RELATIVE TIMING VERIFICATION REVISITED

Technical Report
UPC-DAC-RR-CAP-2008-39

MARC SOLÉ
Department of Computer Architecture
Technical University of Catalonia

ENRIC PASTOR
Department of Computer Architecture
Technical University of Catalonia

Contents

List of Figures

List of Tables

List of Abbreviations

BIFTS	Class of Bounded Inconsistent Fragments Transition Systems, page 191.
BNCTS	Class of Bounded Negative Cycles Transition Systems, page 194.
BRIFTS	Class of Bounded Reduced Inconsistent Fragments TSs , page 194.
CAD/CAV	Computer Aided Design / Computer Aided Verification.
CES	Causal Event Structure, page 106.
CESA	Causal Event Structure with Asymmetric Conflict, page 110.
CESAR	Causal Event Structure with Asymmetric Conflict and Restrictions, page 114.
ECTIA	Enabling Compatible Trace Identifier Automaton, page 92.
ECTRA	Enabling Compatible Trace Remover Automaton, page 97.
EF	Enabling Function, see §2.5.2
ER	Enabling Region, page 15.
FF	Firing Function, see §2.5.2
FR	Firing Region, page 16.
FSSTS	Class of Failure Strongly Synchronized Transition Systems, page 189.
$\mathbf{g_0LP}$	0-substituted generic timing consistency system, page 84.
gLP	Generic timing-consistency system of a trace, page 76.
GRC	Graph of Reachable Configurations, page 107.
GRE	Graph of Reachable Enablings, page 108.
LB	Set of Lower Bounded Events, page 75.
LP	Linear Programming problem to test timing consistent of traces, page 72.
LzCES	Lazy Causal Event Structure, page 130.
LzTS	Lazy Transition System, page 16.
NZTS	Class of Non-Zeno timed Transition Systems.
pTR	Partial Disjunctive Transition Relation, page 23.
Reach	Reachable set of states, page 11.
SD	States not belonging to any of the enabling regions of a set of events, see §6.2.
SE	States that belong to all the enabling regions of a set of events, see §6.2.
SPTE	System of Parametric Timed Equations, page 76.
SSTS	Class of Strongly Synchronized Transition Systems, page 189.
TR	Transition Relation, see §2.5.2.
TS	Transition System, page 10.
TTS	Timed Transition System, page 12.

Chapter 1

INTRODUCTION

1.1 Motivation

How can we be sure that something we have designed will work as intended? Most common practice is to test it. For instance, assume we are given some pieces of wood and we are told to build a chair that lasts at least ten years. We would cut and assemble the different pieces in some way that we find reasonable. But if we had no previous experience constructing chairs, we will not be sure it supports our weight until we finish and we test it.

However, test only provides a partial assurance. It could be that the chair works in our tests, but collapses in some other situations. An obvious improvement is to test it in all possible situations, but this is usually impossible. In this example, as time degrades the chair, we can only be sure that it will last ten years, when this period of time has elapsed. But, ideally, we would like an *a priori* assurance, not an *a posteriori* confirmation.

Compare this situation with the following story. In 1870 a man offered the publication of a puzzle (Figure 1.1) to the editor of a New York newspaper. The first solution sent to the newspaper would have a \$1000 reward. Considering the amount of money, the editor was reluctant, although the man assured that the puzzle was unsolvable. He was so convinced that no solution existed that he offered to pay himself the reward and, finally, the puzzle was published.

The man of the story is Samuel Loyd, one of the most important puzzle authors and recreational mathematicians of XIX century. Although it seems that, in fact, he did not invent this particular puzzle, what is interesting in this story is his confidence. Where it came from? The “secret” was a mathematical proof that the puzzle is unsolvable. With the proof, he could sleep as a baby because his money was at no risk. Without it, one might simply start looking for a solution, but there are just too many possibilities to test them all, and confidence turns into doubt.

We would like to achieve the maximum degree of confidence in every system we design. As the story illustrated, this implies to use mathematics in one or several phases of the design. Broadly speaking, all the techniques devised to increase confidence in a system by incorporating mathematical reasoning into its design/analysis are called *formal methods*.

Historically, formal methods originated to produce better software and hardware systems. However, because of the all-pervasive nature of mathematics, they are now used in many different areas: biology, artificial intelligence, mathematics, etc. For instance they can be used to determine if the puzzle is solvable or not.

What fostered the creation of formal methods in computer science was the increase in size and functionality of hardware and software systems. Unprecedent levels of complexity imply higher

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

Figure 1.1. A famous puzzle. The objective is to order all tiles using their number. In this particular case the objective is to exchange tiles 14 and 15 as the rest are already ordered. To do so, tiles can be displaced to the empty place, but never extracted from the board.

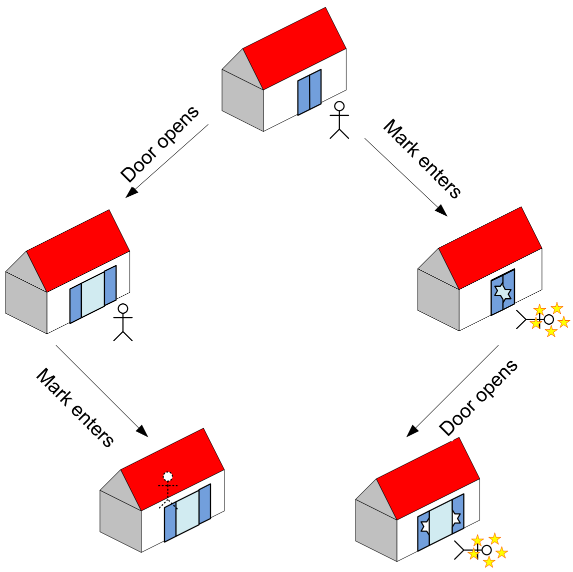


Figure 1.2. Example of a timed system. In initial state Mark is outside the building (top). All the possible future states of the system are shown, without considering the time required by each one of the actions. When these times are taken into account, it can be that all scenarios are possible or that only some of them can happen. For instance, if the door takes two seconds to open, and Marks takes three second to reach the door, only the left sequence of events is possible.

probabilities of introducing design errors. This fact, combined with the ubiquity of such systems, made it necessary to develop techniques that help to reduce the probability of failures.

As formal methods encompass an enormous amount of different techniques, it is very difficult to make a general contribution. In this work we focus on a particular subset of them, namely the methods used for the *verification of timed systems*.

1.2 Verification of timed systems

A timed system is a system whose correct operation depends on the timing of its elements.

EXAMPLE 1.1 *For instance, imagine the system composed by a person (Mark) that wants to enter a building, and the automatic door that is its main entrance. We assume Mark is walking inattentively to the door and is not checking if it opens. He will take some time to reach it and* *enter the building. As he approaches, a sensor detects him, and the control logic orders the door to open. However, it will take some time to open enough to avoid a collision.*

Clearly, if Mark is walking too fast and the door opens slowly, we will witness an accident. As the difference between a happy end and a bad one depends on the timing of the elements of the system, this is a timed system. ■ 1.1

Formal verification consists in building a mathematically-based proof that a system (implementation) behaves according to a given specification. In our example the specification is "Mark must not collide with the door". To do this check, we must consider all possible behaviors of the system *i.e.* all the states in which the system can ever be. Although in our toy example there are only five possible states (Figure 1.2), for complex systems there can be an enormous amount of them. This problem is known as the *state-explosion problem*.

To solve the state-explosion problem one of the most common techniques is to use a symbolic encoding of the states. Instead of storing each state individually, which is impossible in many cases, we store an equation that represents a set of states. For instance in a plane, the equation $x < y$ represents an infinite set of points. Even if we restrict only to integer coordinates, it would be impossible to store all the points explicitely, but is perfectly possible to store the equation.

One of the techniques that try to reduce the complexity gap between timed and untimed verification is *iterative refinement*. The key idea is to start by removing all timing information from a timed system. Then all states are computed, a step that now has a complexity equal to untimed verification. If some violation of the specification is detected, we check if it is a valid behavior considering the timing information. If it is, then a counterexample has been found. Otherwise, some timing information that removes this behavior is added into the system. Then a new iteration is started and all states of this new system are computed. The process continues until we find a counterexample or no specification violation exists.

Obviously, iterative refinement methods have a worst case complexity equal to the timed verification problem: it is possible that we have to consider *all* the timing information to prove the correctness of the system. However, frequently they avoid to compute the exact timed state space, saving a lot of computational effort. In this sense they are perfect examples of a lazy approach.

Iterative refinement is not a single technique, but a family of methods. One of these methods uses *relative timing*: instead of considering the exact time at which events can happen, it only considers their relative orders. For instance, using our initial example, assume that we know that Mark reaches the door in three seconds, and that the door opens in two seconds. In this case the only relevant aspect for relative timing is that the door opens *before* Mark reaches it. By introducing this type of precedences between events is possible to simulate the effects of timing but with a theoretical complexity similar to the verification of untimed systems.

The use of relative timing for the verification of timed systems was introduced in [PCKP00, Peñ03]. However only a very basic methodology was provided. In this work we try to improve it in all directions: giving a sound theoretical foundations as well as improving its efficiency.

1.3 Objectives

The main objective of this work is to *improve the verification of timed concurrent systems using relative timing*. We divide this objective in several subobjectives:

- To improve the efficiency of symbolic reachability analysis for concurrent systems.
- To establish a sound theoretical foundations of relative timing verification and study its theoretical limitations.

- With the insight of the new theory, develop a new methodology that is less conservative and more efficient than previous related approaches ([Peñ03]).

1.4 Structure and contributions

The rest of this document is organized as follows. Part I gives a brief summary of the background needed to read the rest of the parts. It comprises three chapters:

- Chapter 2 introduces the different formal models used in this work. Several types of transition systems are described, together with some of their basic properties. Moreover, we explain how can we represent them symbolically by using Boolean algebras, and which data structures are used for its symbolic manipulation (partial transition relations, ROBDDs, . . .).
- Chapter 3 reviews the mathematical concept of *order* and analyzes how different types of relations between events (causality, disabling, . . .) can be computed in our models.

Part II is dedicated to the fundamentals of the verification of timed concurrent systems using relative timing, with the following structure:

- Chapter 4 introduces general background on the formal verification of timed systems and reviews the significant previous work on this area.
- In Chapter 5 we review the concept of relative timing verification and we establish its theoretical grounds at the trace level. The keystone concept is the *enabling compatible relation* that allows to extrapolate the result of the analysis of one trace to a whole set of traces.
- Chapter 6 shows how we can remove traces that are known to be timing inconsistent, by composing with a special type of automaton. The use of an automaton allows a better treatment of some special situations like indeterminacy.
- To consider traces individually reduces the efficiency of the methodology, hence in Chapter 7 we present a mathematical structure, called *event structure*, that can represent compactly the behavior of a set of traces. Although there are many different types of event structures in the literature, the necessities of our methodology oblige us to develop several tailored types of event structures.
- Chapter 8 deals with the algorithms needed to analyze the timing relations between the events of an event structure. We show that the results of previous approaches [Peñ03] have to be carefully interpreted or, otherwise, wrong conclusions can be achieved. In contrast, we present some algorithms that allow a precise temporal analysis.
- Finally, Chapter 9 introduces a basic verification methodology by putting together all the elements seen in previous chapters. This allows to see how different theoretical parts collaborate to verify a system. Unfortunately, the verification methodology presented contains many defects. However, as these flaws are related to the advanced topics of Part III, the methodolgy is a good starting point for that part.

Part III covers more advanced topics of the relative timing verification and, in particular, addresses all the problems of the preliminar verification methodology presented in Chapter 9. It comprises five chapters:

- Chapter 10 analyzes in depth the indeterminacy problem arising in the ECTRA construction, and how we can solve it by using tailored determinization and minimization algorithms.

- Event structures have to fulfill some conditions if we do not want a conservative analysis. Chapter 11 explains these conditions and provides algorithms to compute event structures that satisfy them.
- Chapter 12 studies the convergence of methods based in the same theoretical tools as our methodology and describes which types of systems can be verified with it.
- Chapter 13 presents the new verification methodology that we propose, based on the theoretical results developed in previous chapters.
- Chapter 14 explains several optimization of the methodology that increase its efficiency, together with some practical implementation aspects.
- Chapter 15 shows examples of the applicability of the methodology, where different types of concurrent systems are verified.
- Finally, Chapter 16 summarizes the conclusions and contributions of this work and outlines some open areas for future research.

Additionally, Appendix A provides a detailed comparison with the most related previous work of [Peñ03], while Appendix B contains most of the lengthier proofs that appear in this work.

PART I

BACKGROUND

Chapter 2

MODELS FOR CONCURRENT SYSTEMS

Summary

Depending on what kind of information is considered relevant, different models abstract different aspects of a system to preserve only its essential elements. As many abstractions are possible, there are many models available. This chapter overviews some of the mathematical models used to represent concurrent systems.

We focus on *transition systems* and its derivatives, that are used in subsequent chapters to model and verify concurrent systems. Untimed systems are directly modeled as transition systems but some extension of the model is needed for timed systems. For this reason we introduce *timed transition systems* and *lazy transition systems*.

Timed transition systems provide a simple way of specifying a timed system, but they are not a suitable model for the computations required by the verification process. Therefore our verification methodology uses a more convenient type of transition systems, called lazy transition systems, as its fundamental model. This type of transition system allows to incorporate timing information incrementally, while keeping the representation requirements low.

The size of the representation of a model is a key element in its efficient manipulation. Computers do not interact directly with the models, but they operate on data structures that represent them. Thus it is important to know the most relevant facts about their representation. We devote the second half of this chapter to explain how lazy transition systems can be encoded for efficient automated operation.

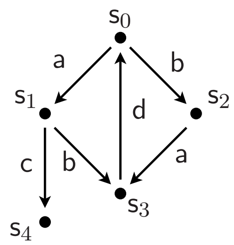


Figure 2.1. An example of transition system.

2.1 Transition systems

Transition systems (TS) are a formalism used to describe systems of concurrent processes [Arn94]. The formalism, although mathematically simple, can model most of the properties of such systems, and so can be used to study their semantics.

Intuitively, a *transition system* consists of the set of possible states of a system, and a set of transitions that the system can produce to change from one state to another. Hence it is a *state-based* model, in contrast with event-based models such as Petri nets, which offer a higher level of abstraction.

DEFINITION 2.1 (TRANSITION SYSTEM)
A transition system (TS) *[NRT92] is a quadruple* $A = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$*, where* S *is a non-empty set of* states, Σ *is a non-empty alphabet of* events, $T \subseteq S \times \Sigma \times S$ *is a* transition relation*, and* s_0 *is the* initial state.
The elements of T *are called* transitions *and are indistinctly denoted by* $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}'$ *or by* $(\mathsf{s}, \mathsf{e}, \mathsf{s}')$.
Given the significance of individual events, the transition relation of a TS can be naturally partitioned into a disjoint set of relations, one for each event $\mathsf{e} \in \Sigma$*:* $T_{\mathsf{e}} = \{\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}' \in T\}$.
Obviously, $T = \bigcup_{\forall \mathsf{e} \in \Sigma} T_{\mathsf{e}}$.
An event e *is* enabled *at state* s *if* $\exists\, \mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}' \in T$. *We will denote by* $\mathcal{E}(\mathsf{s})$ *the set of events enabled at state* s. *The* firing region *of event* e *is defined as* $\mathrm{FR}(\mathsf{e}) = \{\mathsf{s} \in S \mid \mathsf{e} \in \mathcal{E}(\mathsf{s})\}$, i.e. *the set of states where* e *is enabled.*
■ 2.1

A TS is *finite* if S and Σ are finite. A TS is called *deterministic* if for each state s and each event e there is at most one state s′ such that $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}'$. In the sequel, only finite transition systems will be considered. Sometimes we use a more general form of TS, in which there is not a single initial state s_0, but an initial set of states usually denoted by S_{in}. For the type of analysis we perform this difference is not relevant.

EXAMPLE 2.1 *A TS can be represented by an arc-labeled directed graph. A simple example of a TS is shown in Figure 2.1. States are represented as dots, transitions are the directed arcs between the states, and the events are the labels of the transitions. That is,* $S = \{\mathsf{s}_0, \ldots, \mathsf{s}_4\}$, $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ *and* $T = \{\mathsf{s}_0 \xrightarrow{\mathsf{a}} \mathsf{s}_1, \mathsf{s}_1 \xrightarrow{\mathsf{b}} \mathsf{s}_3, \ldots\}$. *As an example, the firing region of event* b *is* $\mathrm{FR}(\mathsf{b}) = \{\mathsf{s}_0, \mathsf{s}_1\}$ *and* $T_{\mathsf{b}} = \{\mathsf{s}_0 \xrightarrow{\mathsf{b}} \mathsf{s}_2, \mathsf{s}_1 \xrightarrow{\mathsf{b}} \mathsf{s}_3\}$, $T_{\mathsf{c}} = \{\mathsf{s}_1 \xrightarrow{\mathsf{c}} \mathsf{s}_4\}$.
■ 2.1

DEFINITION 2.2 (RUN)
A run *of a TS* $A = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$ *is a sequence of transitions* $\rho = \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \xrightarrow{\mathsf{e}_2} \cdots$*, such that* $\mathsf{s}_1 = \mathsf{s}_0$ *and* $\forall i \geq 1 : \mathsf{s}_i \xrightarrow{\mathsf{e}_i} \mathsf{s}_{i+1} \in T$. *Event* e_i *is said to* fire *at step* i *of the run.*

■ 2.2

With an abuse of notation, the expressions $\mathsf{s}_i \in \rho,\ \mathsf{s}_i \xrightarrow{\mathsf{e}_i} \mathsf{s}_{i+1} \in \rho,\ \mathsf{s}_i \xrightarrow{\mathsf{e}_i} \in \rho,\ \xrightarrow{\mathsf{e}_i} \mathsf{s}_{i+1} \in \rho,$ etc, will be often used to denote the fact that different fragments of a sequence belong to a run.

A state s' is said to be reachable from state s if there is a run between both, denoted by $\mathsf{s} \xrightarrow{\rho} \mathsf{s}'$, or simply $\mathsf{s} \xrightarrow{*} \mathsf{s}'$ if the run is not relevant. We assume $\mathsf{s} \xrightarrow{*} \mathsf{s}$ is always true *i.e.* a state is always reachable by itself. The set of states reachable through all possible runs of the system is given by the following definition:

Definition 2.3 (Reachable states)
Given a TS $A = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$*, the set of reachable states from a state* s *is defined as:*

$$\mathrm{Reach}(\mathsf{s}, T) = \{\mathsf{s}' \mid \mathsf{s} \xrightarrow{*} \mathsf{s}'\}$$

■ 2.3

A usual assumption is that, for any TS, $S = \mathrm{Reach}(\mathsf{s}_0, T)$. However, in this work, we only enforce that $S \supseteq \mathrm{Reach}(\mathsf{s}_0, T)$ because it reflects more accurately what happens when TS s are encoded symbolically (see §2.5.2).

Many verification approaches rely on the computation of the set of reachable states. A useful function in this computation is the function Image. Given a set of states and a transition relation, this function returns the successors of these states using the transition relation.

Definition 2.4 (Image states)
Given a set of states X *and a transition relation* T *the set of image states of* X *by* T *is defined as:*

$$\mathrm{Image}(X, T) = \{\mathsf{s}' \mid \mathsf{s} \in X \ \wedge\ \mathsf{s} \longrightarrow \mathsf{s}' \in T\}$$

■ 2.4

For instance in Figure 2.1, $\mathrm{Image}(\{\mathsf{s}_0, \mathsf{s}_1, \mathsf{s}_4\}, T_{\mathsf{b}}) = \{\mathsf{s}_2, \mathsf{s}_3\}$.

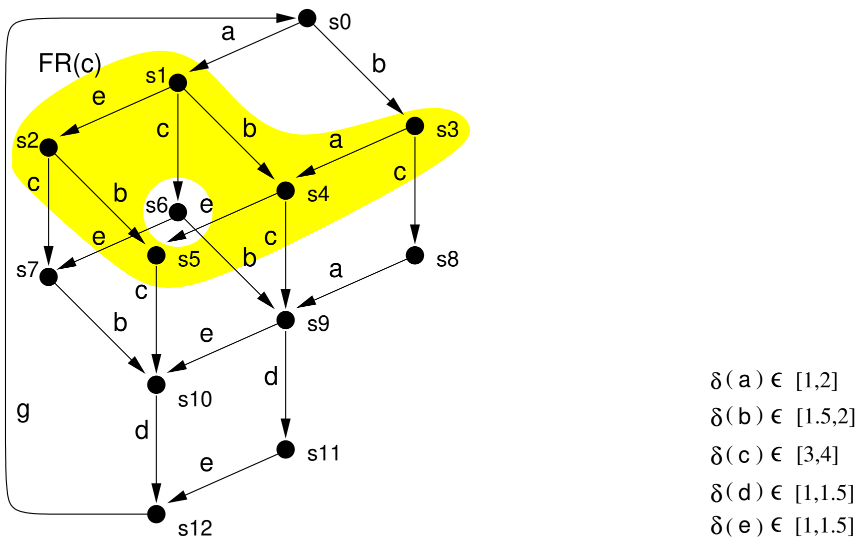
2.2 Timed transition systems

Timed transition systems (TTS) [HMP92a] allow to model systems in which timing information is particularly relevant for their operation, such as real-time systems. TTSs generalize the basic computational model of transition systems by associating minimum and maximum delays to the transitions. The real line is generally used as timed domain although integer and rational values are also used by some authors.

Verification methods for timed transition systems have been developed for various logical specification languages. The methods include both algorithmic techniques for finite-state systems [AH90, HLP90, Ost90] and deductive techniques based on proof systems [Hen90, Ost90, HMP91].

Time is incorporated to transition systems by assuming that transitions happen instantaneously, while minimum and maximum delay bounds restrict the times at which transitions may occur. The delay bounds ensure that transitions occur neither too early (*i.e.* never before the minimum delay bound) nor too late (*i.e.* never after the maximum delay bound). The absence of a lower delay bound requirement is modeled by a minimum delay bound of 0, whereas the absence of an upper delay bound requirement is modeled by a maximum delay bound of ∞. Formally:

Definition 2.5 (Timed Transition System)
A timed transition system (TTS) *[HMP92a] is a triple* $A = \langle A^-, \delta^l, \delta^u \rangle$*, where* $A^- = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$ *is a* TS *called the* underlying transition system*,* $\delta^l : \Sigma \to \mathbb{R}^+$ *and* $\delta^u : \Sigma \to$



(a) (b)

Figure 2.2. An example of timed transition system: (a) underlying TS, (b) associated delay intervals. The firing region of c is shadowed.

$\mathbb{R}^+ \cup \{\infty\}$ *respectively associate a minimum and a maximum* delay bound *to each event, such that* $\forall\, \mathsf{e} \in \Sigma : \delta^l(\mathsf{e}) \leq \delta^u(\mathsf{e})$.

■ 2.5

EXAMPLE 2.2 *Figure 2.2 depicts an example of* TTS *described by means of: the underlying* TS *, where the firing region of event* c *is shadowed, and the delays associated to the events of the system. If the delay of an event is omitted we assume it belongs to the less constraining interval* $[0, \infty)$*. This is the case of event* g*, for example.*

■ 2.2

The TS A^- captures the behavior of the system in the untimed domain, *i.e.* without considering the delay information. The behavior of the system in the timed domain is obtained by associating to each state a time stamp that indicates when the state was reached. Thus, the same state can be visited at different time instants depending on its prehistory, and the time stamps must comply with the delay bounds specified for each event (see Definition 2.6). Conversely, a state can never be reached in the timed domain if its incoming transitions are never fired due to their respective delay constraints. Time is always relative to that of the initial state of the system, and must monotonically increase along the firing sequences.

Because of the time dimension, the computation of the exact timed state space of a system has been proved to be a PSPACE-complete problem [AD90] and demonstrated to be a highly complex task in several contexts such as real-time systems [AD94, HMP91] and asynchronous circuits [DKMW92, Bur92, HB94, MP95a, SY96, VdJL96].

The behavior of a TS is represented by means of runs. Similarly, behavior in a TTS is represented using runs in which time is added, called *timed runs*. A *timed run* assigns a time stamp to each of its states and can be represented as a sequence of the form: $(\mathsf{s}_1, \tau_1) \xrightarrow{\mathsf{e}_1} (\mathsf{s}_2, \tau_2) \xrightarrow{\mathsf{e}_2} \cdots$ where, for all $i > 0$:

- each pair (s_i, τ_i) is called a *timed state* with $\mathsf{s}_i \in S$ and $\tau_i \in \mathbb{R}^+$ and

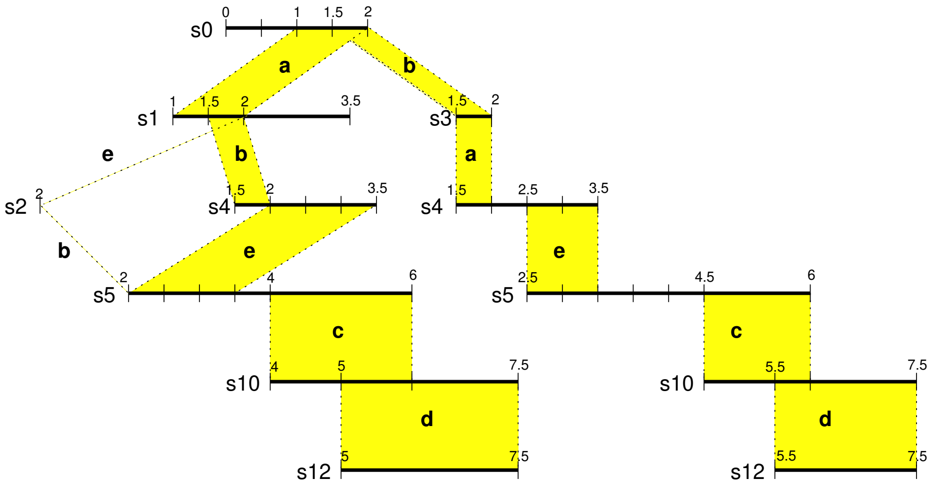


Figure 2.3. Portion of the timed state space of the TTS of Figure 2.2.

- $\mathsf{s}_i \xrightarrow{\mathsf{e}_i} \mathsf{s}_{i+1} \in T$.

Also $\tau_1 = 0$ and $\lim_{i\to\infty} \tau_i = \infty$.

However not all timed runs represent allowed behavior of the system, as they do not necessarily respect the delay bounds of the events. The timed runs that are actually feasible when we take into account these bounds are called *timing consistent runs* and are defined as follows.

Definition 2.6 (Timing-consistent run)
Let $A = \langle A^-, \delta^l, \delta^u \rangle$ *be a* TTS *and let* $\rho = (\mathsf{s}_1, \tau_1) \xrightarrow{\mathsf{e}_1} (\mathsf{s}_2, \tau_2) \xrightarrow{\mathsf{e}_2} \cdots$ *be a timed run of* A. ρ *is* timing consistent *with* A *if the sequence of time stamps* $\tau_1 \le \tau_2 \le \cdots$ *is monotonically increasing and satifies:*

$\forall \mathsf{s}_i \xrightarrow{\mathsf{e}_i} \mathsf{s}_{i+1} \in \rho$ *and* $\forall \mathsf{e} \in \mathcal{E}(\mathsf{s}_i)$ *such that* e *became enabled in state* s_j *and has been been continuously enabled since then, we have that*

- $\mathsf{e} = \mathsf{e}_i \;\Rightarrow\; \delta^l(\mathsf{e}) \le \tau_{i+1} - \tau_j$ i.e. *an event cannot fire before its minimum delay has elapsed, and*
- $\tau_{i+1} - \tau_j \le \delta^u(\mathsf{e})$ i.e. *an event cannot remain enabled after its maximum delay has elapsed.*

■ 2.6

In the definition, time stamp τ_{i+1} corresponds to the *firing time* of event e_i along ρ. Similarly, τ_j corresponds to its *enabling time*. Thus, the firing time of an event only depends on its enabling time plus certain delay amount within the given bounds. On the other hand, if an event is disabled, its disabling occurs before its maximum delay has elapsed, as otherwise it should have already fired.

It is customary to extend the definition to the runs of A^-, so that a run $\rho = \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \xrightarrow{\mathsf{e}_2} \cdots$ is said to be timing consistent if any of its timed runs is timing consistent.

Finally, a run is *Zeno* if it contains an infinite number of transitions in a finite amount of time [AH97], *i.e.* $\sum_{1\le i<\infty}(\tau_{i+1} - \tau_i) < \infty$. For example, that could be the case produced by a cycle of events with 0 delay. Without loss of generality, in the sequel we will only consider systems modeled by *non-Zeno* TTS s, that is, systems that can not produce Zeno runs.

EXAMPLE 2.2 (CONT.) *Figure 2.3 depicts a portion of the timed state space of the* TTS *of Figure 2.2. The horizontal axis represents the time intervals in which the system remains at each state, assuming that time begins at state* s_0. *Notice that some states can be reached, and also left, at different time instants depending on their prehistory.*

To illustrate the behavior of the system in the timed domain assume the execution starts at the timed state $(\mathsf{s}_0, 0)$. *The system remains at such state until the minimum delay of* a *elapses. Then the system can either fire* a *or wait until* b *becomes fireable* 0.5 *time units later. Thus,* a *can fire between time* $0 + \delta^l(\mathsf{a}) = 1$ *and time* $0 + \delta^u(\mathsf{a}) = 2$, *whereas* b *can fire between time* $0 + \delta^l(\mathsf{b}) = 1.5$ *and time* $0 + \delta^u(\mathsf{b}) = 2$. *Assume* a *fires at time* 1.5 *leading to the timed state* $(\mathsf{s}_1, 1.5)$. *Now* b *is still enabled and can fire, while* e *and* c *become newly enabled.* e *cannot fire until time* $1.5 + \delta^l(\mathsf{e}) = 2.5$, *whereas* c *is much slower and cannot fire until* $1.5 + \delta^l(\mathsf{c}) = 4.5$. *Therefore, if for example* b *fires at time* 1.5 *leading to the timed state* $(\mathsf{s}_4, 1.5)$, *it will be followed by* e *and* c, *leading for example to the timed states* $(\mathsf{s}_5, 2.5)$ *and* $(\mathsf{s}_{10}, 4.5)$ *successively. The execution continues by firing* d, *and so on.*

None of the states s_5, s_6, s_7, s_8, s_9 *or* s_{11} *is reachable in the portion of the timed domain shown in the figure. For example, state* s_8 *can not be reached from* s_3 *by firing* c, *since* c *is slower than* a, *even in the case that both events were enabled at the same instant. In fact, only the following three runs are potentially timing consistent if appropriate time stamps are assigned to each state:* $\mathsf{s}_0 \xrightarrow{\mathsf{a}} \mathsf{s}_1 \xrightarrow{\mathsf{e}} \mathsf{s}_2 \xrightarrow{\mathsf{b}} \mathsf{s}_5 \xrightarrow{\mathsf{c}} \mathsf{s}_{10} \xrightarrow{\mathsf{d}} \mathsf{s}_{12} \cdots$, $\mathsf{s}_0 \xrightarrow{\mathsf{a}} \mathsf{s}_1 \xrightarrow{\mathsf{b}} \mathsf{s}_4 \xrightarrow{\mathsf{e}} \mathsf{s}_5 \xrightarrow{\mathsf{c}} \mathsf{s}_{10} \xrightarrow{\mathsf{d}} \mathsf{s}_{12} \cdots$ *and* $\mathsf{s}_0 \xrightarrow{\mathsf{b}} \mathsf{s}_3 \xrightarrow{\mathsf{a}} \mathsf{s}_4 \xrightarrow{\mathsf{e}} \mathsf{s}_5 \xrightarrow{\mathsf{c}} \mathsf{s}_{10} \xrightarrow{\mathsf{d}} \mathsf{s}_{12} \cdots$. *For example, the following timed run is timing consistent:*

$$(\mathsf{s}_0, 0) \xrightarrow{\mathsf{a}} (\mathsf{s}_1, 1) \xrightarrow{\mathsf{b}} (\mathsf{s}_4, 1.5) \xrightarrow{\mathsf{e}} (\mathsf{s}_5, 2) \xrightarrow{\mathsf{c}} (\mathsf{s}_{10}, 4) \xrightarrow{\mathsf{d}} (\mathsf{s}_{12}, 5) \cdots$$

Conversely, the following timed run is not timing consistent:

$$(\mathsf{s}_0, 0) \xrightarrow{\mathsf{a}} (\mathsf{s}_1, 1) \xrightarrow{\mathsf{b}} (\mathsf{s}_4, 2) \xrightarrow{\mathsf{e}} (\mathsf{s}_5, 3.5) \xrightarrow{\mathsf{c}} (\mathsf{s}_{10}, 5) \xrightarrow{\mathsf{d}} (\mathsf{s}_{12}, 5) \cdots$$

since the firing of e *happens at time* 3.5, *but according to its delays and the firing of its trigger* a *at time* 1, e *must fire between time* 2 *and time* 2.5. ■ 2.2

To conclude this section, we want to remark that different notions of timed transition systems with timing constraints for the discrete-time paradigm have been also proposed by several authors [PH88, Ost90, HMP91]. For instance *timed transition models* [Ost90] are fair transition systems [MP95b] extended with an explicit global integer clock. Similarly, *clocked transition systems* were defined in [MP96] as an alternative model for real-time systems and an evolution of TTSs.

In TTSs time is implicit in the sense that the states do not contain timing information and a timestamp must be used to represent it explicitly. On the other hand, in *clocked transition systems*, time is explicitly represented by means of a set of timers (often called clocks) which increase uniformly whenever time progresses, and can be set to specific values by the firing of transitions. This type of transition systems is inspired by a more commonly used model called *timed automata* which is described in detail in Chapter 4.

The use of explicit clocks has the advantage that no specially modified logic is needed to specify timing properties, but is tedious to use when describing digital systems (because transitions must reset clocks very often, which decreases readability). In this work we focus on safety properties, so we will have no benefit for this extra burden, and therefore we use TTSs.

2.3 Lazy transition systems

When time becomes an essential magnitude in a model for concurrent systems, a complexity dimension is added to the analysis. In formalisms where time is an explicit magnitude, such

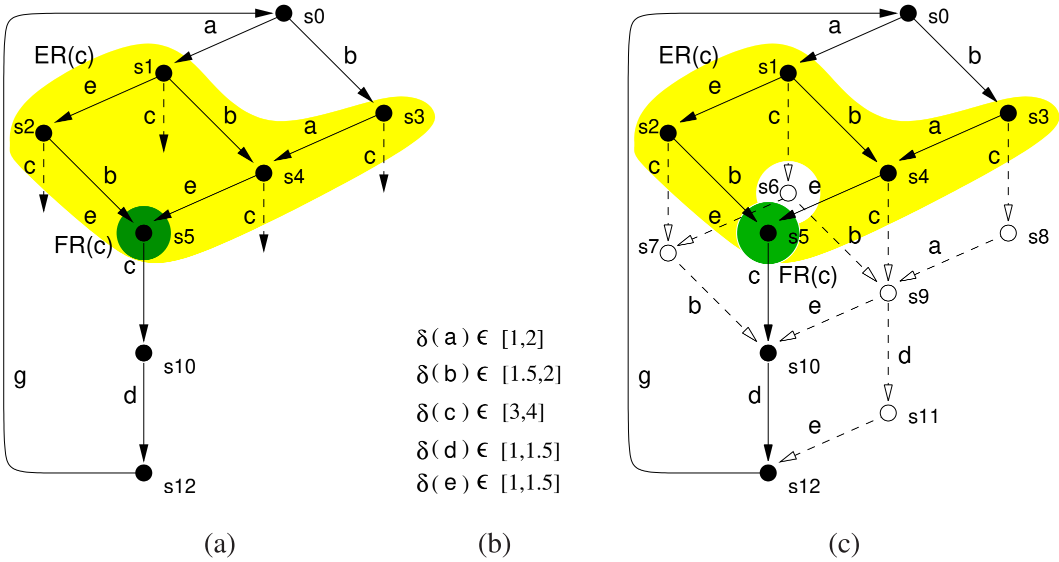


Figure 2.4. An example of lazy transition system: (a) *Classical* LzTS corresponding to the TTS in Figure 2.2. (b) Delays associated to the events. (c) *New* LzTS for the TTS in Figure 2.2.

complexity is specially noticeable when computing the timed state space of the system. For example, the problem of reachability in *timed automata* is proved to be PSPACE-hard [AD94], where the exponentiality depends on the number of clocks and on the encoding of the maximum values that can be taken by the clocks (see §4.2.2). This complexity makes the analysis of systems with a moderate amount of untimed states almost impractical.

Several approaches have been devised to represent timed states in a succinct form, *e.g.* [BM00]. However, the incorporation of the timed domain in the representation of the states hampers an efficient representation of large state spaces with BDDs [Bry86]. Even the discretization of time [HMP91] poses serious problems when the number of clocks or the constants of the timing constraints are large. An interesting approach to face this complexity problem was proposed in [AK95], where the clocks used during the analysis of the system and their accuracy are determined dynamically upon demand. In this way, only that timing information relevant to the analysis emerges during the calculation of the reachable states.

In some cases, however, rather than calculating the exact time intervals in which each state can be visited by any valid run of the system, the only information required is to know whether each state is visited by some timing-consistent run and which events can actually fire there. In other words, only the set of reachable states in the timed domain and the validity (according to the timing information) of each of the transitions are required. This information can be represented by abstracting absolute time information out of the model.

The abstraction leads to the definition of a new computational model called *lazy transition systems* [CKK+98], in which timing information is represented in terms of the notion of *laziness*. This notion explicitly distinguishes among the enabling and the firing of an event, assuming certain implicit delay between them. Formally:

Definition 2.7 (Lazy Transition System [CKK+98])

A lazy transition system (LzTS) *is a five-tuple* $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{ER}\rangle$, *where* $\langle S, \Sigma, T, \mathsf{s}_0\rangle$ *is a* TS*, and the function* $\mathrm{ER} : \Sigma \rightarrow 2^S$ *defines the* enabling region *of each event.*

■ 2.7

Event e is said to be *enabled* at state $\mathsf{s} \in S$ if $\mathsf{s} \in \mathrm{ER}(\mathsf{e})$. Similarly, event e is said to be *fireable* at state $\mathsf{s} \in S$ if $\mathsf{s} \in \mathrm{FR}(\mathsf{e})$. For each event $\mathsf{e} \in \Sigma$ the condition $\mathrm{FR}(\mathsf{e}) \subseteq \mathrm{ER}(\mathsf{e})$ must hold. Event e is said to be *lazy* if $\mathrm{FR}(\mathsf{e}) \neq \mathrm{ER}(\mathsf{e})$. Therefore, any state of a LzTS in the set $\mathrm{ER}(\mathsf{e}) \setminus \mathrm{FR}(\mathsf{e})$ is a state in which the event e is enabled but cannot fire due to the delays associated to the events of the system.

EXAMPLE 2.2 (CONT.) *Figure 2.4 (a) shows the lazy transition system corresponding to the timed transition system in Figure 2.2. Analyzing the delays, it can be proved that event* c *is always slower to fire than events* a, b *and* e*. Therefore,* c *becomes lazy in those states where it is concurrently enabled with those faster events. Thus, although* c *is enabled in states* $\mathrm{ER}(\mathsf{c}) = \{\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3, \mathsf{s}_4, \mathsf{s}_5\}$*, it can only fire in* $\mathrm{FR}(\mathsf{c}) = \{\mathsf{s}_5\}$*, once* a, b *and* e *have already fired.*
■ 2.2

Although this definition of LzTS has many advantages and, in fact, we use it to implement our methodology, it loses information: transitions in the lazy region are *lost*. This can cause some trouble depending on what kind of information you want to obtain from the system. For this reason we prefer to develop our theoretical model using a slightly different definition of a LzTS in which this information is preserved.

DEFINITION 2.8 (LAZY TRANSITION SYSTEM)
A lazy transition system (LzTS) *is a five-tuple* $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR}\rangle$ *, where* $\langle S, \Sigma, T, \mathsf{s}_0\rangle$ *is a TS, and the function* $\mathrm{FR} : \Sigma \rightarrow 2^S$ *defines the* firing region *of each event. The* enabling region *of each event, on the other hand, is defined as the set of states in which that particular event is enabled* i.e. *all the states from where there is an outgoing transition labeled with it.*
■ 2.8

The main difference with Definition 2.7 is that now the enabling regions are implicitly defined by the transition relation T, and we do restrict the firing of the events using their firing regions. In former definition, on the contrary, the enabling region was explicitly defined, while the firing regions were obtained implicitly from T.

Note that from this kind of LzTS one can always obtain the *classical* LzTS. Henceforth whenever we talk about a LzTS we refer to Definition 2.8. If we need to refer to Definition 2.7 we will use the term *classical LzTS* . As Figure 2.4 shows, the graphical representation of the classical form is simpler. Thus, in many cases, we use this representation also for the new LzTSs, to make figures easier to understand.

EXAMPLE 2.2 (CONT.) *Figure 2.4 (c) shows the lazy transition system corresponding to the timed transition system in Figure 2.2 using Definition 2.8. The only difference is that arcs that are not allowed to fire due to laziness of events are preserved. Unreachable states due to laziness are drawn white and arcs not fireable due to laziness of events are dashed and have white arrow.* ■ 2.2

Notice that a TS is just a particular case of LzTS in which both enabling and firing regions coincide for all the events.

The introduction of the new definition of LzTS obliges to make some modifications to the concept of run and image states.

DEFINITION 2.9 (RUN OF A **LzTS**)
A run *of a lazy transition system* $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR}\rangle$ *is a sequence of transitions*

$\rho = \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \xrightarrow{\mathsf{e}_2} \cdots$, *such that* $\mathsf{s}_1 = \mathsf{s}_0$ *and* $\forall i \geq 1 : \mathsf{s}_i \xrightarrow{\mathsf{e}_i} \mathsf{s}_{i+1} \in T$. *If all transitions are taken from the firing region of the corresponding event* i.e. $\forall i \geq 1 : \mathsf{s}_i \in \mathrm{FR}(\mathsf{e}_i)$, *then we call the run a* valid run. *Otherwise we will call it a* lazy run.

■ 2.9

The set of lazy runs represents the forbidden behavior of the system due to the laziness of events.

Changes also enforce the modification of the definition of reachability. Reachable states are only those who are reachable by *firing* events, *i.e.* are reachable through a valid run of the system. This allows to directly extend Definition 2.3 to LzTSs.

The use of LzTSs enables to reason in terms of partial orders of events (see Chapter 3), *i.e.* in terms of the so-called *relative timing* paradigm [SGR99], which is much more intuitive than defining absolute time separations between pairs of events (see also Chapter 4). Moreover, whereas absolute timing information requires complex techniques to represent the space of reachable timed regions or states [Alu98] (*e.g.* using difference bound matrices, time polyhedra, etc.), the generation of the reachable state space for relative timing is of the same theoretical complexity as for untimed systems.

2.4 Models need a representation

In previous sections we have presented the formal models for concurrent systems that we use in subsequent chapters. From the three models presented (TS, LzTS, TTS) our algorithms focus on two of them, since TTSs are used mainly to describe the system we want to verify.

It is theoretically possible to work with an explicit representation of both models, but this greatly restricts the size of the systems that can be verified. Hence an alternative representation is needed.The following sections explain how we can represent these models symbolically using Boolean algebras. As LzTSs generalize TSs, we only discuss LzTSs. Moreover we show what kind of data structures are used to work efficiently with the resulting Boolean expressions.

2.5 Representation of LzTSs

To provide an efficient symbolic representation of LzTSs, we map them onto Boolean algebras. Each state of the system is described by a unique vertex in the algebra. Thus, the sets of states of the system, the functions and transition relations that define the system behavior, and the properties for verification, are all modeled as Boolean functions.

2.5.1 Boolean algebras

A *Boolean algebra* is a fifth-tuple $\langle B, +, \cdot, 0, 1\rangle$, where: B is a set, $+$ and $\cdot$ are binary operators on B that satisfy the commutative and distributive laws; and 0 and 1 belong to B and are respectively the neutral elements of $+$ ($b + 0 = b$) and $\cdot$ ($b \cdot 1 = b$), with $b \in B$. Also, for all $b \in B$ there exists a complement $\overline{b} \in B$ such that $b + \overline{b} = 1$ and $b \cdot \overline{b} = 0$. Under this conditions, the system $\langle \mathbb{B}, +, \cdot, 0, 1\rangle$, with $\mathbb{B} = \{0, 1\}$ and with $+$ and $\cdot$ being the *logic OR* and the *logic AND* operations respectively, is a Boolean algebra.

An n-variable *logic function* $f : \mathbb{B}^n \to \mathbb{B}$ (*i.e.* a *boolean function*) transforms each element $(v_1, \ldots, v_n) \in \mathbb{B}^n$ into an element of $\mathbb{B}$. Let $F_n(\mathbb{B})$ be the set of n-variable logic functions on $\mathbb{B}$, then the system $\langle F_n(\mathbb{B}), +, \cdot, 0, 1\rangle$ is also a boolean algebra, where $+$ and $\cdot$ stand for addition and multiplication of n-variable logic functions, and 0 and 1 stand for the "zero" and "one" functions ($f(v_1, \ldots, v_n) = 0$ and $f(v_1, \ldots, v_n) = 1$, respectively). Given the Boolean algebra of n-variable logic functions, with n symbols $v_1, \ldots, v_n$, we call a *vertex* each element of $\mathbb{B}^n$. A *literal* is either a variable v_i or its complement $\overline{v_i}$. A *cube* c is a set of literals, such that if $v_i \in c$ then $\overline{v_i} \notin c$ and vice versa. A cube is interpreted as the Boolean product of its literals. Note that the cubes with n literals are in one-to-one correspondence with the vertexes of $\mathbb{B}^n$.

Cofactors and *abstractions* are useful operations for the manipulation of Boolean functions. The following functions denote respectively the positive and negative *cofactors* of an n-variable Boolean function $f(v_1, \ldots, v_n)$ with respect to a variable v_i:

$$\begin{aligned} f_{v_i} &= f_{|v_i=1} &= f(v_1, \ldots, v_{i-1}, 1, v_{i+1}, \ldots, v_n), \\ f_{\overline{v_i}} &= f_{|v_i=0} &= f(v_1, \ldots, v_{i-1}, 0, v_{i+1}, \ldots, v_n). \end{aligned}$$

An interesting property of cofactors is given by the Boole's expansion theorem (also known as Shannon's expansion theorem), which shows that a Boolean function can be represented in terms of its cofactors. That is:

$$f(v_1, \ldots, v_n) = \overline{v_i} \cdot f_{\overline{v_i}} + v_i \cdot f_{v_i} = [\overline{v_i} + f_{v_i}] \cdot [v_i + f_{\overline{v_i}}] \tag{2.1}$$

The *existential* and *universal abstractions* of a n-variable Boolean function $f(v_1, \ldots, v_n)$ with respect to a variable v_i are respectively defined in terms of cofactors as follows:

$$\exists_{v_i} f = f_{v_i} + f_{\overline{v_i}} \quad \textit{and} \quad \forall_{v_i} f = f_{v_i} \cdot f_{\overline{v_i}} \ .$$

EXAMPLE 2.3 *To illustrate these concepts, let us consider the function: $f(a, b, c) = bc + a\overline{b}\overline{c} + \overline{a}c$. The positive and negative cofactors with respect to variable a are: $f_a = bc + \overline{b}\overline{c}$ and $f_{\overline{a}} = c$. The abstractions with respect to variable a are: $\exists_a f = f_a + f_{\overline{a}} = \overline{b} + c$ and $\forall_a f = f_a \cdot f_{\overline{a}} = bc$. The existential abstraction $\exists_a f$ is the function that evaluates to 1 for all those values of b and c such that there is a value of a for which f evaluates to 1. The universal abstraction $\forall_a f$ is the function that evaluates to 1 for all those values of b and c such that f evaluates to 1 for any value of a.* ■ 2.3

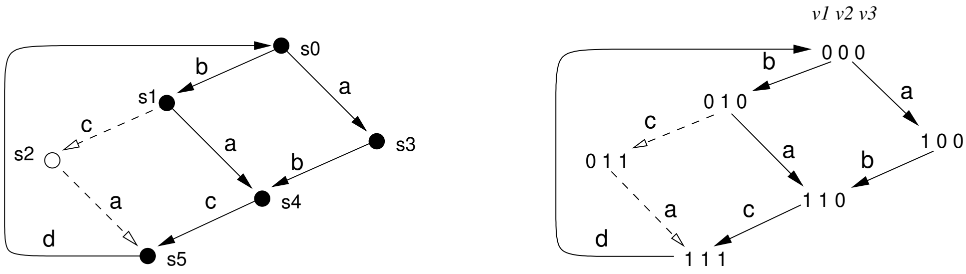
On the other hand, is well known that given a finite set S, the system $\langle 2^S, \cup, \cap, \emptyset, S\rangle$ is also a Boolean algebra, *i.e.* the algebra of subsets of S. The *representation* theorem (Stone, 1936) says that: "*every finite Boolean algebra is isomorphic to the Boolean algebra of subsets of some finite set S*". Therefore, according to this result, reasoning in terms of *union*, *intersection*, etc. , on a finite set is isomorphic to performing logic operations ($+$ and $\cdot$) with logic functions. This result establishes the basis of the symbolic techniques used in this work since it allows the manipulation of sets of states using Boolean operations.

For further reference, a complete introduction to Boolean algebras can be found in [Bro90].

2.5.2 Encoding of LzTSs using Boolean algebras

Let $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR}\rangle$ be a LzTS. As previously discussed, the system $\langle 2^S, \cup, \cap, \emptyset, S\rangle$ is the Boolean algebra of sets of states of system A. Therefore, there is a one-to-one correspondence between the states of S and the vertexes of $\mathbb{B}^n$, given an appropriate value of n. Henceforth we will consider that S contains as many states as vertices are in $\mathbb{B}^n$. This implies that $S \supseteq \mathrm{Reach}(\mathsf{s}_0, T)$.

Each state $\mathsf{s} \in S$ can be represented by means of an *encoding function* $\mathcal{Q} : S \rightarrow \mathbb{B}^n$, such that $n \geq \lceil log_2(|S|)\rceil$. That is, given the set of Boolean variables $\mathcal{V} = \{v_1, \ldots, v_n\}$, each state $\mathsf{s} \in S$ is encoded into a vertex $(v_1, \ldots, v_n) \in \mathbb{B}^n$. Provided such encoding, any set of states $P \in S$ can be represented by a *characteristic (Boolean) function* $\mathcal{X}_P^{\mathcal{Q}} : \mathbb{B}^n \rightarrow \mathbb{B}$ that evaluates to 1 for those vertexes of $\mathbb{B}^n$ that correspond to states in the set P, encoded using $\mathcal{Q}$. Whenever the encoding is understood, we simply write $\mathcal{X}_P$. When implemented with BDDs (§2.6.1), characteristic functions provide, in general, compact and efficient representations.



(a) (b)

Figure 2.5. (a) A simple LzTS and (b) the same LzTS but binary-encoded.

Characteristic functions can also be used to represent *binary relations* between sets of states. Given two sets of states P_1 and P_2, to represent the binary relation $\mathcal{R} \subseteq P_1 \times P_2$ it is necessary to use two different sets of variables to identify the elements of each set. For example, variables $v_1, \ldots, v_n$ for P_1 and variables $v'_1, \ldots, v'_n$ for P_2. Provided the two sets of variables, the cartesian product of a relation between P_1 and P_2 can be simply expressed as the product of the respective characteristic functions. Since the binary relations we will represent are the transition relations of the transition system, we will call these two sets as the *current-state* set of variables and the *next-state* set of variables.

Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ and $\mathcal{V}' = \{v'_1, \ldots, v'_n\}$ be respectively, the set of current and next-state Boolean variables used to encode the states and transitions of the LzTS $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR}\rangle$. In such a way that v'_i is the next-state variable corresponding to the current-state variable v_i , and vice versa. The usual definition of LzTS can be extended to contain $\mathcal{V}$ and $\mathcal{V}'$

DEFINITION 2.10 (BOOLEAN ENCODED **LzTS**)

A Boolean encoded lazy transition system *is a six-tuple* $A = \langle \mathcal{V}, \mathcal{V}', \Sigma, \mathrm{TR}, \mathcal{X}_{\mathsf{s}_0}, \mathrm{FR}\rangle$ *if the LzTS is not classical and* $A = \langle \mathcal{V}, \mathcal{V}', \Sigma, \mathrm{TR}, \mathcal{X}_{\mathsf{s}_0}, \mathrm{ER}\rangle$ *if it is. The functions* EF, FF *and* TR *represent, respectively, its enabling region, its firing region and its transitions relation for each one of the events* $\mathsf{e} \in \Sigma$*, by means of the following characteristic functions:*

- $\mathrm{EF}(\mathsf{e}) : \mathbb{B}^n \rightarrow \mathbb{B}$ *such that* $\mathrm{EF}(\mathsf{e}) = 1$ *for all the states (encoded using* $\mathcal{V}$*) belonging to the enabling region of* e , i.e. $\mathrm{ER}(\mathsf{e})$.
- $\mathrm{FF}(\mathsf{e}) : \mathbb{B}^n \rightarrow \mathbb{B}$ *such that* $\mathrm{FF}(\mathsf{e}) = 1$ *for all the states (encoded using* $\mathcal{V}$*) belonging to the firing region of* e , i.e. $\mathrm{FR}(\mathsf{e})$.
- $\mathrm{TR}(\mathsf{e}) : \mathbb{B}^{2n} \rightarrow \mathbb{B}$ *such that* $\mathrm{TR}(\mathsf{e}) = 1$ *for all the relations* $(\mathsf{s}_1, \mathsf{s}_2)$ *such that there is a transition of event* e , $\mathsf{s}_1 \xrightarrow{\mathsf{e}} \mathsf{s}_2 \in T$*, that is for all elements in* T_{e}*. The part of the relation corresponding to state* s_1 *is encoded using the current-state variables in* $\mathcal{V}$*, whereas the part of the relation corresponding to state* s_2 *is encoded using the next-state variables in* $\mathcal{V}'$.

When characteristic functions of the enabling and firing regions are expressed using the set of next-state variables $\mathcal{V}'$*, we write* $\mathrm{EF}'(\mathsf{e})$ *and* $\mathrm{FF}'(\mathsf{e})$*, respectively. Also, when the sets of variables in a transition relation are interchanged we will write* $\mathrm{TR}(\mathsf{e})^{-1}$. ■ 2.10

EXAMPLE 2.4 *Figure 2.5 (a) shows a simple LzTS. The states of the system can be encoded using at least three (current-state) Boolean variables,* i.e. $\mathcal{V} = \{v_1, v_2, v_3\}$*. In (b) we show the same LzTS but encoded using an arbitrary binary encoding of the states.*

Given a set of states $P = \{\mathsf{s}_0, \mathsf{s}_1, \mathsf{s}_2\}$*, its characteristic function is:*

$$\mathcal{X}_P = \overline{v_0}\,\overline{v_1}\,\overline{v_2} \;+\; \overline{v_0}\,v_1\,\overline{v_2} \;+\; \overline{v_0}\,v_1\,v_2 = \overline{v_0}\,(v_1 \;+\; \overline{v_2}).$$

Similarly, $\mathrm{EF}(\mathsf{c}) = v_2\,\overline{v_3}$ *and* $\mathrm{FF}(\mathsf{c}) = v_1\,v_2\,\overline{v_3}$*. Note that states represented by encodings* $\overline{v_0}\,v_1\,v_2$*,* $\overline{v_0}\,\overline{v_1}\,v_2$ *and* $v_0\,\overline{v_1}\,v_2$ *are not reachable.*

To encode the transition relations, corresponding next-state variables, $\mathcal{V}' = \{v'_1, v'_2, v'_3\}$*, are also required. For instance we have that* $\mathrm{TR}(\mathsf{b}) = \overline{v_1}\,\overline{v_2}\,\overline{v_3}\,\overline{v'_1}\,v'_2\,\overline{v'_3} \;+\; v_1\,\overline{v_2}\,\overline{v_3}\,v'_1\,v'_2\,\overline{v'_3}$*.* ■ 2.4

Now we can translate the Image function (Definition 2.4) using this encoding. This function is defined for TS s but is straightforward to extend it to LzTS. The only additional thing to take into account is that it does not restrict to the fireable transitions. Hence, $\mathrm{Image}(P_1, T_{\mathsf{e}})$ returns the set of states $\{\mathsf{s}' \mid \mathsf{s} \in P_1 \wedge \mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}' \in T_{\mathsf{e}}\}$. If we name this set as P_2, we can compute its characteristic function as follows:

$$\mathcal{X}'_{P_2} \;=\; \exists_{v_1,\ldots,v_{|\mathcal{V}|}}\,(\;\mathrm{TR}(\mathsf{e}) \cdot \mathcal{X}_{P_1}\;)$$

To obtain the reachable states we must intersect with the firing regions of the events[1]. For instance, in the example, in order to compute the states reached after firing event b from state s_0 we just have: $\exists_{v_1,v_2,v_3}(\;\mathrm{TR}(\mathsf{b}) \cdot \mathrm{EF}(\mathsf{b}) \cdot \mathcal{X}_{\{\mathsf{s}_0\}}\;) = \exists_{v_1,v_2,v_3}(\,\overline{v_1}\,\overline{v_2}\,\overline{v_3}\,\square\overline{v'_1}\,v'_2\,\overline{v'_3}\,) = \overline{v'_1}\,v'_2\,\overline{v'_3}$. As expected, this corresponds to state s_1 but encoded using next-state variables. To convert expression encoded with such variables to expressions encoded with present-state variables we do use the substitution operator $\mathcal{V} \leftarrow \mathcal{V}'$. For instance $[\overline{v'_1}\,v'_2\,\overline{v'_3}]_{\mathcal{V}\leftarrow\mathcal{V}'} = \overline{v_1}\,v_2\,\overline{v_3}$. Formally, the substitution $\mathcal{V} \leftarrow \mathcal{V}'$ replaces each occurrence of any variable v' by its corresponding unprimed variable of $\mathcal{V}$.

2.6 Representing Boolean functions

Systems can be specified with a variety of models, including the ones presented in Chapter 2. However, when trying to compute something useful with these models, we face the problem of finding some good data representation for them. We must simply answer the question of how we store models in a computer so that operations on the model are fast and the space occupied is small.

In §2.5 we have seen how to map LzTSs into Boolean formulas. However, we still have the question of how this formulas are stored in a computer so that we can operate with them efficiently. Taking into account the most frequent operations in formal verification, ideally, the data structure used should provide an efficient implementation of the following operations:

> Conjunction, disjuction, existential quantification, tautology/contradiction test, and equality test between expressions.

Unfortunately it can be shown that no efficient data structure exists that can compactly represent *any* Boolean formula and, at the same time, provide fast operations. We show some of the possible representations of a given Boolean function in Figure 2.6.

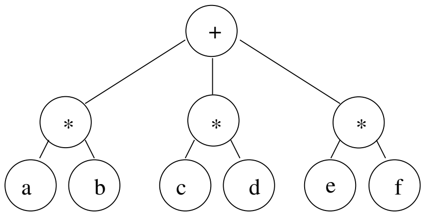
We want to have the advantages of canonicity (in order to have fast equality operation) so we should look for a normal form. Disjunctive or Conjunctive Normal Forms (DNF and CNF respectively) are a possibility, but either the tautology checking or the contradiction checking is an NP-complete problem. The If-then-else Normal Form (INF) has not this problem. Figure 2.7 (a) shows the INF of a function. The most successful implementation of the INF are Reduced Ordered Binary Decision Diagrams, which basically are the same diagrams produced by the INF but with the common subtrees merged.

[1] Classical LzTS do not have this problem since the information of the firing regions is already incorporated in the transition relation.

POLINOMIC FORM

f(a, b, c, d, e, f) = a·b + c·d + e·f

TREE OF OPERANDS



TRUTH TABLE

abc \ def	000	001	010	011	100	101	110	111
000	0	0	0	1	0	0	0	1
001	0	0	0	1	1	1	1	1
010	0	0	0	1	0	0	0	1
011	0	0	0	1	1	1	1	1
100	0	0	0	1	0	0	0	1
101	0	0	0	1	1	1	1	1
110	1	1	1	1	1	1	1	1
111	1	1	1	1	1	1	1	1

DISJUNCTIVE NORMAL FORM

f(a, b, c, d, e, f) = abcdef + abcdef' + abcde'f + abcde'f' + abcd'ef + abcd'ef' + abcd'e'f + abcd'e'f' +
+ abc'def + abc'def' + abc'de'f + abc'de'f' + abc'd'ef + abc'd'ef' + abc'd'e'f + abc'd'e'f'
+ ab'cd'ef + ab'cde'f' + ab'cde'f + ab'cdef' + ab'cdef + ab'c'd'ef + ab'c'def + a'bcd'ef +
+ a'bcde'f' + a'bcde'f + a'bcdef' + a'bcdef + a'bc'd'ef + a'bc'def + a'b'cd'ef + a'b'cde'f'
+ a'b'cde'f + a'b'cdef' + a'b'cdef + a'b'c'd'ef + a'b'c'def.

Figure 2.6. Several representations of the same Boolean function.

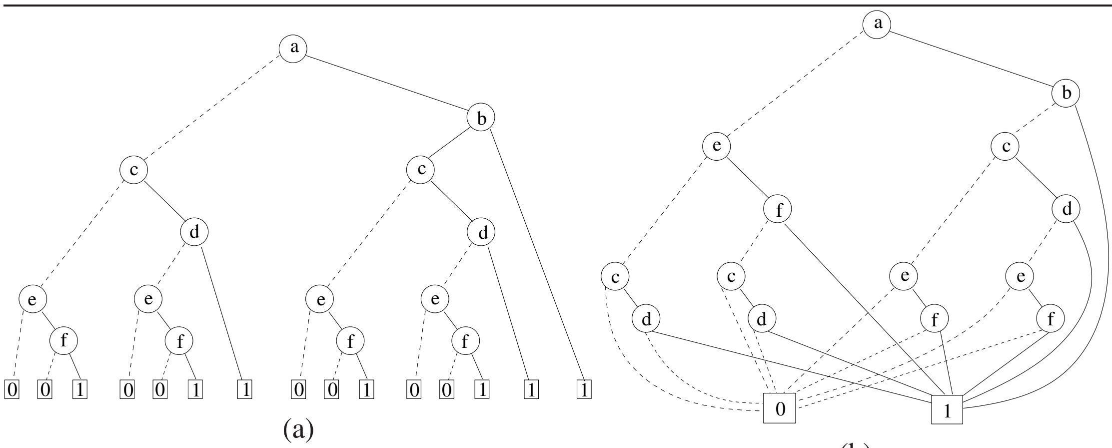


Figure 2.7. INF representation vs. BDD: (a) If-then-else Normal form of function shown in Figure 2.6. (b) One of the possible BDDs that represent the same function. Note that the order in which variables are checked is not the same for all paths.

2.6.1 Reduced Ordered Binary Decision Diagrams (ROBDDs)

Introduced in [Lee59], further developed in [S.B78] and finally popularized by [Bry86, Bry92], this data structure has come one of the milestones of formal verification in the last decades.

DEFINITION 2.11 (BINARY DECISION DIAGRAM (BDD))
A Binary Decision Diagram *is a rooted directed acyclic graph with:*

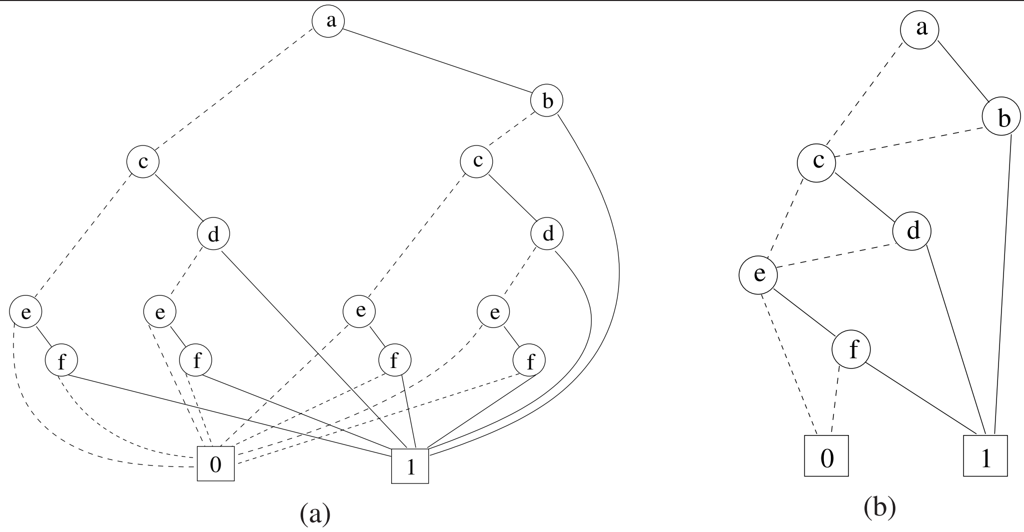


Figure 2.8. OBDD and ROBDD examples: (a) OBDD equivalent to the BDD of Figure 2.7 (b). For all paths variables are checked in lexicographical order. (b) Its equivalent ROBDD.

- *one or two terminal nodes with out-degree zero labeled 0 or 1, and*
- *a set of variable nodes* u *of out-degree two. The two outgoing edges are given by functions* $\mathrm{low}(u)$ *and* $\mathrm{high}(u)$. *A variable* $\mathrm{var}(u)$ *is associated to each variable node.*

■ 2.11

As one can see this definition is quite broad. This has a big impact on the complexity of the algorithms needed to perform logical operations between BDDs and also does not guarantee the canonicity of the representation. Figure 2.7 (b) shows one of the possible BDDs for the function of Figure 2.6.

To increase the simplicity of the algorithms and enforce canonicity is much easier to define a common order of variable testing in all paths of the BDD. Although this has the side effect of making some representations exponential, the benefits usually make up for this problem.

Definition 2.12 (Ordered Binary Decision Diagram (OBDD))
An Ordered Binary Decision Diagram *is a BDD in which all paths through the graph satisfy a given linear order* $x_1 < x_2 < \ldots < x_n$.

■ 2.12

An example of OBDD can be seen in Figure 2.8 (a).

Finally one can compact more the representation by applying some simplifications in the resulting graph. This has the only impact of reducing the size of the representation by making the manipulation algorithms slightly more complex, but the properties of the representation remain the same (canonicity and constant checking of tautology and contradiction).

Definition 2.13 (Reduced OBDD (ROBDD))
A Reduced Ordered Binary Decision Diagram *is an OBDD that satisfies the following conditions:*

Uniqueness *Distinct nodes* u *and* v *cannot have the same variable name and low and high successors,* i.e.

$$(\mathrm{var}(u) = \mathrm{var}(v)) \wedge (\mathrm{low}(u) = \mathrm{low}(v)) \wedge \mathrm{high}(u) = \mathrm{high}(v) \Rightarrow u = v$$

Non-redundant tests *No variable node u has identical low and high successors,* i.e.

$$\text{low}(u) \neq \text{high}(u)$$

■ 2.13

Figure 2.8 (b) shows an example of ROBDD. Although ROBDDs are just a subtype of BDDs, their widespread use made them more known as just simply BDDs. From now on we will also use the term BDD to refer to a ROBDD.

2.7 Partial specification of transition relations

Usually concurrent systems are made up of events that affect only a few of the variables used to encode the states. Thus almost all the specification languages that are able to describe concurrent systems allow only to describe the transition relations of the events in terms of the variables that change, avoiding the overhead of specifying that the rest of the variables remain the same.

EXAMPLE 2.5 *For instance, imagine you want to express the transition relation of NOT gate in a big asynchronous system that has a hundred state variables ($V = \{v_1, v_2, \ldots, v_{100}\}$). If the NOT gate defines the state of variable v_1 it is much easier to say* $\mathrm{TR}(\mathrm{NOT}) = v_1\overline{v_1'} + v_1'\overline{v_1}$ *than to write* $\mathrm{TR}(\mathrm{NOT}) = (v_1\overline{v_1'} + v_1'\overline{v_1})(v_2 = v_2')(v_3 = v_3') \cdots (v_{100} = v_{100}')$. ■ 2.5

First tools accepted this much shorter description and then converted it internally to the complete expression. This caused a non-trivial increase in the size of the TRs. However, it was soon recognized [BCL+94] that computations could be done using the shorter version of the TR.

In [BCL+94] they focused on circuit verification, thus their proposal (for which no implementation details were given) considered that only one next state variable was present in each TR. However this is not suitable for protocol (or automata in general) representation, so as early as in [RCP95] a more general version of the same concept was developed.

Recently some other researchers independently came to the same solution [BR03] and named it *partial disjunctive transition relation*. For simplicity in the rest of the text we use the term *partial transition relation* (pTR) to refer to such objects, or simply *transition relation* when the context is clear.

Before we can give a formal definition of pTRs, we need some auxiliary functions:

DEFINITION 2.14 (SUPPORT OF A FORMULA)
*The set of variables which constitute a formula (*i.e. *a Boolean expression) φ is called* support *of φ, written as* $\mathrm{Sup}(\varphi)$. ■ 2.14

Now we can define what is a partial transition relation.

DEFINITION 2.15 (PARTIAL DISJUNCTIVE TRANSITION RELATION)
Given a Boolean encoded LzTS $A = \langle \mathcal{V}, \mathcal{V}', \Sigma, \mathrm{TR}, \mathcal{X}_{s_0}, \mathrm{FR} \rangle$, a partial disjunctive Transition Relation *(pTR) is just a form to express transition relations. Let E be a Boolean expression of a transition relation using the sets of variables $\mathcal{V}$ and $\mathcal{V}'$. Its partial Transition Relation form is a tuple $\langle E_p, \xi \rangle$, where E_p is also a Boolean expression on variables of $\mathcal{V}$ and $\mathcal{V}'$ and ξ is a set of variables called the* set of changing variables*, such that $\xi \subseteq \mathcal{V}$ and* $\mathrm{Sup}(E_p) \cap \mathcal{V}' \subseteq \xi' \subseteq \mathcal{V}'$*, so that*

$$E = E_p \cdot \prod_{\forall v' \in \mathcal{V}' \setminus \xi'} (v' = v)$$

■ 2.15

We can enforce this form of expressing relations to be a normal form by requiring that E' is in a normal form (*e.g.* ROBDD, CNF, ...) and the set of changing variables must be maximal. When this happens, the representation is minimal *i.e.* there is no other partial transition relation with a smaller E_p representing the same function. We define an auxiliary function $\mathrm{total}(\langle E_p, \xi\rangle)$ that given a partial transition relation returns the transition relation in “full” form *i.e.*

$$\mathrm{total}(\langle E_p, \xi\rangle) = E_p \cdot \prod_{\forall v' \in \mathcal{V}' \setminus \xi'} (v' = v)$$

EXAMPLE 2.5 (CONT.) *For instance, following the previous example,*

$$\mathrm{total}(\langle v_1\overline{v_1'} + v_1'\overline{v_1}, \{v_1\}\rangle) = (v_1\overline{v_1'} + v_1'\overline{v_1})(v_2 = v_2')(v_3 = v_3')\cdots(v_{100} = v_{100}'). \qquad \blacksquare \; 2.5$$

Note that the set of changing variables is defined as $\mathrm{Sup}(E_p) \cap \mathcal{V}' \subseteq \xi' \subseteq \mathcal{V}'$, thus more variables than the ones appearing in E_p can belong to the set. The reason for this is to allow indeterministic transition relations. For instance, if we want to express that if variable v_0 is 1, then we want to be v_1 either 0 or 1, we should write something like $v_0(v_1' + \overline{v_1'})$. However, this function is equal to v_0 in many normal forms. How we can determine which variables change if we only have the expression v_0? We need to explicitely define the set of changing variables.

There is a subtlety in Definition 2.15: in case of an expression like $v_0v_1'\overline{v_1'} + v_0'$, the definition enforces to be the set of changing variables $\xi \supseteq \{v_0, v_1\}$. Although the first minterm is clearly false, as variable v_1' appears in the expression, is assumed that the expression sets its value in all of its minterms.

As in all this work we use pTRs, henceforth when we use the term TR in fact we refer to a pTR, unless otherwise stated. From the theoretical point of view, the difference is irrelevant, but in terms of practical application is important.

2.7.1 Operations with partial TRs

The space savings can be considerable by using partial transition relations, however, we must ensure that all needed operations can still be done. In particular, transition relations must be able to perform images and preimages as well as to be composable between them *e.g.* compute the transition relation of the union/intersection of transition relations.

Image computation

Partial TRs are particularly well suited for this operation. Given a pTR $\langle E, \xi\rangle$, as variables not in ξ do not change their value the $\mathrm{Image}(\mathrm{Set}, E)$ function can be computed as

$$\Big[\exists_\xi\big(E \cdot \mathcal{X}_{\mathrm{Set}}\big)\Big]_{V \leftarrow V'}$$

Preimage computation

The preimage function is the inverse of the Image function. Given some states and a set of transitions, its preimage returns the states from which there is a transition that reaches any of the states specified. It is frequently used in model checking algorithms, specially in CTL formula verification, thus it is important to define it.

To compute the preimage of a set of states Set we will need to change current-state by next-state variables of Set, but only for the changing variables.The preimage is then computed as

$$\Big[\exists_{\xi'}\big(E \cdot [\mathcal{X}_{\mathrm{Set}}]_{\xi' \leftarrow \xi}\big)\Big]_{V \leftarrow V'}$$

Normalization

Like with rational numbers that need a common denominator to be able to add or subtract them, also pTRs need a common set of changing variables to be able to perform operations like union or intersection between them.

It is very easy to show that trying to do otherwise creates problems. For instance, imagine we want to do the union of two pTRs $\langle \overline{v_0}v_0', \{v_0\}\rangle$ and $\langle \overline{v_1}v_1', \{v_1\}\rangle$ The result should be something like: v_0 can become 1 if it was 0 (and then v_1 preserves its value) or v_1 can become 1 if it was 1 (and then v_0 preserves its value). It is clear than in the resulting set of changing variables all the changing variables of the two pTRs must be present, then doing the direct union of both results in pTR $\langle \overline{v_0}v_0' + \overline{v_1}v_1', \{v_0, v_1\}\rangle$. But this has different semantics than the intended as now v_0 can become 1 if it was 0 but then v_1 gets an indeterminate value, and vice versa.

We call normalization of two pTRs the process by which the set of changing variables are made identical. This usually implies to convert the pTRs into a non minimal form before operating, equivalent to putting a rational number in a non simplified form to be able to add it to another.

Definition 2.16 (Normalized pTRs)

Two pTRs are said to be normalized iff their set of changing variables is equal. ■ 2.16

Two pTRs can be enforced to be normal by intersecting them with the identity functions of the variables that are different in both pTRs. Formally, given two pTRs $p_1 = \langle E_1, \xi_1\rangle$ and $p_2 = \langle E_2, \xi_2\rangle$, then $p_1^n = \langle E_1 \cdot \prod_{\forall v \in \xi_2 \setminus \xi_1}(v' = v), \xi_1 \cup \xi_2\rangle$ and $p_2^n = \langle E_2 \cdot \prod_{\forall v \in \xi_1 \setminus \xi_2}(v' = v), \xi_1 \cup \xi_2\rangle$ are normalized between them.

Note that the previous procedure gives the minimal normalized pTRs if both pTRs were already minimal (equivalent to using the minimum common multiple as denominator for rational operations), but that the definition allows for larger representations also.

Union of pTRs

Given two normalized pTRs $p_1 = \langle E_1, \xi_1\rangle$ and $p_2 = \langle E_2, \xi_2\rangle$, the union pTR is defined as $p_{1+2} = \langle E_1 + E_2, \xi_1 \cup \xi_2\rangle$.

Lemma 2.1

If p_1 and p_2 were in minimal normalized form, then p_{1+2} is in minimal form. ■ 2.1

Proof: *A pTR is in minimal from if and only if the set of changing variables is minimal. As union preserves all arcs from both transitions all the variables that previously changed must still change. Thus $\xi_1 \cup \xi_2$ is the minimal set of changing variables.*

Intersection of pTRs

This operation has not the same nice properties as the union of pTRs. Given two normalized pTRs $p_1 = \langle E_1, \xi_1\rangle$ and $p_2 = \langle E_2, \xi_2\rangle$, the intersection pTR is defined as $p_{1\cdot 2} = \langle E_1 \cdot E_2, \xi_1 \cup \xi_2\rangle$.

As in intersection only common arcs are preserved, it is possible that some of the variables that previously changed, do not change any more, thus $\xi_1 \cup \xi_2$ is not guaranteed to be the minimal set of changing variables, thus it is neither guaranteed that $p_{1\cdot 2}$ is in minimal form.

Composition of pTRs

The composition of two pTRs, p_1 and p_2, is a new pTR, $p_{2\circ 1}$, such that $\mathrm{Image}(\mathrm{Set}, p_{2\circ 1}) = \mathrm{Image}(\mathrm{Image}(\mathrm{Set}, p_1), p_2)$. Given two normalized pTRs $p_1 = \langle E_1, \xi_1\rangle$ and $p_2 = \langle E_2, \xi_2\rangle$, we define the set of present state variables of E_2 that are not set by E_1 as $\mathrm{Nonset} = (\mathrm{Sup}(E_2) \cap V) \setminus \xi_1$, and V'' to be an auxiliary set of variables related to the other two sets V and V' by bijective functions such that we can get the corresponding variable v''_i of V'' from v_i in V or from v'_i in V'. Then the composed pTR is defined as

$$p_{2\circ 1} = \langle \exists_{V''} \Big([E_1]_{V'' \leftarrow V'} \cdot \prod_{\forall v \in \text{Nonset}} (v = v'') \cdot [E_2]_{V'' \leftarrow V} \Big), \xi_1 \cup \xi_2 \rangle.$$

Composition does not guarantee give a minimal pTR for two reasons: first is the intersection that is performed which can remove arcs and, second, the functions composed may be such that some of the previously changing variables do not change any more. The most extreme case of such phenomenon is where a function is composed with its inverse function. In that case our computations give a set of changing variables equal to $\xi_1 \cup \xi_2$ but it is trivial to see that in fact no variable changes at all!

Support of pTRs

In Definition 2.14 we introduced the support of a Boolean expression. Intuitively, this is the set of variables to which a Boolean expression is related, that is, the variables it *reads* or *writes*. This set of variables is important when computing causality between events, so we would like to compute the support of the pTR of an event.

DEFINITION 2.17 (SUPPORT OF A PARTIAL TRANSITION RELATION)
Given a pTR $p = \langle E, \xi \rangle$ *we define its support as* $\text{Sup}(p) = \text{Sup}(E) \cup \xi'$. ■ 2.17

Note that if the set ξ' is excluded from the support, is possible that some *written* variables are left out from it. The problem appears with events with indeterministic transitions. For instance the event $p = \langle v_0, \{v_1\} \rangle$ sets the variable v_1 to both values 0 and 1 when v_0 is 0. The expression of p could have been written $v_0(v'_1 + \overline{v'_1})$ but remember that ROBDD reduce redundant nodes so in practice the expression when represented with a BDD becomes v_0.

Simplification of pTRs

Given a pTR $p = \langle E, \xi \rangle$ that is not in its minimal form, we call *simplification* the process by which we obtain a pTR $p_s = \langle E_s, \xi_s \rangle$ that is minimal and $\text{total}(p) = \text{total}(p_s)$. Formally, p_s is minimal when the number of minterms (over $\mathcal{V} \cup \mathcal{V}'$) of E_p is maximal and $|\xi_s|$ is minimal.

The minimization process consists in transferring, as much as possible, the information contained in E to the implicit conjunctions of the set of non-changing variables. This is achieved by detecting variables whose behavior can be represented by these implicit conjunctions. Using Boole's expansion theorem for a given variable v_i we can write:

$$E = \overline{v_i} \cdot \overline{v'_i} \cdot E_{\overline{v_i}, \overline{v'_i}} + \overline{v_i} \cdot v'_i \cdot E_{\overline{v_i}, v'_i} + v_i \cdot \overline{v'_i} \cdot E_{v_i, \overline{v'_i}} + v_i \cdot v'_i \cdot E_{v_i, v'_i}$$

now $\text{total}(p)$ can be rewritten as:

$$\text{total}(p) = \left(\overline{v_i} \cdot \overline{v'_i} \cdot E_{\overline{v_i}, \overline{v'_i}} + \overline{v_i} \cdot v'_i \cdot E_{\overline{v_i}, v'_i} + v_i \cdot \overline{v'_i} \cdot E_{v_i, \overline{v'_i}} + v_i \cdot v'_i \cdot E_{v_i, v'_i} \right) \cdot \prod_{\forall v \notin \xi} (v = v')$$

Clearly, if the information of v_i has to be transferred to $\prod_{\forall v \notin \xi} (v = v')$, then the only possibility is that $\overline{v_i} \cdot v'_i \cdot E_{\overline{v_i}, v'_i} + v_i \cdot \overline{v'_i} \cdot E_{v_i, \overline{v'_i}} = 0$. In such case the expression reduces to

$$\text{total}(p) = \left(\overline{v_i} \cdot \overline{v'_i} \cdot E_{\overline{v_i}, \overline{v'_i}} + v_i \cdot v'_i \cdot E_{v_i, v'_i} \right) \cdot \prod_{\forall v \notin \xi} (v = v').$$

Now $\overline{v_i} \cdot \overline{v_i'} \cdot E_{\overline{v_i},\overline{v_i'}} + v_i \cdot v_i' \cdot E_{v_i,v_i'}$ which can be rewritten as $\left(\overline{v_i} \cdot E_{\overline{v_i},\overline{v_i'}} + v_i \cdot E_{v_i,v_i'}\right) \cdot (v_i = v_i')$ or $\left(\overline{v_i'} \cdot E_{\overline{v_i},\overline{v_i'}} + v_i' \cdot E_{v_i,v_i'}\right) \cdot (v_i = v_i')$. However, in the former expression $v_i \notin \mathrm{Sup}(\overline{v_i} \cdot E_{\overline{v_i},\overline{v_i'}} + v_i \cdot E_{v_i,v_i'})$, hence the variable can be removed from ξ, while in the latter expression is not possible.

Following this guidelines it is possible to implement a simple algorithm to minimize a pTR (Algorithm 2.1).

Algorithm 2.1: Simplification of a pTR

---

input : $p = \langle E, \xi \rangle$ a pTR.
input : $p_s = \langle E_s, \xi_s \rangle$ a minimal pTR such that $\mathrm{total}(p) = \mathrm{total}(p_s)$.

$V \leftarrow \emptyset$
forall $v_i \in \xi$ **do**
  if $E \cdot (v_i \oplus v_i') = 0$ **then** $V \leftarrow V \cup \{v_i\}$
$E_s \leftarrow \exists_{\forall v_i' \in V'} E$
$\xi_s \leftarrow \xi \setminus V$
return $\langle E_s, \xi_s \rangle$

EXAMPLE 2.6 *Given a pTR* $p = \langle(\overline{v_1}\,\overline{v_1'} + v_1 v_1')v_2 v_2' v_3 \overline{v_3'}, \{v_1, v_2, v_3\}\rangle$*, in the first iteration of the algorithm we obtain that* $(\overline{v_1}\,\overline{v_1'} + v_1 v_1')v_2 v_2' v_3 \overline{v_3'} \cdot (v_1 \oplus v_1') = 0$*, hence* v_1 *is added to* V*. In second iteration, again,* $(\overline{v_1}\,\overline{v_1'} + v_1 v_1')v_2 v_2' v_3 \overline{v_3'} \cdot (v_2 \oplus v_2') = 0$ *and* v_2 *is added to* V*. Finally, in the last iteration* $(\overline{v_1}\,\overline{v_1'} + v_1 v_1')v_2 v_2' v_3 \overline{v_3'} \cdot (v_3 \oplus v_3') = (\overline{v_1}\,\overline{v_1'} + v_1 v_1')v_2 v_2' v_3 \overline{v_3'} \neq 0$*. Thus* $V = \{v_1, v_2\}$.

The simplified pTR is then obtained by abstracting the variables $\{v_1', v_2'\}$ *from the expression* $(\overline{v_1}\,\overline{v_1'} + v_1 v_1')v_2 v_2' v_3 \overline{v_3'}$ *which results in* $(\overline{v_1} + v_1)v_2 v_3 \overline{v_3'} = v_2 v_3 \overline{v_3'}$*. The set of changing variables is also updated accordingly and the minimized pTR of p is* $\langle v_2 v_3 \overline{v_3'}, \{v_3\}\rangle$. ■ 2.6

Note that, altough no abstraction of present-state variables is performed by the algorithm, they can be naturally elliminated in the case that the factor $(v_i = v_i')$ already forms part of the expression, as shown in the example. If ROBDDs are used to represent the Boolean expression this process is automatic by the reduction rules of the ROBDDs.

Finally, the algorithm can be used to convert a "normal" TR t (*i.e.* not partial) into a pTR, by simply calling Algorithm 2.1 on $\langle t, \mathcal{V}\rangle$.

Chapter 3

RELATIONS, ORDER AND CAUSALITY ANALYSIS

Summary

In this chapter we review the mathematical notions of relation and order, together with several types of important relations between the events of a LzTS. These relations, namely *triggering*, *disabling*, *concurrency* and *independence*, are fundamental in concurrent systems.

LzTSs define two kinds of behavior through the set of lazy traces and the set of valid traces. For this reason we also define the relations between events at two different levels, either taking into account all possible behaviors or just the valid traces.

Finally, we pay special attention to how these relations can be determined in a symbolic setting (*e.g.* to determine if two events of a LzTS are independent, concurrent, they mutually disable, etc.). This is important because we use symbolically encoded LzTSs and these relations have to be computed in several stages of our verification methodology.

3.1 Relations

In this work we will only use binary relations, hence the definition of *relation* we use reflects this restriction.

DEFINITION 3.1 (RELATION)
Given two sets A_1 and A_2 a relation R over the sets A_1 and A_2 is a subset of their cartesian product. Formally, $R \subseteq A_1 \times A_2$. ■ 3.1

Relations can have several types of properties.

DEFINITION 3.2 (PROPERTIES OF A RELATION)
A relation $R \subseteq A \times A$ on a set A is said to be:

- reflexive *if* $\forall a \in A,\ (a, a) \in R$.
- irreflexive *if* $\forall a \in A,\ (a, a) \notin R$.
- transitive *if* $\forall a, b, c \in A,\ \{(a, b), (b, c)\} \subseteq R \Rightarrow (a, c) \in R$.
- acyclic *if* R^* *is irreflexive.*
- symmetric *if* $\forall a, b \in A,\ (a, b) \in R \Rightarrow (b, a) \in R$.
- assymmetric *if* $\forall a, b \in A,\ (a, b) \in R \Rightarrow (b, a) \notin R$.
- antisymmetric *if* $\forall a, b \in A,\ \{(a, b), (b, a)\} \subseteq R \Rightarrow a = b$.
- total *or* connected *if* $\forall a, b \in A,\ (a, b) \in R \vee (b, a) \in R$.

■ 3.2

Relations can be classified according to which set of the previous properties satisfy. In particular we are interested in relations that express the notion of order in a set.

3.2 Orders

All throughout this work order relations will be used. As there is no consensus on how they are described in the literature we give here a brief explanation of the definitions we use.

DEFINITION 3.3 (ORDERS)
A relation $R \subseteq A \times A$ on a set A that:

- *is acyclic, irreflexive and asymmetric is called a* pseudo-order.
- *is reflexive, antisymmetric and transitive is called a* partial order.
- *is irreflexive, asymmetric and transitive is called a* strict order.

A strict order does not necessarily relate (order) all the elements in the set, for this reason it is also called a strict partial order. *If it relates all the elements, then it is called a* strict total order. ■ 3.3

From a pseudo-order R we can obtain a strict order by doing the transitive closure of R, noted as R^*. We consider two pseudo-orders R and R' equivalent iff $R^* = R'^*$.

EXAMPLE 3.1 *Let R be a pseudo-order over the set $\{a, b, c, d\}$, such that $R = \{(a, b), (b, c)\}$, then $R^* = \{(a, b), (b, c), (a, c)\}$.*

DEFINITION 3.4 (ORDERED SET)
A set A equipped with an order relation R is called an ordered set. *Formally, the tuple $\langle A, R\rangle$ is an ordered set.* ■ 3.4

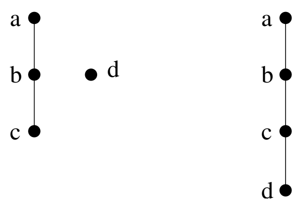


Figure 3.1. (left) Hasse diagram of set $\{a, b, c, d\}$ ordered by relation $R' = \{(a, b), (b, c), (a, c)\}$. (right) Hasse diagram of one linearization of R'

Ordered sets are sometimes called *partial ordered sets* or just *posets* in the literature. One of the most useful and attractive features of ordered sets is that, in the finite case, they can be represented with diagrams. But first we need to introduce the concept of covering.

DEFINITION 3.5 (COVERING RELATION)
Let $P = \langle A, < \rangle$ be an ordered set and let $x, y \in P$. We say that x covers *y if $x > y$ and there is no element $z \in P$ such that $x > z > y$. We will denote this by $x >_\mu y$.* ■ 3.5

The covering relation $<_\mu$ is a pseudo-order relation. In fact is the smallest element in its equivalency class *i.e.* there is no other pseudo-order that specifies the same strict order with less elements. Given a partial order R we will denote its covering relation as R_μ. Then we can draw a *diagram* (or *Hasse diagram*) of ordered set P by using the following definition.

DEFINITION 3.6 (HASSE DIAGRAM)
Let $P = \langle A, < \rangle$ be an ordered set. A Hasse diagram *of P is a draw created by the following algorithm:*

1. *For each $x \in P$, draw a point $p(x)$.*
2. *For each covering pair such that x covers y, connect $p(x)$ and $p(y)$ by a line $l(x, y)$.*
3. *Carry out previous steps in a way such that*
   (i) if x covers y then $p(x)$ is higher than $p(y)$.
   (ii) no point $p(z)$ intersects line $l(x, y)$ if $z \neq x$ and $z \neq y$.

■ 3.6

EXAMPLE 3.1 (CONT.) *In the left of Figure 3.1 we can see the Hasse diagram of set $\{a, b, c, d\}$ ordered by relation $R' = \{(a, b), (b, c), (a, c)\}$. The covering relation of R' is $\{(a, b), (b, c)\}$*

3.3 Relations between the events of a LzTS

State based models like TSs are very flexible in the sense that many other formalisms can be mapped on top of them. However the advantages of such a low-level formalism have also its counterpart. One disadvantage is that the causality information is lost, as now gets "hidden" in the transition relation. In some cases this information could be used to analyze more efficiently concurrent systems [SP04a, SP04b] or to describe in a high level formalism a fragment of the system (Chapter 11).

In this section we explain how we can retrieve causality information from the transition relation definition without having to generate all the state space. To do so we first define some important relations between the events in a LzTS in §3.3.1 and then we show how they can be computed in a symbolic setting (§3.3.2). *In the rest of this chapter we assume that the transition relations of a* TS *are represented using partial transition relations.* As LzTSs generalize TSs, we use a LzTS as model in our definitions.

3.3.1 Fundamental relations

Depending on the effect that the firing of an event might have on some of the other events, several types of relations can be defined.

DEFINITION 3.7 (LOGICAL RELATIONS)
Given a lazy transition system $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR}\rangle$*, two events* $\mathsf{e}_1, \mathsf{e}_2 \in \Sigma$ *and a set of states* $\mathsf{S} \subseteq S$*:*

- e_1 logically triggers e_2 in S *if the firing of* e_1 *enables* e_2 *in some of the states of* S. *Formally,*

$$\exists \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \in T \wedge \mathsf{s}_1, \mathsf{s}_2 \in \mathsf{S} \; : \; (\mathsf{s}_1 \notin \mathrm{ER}(\mathsf{e}_2) \vee \mathsf{e}_1 = \mathsf{e}_2) \wedge \mathsf{s}_2 \in \mathrm{ER}(\mathsf{e}_2)$$

- e_1 logically disables e_2 in S *if the firing* e_1 *renders* e_2 *not enabled in some of the states of* S. *Formally,*

$$\exists \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \in T \wedge \mathsf{s}_1, \mathsf{s}_2 \in \mathsf{S} \; : \; \mathsf{s}_1 \in \mathrm{ER}(\mathsf{e}_2) \wedge \mathsf{s}_2 \notin \mathrm{ER}(\mathsf{e}_2)$$

- e_1 and e_2 are logically concurrent in S *if they are both enabled in some state of* S *and exists another state in* S *that can be reached by firing both events no matter in which order. Formally,*

$$\exists \mathsf{s} \in \mathrm{ER}(\mathsf{e}_1) \cap \mathrm{ER}(\mathsf{e}_2) \cap \mathsf{S} \;\wedge\; \exists \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3 \in \mathsf{S} \; : \\ \mathsf{s} \xrightarrow{\mathsf{e}_1} \mathsf{s}_1, \mathsf{s} \xrightarrow{\mathsf{e}_2} \mathsf{s}_2 \in T \;\Rightarrow\; \mathsf{s}_1 \xrightarrow{\mathsf{e}_2} \mathsf{s}_3, \mathsf{s}_2 \xrightarrow{\mathsf{e}_1} \mathsf{s}_3 \in T$$

- e_1 and e_2 are logically independent in S *if the firing of one of the events does not change the enabledness of the other and, for the states where they are concurrently enabled, no matter in which order they are fired, the same state is reached. Formally,*

$$\begin{aligned} &\forall\, \mathsf{s} \in \mathsf{S} : \mathsf{s} \xrightarrow{\mathsf{e}_1} \mathsf{s}_1 \in T, \quad \mathsf{s} \in \mathrm{ER}(\mathsf{e}_2) \Leftrightarrow \mathsf{s}_1 \in \mathrm{ER}(\mathsf{e}_2) \quad \wedge \\ &\forall\, \mathsf{s} \in \mathsf{S} : \mathsf{s} \xrightarrow{\mathsf{e}_2} \mathsf{s}_2 \in T, \quad \mathsf{s} \in \mathrm{ER}(\mathsf{e}_1) \Leftrightarrow \mathsf{s}_2 \in \mathrm{ER}(\mathsf{e}_1) \quad \wedge \\ &\forall\, \mathsf{s} \in \mathrm{ER}(\mathsf{e}_1) \cap \mathrm{ER}(\mathsf{e}_2) \cap \mathsf{S}, \\ &\quad (\forall \mathsf{s} \xrightarrow{\mathsf{e}_1} \mathsf{s}_1, \mathsf{s}_1 \xrightarrow{\mathsf{e}_2} \mathsf{s}_3 \in T \;\wedge\; \{\mathsf{s}_1, \mathsf{s}_3\} \subseteq \mathsf{S}, \exists\, \mathsf{s} \xrightarrow{\mathsf{e}_2} \mathsf{s}_2, \mathsf{s}_2 \xrightarrow{\mathsf{e}_1} \mathsf{s}_3 \in T \;\wedge\; \mathsf{s}_2 \in \mathsf{S}) \;\wedge \\ &\quad (\forall \mathsf{s} \xrightarrow{\mathsf{e}_2} \mathsf{s}_2, \mathsf{s}_2 \xrightarrow{\mathsf{e}_1} \mathsf{s}_3 \in T \wedge \{\mathsf{s}_2, \mathsf{s}_3\} \subseteq \mathsf{S},\ \exists\, \mathsf{s} \xrightarrow{\mathsf{e}_1} \mathsf{s}_1, \mathsf{s}_1 \xrightarrow{\mathsf{e}_2} \mathsf{s}_3 \in T \;\wedge\; \mathsf{s}_1 \in \mathsf{S}) \end{aligned}$$

■ 3.7

We have used the term *logical* in each one of the relations defined because we can think of LzTSs as describing two different levels of behavior. The first level only considers the transition relation definition and the firing region information is completely disregarded. We term this level as the *logical* level. For instance this is very clear when the firing region represents timing information: conceptually is not the same that two events are concurrent but one is forced to happen before the other, than if one of them would disable the other but this disabling is never observed due to the timing (Figure 3.2). The observable behavior of the system is the same in both situations, but clearly at the logical level the relation between both events must be different.

In contrast, in the other level, which we term as the *observable* level, the firing regions of the events are considered. This level is usually the most important because most of the properties

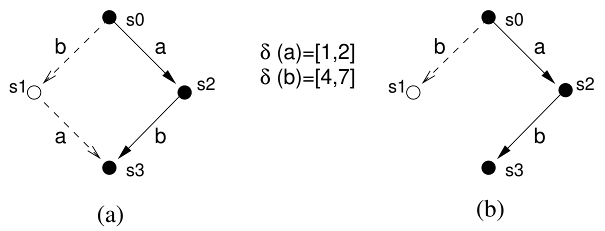


Figure 3.2. Two LzTSs where the observable behavior is the same, but logical behavior is different. In (a) events a and b are logically independent, while in (b) event b logically disables a.

checked on LzTSs depend on the observed behavior. Hence relations of Definition 3.7 have their *observable* counterparts in Definition 3.8.

DEFINITION 3.8 (OBSERVABLE RELATIONS)
Given a lazy transition system $A = \langle S, \Sigma, T, s_0, \mathrm{FR}\rangle$, *two events* $e_1, e_2 \in \Sigma$ *and a set of states* $S \subseteq S$*:*

- e_1 observably triggers e_2 in S *if*

$$\exists s_1 \xrightarrow{e_1} s_2 \in T \wedge s_1, s_2 \in S \ : \ s_1 \in \mathrm{FR}(e_1) \wedge (s_1 \notin \mathrm{ER}(e_2) \vee e_1 = e_2) \wedge s_2 \in \mathrm{ER}(e_2)$$

- e_1 observably disables e_2 in S *if*

$$\exists s_1 \xrightarrow{e_1} s_2 \in T \wedge s_1, s_2 \in S \ : \ s_1 \in \mathrm{FR}(e_1) \wedge s_1 \in \mathrm{ER}(e_2) \wedge s_2 \notin \mathrm{ER}(e_2)$$

- e_1 and e_2 are observably concurrent in S *if*

$$\exists s \in \mathrm{FR}(e_1) \cap \mathrm{FR}(e_2) \cap S \ \wedge \ \exists s_1 \in \mathrm{FR}(e_2) \cap S \ \wedge \ \exists s_2 \in \mathrm{FR}(e_1) \cap S \ \wedge \ \exists s_3 \in S \ : \\ s \xrightarrow{e_1} s_1, s \xrightarrow{e_2} s_2 \in T \ \Rightarrow \ s_1 \xrightarrow{e_2} s_3, s_2 \xrightarrow{e_1} s_3 \in T$$

- e_1 and e_2 are observably independent in S *if*

$$\forall s \in S : s \xrightarrow{e_1} s_1 \in T, \quad s \in \mathrm{FR}(e_1) \wedge (s \in \mathrm{ER}(e_2) \Leftrightarrow s_1 \in \mathrm{ER}(e_2)) \quad \wedge$$
$$\forall s \in S : s \xrightarrow{e_2} s_2 \in T, \quad s \in \mathrm{FR}(e_2) \wedge (s \in \mathrm{ER}(e_1) \Leftrightarrow s_2 \in \mathrm{ER}(e_1)) \quad \wedge$$
$$\forall s \in \mathrm{FR}(e_1) \cap \mathrm{FR}(e_2) \cap S,$$
$$(\forall s \xrightarrow{e_1} s_1, s_1 \xrightarrow{e_2} s_3 \in T : s_1 \in \mathrm{FR}(e_2) \wedge \{s_1, s_3\} \subseteq S, \exists\, s \xrightarrow{e_2} s_2, s_2 \xrightarrow{e_1} s_3 \in T \ \wedge \ s_2 \in \mathrm{FR}(e_1) \cap S) \ \wedge$$
$$(\forall s \xrightarrow{e_2} s_2, s_2 \xrightarrow{e_1} s_3 \in T : s_2 \in \mathrm{FR}(e_1) \wedge \{s_2, s_3\} \subseteq S, \exists\, s \xrightarrow{e_1} s_1, s_1 \xrightarrow{e_2} s_3 \in T \ \wedge \ s_1 \in \mathrm{FR}(e_2) \cap S)$$

■ 3.8

We use the terms *locally*, *effectively* and *structurally* in combination with the previous definitions. We use the term *locally* when $S \subset \mathrm{Reach}(s_0, T)$, *effectively* when $S = \mathrm{Reach}(s_0, T)$ and *structurally* when $S = S$.

To avoid overly long descriptions like *"event* a *effectively observably triggers event* b *in states* $\{s_1, s_2, \ldots, s_{15}\}$*"* we use a number of conventions. For instance we assume that we usually speak about observable behavior (which is the most common situation) hence we skip the term *observably* if no confusion is possible. We also avoid stating the set of states in which the relation arises when the set of states is clear from the context (for instance in a figure of a LzTS we assume that we are talking about all the states depicted, if we are talking about a run, then the set is all the states in the run, etc.).

EXAMPLE 3.2 *In Figure 3.3 we can see a* LzTS *(in fact a* TS*). Events* a *and* b *are* locally independent in state s_0. *They are not structurally independent as* a *triggers* b *in the transition* $s_4 \xrightarrow{a} s_5$. *If the initial state of the system is* s_0*, then the set of reachable states is* $\{s_0, s_1, s_2, s_3\}$ *and events* a *and* b *are considered* effectively independent *as they are independent in all the reachable states.*

Conversely, we can say that a structurally triggers b *because of the transition* $s_4 \xrightarrow{a} s_5$*, however it does not* effectively trigger *it because* s_4 *does not belong to the reachable set of states.* ■ 3.2

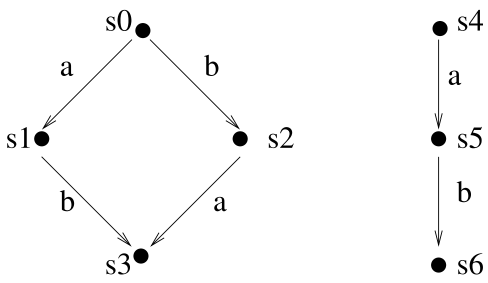


Figure 3.3. Example of local independence.

The are several implications between the different types of relations we have defined. For instance *independence* is stronger than just *concurrency* between two events (independence implies concurrency, but not the other way round). We can see this in Figure 3.5 (a,b). The following equations summarize some of these implications:

$$\begin{array}{lcr} e_i \text{ and } e_j \text{ are logically independent} & \Rightarrow & e_i \text{ and } e_j \text{ are logically concurrent} \\ \Downarrow & & \Uparrow \\ e_i \text{ and } e_j \text{ are observably independent} & \Rightarrow & e_i \text{ and } e_j \text{ are observably concurrent} \end{array} \tag{3.1}$$

$$\begin{aligned} e_i \text{ and } e_j \text{ are independent } \Rightarrow & \neg(e_i \text{ triggers } e_j) \wedge \neg(e_j \text{ triggers } e_i) \wedge \\ & \neg(e_i \text{ disables } e_j) \wedge \neg(e_j \text{ disables } e_i) \end{aligned} \tag{3.2}$$

$$\text{if } S \subseteq S' \text{ then } \begin{cases} e_i \text{ triggers } e_j \text{ in } S \Rightarrow e_i \text{ triggers } e_j \text{ in } S' \\ e_i \text{ disables } e_j \text{ in } S \Rightarrow e_i \text{ disables } e_j \text{ in } S' \\ e_i \text{ and } e_j \text{ are independent in } S \Leftarrow e_i \text{ and } e_j \text{ are independent in } S' \end{cases} \tag{3.3}$$

3.3.2 Computing symbolically the relations between the events of a LzTS

The relationship between two events given a set of states S in a Boolean encoded LzTS $A = \langle \mathcal{V}, \mathcal{V}', \Sigma, \mathrm{TR}, \mathcal{X}_{s_0}, \mathrm{FR} \rangle$ can be symbolically computed with the methods presented in the following sections.[1]

Triggering computation

Let T_{e_i} and T_{e_j} be the TRs of two events e_i and e_j and be S a set of states. Event e_i logically triggers e_j in S iff

$$\mathrm{Image}(S \cap \overline{\mathrm{ER}(e_j)},\ T_{e_i}) \cap \mathrm{ER}(e_j) \cap S \neq \emptyset \tag{3.4}$$

In Figure 3.4 there is a graphical depiction of the elements involved in the equation.

[1] If classical LzTSs are used, it is more optimal to use the computation method defined for *logical* relations, as in these models the definition for observable relations redundantly checks for firing regions, while this information is already stored in the transition relation.

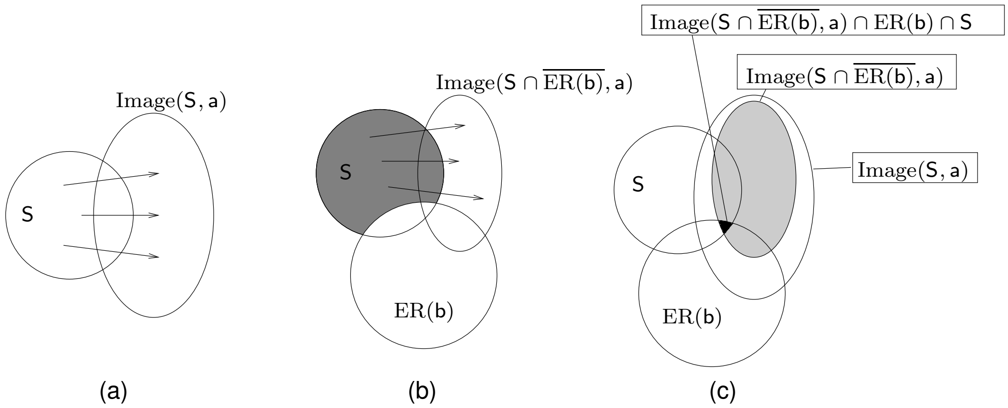


Figure 3.4. (a) The image of set S using event a *i.e.* $\mathrm{Image}(\mathsf{S},\mathsf{a})$, (b) the image of the states of S in which event b is not enabled, and (c) the relationship between previous sets and the set that will identify the states of S in which the event of a triggers b (the black zone).

We can translate this definition into operations with the Boolean algebra representation of a LzTS (see §2.5). Then e_i triggers e_j in S iff

$$\big[\exists_{v\in\mathcal{V}}[\mathrm{TR}(\mathsf{e}_i)\cdot\mathcal{X}_\mathsf{S}\cdot\overline{\mathrm{EF}(\mathsf{e}_j)}]\big]_{\mathcal{V}\leftarrow\mathcal{V}'}\cdot\mathrm{EF}(\mathsf{e}_j)\cdot\mathcal{X}_\mathsf{S}\neq 0$$

That is, if the conjunction of the Boolean formulas is different than the 0 Boolean value.

Using the previous formulas with $\mathcal{X}_\mathsf{S}=1$ (1 is the Boolean representation of the whole state space S) we obtain the structural triggering between two events. Structural triggering is important because it is much easier to compute than effective triggering. Although in many circumstances we would like to obtain the *effective* relationships between events, this would imply to compute the set of reachable states, thus is computationally expensive. On the other hand structural information can be obtained the transition relation definitions (*i.e.* all the T_e) which are already available at the beginning of the process.

We have shown how to compute the logical triggering, to compute the observable triggering we should simply replace equation (3.4) with the following one:

$$\mathrm{Image}(\mathsf{S}\cap\mathrm{FR}(\mathsf{e}_i)\cap\overline{\mathrm{ER}(\mathsf{e}_j)},\ T_{\mathsf{e}_i})\cap\mathrm{ER}(\mathsf{e}_j)\cap\mathsf{S}\neq\emptyset \tag{3.5}$$

Disabling computation

Let T_{e_i} and T_{e_j} be the TRs of two events e_i and e_j and be S a set of states. Event e_i logically disables e_j in S iff

$$\mathrm{Image}(\mathsf{S}\cap\mathrm{ER}(\mathsf{e}_j),\ T_{\mathsf{e}_i})\cap\overline{\mathrm{ER}(\mathsf{e}_j)}\cap\mathsf{S}\neq\emptyset \tag{3.6}$$

In the Boolean algebra representation, e_i disables e_j in S iff

$$\big[\exists_{v\in\mathcal{V}}[\mathrm{TR}(\mathsf{e}_i)\cdot\mathcal{X}_\mathsf{S}\cdot\mathrm{EF}(\mathsf{e}_j)]\big]_{\mathcal{V}\leftarrow\mathcal{V}'}\cdot\overline{\mathrm{EF}(\mathsf{e}_j)}\cdot\mathcal{X}_\mathsf{S}\neq 0$$

Again, using the previous formulas with $\mathcal{X}_\mathsf{S}=1$ we obtain the structural disabling between two events.

We have shown how to compute the logical disabling, to compute the observable disabling we should simply replace equation (3.6) with the following one:

$$\mathrm{Image}(\mathsf{S}\cap\mathrm{FR}(\mathsf{e}_i)\cap\mathrm{ER}(\mathsf{e}_j),\ T_{\mathsf{e}_i})\cap\overline{\mathrm{ER}(\mathsf{e}_j)}\cap\mathsf{S}\neq\emptyset \tag{3.7}$$

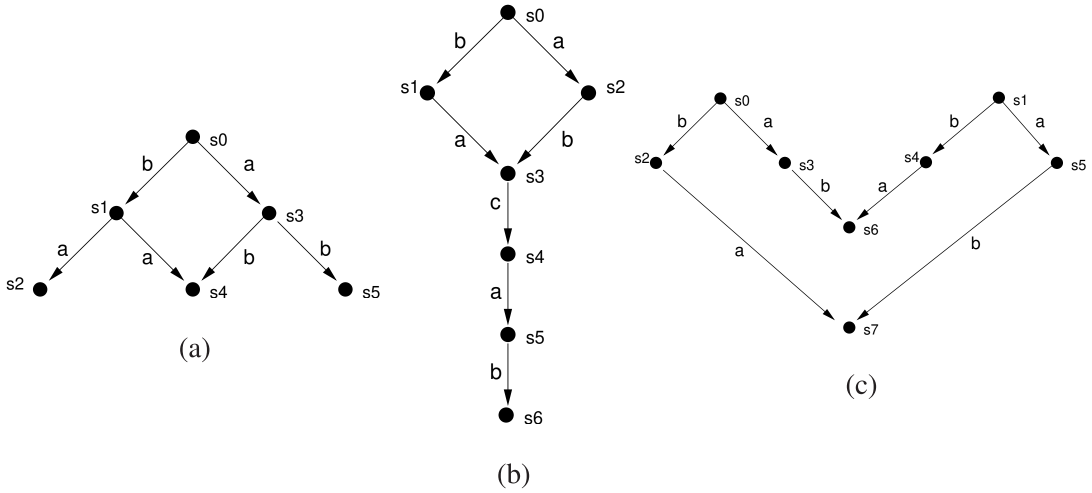


Figure 3.5. Two TSs where events a and b are concurrent but not independent. (a) Finite non-deterministic TS (b) Finite and deterministic TS. (c) Finite and deterministic TS where events a and b are neither concurrent nor independent, although they satisfy necessary conditions (3.2) and (3.8) for concurrency and independence.

Concurrency computation

Concurrency is a little bit more challenging to compute. A state in S must be found such that at least one of the states that can be reached by the sequence $\xrightarrow{\mathsf{e}_i\mathsf{e}_j}$ is also reachable by $\xrightarrow{\mathsf{e}_j\mathsf{e}_i}$. One might be tempted to try with the following equation:

$$\mathrm{Image}(\mathrm{Image}(\mathsf{S}, T_{\mathsf{e}_j}) \cap \mathsf{S}, T_{\mathsf{e}_i}) \cap \mathrm{Image}(\mathrm{Image}(\mathsf{S}, T_{\mathsf{e}_i}) \cap \mathsf{S}, T_{\mathsf{e}_j}) \neq \emptyset \tag{3.8}$$

However although this corresponds to a consequence of concurrency (hence it is a necessary condition) it is not a sufficient condition as Figure 3.5 (c) shows: $\mathrm{Image}(\{\mathsf{s}_0, \mathsf{s}_1, \ldots, \mathsf{s}_7\}, T_a) = \{\mathsf{s}_3, \mathsf{s}_5, \mathsf{s}_6, \mathsf{s}_7\}$ and $\mathrm{Image}(\{\mathsf{s}_3, \mathsf{s}_5, \mathsf{s}_6, \mathsf{s}_7\}, T_b) = \{\mathsf{s}_6, \mathsf{s}_7\}$, while $\mathrm{Image}(\{\mathsf{s}_0, \mathsf{s}_1, \ldots, \mathsf{s}_7\}, T_b) = \{\mathsf{s}_2, \mathsf{s}_4, \mathsf{s}_6, \mathsf{s}_7\}$ and $\mathrm{Image}(\{\mathsf{s}_2, \mathsf{s}_4, \mathsf{s}_6, \mathsf{s}_7\}, T_b) = \{\mathsf{s}_6, \mathsf{s}_7\}$.

The solution comes by using the composition of transition relations. However, first of all some auxiliary notation must be introduced. We will note as $T_\mathsf{e}[\mathsf{S}]$ the transition relation T_e when restricted to set of states S. Formally,

$$T_\mathsf{e}[\mathsf{S}] \stackrel{\mathrm{def}}{=} \{\mathsf{s}_1 \xrightarrow{\mathsf{e}} \mathsf{s}_2 \in T_\mathsf{e} \mid \mathsf{s}_1, \mathsf{s}_2 \in \mathsf{S}\}$$

which in Boolean algebra is computed as $\mathcal{X}_\mathsf{S} \cdot \mathrm{TR}(\mathsf{e}) \cdot \mathcal{X}'_\mathsf{S}$.

Now two events e_i and e_j are logically concurrent in S iff

$$\left(T_{\mathsf{e}_i}[\mathsf{S}] \circ T_{\mathsf{e}_j}[\mathsf{S}]\right) \cap \left(T_{\mathsf{e}_j}[\mathsf{S}] \circ T_{\mathsf{e}_i}[\mathsf{S}]\right) \neq \emptyset$$

as the expression $T_{\mathsf{e}_i}[\mathsf{S}] \circ T_{\mathsf{e}_j}[\mathsf{S}]$ represents the transition relation that relates each state s of S with all the states in S that can be reached by firing the sequence $\xrightarrow{\mathsf{e}_j\mathsf{e}_i}$ from s (and in which the middle state is also in S).

For observable concurrency the concept is almost the same, the only difference is that now the fireability of the event must be checked before the transition can be taken. Hence we overload our auxiliary notation as follows:

$$[\mathsf{S}_1]T_\mathsf{e}[\mathsf{S}_2] \stackrel{\mathrm{def}}{=} \{\mathsf{s}_1 \xrightarrow{\mathsf{e}} \mathsf{s}_2 \in T_\mathsf{e} \mid \mathsf{s}_1 \in \mathsf{S}_1 \ \wedge\ \mathsf{s}_2 \in \mathsf{S}_2\}$$

which is trivially implemented as $\mathcal{X}_{\mathsf{S}_1} \cdot \mathrm{TR}(\mathsf{e}) \cdot \mathcal{X}'_{\mathsf{S}_2}$. Then we say that two events e_i and e_j are observably concurrent in S if and only if

$$\Big([\mathsf{S} \cap \mathrm{FR}(\mathsf{e}_i)]T_{\mathsf{e}_i}[\mathsf{S}] \circ [\mathrm{FR}(\mathsf{e}_j)]T_{\mathsf{e}_j}[\mathsf{S}]\Big) \cap \Big([\mathsf{S} \cap \mathrm{FR}(\mathsf{e}_j)]T_{\mathsf{e}_j}[\mathsf{S}] \circ [\mathrm{FR}(\mathsf{e}_i)]T_{\mathsf{e}_i}[\mathsf{S}]\Big) \neq \emptyset$$

Independence computation

Computation of independence follows the same guidelines as in the case of concurrency but with stronger requirements, due to the universal quantifiers that appear in its definition. Two events e_i and e_j are logically independent in S, if and only if, they do not logically trigger nor disable each other and

$$\Big(T_{\mathsf{e}_i}[\mathsf{S}] \circ T_{\mathsf{e}_j}[\mathsf{S}]\Big) \Leftrightarrow \Big(T_{\mathsf{e}_j}[\mathsf{S}] \circ T_{\mathsf{e}_i}[\mathsf{S}]\Big)$$

If this condition is true, any state reachable through one of the paths can be reached by one of the other paths (and all the states involved in the path belong to S). In fact one can found the states of S that satisfy the condition by simply tweaking a little bit the equation: $\Big(T_{\mathsf{e}_i}[\mathsf{S}] \circ T_{\mathsf{e}_j}[\mathsf{S}]\Big) \cap \Big(T_{\mathsf{e}_j}[\mathsf{S}] \circ T_{\mathsf{e}_i}[\mathsf{S}]\Big)$ gives the common transitions and now we do only have to obtain the domain of the transition by abstracting the next state variables from the expression.

For observable independence, again, the only change is that now the fireability of the event must be checked before the transition can be taken. So we will say that two events e_i and e_j are observably independent in S if and only if they do not observably trigger nor disable each other and

$$\Big([\mathsf{S} \cap \mathrm{FR}(\mathsf{e}_i)]T_{\mathsf{e}_i}[\mathsf{S}] \circ [\mathrm{FR}(\mathsf{e}_j)]T_{\mathsf{e}_j}[\mathsf{S}]\Big) \Leftrightarrow \Big([\mathsf{S} \cap \mathrm{FR}(\mathsf{e}_j)]T_{\mathsf{e}_j}[\mathsf{S}] \circ [\mathrm{FR}(\mathsf{e}_i)]T_{\mathsf{e}_i}[\mathsf{S}]\Big)$$

Although this expression allows to check the independence of two events, it is quite computationally expensive, specially if a big number of tests have to be performed. Fortunately there are some sufficient conditions for independence that are easy to check if the representation of the transition relations allows a fast check of which variables are involved in each event (like in pTRs). However before presenting them, some additional definitions must be given.

DEFINITION 3.9

Let e_i *and* e_j *be two events, and* $\mathrm{TR}(\mathsf{e}_i) = \langle E_{\mathsf{e}_i}, \xi_{\mathsf{e}_i}\rangle$ *and* $\mathrm{TR}(\mathsf{e}_j) = \langle E_{\mathsf{e}_j}, \xi_{\mathsf{e}_j}\rangle$ *be their corresponding partial transition relation. Then event* e_i *is said to have* independent causal support *from event* e_j *if* $\big(\xi_{\mathsf{e}_j} \cup \xi'_{\mathsf{e}_j}\big) \cap \mathrm{Sup}(\mathrm{TR}(\mathsf{e}_i)) = \emptyset$. ■ 3.9

Remember that the transition relations we use are *partial* (see §2.7) thus $\mathrm{Sup}(\mathrm{TR}(\mathsf{e}_i))$ follows Definition 2.17 and can be a very small subset of $\mathcal{V} \cup \mathcal{V}'$.

LEMMA 3.1

Events that have mutual independent causal support one from each other, are structurally independent. ■ 3.1

Proof: $\big(\xi_{\mathsf{e}_j} \cup \xi'_{\mathsf{e}_j}\big) \cap \mathrm{Sup}(\mathrm{TR}(\mathsf{e}_i)) = \emptyset$ *implies that* $\xi_{\mathsf{e}_j} \cap \mathrm{Sup}(\mathrm{TR}(\mathsf{e}_i)) = \emptyset$ *and* $\xi'_{\mathsf{e}_j} \cap \mathrm{Sup}(\mathrm{TR}(\mathsf{e}_i)) = \emptyset$.

If $\xi_{\mathsf{e}_j} \cap \mathrm{Sup}(\mathrm{TR}(\mathsf{e}_i)) = \emptyset$ *is true, then event* e_j *is not able to* write *on the variables on which the enabledness of* e_i *depends. Then, any state obtained from the activation of* e_j *will preserve the enabledness of* e_i*. Moreover, as* e_i *cannot* read *the variables that* e_j *modified, it cannot perform different kinds of writings depending on whether* e_j *already fired or not.*

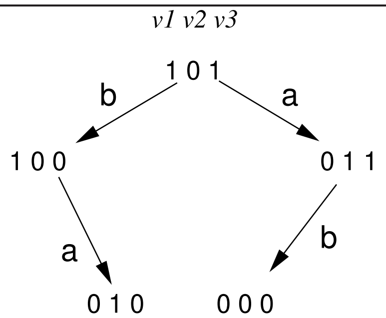


Figure 3.6. A TS in which the firing of events a and b does not change the enabledness of the other event, but $101 \xrightarrow{ab} 000$ does not coincide with $101 \xrightarrow{ba} 010$.

If the same condition is satisfied by e_i with respect to e_j then we guarantee that none of the events is able to change the enabledness status of the other, but it is not guaranteed that all states reached by $\xrightarrow{\mathsf{e}_i \mathsf{e}_j}$ are also reachable by $\xrightarrow{\mathsf{e}_j \mathsf{e}_i}$ as the example in Figure 3.6 shows. In the figure, a TS is shown that has two events, namely a and b. The pTRs for both events are given below:

- $\mathrm{TR}(\mathsf{a}) = \langle v_1\ \overline{v_1'}\ v_2', \{v_1, v_2\}\rangle$.
- $\mathrm{TR}(\mathsf{b}) = \langle v_3\ \overline{v_2'}\ \overline{v_3'}, \{v_2, v_3\}\rangle$.

The example satisfies previous condition for both events as $\xi_{\mathsf{a}} \cap \mathrm{Sup}(\mathrm{TR}(\mathsf{b})) = \{v_1, v_2\} \cap \{v_3, v_2', v_3'\} = \emptyset$ as well as $\xi_{\mathsf{b}} \cap \mathrm{Sup}(\mathrm{TR}(\mathsf{a})) = \{v_2, v_3\} \cap \{v_1, v_1', v_2'\} = \emptyset$. However, as both event are able to overwrite *some of the variables the other event writes, the ending state differs depending on the firing order.*

If $\xi'_{\mathsf{e}_j} \cap \mathrm{Sup}(\mathrm{TR}(\mathsf{e}_i)) = \emptyset$ the event e_j is not able to write *in the same variables as e_i does. If the same condition applies to e_i with respect to e_j, then it is not possible to create a different path that remembers the firing order of the events like in Figure 3.6.*

This lemma states that only events that, at most, *read* from common elements are guaranteed to be independent –any combination of read/write or write/write on common variables is dangerous. This is well-known result in the partial order domain [God96] and allows to implement a fast independence test.

For instance in Figure 2.5 (b) we can see that events a and b are effectively independent. Imagine that the two events have the following transition relations: $\mathrm{TR}(\mathsf{a}) = \langle \overline{v_1} v_1', \{v_1\}\rangle$ and $\mathrm{TR}(\mathsf{b}) = \langle \overline{v_2} v_2', \{v_2\}\rangle$. In this case they are also structural independent as each of the events reads and writes on different sets of variables.

PART II

TIMING VERIFICATION - FUNDAMENTALS

Summary of Part II

Verification of timed systems is a hard problem, so several strategies have been developed to alleviate its complexity. One of these strategies is to use iterative refinements, based in the observation that the computation of the exact timed state space is often unfeasible but that a less exact knowledge (easier to compute) is sometimes sufficient.

In this part we present the fundamental notions of our work in the verification of timed concurrent systems. Our contributions are an extension of the work of [Peñ03] which, in turn, extends the ideas presented in [PCKP00]. The core of all these methodologies is to perform iterative refinements using information about the relative timing of the events of the system (*i.e.* information about which event is forced to happen before another in a given context).

In the first chapter of this part, we present the problem of the verification of timed systems and we review the most common approach to solve it. Basically, we explain how the exact timed state space, which is infinite, can be represented in a finite structure. We also pay special attention to previous iterative refinement approaches.

In Chapter 5 we outline the overall verification strategy of [Peñ03, PCKP00] using two small examples. Next, the different theoretical aspects of the verification approach are introduced. A trace semantics is defined to unify the reasoning with the different computational models used by the verification. We present a naive individual trace verification method which is not suitable for concurrent systems due to the potential huge number of traces reaching an error state. A very general criterion is introduced that allows to reuse the result of the timing consistency analysis of a trace on other traces. Unfortunately the criterion is too much general to be easily implemented in a symbolic setting. Hence we derive more restricted criterion, called *enabling compatibility*, which can be easily implemented.

Enabling compatibility makes it possible that the timing analysis of a particular trace can also be applied to the set of traces that share the same enabling orderings. To do so, an automaton which identifies enabling compatible traces is presented. Relative timing information can be incorporated into the automaton, which is then composed with the original system (Chapter 6). The effect of such composition is to remove from the system a timing inconsistent trace as well as all their enabling compatible traces. However, due to the huge amount of concurrent traces usually reaching a state in a concurrent system, enabling compatibility alone is not efficient enough.

Event structures are then introduced in Chapter 7 as a model that represents succinctly a set of concurrent traces and for which timing analysis is possible and quite efficient. Algorithms for such a task are provided (Chapter 8). From an event structure that has been timing analyzed, it is also possible to construct an automaton that removes all the enabling compatible traces with some timing inconsistent trace represented by the event structure. The chapter ends with the description of the process of construction of such automaton.

Finally, in Chapter 9, we combine all the ideas developed in previous chapters to form the skeleton of a fully automated iterative verification methodology. However, it is far from being perfect. In Part III this verification methodology is corrected, completed and optimized.

Chapter 4

VERIFICATION OF TIMED SYSTEMS

Summary

This chapter reviews the previous work on the formal verification of timed systems. The attention is focused in those modeling, specification and analysis alternatives more widely used. For a deeper insight, the interested reader is addressed to the provided references.

The correctness of timed systems depends on their timing properties, hence considering quantitative time is essential for their analysis. The main paradigms for the incorporation of quantitative timing information to the system's models are first reviewed. A widely used approach is that of *continuous-time* for which the most popular representative, the *timed automata* modeling formalism, is analyzed.

Since the properties we aim to verify do not require very expressive specifications, we do only briefly review the most relevant specification formalisms for timed systems, mainly timed temporal logics.

Regarding the analysis of timed systems, the main verification method relies on the construction of a finite respresentation of the timed state space. Therefore we introduce the reachability analysis problem in timed systems and we comment many of the available alternatives to solve it. We particularly focus on iterative verification methods, because our methodology falls into this category.

4.1 Introduction

The verification of a timed system consists in determining whether the system, whose behavior depends on the timing of its events, satisfies or not a set of given properties. To accomplish this task we need the following ingredients:

- A model of the system, which is capable to capture those behaviors of the system that are relevant for the verification.
- A specification language, expressive enough to state the properties of interest.
- A verification methodology, which is suitable to be used in conjunction with the modeling and the specification formalisms.

For each one of this elements, we review the most relevant contributions in the following sections. In §4.2 we describe and compare some of the most popular timed models. In §4.3 we briefly consider the specification languages, and in §4.4 we overview the available verification methodologies.

4.2 The models

Obviously, if we want to analyze a system whose behavior depends on time, then this type of information must appear in the model. There are many models that incorporate timing information. Among them, the most popular are *timed automata* [AD94]. Given their importance, we briefly introduce them in §4.2.2.

Other relevant formalisms exist, such as *timed transition systems* [HMP92a] or *timed Petri nets* [Zub80], to cite some, but they do not enjoy the same level of acceptance. However, they are still valuable models for research as each of them has its own interesting properties. For instance in this work we do use timed transition systems rather than timed automata for the reasons we explain in §4.2.3.

4.2.1 Models of time

Most of the early works in formal verification were only focused at verifying the functional properties of systems, and only a qualitative notion of time was present into the models (mostly finite automata) and the specifications (mostly temporal logic). In such cases, properties only assert, for example, that a certain condition is always true or that a expected response of the system eventually occurs.

While qualitative modeling of time allows the efficient verification of certain properties, it is not satisfactory for verifying the correctness of systems that depend crucially on timing: combinatorial circuits must meet some given clock requirements, embedded controllers must respond to interrupts within some time interval, etc. As a more precise example, consider the statement "*open the door upon detection of a person*" of an automatic door. This temporal sequencing carries no quantitative information on the delay allowable between the detection and the opening action. Hence, it is not possible to directly model the opening of the door "*less than 2 seconds after detecting the person*".

For those systems whose correctness depends on a proper timing, often called *time-critical* or *real-time* systems, a quantitative notion of time must be incorporated both into the system models and also into the specification formalisms. The way time is represented has a crucial impact on the size of the resulting timed state space. Three main approaches exist for that purpose: *discrete-time*, *fictitious-clock* and *continuous-time*.

Discrete time

Formalisms based on the *discrete-time* notion (*e.g.* [AK83, JM86, EMSS90, BMPY97]) map time onto the integer domain. This approach is appropriate to describe the behavior of *synchronous* systems where all components are driven by a common global clock. However, to be able to model *asynchronous* systems they require to discretize time by choosing some fixed *time quantum*, so that the separation of two events in the timed domain is always a multiple of such quantum. Figure 4.1 (a) illustrates this concept.

The main advantage of this approach is that the timing analysis and timed state exploration techniques are generally simpler than their counterparts for continuous-time. Its main drawback is that determining the time quantum *a priori* may not be easy and therefore may compromise the accuracy of the resulting model. In this sense, in [BS91] it is shown that the reachability problem for asynchronous circuits with bounded delays cannot be solved correctly using the discrete-time model. On the other hand, it has also been proved that discrete time can give equivalent results to continuous time if a number of conditions in the model are met [HMP92b, AMP98, BMT99]. The required conditions are not very strong (our timed transition system fulfill them), but still the choice of a sufficiently small time quantum to model the system accurately enough may blow up the timed state space so that verification becomes unfeasible.

Fictitious clock

The *fictitious-clock* approach introduces a special *tick* event into the model (*e.g.* [AH89, Bur89, Ost90, HLP90]). Thus, time is understood as a global state variable that ranges over the domain of natural numbers, and is incremented with every *tick* event. Generally, this paradigm allows arbitrarily many events of any process between two successive *ticks*.

The delay between two events is measured by counting the number of *ticks* between them. When it is required that there be k *ticks* between two events, it can only be inferred that the actual delay between them is at least $k-1$ time units and at most $k+1$ time units. Therefore, it is impossible to determine precisely some typical and simple requirements on the delays between events, *e.g.* "*the delay between the detection and the alarm equals 2 seconds*". Figure 4.1 (b) illustrates the fictitious-clock approach. Although events may occur at any time, the precise occurrence instant can only be approximated to be in between of two *tick* events.

In general, the models based in the fictitious-clock approach require a somewhat cumbersome encoding mechanism to measure time intervals. This reduces the readability of the model and makes modifying the model a tricky and prone to errors task. This might explain why this model of time is the least popular of the three approaches.

Continuous time

The third approach for the modeling of real-time behavior, models time more realistically as a continuous magnitude. Some examples of the use of the so-called *continuous-time* or *dense-time* can be found in [Dil89, Koy90, ACD90, HMP92a, HNSY92, YSSC93, RM94, LPY95, SB97]. These approaches associate a non-negative real value to each event of the system, and therefore to each reachable state. Continuous-time differs from the other time models because the exact bounds on the actual delays between the events can be expressed. Moreover, the use of continuous-time allows a more precise modeling of analog or asynchronous systems, as well as systems that operate at different clock frequencies. Figure 4.1 (c) illustrates the continuous-time notion.

Since this approach does not rely on the use of a discretization constant, one possible drawback of using the reals as time domain is the added complexity. However, it has been shown that with

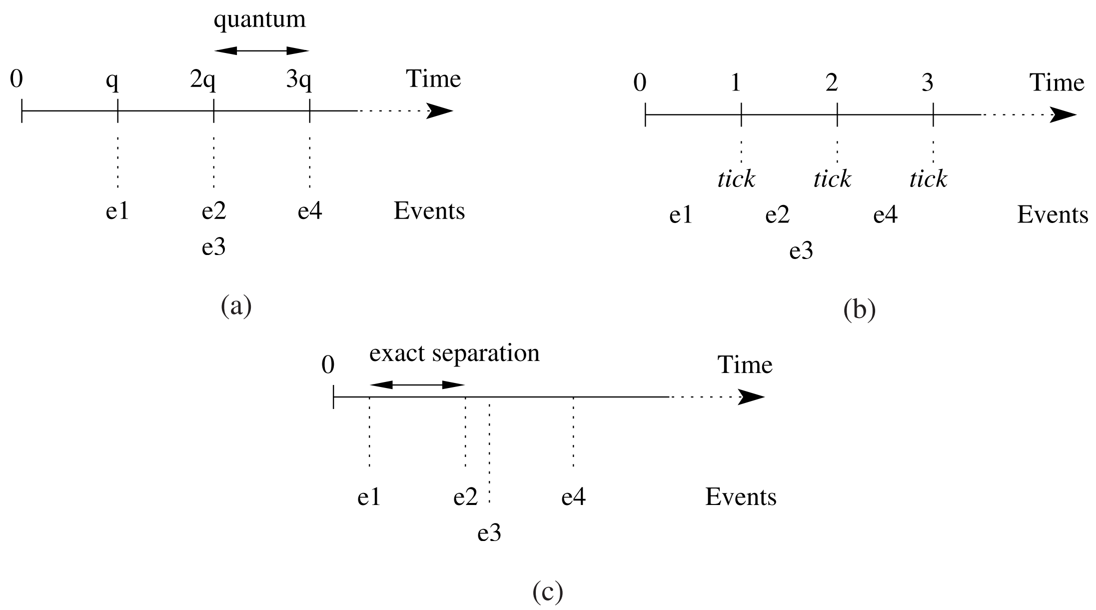


Figure 4.1. Three representations of time: (a) discrete-time, (b) fictitious-clock and (c) continuous-time.

appropriate techniques (*e.g.* [AIKY92, HMP92b, ABH$^+$97, TKY$^+$98]), the analysis of continuous-time models does not increase in complexity, if compared to the discrete-time counterparts. The main idea behind such techniques consists in breaking the infinite continuous timed state space into equivalence classes, such that all states in the same class lead to the same behavior and can be analyzed together.

4.2.2 Timed automata

With the wide adoption of the continuous-time paradigm, the *timed automata* framework, pioneered by [Dil89, ACD90], has become one of the most popular choices to incorporate quantitative time into the system's models. Several timing verification tools use this formalism as their basis: COSPAN [AK95], KRONOS [Yov97], UPPAAL [BLL$^+$95], MOCHA [AHM$^+$98], RED [Wan01], IF [BGO$^+$04] among others.

A timed automaton is a classical finite automaton augmented with a finite set of real-valued *clocks*. That is, a timed automaton is built from two elements: a finite automaton which describes the (control) states or *locations*, and the transitions between them; and a set of clocks used to specify the quantitative time constraints. Transitions are assumed to happen instantaneously, whereas time can elapse when the automaton is at a given state. In the initial location all clock values are set to zero. Then, the clocks evolve at a uniform rate taking non-negative real values. At any instant, *reading* a clock tells how much time has elapsed since the last time the clock was reset.

Besides the source and target locations, a *transition* is formed by other three elements: a *guard*, also called *clock constraint* or *firing condition*, such that the transition cannot be taken unless the current values of the clocks satisfy the guard; a *label*, or action name; and a set of clocks that must be reset after the transition has been taken.

A clock constraint is often associated to each location of the automaton. This type of constraint, called the *invariant* of the location, forces that time can elapse in the location only as long as the invariant is still satisfied.

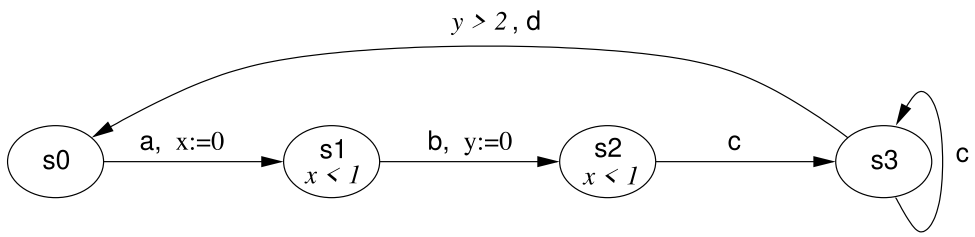


Figure 4.2. An example of timed automaton.

To provide a formal definition of a timed automaton, clock constraints must be precisely defined first. Let X be a set of real clocks, the set $\Phi(X)$ of clock constraints φ allowable as location invariants and enabling conditions, is defined as:

- All inequalities of the form $x < c$, $x \leq c$, $c < x$, $c \leq x$ are in $\Phi(X)$ where x is a clock and c is a non-negative real number.
- If φ_1 and φ_2 are in $\Phi(X)$ then the constraint $\varphi_1 \wedge \varphi_2$ is in $\Phi(X)$.

Notice that if X contains k clocks, then each clock constraint delimits a convex region in a k-dimensional Euclidean space. This observation provides a way for representing the timed state space of a timed automaton (see §4.4.1).

The formal definition of a timed automaton follows:

DEFINITION 4.1 (TIMED AUTOMATA)
A timed automaton *is a 6-tuple* $A = \langle \Sigma, S, S_o, X, I, T \rangle$ *such that:* Σ *is a finite* alphabet*;* S *is a finite set of* locations *(states);* $S_o \subseteq S$ *is a set of* initial locations*;* X *is a set of* clocks*;* $I : S \longrightarrow \Phi(X)$ *is the* location invariant*; and* $T \subseteq S \times \Sigma \times \Phi(X) \times 2^X \times S$ *is a set of* transitions.

The 5-tuple $\langle \mathsf{s}, \mathsf{a}, \varphi, \lambda, \mathsf{s}' \rangle \in T$ *is a transition from location* s *to location* s′ *corresponding to the action labeled as* a*. The clock constraint* φ *specifies when the transition is enabled, and* $\lambda \subseteq X$ *is the set of clocks that are reset when the transition is taken.* ■ 4.1

EXAMPLE 4.1 *Consider the timed automaton in Figure 4.2.*

When the system switches from the initial location s_0 *to location* s_1 *by the action* a*, the clock* x *is reset to* 0*. Therefore, in all the other locations, the value of clock* x *shows the time elapsed since the last occurrence of action* a*.*

The invariant $x < 1$ *associated to locations* s_1 *and* s_2*, ensures that the arc* $\mathsf{s}_2 \xrightarrow{\mathsf{c}}$ *is taken within time* 1 *of the preceding* a*. Similarly, the fact that the arc* $\mathsf{s}_1 \xrightarrow{\mathsf{b}}$ *resets clock* y *together with the condition* $y < 2$ *on arc* $\mathsf{s}_3 \xrightarrow{\mathsf{d}}$ *ensures that the delay between* b *and the following* d *is always greater that* 2*.*

Locations s_0 *and* s_3 *have no invariant constraint. This means that the system can spend an arbitrary amount of time in them. As a consequence, there is no guarantee that arc* $\mathsf{s}_0 \xrightarrow{\mathsf{a}}$ *or* $\mathsf{s}_3 \xrightarrow{\mathsf{d}}$ *are taken at some time instant.* ■ 4.1

The semantics of a timed automaton A is defined by associating a transition system, $\mathcal{T}(A)$, to it. At any time, the *configuration* or global state of the system modeled by the timed automaton is given by a location, s, of the automaton and a clock interpretation, v, that assigns a real value to each clock. Thus, a configuration is a pair (s, v) where $\mathsf{s} \in S$ and $v : X \longrightarrow \mathbb{R}^+$. The set of *initial configurations* is given by the set $\{(\mathsf{s}, v) \mid \mathsf{s} \in S_o \ \wedge \ \forall x \in X \ [v(x) = 0]\}$, *i.e.* the set of initial locations in which all the clocks are set to 0.

The system changes from one configuration to another by means of two types of transitions:

- *Delay transition*: which lets a time delay $\delta \in \mathbb{R}$ to elapse, *i.e.* increasing the value of all clocks by δ. Then, the system moves from configuration (s, v) to configuration (s, v'), written as $(\mathsf{s}, v) \xrightarrow{\delta} (\mathsf{s}, v')$, where $\forall x \in X \ v'(x) = v(x) + \delta$.
- *Action transition*: which executes an actual transition $\langle \mathsf{s}, \mathsf{a}, \varphi, \lambda, \mathsf{s}' \rangle \in T$ of the automaton. This is written as $(\mathsf{s}, v) \xrightarrow{\mathsf{a}} (\mathsf{s}', v')$, such that v satisfies the guard φ and $v' = v[\lambda := 0]$.

Thus, the timed state space of a timed automaton can be seen as an *infinite* transition system $\mathcal{T}(A) = \langle Q, \Sigma \cup \mathbb{R}, R, Q_o \rangle$, where: Q and Q_o are the set of configurations and the initial configurations, respectively; the original alphabet Σ is augmented with the real numbers to include the delay transitions; and R is the *transition relation* obtained by combining the delay and the action transitions.

EXAMPLE 4.1 (CONT.) *Let the timed automaton in Figure 4.2 be called A.*

The state-space of $\mathcal{T}(A)$ is given by $Q \subseteq \{\mathsf{s}_0, \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3\} \times \mathbb{R}^2$. A sequence of possible transitions is, for example:

$$(\mathsf{s}_0, 0, 0) \xrightarrow{1.2} (\mathsf{s}_0, 1.2, 1.2) \xrightarrow{\mathsf{a}} (\mathsf{s}_1, 0, 1.2) \xrightarrow{0.7} (\mathsf{s}_1, 0.7, 1.9) \xrightarrow{\mathsf{b}} (\mathsf{s}_2, 0.7, 0) \xrightarrow{0.1} (\mathsf{s}_2, 0.8, 0.1) \xrightarrow{0.1} \cdots$$

where the numbers at each configuration are (from left to right) the values of the clocks x and y.

■ 4.1

Solving the reachability problem for a timed automaton is a nontrivial task since the number of potential configurations is infinite. In order to solve the task, finite representations of the infinite state space are required (see §4.4.1). However, even using such representations, the state explosion problem often limits the practical applicability of the algorithms and tools that rely on the reachability set. More precisely, the problem is PSPACE-hard [AD94]. Moreover, in [CY91] it was proved that both sources of complexity, the number of clocks and the magnitude of the constraints yield to PSPACE-hardness independently of each other.

4.2.3 Timed transition systems versus timed automata

In this work we use timed transition systems (see §2.2) instead of timed automata. The reasons for this choice are practical and theoretical. From a practical point of view our work is an extension of [Peñ03], in which the same model was chosed. This allows to compare much more easily both works and makes it possible to have a common basis.

From a theoretical perspective, the main idea behind [Peñ03] and this work is to use the relative timing of the events in substitution of quantitative timing. For instance, to use the fact that some event has happened before another to infere something about the timing information, instead of using an explicit clock to keep it. In this scenario the methodology greatly benefits from a model in which there is a strong relationship between the firing of the events and the clocks of the system. This is the case of timed transition systems but not of timed automata.

4.3 Timed specifications

The wide spectrum of specification formalisms can be roughly divided into two categories: logic-based and automata-based approaches.

In the logic-based approach, originally introduced in [Pnu77], the properties under verification are stated as formulas using a *temporal logic* (see [Eme90] for an overview). Despite of a number of derivatives, two main families of temporal logics exist: *linear temporal logic* (LTL) pioneered by [Pnu77, OL82], and *computational tree logic* (CTL) pioneered by [BAPM81, EC82]. Provided the specification in the form of a set of formulas of the logic, the state space of the system is explored checking whether each formula is satisfied in all possible behaviors of the system. The resulting verification methods, *i.e.* the so-called *model-checking* methods, were pioneered by [LP85, CES86, BCM$^+$92, GW91] among others.

In automata-based approaches, the same formalism is used for describing both the system and the specification containing the properties of interest. Then, the verification consists in showing that all behaviors of the system are also part of the specification. This is often achieved by showing an *implementation relation* between the system and its specification in terms of *language containment* [Tho81], *simulation* [DHWT91] or an *homomorphism* [Kur94],for example.

Although timed specifications are one of the major ingredients of formal verification and has been a very active research area, we will not describe them in more detail. The reason for this is that our work only focuses in safety properties which can easily described using a (Boolean) formula that identifies the good/bad states/transitions. Although this might seem very restrictive when compared with the expressiveness of the abovementioned specifications, in fact many of its limitations can be bypassed by modifying the system under analysis [ABABL03], a strategy that is inscribed in the automata-based approaches of the previous classification.

4.4 Verification methods

The basic verification strategy is to generate the timed state space of the system under analysis. In fact, this strategy can be viewed as the only possible one, as the differences between methods are, fundamentally:

- How they generate the timed state space.
- If they do use an abstraction of the model or not, and which abstraction is used.
- If they test the properties during the generation of the state space or once this process has finished.

4.4.1 Construction of the timed reachability space

Most approaches for the verification of timed systems rely on the construction of the timed reachability space. However, the number of timed states is infinite (in fact uncountable). For example, in the case of timed automata such infiniteness has two sources: the clock values are potentially unbounded, and even when they are restricted to a bounded interval, the set of real valuations is dense. Therefore, typical model-checking algorithms are no longer feasible. In order to overcome such complexity, finite representations of the timed state space must be provided.

Although the remaining of this section refers to timed automata, the presented techniques are generally applicable to the reachability problem in timed systems.

Clock regions

The main idea to overcome the aforementioned complexity is the use of *clock regions* [ACD90] which we introduce intuitively as follows. Consider two configurations (s, v) and (s, v') of a timed automaton, where the clock valuations v and v' are *very close*. Assume, for example, that there is a single clock x and that $v(x) = 1.2347$ and $v'(x) = 1.235$. Given a certain notion of

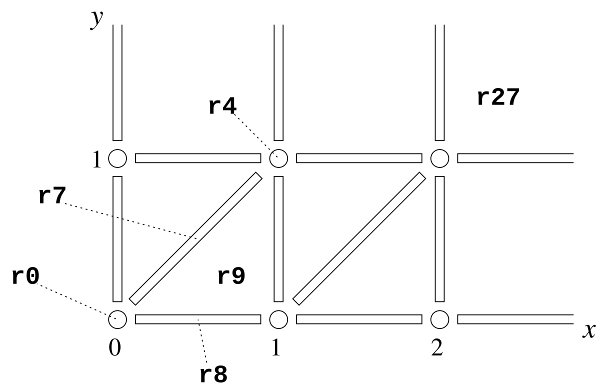


Figure 4.3. Regions for two clocks x and y, with constraints $x \sim k$ ($k \in \{0, 1, 2\}$) and $y \sim k$ ($k \in \{0, 1\}$).

closeness for configurations (see below), the automata will behave in roughly the same way from either of both configurations, and hence the same properties will be satisfied.

If the the clock constraints only contain integer numbers, an equivalence relation [ACD90] can be defined on the space of configurations, that equates two configurations if: they correspond to the same location, they agree on the integral part of the clock valuations, and they agree on the ordering of the fractional part of the clock valuations. The integral parts of the clocks are needed to decide if a particular clock constraint is met, whereas the ordering of the fractional parts is needed to decide which clock will change its integral part first. Although the integral part of a clock x can get arbitrarily large values, if x is never compared with a constant greater that c_x , the actual value of x beyond c_x does not affect the behavior of the automaton.

More formally, let $v(x) \in \mathbb{R}^+$ be the valuation of a clock, $\lfloor v(x) \rfloor$ denotes the integral part of the valuation whereas $\mathrm{frac}(v(x))$ denotes its fractional part, such that $v(x) = \lfloor v(x) \rfloor + \mathrm{frac}(v(x))$. The *region equivalence* $v_1 \cong v_2$ for two clock valuations v_1 and v_2 is defined by the following conditions:

- $\forall\, x \in X$, either $\lfloor v_1(x) \rfloor = \lfloor v_2(x) \rfloor$ or, $\lfloor v_1(x) \rfloor > c_x$ and $\lfloor v_2(x) \rfloor > c_x$.
- $\forall\, x, y \in X$ with $v_1(x) \leq c_x$ and $v_1(y) \leq c_y$, $\mathrm{frac}(v_1(x)) \leq \mathrm{frac}(v_1(y))$ iff $\mathrm{frac}(v_2(x)) \leq \mathrm{frac}(v_2(y))$.
- $\forall\, x \in X$ with $v_1(x) \leq c_x$, $\mathrm{frac}(v_1(x)) = 0$ iff $\mathrm{frac}(v_2(x)) = 0$.

Then, a *clock region* is an equivalence class of clock valuations induced by $\cong$. Each region can be characterized by the finite set of constraints it satisfies. For example, given a clock valuation $v(x) = 0.5$ and $v(y) = 0.8$ for clock x and y, every clock valuation in the clock region for v, denoted $[v]$, satisfies the constraint $0 < x < y < 1$. The following example illustrates these ideas more intuitively.

EXAMPLE 4.2 *Figure 4.3 [Kro99] shows the set of possible clock regions for two clocks x and y. Only constraints of the form $x \sim k$ with $k \in \{0, 1, 2\}$, and $y \sim k$ with $k \in \{0, 1\}$ have been considered. Recall that $\sim \in \{<, \leq, =, \geq, >\}$.*

The example contains 28 clock regions. Some of them correspond to corner points, *like* `r0`, *characterized by the constraint $[x = y = 0]$. Other regions are* open surfaces *in the plane, like* `r9` *characterized by $[0 < y < x < 1]$, or* `r27` *characterized by $[x > 2 \,\wedge\, y > 1]$. Finally, the other regions are* open segments, *like* `r7`, *characterized by $[0 < x = y < 1]$.*

The system starts in $\mathtt{r0}$ *and as time passes, the clocks increase their values simultaneously. Thus,* $\mathtt{r7}$ *is visited next, then* $\mathtt{r4}$*, etc. If instead of letting time to elapse, a transition that resets some of the clocks is performed, a region on the axes is reached. For example, the reset of clock* y *while in* $\mathtt{r7}$ *leads to* $\mathtt{r8}$*.* ■ 4.2

Although the set of clock regions is finite, its cardinality grows exponentially with the number of clocks: for n clocks with constraints in which every constant k is upper bounded by K, the number of regions is $O(n!K^n)$. As a consequence, whereas determining the truth of a CTL formula has linear complexity, the problem is PSPACE-complete for a timed automaton and a TCTL formula [ACD93]. Therefore, efficient representations for handling sets of regions must be devised.

Region automata

The equivalence relation $\cong$ over the clock valuations can be extended over the set of possible configurations of the timed automaton. Thus, two configurations are equivalent, *i.e.* $(\mathsf{s}_1, v_1) \cong (\mathsf{s}_2, v_2)$ iff $\mathsf{s}_1 = \mathsf{s}_2$ and $v_1 \cong v_2$. The resulting equivalence classes of configurations of a timed automaton A, are captured by the so-called *region automaton* [AD94], denoted by $\mathcal{R}(A)$. A state in $\mathcal{R}(A)$ is of the form (s, α) where s is a location of A and α is a clock region.

The interpretation is that whenever the configuration in A is (s, v), the state of $\mathcal{R}(A)$ is $(\mathsf{s}, [v])$. Thus, the initial states of $\mathcal{R}(A)$ are of the form $(\mathsf{s}_o, [v_o])$ where $\mathsf{s}_o \in S_o$ and $\forall\, x \in X\ v_o(x) = 0$. Also, there is an edge $(\mathsf{s}, \alpha) \xrightarrow{\mathsf{a}} (\mathsf{s}', \alpha')$ in $\mathcal{R}(A)$ iff $(\mathsf{s}, v) \xrightarrow{\mathsf{a}} (\mathsf{s}', v')$ in A for some $v \in \alpha$ and $v' \in \alpha'$.

EXAMPLE 4.3 *Consider the timed automaton and its corresponding region automaton shown in Figure 4.4. Only the regions reachable from the initial region* $(\mathsf{s}_0, [x = y = 0])$ *are shown. Notice that the timing constraints cause that the switch from* s_2 *to* s_3 *is never taken. The only reachable region for location* s_2 *satisfies the clock constraint* $[1 = y < x]$*. This region has no outgoing edges because, in order for event* $\mathtt{c}$ *to happen, the constraint* $[x < 1]$ *must hold, and that is not possible.* ■ 4.3

Zone automata

Region automata can be easily simplified by collapsing groups of regions into *convex geometric regions* or *clock zones* [BD91, ACD+92, AD94]. For example, in the region automaton of Figure 4.4 (b), there are three regions for location s_1 with associated clock regions $[y = 0 < x < 1]$, $[y = 0, x = 1]$ and $[y = 0, x > 1]$. These regions could be collapsed to obtain the union $[y = 0 < x]$, for example. More precisely, a clock zone is formed by a conjunction of clock constraints each of which puts a lower or upper bound on a clock or a difference of two clocks. Given a timed automaton A, its zone automaton $\mathcal{Z}(A)$ can be obtained in a similar way as for the region automaton.

EXAMPLE 4.4 *Figure 4.5 depicts the zone automaton for the timed automaton in Figure 4.4. Notice that unlike the region automaton, in the zone automaton each vertex has at most one successor per symbol. Also, the number of vertexes of* $\mathcal{Z}(A)$ *is less than those in* $\mathcal{R}(A)$*.* ■ 4.4

Theoretically, in the worst case, the number of zones is exponential with respect to the number of regions, therefore the zone automaton may be exponentially bigger than the region automaton. However, in most practical cases, the zone automaton has less reachable vertexes and provides an improvement in performance. The reason is that, while the number of clock regions depends on the

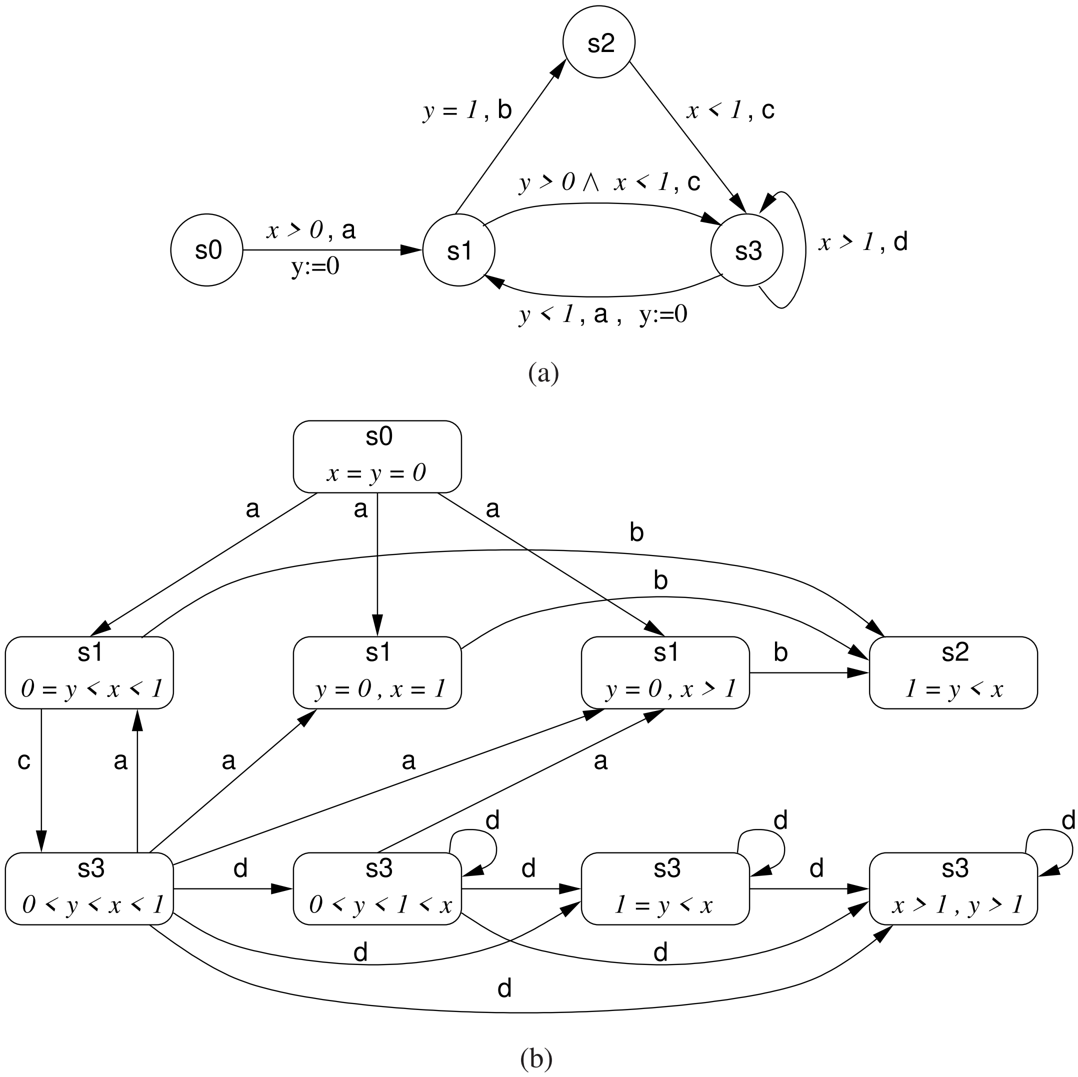


Figure 4.4. Timed automaton (a) and corresponding region automaton (b).

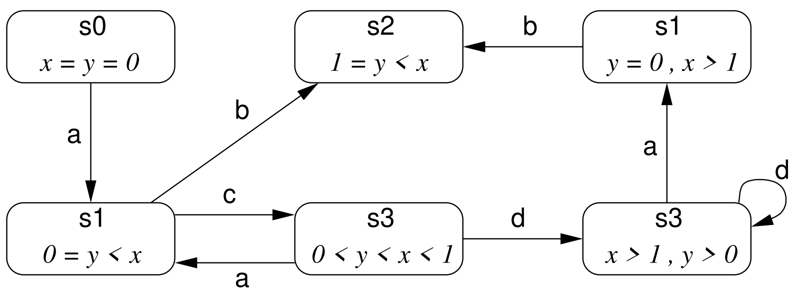


Figure 4.5. Zone automaton for the region automaton in Figure 4.4 (b).

magnitudes of the constants used by the clock constraints, the number of zone regions is relatively insensitive to such fact.

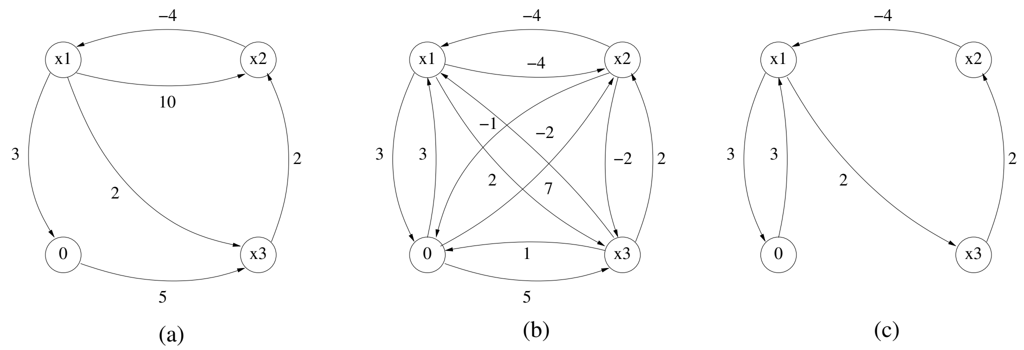


Figure 4.6. (a) Graph representing a DBM, (b) its shortest-path closure and (c) its shortest-path reduction [LLPY03]

Difference-bound matrices

Clock zones can be efficiently represented by sets of linear inequalities using *difference-bound matrices* (DBM) [Dil89]. Suppose a timed automaton has k clocks, $x_1, \ldots, x_k$. Then a clock zone can be represented by a $(k+1) \times (k+1)$ matrix D. The matrix has $(k+1)$ entries because the clock x_0 is used to represent the constant 0. Thus the entry D_{i0} gives an upper bound of the clock x_i, whereas the entry D_{0i} gives a lower bound of the clock. For every pair i, j the entry D_{ij} gives an upper bound on the difference of clocks x_i and x_j.

To distinguish between a strict and a non-strict bound and allow the absence of a bound, the so-called *bounds-domain* $\mathbb{D}$ for the entries of the matrix is defined to be $\mathbb{Z} \times \{0,1\} \cup \{\infty\}$: the constant ∞ denotes the absence of a bound; the bound $(c,1)$ for $c \in \mathbb{Z}$, denotes the non-strict bound $\leq c$; and the bound $(c,0)$ denotes the strict bound $< c$. A clock valuation v satisfies a DBM D if, and only if, for all $1 \leq i \leq k$, $x_i \leq D_{i0}$ and $-x_i \leq D_{0i}$, and for all $1 \leq i,j \leq k$, $x_i - x_j \leq D_{ij}$. Every DBM represents a clock zone, and every clock zone is represented by some DBM.

EXAMPLE 4.5 *Consider the clock zone defined by the following constraints:*

$$[0 \leq x_1 < 2] \;\wedge\; [0 < x_2 < 1] \;\wedge\; [x_1 - x_2 \geq 0]$$

It can be represented by the following difference-bound matrix:

$$\begin{array}{cc} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \left(\begin{array}{ccc} \infty & (0,1) & (0,0) \\ (2,0) & \infty & \infty \\ (1,0) & (0,1) & \infty \end{array}\right) \end{array}$$

■ 4.5

DBMs can be also easily interpreted and depicted as weighted graphs. For instance the system

$$[-x_1 \leq 3] \;\wedge\; [x_3 \leq 5] \;\wedge\; [x_3 - x_1 \leq 2] \;\wedge\; [x_2 - x_3 \leq 2] \;\wedge\; [x_2 - x_1 \leq 10] \;\wedge\; [x_1 - x_2 \leq 4]$$

appears as a directed weighted graph in Figure 4.6 (a). In general the same set of clock assignments may be described by several constraint systems (and hence graphs, for instance all the graphs of Figure 4.6 have the same solution set). To test inclusion between two constraint systems lets say D and D' it is advantageous if they are closed under entailment in the sense that no constraint of D can be strengthened without reducing the solution space. This is usually done by applying

Algorithm 4.1: Reachability analysis of a zone automaton

input : $\mathcal{A}$, a timed automaton.
output : FALSE if failure states are reachable, TRUE otherwise.

Passed $\leftarrow \emptyset$
Wait $\leftarrow \{\langle l_0, D_0 \rangle\}$
while Wait $\neq \emptyset$ **do**
    take $\langle l, D \rangle$ from Wait
    if $l = l_f$ *and* $D \cap \phi_f \neq \emptyset$ **then return** FALSE
    if $\forall \langle l, D' \rangle \in$ Passed, $D \not\subseteq D'$ **then**
        Passed $\leftarrow$ Passed $\cup \{\langle l, D \rangle\}$
        forall $\langle l', D' \rangle$ *such that* $\langle l', D' \rangle$ *is a successor of* $\langle l, D \rangle$ **do**
            Wait $\leftarrow$ Wait $\cup \{\langle l', D' \rangle\}$
return TRUE

an all-shortest-path algorithm (typically the Floyd-Warshall) to produce a canonical closed form of the DBM, which is usually stored as a dense matrix, although it often turns out that most of these bounds are redundant. Figure 4.6 (b) shows the canonical closed form of Figure 4.6 (a) and contains many arcs that are redundant. To save space, [LLPY03] proposed an algorithm that produces a canonical reduced form of a DBM that contains only the minimal number of constraints (see Figure 4.6 (c)).

The system of constraints has no solution if, and only if, a negative cycle exists in the graph. It is easy to detect such a case when computing the canonical closed form because in this situation at least one of the entries in the diagonal of the matrix becomes negative. This has some relevance in our case because the timing consistency checking of a trace can be represented with a DBM.

A good source of information on the construction of difference-bound matrices can be found in [CGP00].

Standard reachability analysis of the zone automaton

The first algorithms developped to obtain the reachable state space of a zone automaton were based in an explicit search of zone states. As a representative, Algorithm 4.1 shows the core of the verification engine UPPAAL as presented in [BY03]. Each pair of location and DBM forms a zone state and one can generate the successor states by applying the delay and action transitions of the timed automaton under analysis.

The algorithm starts with initial state $\langle l_0, D_0 \rangle$, where l_0 represents the initial location and D_0 enforces that all clocks equal 0. While there are zones to process, it takes one of them and computes its successors. If at some point a zone in which a failure is defined is found (represented by zone $\langle l_f, \phi_f \rangle$) then a timing consistent failure trace exists. Otherwise, if there is no pending zone to process, then all zones have been generated and the system is correct.

The problem of this approach is that every zone must be stored explicitely. Usually locations are stored in a hash table, and for each location in the table there is a list of the DBMs of that particular location. This makes it only possible to verify systems in which the number of visited locations is not very large (up to the memory resources of the available hardware). Systems that do not fulfill this condition require strategies that allow to reduce the number of states (like abstractions).

Another source of inefficiency comes from the fact of having two separate data structures: one for the locations and one for the timing information. It was soon recognized that a data structure containing both informations would yield better results. Two main approaches exist to be able to use a single data structure:

- discretize the timed automaton and then use any of the data structures available for untimed verification (this usually involves encoding the integer-valued clocks into sets of Boolean variables), or

- use a new data structure capable of containing both the Boolean values of the location as well as the clock differences of the DBM.

Discretization has been investigated on different models by several researchers. As we commented in §4.2.1, the reachability problem for asynchronous circuits with bounded delays cannot be satisfactorily solved using the discrete-time model [BS91]. However, in [BS95] the same authors define as an open problem if there is a discretization of time of a digital circuit with real-valued delays inside integer-bounded intervals, such that the qualitative behavior of the circuit is preserved.

The questions was answered in [AMP98]: every acyclic circuit can be discretized, as well as all circuits with closed delay intervals[1], however there are cyclic circuits (which contain open intervals) for whom there is no possible discretization.

For timed automata the first results were published in [GV94]. The authors give two discretizations that can be applied to any timed automaton. However their technique requires to distort the passage of time, as noted in [AMP98]. Precisely in this latter paper it is proved that any untimed property can be proved in a closed automaton (*i.e.* a timed automaton which contains only non-strict inequalities in clock conditions) using discrete semantics. For non-closed automata, only a conservative verification, which can provide false negatives, is possible.

Timed transition system are all discretizable as shown in [HMP92b].

Once the system can work with discrete-time, a number of available data structures can be used to obtain the reachable state space, as BDDs [BLN03], *Numerical Decision Diagrams* (NDDs) [ABK$^+$97] or *Interval Decision Diagrams* (IDDs) [Str98]. Some of these data structures are very sensitive to the size of the clock values they have to store (BDDs and NDDs) while some others are not so directly affected. However all of them are variations of decision diagrams and greatly depend on the variable ordering to achieve reasonable space requirements.

Besides discretization, the other possible strategy is to use a single data structure to contain both the state and timing information. Several new data structures have been proposed to achieve this objective: *Clock Difference Diagrams* (CDDs) [BLP$^+$99], *Relaxed CDDs* [PS07a], *Clock Restriction Diagrams* [Wan04]. Typically, these structures are more resilient to increasing the values of the delay bounds (as long as this change does not affect the behavior) but still variable order, which includes the place in which clocks are tested when performing operations on the structure, becomes of capital importance in its performance.

Variable order is a common problem in untimed verification with decision diagrams, as well as the existence of problems for which there is no polynomical representation. To circumvent these limitations SAT solving techniques were introduced for untimed verification. The idea is to create a formula in CNF such that it is only satisfied if the system is not correct up to a certain depth (bounded model checking). This is convenient because the size of the formula is polynomical with

[1] This last result is only valid under the assumption that behavior preservation does not need to mantain the same order ($<$ or $\leq$) between timestamp in a trace under discrete-time and continuous-time. Only “logical” order is mantained and two event with strictly ordered timestamps in the continuous domain might be assigned the same timestamp in the discrete domain. This preservation notion is the one used in [HMP92b].

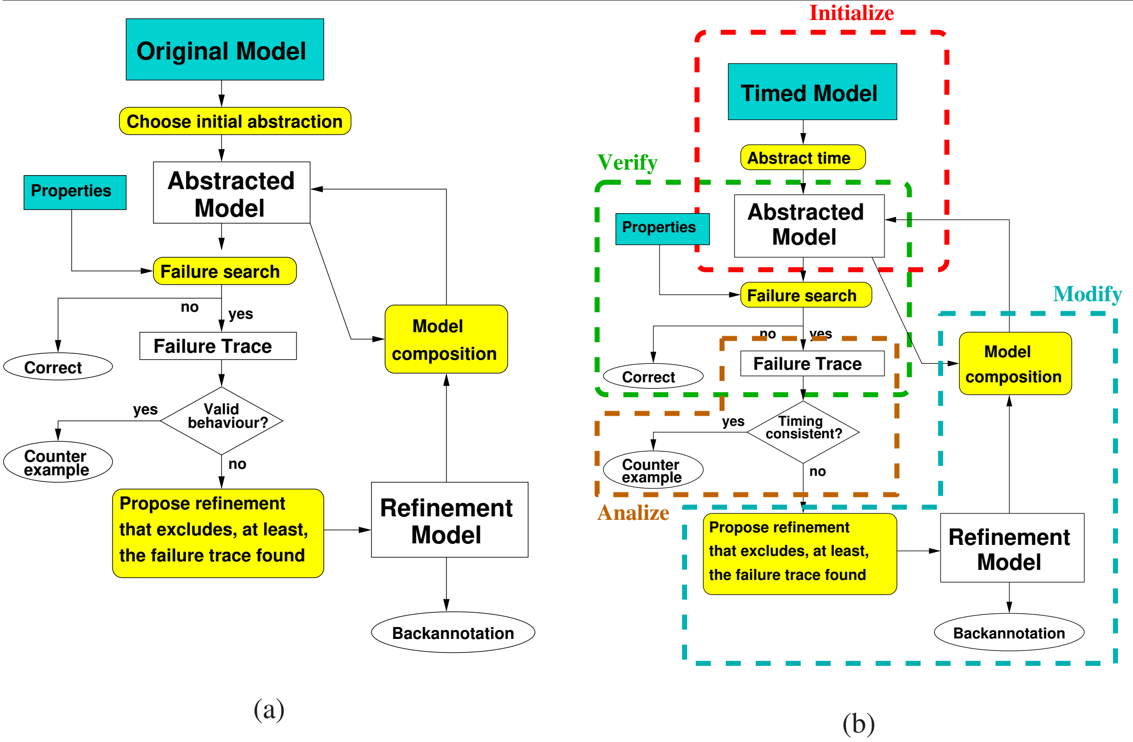


Figure 4.7. (a) General iterative verification flow. (b) Specialized iterative verification flow using time abstraction divided in four phases as described by Balarin [Bal95].

respect to the system description and the depth at which an error is searched. Later improvements allowed to perform unbounded verification using the same technique.

Trying to achieve the same benefits in timed verification, it has been studied the application of SAT-solving techniques to timed automata [NMA$^+$02, SB03, WZP03, ACGM04, PS07b]. Approaches can be classified into two families: one is formed by all methods that use a discretization of time and a standard SAT-solver, and the other is formed by methods that incorporate real-time into the formulas that are treated with tailored SAT-solvers, which include some type of decision procedure to deal with propositions contaning real-valued variables.

Although there are some promising results, these techniques have to face that translating asynchronous systems to “nice” SAT formulas is more complex than for synchronous circuits. Moreover the depth of concurrent systems is usually larger due to the interleaving of the events, which means that SAT solvers have to deal with bigger formulas. For all these reasons there are no clear winners in the experiments appearing in the literature when they are compared to non-SAT techniques we have previously discussed.

4.5 Iterative refinement methods

The verification methodology we will propose is iterative, thus former approaches using the same strategy deserve a comment. In Figure 4.7 (a) we can see a general version of an iterative refinement verification flow. First of all we have a mathematical formalism used to describe the system, named as *original model* in the figure. It might be the case that the model contains much more information than the one needed to show the correctness of the system, and that, due to this excess in the detail, the size of the model makes the verification problem intractable. So the iterative approach makes a first coarse abstraction in which only the most "relevant" facts of the model are retained. The result of this abstraction step is an *abstracted model*.

Once the abstracted model is available, we analyze it (typically with a reachability analysis) to check if some violation of the specification is encountered. If no trace is found the system is correct and no further processing must be done. On the other hand, if a trace to a failure is found we must check if it has a corresponding trace in the original model, that is, if we have encountered a real counterexample or just a failure trace induced by the abstraction.

If the failure trace corresponds to a valid behavior in the original model, then we generate that corresponding trace which is returned as counterexample, proving that the system does not satisfy the specification. Otherwise, we know that we have a *fake* trace, so in order to have some progress in our verification we must somehow modify the abstracted model to avoid reproducing, at least, this particular fake trace. If we do not enforce such condition, it is easy to see that the method can go into an infinite loop *e.g.* if failure search is done in a deterministic way the same fake trace will be encountered again and again. Typically to remove this trace from the behaviors of the abstracted model some analysis on the trace must be performed (*i.e.* why this trace is a fake trace?) which takes into account all the abstracted information and from this analysis a *refinement model* is built. Finally this refinement model is composed with the abstracted model to produce a more accurate model.

In Figure 4.7 (b) the general version of an iterative refinement verification flow has been specialized to use a time abstraction. The key difference with Figure 4.7 (a) is that initial abstraction consists in removing the timing information from the model (so in fact we do check if the system satisfies the specification regardless of the delays of the events) and that the *valid behavior* checking is now a *timing consistency* checking (*i.e.* the found failure trace is consistent when the delays in the original model are used?). In the figure we show also the division in four steps proposed in [Bal95] for the iterative methods.

The use of time abstractions in iterative refinement has been investigated by few researchers. Now we review their contributions.

Modification of the delays

In [AIKY92] they proposed an iterative verification flow which follows Figure 4.7 (b). In their approach a parallel composition of ω-automata is used as *original model* from which a region automaton is built. When a timing inconsistent failure trace θ is found, it is added to a set of failure traces Θ.

Then they identify the minimal set of processes whose delays are needed to still prove that all $\theta \in \Theta$ are timing inconsistent. This problem is NP-hard and they proceed by exhaustively considering subsets of processes in order of increasing size. Having identified such a minimal set of processes I they find out the minimal subset of delays (out of all the delays of processes in I) that still proves timing inconsistency of all traces in Θ. Again this is done by exhaustive search. Clearly this minimal subset Δ_Θ will consist of only the delays corresponding to the edges involved in negative cycles in the graph theoretical approach to the timing consistency checking of traces in Θ.

Finally, after Δ_Θ has been identified, all the delay constraints are scaled down using their greatest common divisor (gcd) as this does not affect the language accepted by the region automaton but might have a big impact on its representation size. In order to achieve a greater reduction they try to maximize the gcd by *relaxing* the delay bounds so as to still falsify all the traces in Θ. With these new delays a new region automaton is built and the process starts again.

To summarize, their technique consists in progressively adjusting the delays of the events from the unbounded interval $[0, \infty)$ up to their original value $[\alpha, \beta]$ as long as a timing inconsistent trace is present in the model.

Composition with a safety automaton

In [WT94] a more general framework is used in which both overapproximations and underapproximations are iteratively computed until the system is shown correct by the overapproximation or a failure trace is found by the underapproximation. This framework is more general than the one in Figure 4.7 (b) because approximations are not restricted to the timing information but also to the states of the system. The general version of the method in Figure 4.7 (a) could be used instead, but again their approach is more general in the sense that they also use underapproximations while the verification flow of Figure 4.7 (a) relies only on overapproximations. Underapproximations describe a subset of already valid traces, thus if a failure trace is found in an underapproximation then it is automatically a valid counterexample.

However the main principle is the same: the behavior of the system is abstracted coarsely in the beginning and, until no failure exists or a counterexample is found, the algorithm progressively refines the partitions used for the abstraction. In [WT94] a non-Zeno timed safety automata (TSA) is used as *original model* while the same TSA enriched with an *approximation structure* (which defines the partitions used in abstractions) is the *abstracted model* of Figure 4.7 (a).

Composition with guard automata

Balarin et al proposed two related iterative verification methods. First one [BSV93] is similar to [AIKY92] which we have commented before, but in composition they add subprocesses, called *guard automata*, that monitor difference constraints between clocks in an approximate manner (as they keep timer values approximatively up to the closest integer). This approach still suffers from *stiffness*, that is, representation sizes depends heavily on delay values. The stiffness problem can cause that, even if a change in a delay value does not change the *reasons* why the system can be shown correct, it is perfectly possible that in one scenario the automaton is small while in the other can in no way fit into memory.

To solve this problem in [Bal95] they developed a new algorithm. When a timing inconsistent trace is found it is analyzed and a negative weighted loop is returned (overconstrained loop). Then all the clocks that appear in the edges of the overconstrained loop are transitively related by means of guard automata. Instead of trying to keep an approximate value of the difference between the clocks of the guard automaton they just use the fact that in some states (when clocks are reset) they can know if the difference constraint they want to enforce is satisfied or not and that this knowledge is still valid for all successor states until a new reset is performed, so after composition they can safely eliminate arcs that are known to violate some of the difference constraints.

The approach in [Bal95] allowed the use of implicit state enumeration techniques (although they do not give details on how they handle timing constraints) and does not suffer from the stiffness problem. Although this was previously unrealized because of the difference in formalisms, it is the work more closely related with [Peñ03], as it is the information on the structure of the original automaton (*i.e.* when clocks are reset) which is used instead of timer values which do always enforce to build the region automaton.

The main problem with this approach is that the algorithm proceeds “blindly” hence there is no criterion of which of the negative cycles in the original timing inconsistent trace is better to choose. This can make the algorithm to apply many different compositions that have a very small impact but increase the representation size. This is not a minor question and forced them to introduce *hints* (*i.e.* we tell the algorithm at the beginning that we want to track the time differences between a given set of events) to be able to verify their bigger examples. However this is not a very promising approach because it depends on having already the solution to the problem (or a partial solution) to solve the problem itself!

In [Bal95] this option was defended by arguing that designers have already an idea of why their design should work, but still we believe it is a practice that verification methods should avoid. Note that the original work of [Peñ03], which will be our starting point, can also suffer from this problem of performing several compositions that contribute very few to the verification task while they increase enormously the size of the representation. This sometimes can simply cause a delay in the result but in other cases can be the difference between success or failure. In contrast with [Bal95], in [Peñ03] a heuristic is used to select which subset of events must be analyzed, hence usually results are more stable so no hints mechanism was introduced.

4.6 Conclusions

The chapter has reviewed the most relevant approaches for the modeling, specification and verification of timed systems. We have focused on timed automata and available verification methodologies for invariant untimed properties.

A number of verification tools have resulted from all these approaches, but all of them have their own weak spots. This should come at no surprise given the difficulty of the problem to solve. In this context new methodologies are always wellcomed because they usually propose new tradeoffs, which can be very successful in some settings (and awfull in many others) but increase the amount of tools available to address the verification of timed systems.

In next chapters we present a different verification methodology that might improve the verification of untimed invariants. It is an iterative refinement method that uses timed transition systems as the underlying formalism to model timed systems under the continuous-time paradigm. Instead of computing the exact timed state space, the relative timing paradigm is used to abstract exact time information from the representation. Hence, LzTSs are used[2], which represent the ordering relations between events in the timed domain, by explicitly distinguishing between their enabling and their actual firing conditions.

The approach is applicable to systems modeled by timed transition systems

[2]In terms of Figure 4.7, the *Original model* is a TTS in our case and a LzTS acts as the *Model*.

Chapter 5

INTRODUCTION TO RELATIVE TIMING VERIFICATION

Summary

The verification of timed systems is a hard problem. One of the main difficulties is to keep track of the timing information (*i.e.* the value of the clocks) associated to each state.

In models where there is a strong relationship between the value of the clocks and the firing of the events, like TTSs, we have an alternative way to keep track of the timing information. Instead of using explicit clocks, we can use the information of which events have fired or are enabled to determine if a particular set of traces exists in the timed domain.

In this chapter we present the fundamentals of the verification methodology introduced in [Peñ03], which is an iterative refinement method based in the previous observation. Given the complexity of the subject, in which several different mathematical entities are involved, the complete formal explanation of the basic methodology spans from this chapter to Chapter 9.

In this first chapter we overview the methodology using an example and we present the basic theoretical tools used in the rest of this work. In particular, we introduce trace semantics and the concept of enabling compatibility between traces. If a trace is enabling compatible with another, then the timing inconsistency of the latter proves the inconsistency of the former, thus allowing to apply the result of a timing analysis on a single trace to a set of traces.

5.1 Introduction

The verification of concurrent systems typically suffers from the well known state explosion problem. In systems with a finite number of states, the problem is often alleviated by using symbolic techniques to represent the reachable states. This is also combined with partial order techniques or abstractions that reduce the complexity of the models. However, when time is an essential dimension in the verification problem, complexity is drastically affected. Since the correctness of the system depends on the actual values of event delays and not only on its functional behavior, the verification becomes unmanageable even for moderate-size systems. More precisely, computing the reachability space of a timed system is proved to be a PSPACE-complete problem [AD94], and demonstrated to be highly complex in several practical contexts. Although several techniques have been devised to alleviate such complexity (see Chapter 4), the size of the untimed state space is still the major bottleneck for the analysis of highly concurrent systems.

This chapter describes an actualization of the approach presented in [Peñ03] that extends the applicability of the conventional methods based on symbolic reachability analysis, to the verification of safety properties in timed systems. The approach is based on two fundamental facts:

- The observation that the set of runs of a transition system can be covered by a set of event structures [NPW81]. This reduces the verification problem to performing timing analysis over small sets of events from which timing constraints that prove the correctness or incorrectness of a system can be derived; and the incorporation of such constraints into the system along an incremental refinement process.
- The use of *relative timing* [SGR99] allows to represent the timed domain of a system in an efficient way. When considering precise delay bounds in timed systems, the complexity blow-up often makes the analysis an intractable problem, even for small systems. Instead, relative timing considers the *effect* of delays in a system in terms of relative ordering of events (*e.g.* a happens before b).

The verification approach can be briefly summarized as follows. Rather than calculating the exact timed state space, an *off-line* timing analysis is performed on a set of event structures that covers the runs leading to system failures. The timing analysis is efficiently performed by using McMillan and Dill's algorithm [MD92] over a small set of events. The resulting timing constraints are incorporated to the system in the form of relative timing information along a series of iterative refinements of the original untimed state space. If some of the runs leading to failure situations cannot be proved to be timing-inconsistent, then the system is incorrect and the failure run is a counterexample.

The approach presented here, not only verifies the correctness of the system with respect to a set of given safety properties, but also provides as back-annotation a set of timing constraints sufficient to prove such correctness. This information is crucial in frameworks in which synthesis and verification are iteratively invoked to design systems that must meet functional and non-functional constraints.

We want to remark that the application of the method for the verification of untimed systems does not involve any additional overhead with respect to the conventional symbolic methods (*e.g.* [BCM$^+$92]).

5.1.1 Relative Timing

So far we have talked about the idea of relative timing but no illustrative example has been provided that can help to understand some of its benefits, specially in areas other than the verification of

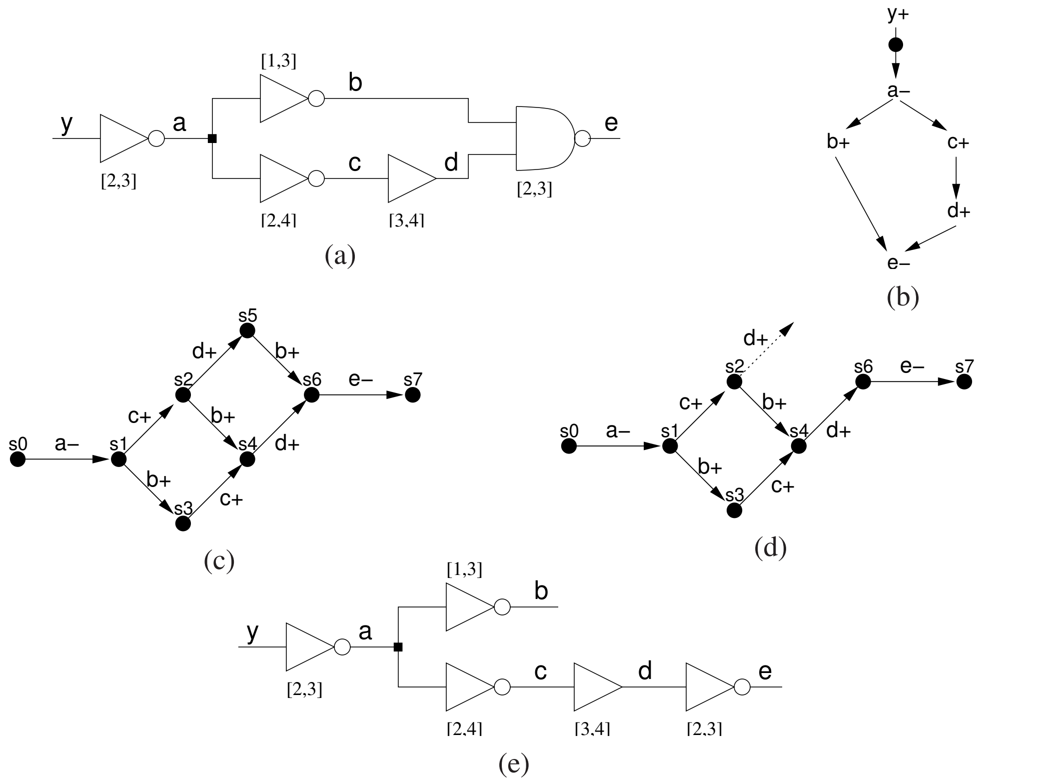


Figure 5.1. Relative timing in the synthesis of circuits: (a) timed circuit, (b) portion of the STG, (c) corresponding TS for the untimed behavior, (d) corresponding classical LzTS and (e) optimized circuit.

timed systems. In this section we reproduce partially an example from [CKK$^+$02] where relative timing is used to improve the synthesis of asynchronous control circuits. The synthesis process takes relative timing information into account thus allowing the generation of smaller and faster circuits.

EXAMPLE 5.1 *Consider the asynchronous circuit in Figure 5.1 (a). The delays of the gates are represented by intervals of the form $[d, D]$, which indicate that the output of the gate driving a given signal* x *will change $\delta(\mathsf{x})$ time units after the gate became enabled, with $d \leq \delta(\mathsf{x}) \leq D$. That is, the firing time is bounded by the given delay interval.*

After the occurrence of a rising transition of signal y*, the behavior represented by the STGof Figure 5.1 (b) is enabled to happen. The rising transition of signal* b *appears to be concurrent with the rising transition of signals* c *and* d*. The corresponding underlying TS is depicted in (c).*

If the actual delays of the gates driving these signals are considered, it is easy to realize that b+ *will always happen before* d+*. Clearly, the earliest time for* d+ *to occur is 5 time units after* a−*, whereas the latest time for* b+ *to occur is only 3 time units after* a−*. This observation can be translated into the fact that state* s_5 *of the untimed TS will never be reached (see the resulting LzTS in (d)).*

Provided that b+ *will always happen before* d+*, the causality relation* b+ $\longrightarrow$ e− *is always guaranteed by the actual delays and the causality relation* d+ $\longrightarrow$ e−*. Thus, a potential optimization of the circuit may consider the relative timing constraint between* b+ *and* d+*, and ignore the explicit causality relation* b+ $\longrightarrow$ e−*, which leads to the optimized circuit of Figure 5.1 (e).* ■ 5.1

Along the process described in the example, neither the exact times at which each event occurs nor the exact times at which the states are reached need to be determined. Instead the reasoning is done in terms of "*which event occurs before each other*". This type of reasoning is particularly useful in the early stages of the design flow, when the exact timed behavior of a system is difficult to determine and precise delay constraints are hard to satisfy. Conversely, it is much simpler to deal with constraints that just state which event must be faster than other, without taking care of the exact delay slack between them. Moreover, it is much easier to keep these type of constraints satisfiable along the successive design steps.

Using similar ideas, Intel's Strategic CAD Lab designed an asynchronous instruction length decoder for the x86 instruction set [RSG+99]. The circuit exhibits a promising increase in performance with respect to its synchronous counterpart, thanks to the optimizations achieved using the relative timing information. The techniques pioneered by this design have been evolved and formalized using the LzTS model [CKK+98] and automated in the logic synthesis tool PETRIFY [CKK+97].

Finally remark, that although this section has referred to asynchronous circuits, they are just an example of application. The relative timing paradigm is applicable to the design, synthesis and verification of timed systems in general.

5.2 Overview

This section provides an overview of the verification approach with relative timing presented in this chapter. For that purpose, an simple illustrative example is developed.

This work develops a formal approach to verify that a system with certain timing constraints satisfies a given safety property P. The system is modeled by means of a timed transition system, A, composed by an underlying transition system, A^-, and two functions, δ^l and δ^u, which associate minimal and maximal delays, respectively, to each event of the system. A given sequence of events of a TTS is said to be timing-consistent if it is possible to assign increasing time values to all the events such that their firing times are within the allowed bounds. The modeling formalism of timed transition systems was introduced in §2.2.

The verification problem is posed in terms of the following language inclusion test: $\mathcal{L}(A) \subseteq \mathcal{L}(P)$ [Gup92], where $\mathcal{L}(A)$ corresponds to the set of all possible behaviors of A, and $\mathcal{L}(P)$ is the set of all possible behaviors satisfying property P. The approach consists in building successive conservative approximations of $\mathcal{L}(A)$ starting from $\mathcal{L}(A^-)$, by adding relative timing constraints [SGR99] in an iterative manner. We start from the TS $A_0 = A^-$, *i.e.* the original system without timing constraints, and try to prove the inclusion $\mathcal{L}(A_0) \subseteq \mathcal{L}(P)$ by applying well-known *symbolic model checking* techniques [CMB90, BCM+92]. If the inclusion holds, then $\mathcal{L}(A) \subseteq \mathcal{L}(A_0) \subseteq \mathcal{L}(P)$, which indicates that A satisfies P without any timing assumption. The verification succeeds.

If P is not satisfied in some state, a run ρ that leads to a failure is generated. If the run is timing-consistent, then the system is incorrect, *i.e.* violates the required property. However, if the run is not timing-consistent, it can be used to refine the untimed state space and remove other timing-inconsistent runs leading to failure states. To do this, a suffix ρ' of the run ρ is taken and an event structure that covers ρ' is built. Timing analysis on the event structure is performed by using the polynomial algorithm for acyclic marked graphs in [MD92]. A set of relative timing constraints are derived that prove the timing-inconsistency of ρ'.

The state space of the (timed) event structure is composed with the untimed abstraction of the system A_0, in such a way that at least the failure run ρ is removed and no timing-consistent run is removed. A series of successive approximations A_i of A are constructed iteratively, with

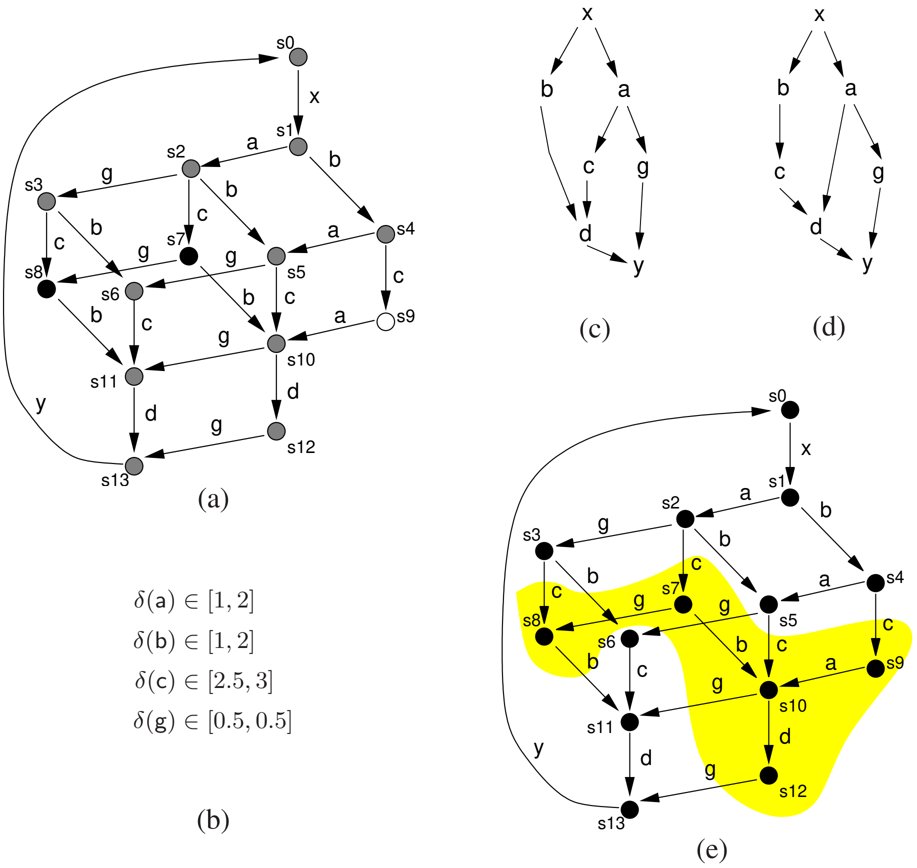


Figure 5.2. Example of verification with relative timing: (a,b) TTS. (c,d) Event structures covering the runs starting from s_0. (e) Timed state space (shaded states are unreachable).

containment $\mathcal{L}(A) \subseteq \mathcal{L}(A_i)$ and monotonic convergence, $\mathcal{L}(A_{i+1}) \subseteq \mathcal{L}(A_i)$. At every step $\mathcal{L}(A_i) \subseteq \mathcal{L}(P)$ is checked. Verification stops successfully if the inclusion holds, or fails if a counterexample run is found.

Iterative approaches for the verification of real-time systems have been also presented in [AK95, BSV95]. The novelty of the approach presented in [PCKP00] was that the underlying timed model used was TTS (the previous methods focused on timed automata) and the use of event structures to perform efficient off-line timing analysis, and to incorporate the resulting timing information in the form of relative timing constraints.

EXAMPLE 5.2 *Figure 5.2 depicts the TTS modeling a simple timed system. Subfigures (a) and (b) show, respectively, the underlying (untimed) TS and the delay intervals of events* a, b, c *and* g*. The delay interval for the rest of events is assumed to be unbounded,* i.e. $[0, \infty)$*. Subfigure (e) depicts the state space of the system, when the delays are taken into account the shadowed states are not reachable. A crucial observation is that all runs in the TS of Figure 5.2 (a) that start and end at state* s_0 *can be covered by the two event structures depicted in subfigures (c) and (d): black states are covered by the event structure (c), white states are covered by the event structure (d) and gray states are covered by both event structures. Thanks to this fact the later verification process can be carried out with just a couple of refinements.*

Assume that the property to be verified indicates that event g *must always precede event* d *in any possible run after having visited state* s_0*. The property holds in the timed state space since no*

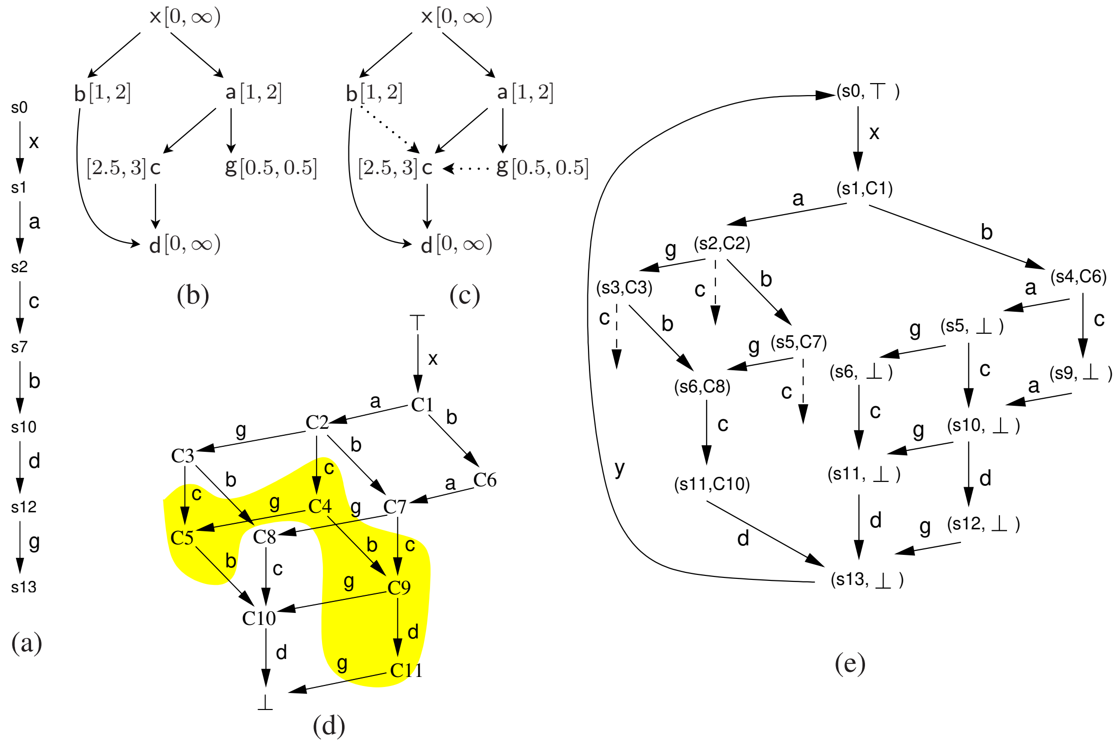


Figure 5.3. Example of verification with relative timing (first iteration): (a) A failure run. (b) Its corresponding event structure. (c) The event structure annotated with timing arcs. (d) State space of the event structure (shaded states are unreachable). (e) Classical LzTS obtained after composition.

state where d *can fire before* g *is reachable. Conversely, the property does not hold in the untimed state space, for example in state* s_{10} *where* d *can fire before* g.

The analysis starts by generating a run that leads to the failure situation, for example a run from s_0 *to* s_{10} *followed by the firing of* d *before* g *can be generated (Figure 5.3 (a)). Next, an event structure that captures the causality relations of the events in the run is derived (Figure 5.3 (b)). Notice that, in the event structure,* c *is only triggered by* a *but not triggered by* b, *as one may expect by looking at the transition system. This is due to the fact that the event structure only contains those causality relations derived from the run. In the failure run under analysis,* c *is not enabled in* s_1 *and is enabled after having fired* a *from* s_1. *Thus* a *triggers* c, *while* b *is concurrent to it.*

By timing analysis over the event structure, we find that b *and* g *always precede* c. *These timing relations are shown as the dotted arcs incorporated to the event structure in subfigure (c). Such timing analysis is only valid for the causal relations expressed in the event structure, but it is not valid, for example, in the case when* b *triggers* c. *Figure 5.3 (d) depicts the state space of the timed event structure, where the shadowed states are not reachable due to the timing relations. Namely, event* c *is prevented to fire in some states, where its firing would be inconsistent with the timing analysis.*

Finally, all this information is incorporated into the system (Figure 5.3 (e)) by composing the original system and the event structure. An event structure being derived from a particular run gives only partial behaviors of the original system. When the behaviors of the system and the event structure mismatch, the special symbol $\perp$ *is used. Some states in the composed system are split into two instances depending on whether they are reached by runs matching (*enabling compatible*)*

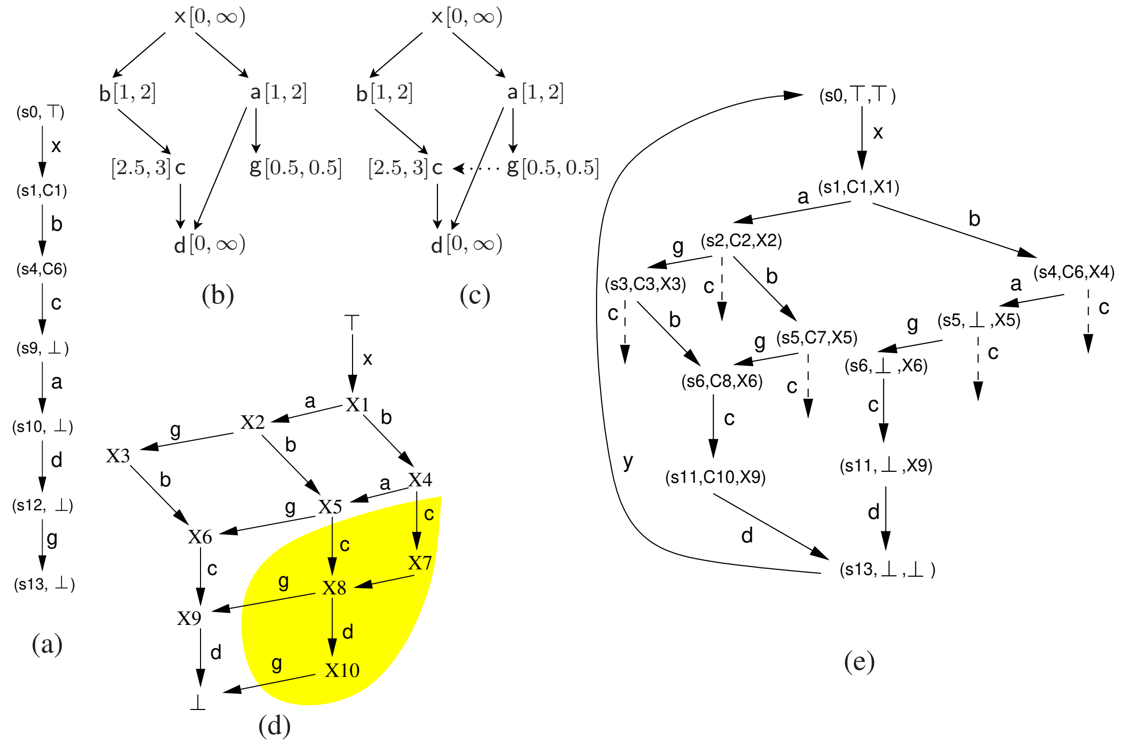


Figure 5.4. Example of verification with relative timing (second iteration): (a) A failure run. (b) Its corresponding event structure. (c) The event structure annotated with timing arcs. (d) State space of the event structure (shaded states are unreachable). (e) Classical LzTS obtained after composition.

the event structure or not (see states s_5*,* s_6*,* s_{11} *and* s_{13}*). Figure 5.3 (e) shows the resulting system. Notice that the set of runs is smaller than that of the original system, but larger than that of the actual state space when the delays are considered (Figure 5.2 (e)), and that only timing inconsistent runs have been removed.*

This completes the first refinement of the untimed state space taken as starting point. This step has removed some of the failure runs but not all of them. For example, if the new state $(s_{10}, \perp)$ *is reached,* d *can fire before* g *and this contradicts the property under verification. Figure 5.4 summarizes one more refinement. In the resulting system all the failure runs have been removed, which proves that the system satisfies the property. Although it is not generally true, in this case the final state space contains exactly the same runs than the actual state space shown in Figure 5.2 (e).*

■ 5.2

Several objects and notions have been mentioned along the previous example, such as event structures, enabling compatibility, etc. These and other notions, as well as the theoretical aspects of the verification with relative timing are presented in detail in the following sections and chapters. In this chapter we introduce traces and trace-related concepts that constitute the theoretical basis of the proposed methodology.

5.3 Trace semantics

As we have discussed in the previous section, the verification problem is posed in terms of the language generated by the system under verification and the language of all behaviors satisfying a given property. The verification process involves lazy transition systems and event structures (see

Chapter 7) as major models. The process consists in an iterative incremental refinement of the system under verification with the timing information derived from the event structures.

A common semantics that unifies the models involved in the verification process can be defined in terms of *traces*. Based on traces, we will derive several notions that formalize our refinement approach for verification.

5.3.1 Traces and languages

We extend the usual notion of trace [Maz88] by associating the set of enabled events to the firing of each event in a sequence of event firings. Thus, each element of the trace keeps track of which events are enabled and which event fires at each step.

DEFINITION 5.1 (TRACE)

Let Σ *be an alphabet of events. A trace* $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots$ *is a sequence such that* $\forall i \geq 1 \;:\; E_i \subseteq \Sigma$ *and* $\mathsf{e}_i \in E_i$*, where* E_i *denotes the set of events enabled when* e_i *fires. We will use the term* enabling *to refer to any set of enabled events.* ■ 5.1

This definition of trace allows repeated events. However in some cases we will need all events to be distinct. In these situations we will use a *distinct trace* that is a normal trace in which events have been renamed by adding an occurrence number to them.

DEFINITION 5.2 (DISTINCT TRACE)

Let $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots$ *be a trace over an alphabet of events* Σ*. The* distinct trace *of* θ *is a trace* $\theta' = E'_1 \xrightarrow{\mathsf{e}'_1} E'_2 \xrightarrow{\mathsf{e}'_2} \cdots$ *over an alphabet of events* $\Sigma' \subseteq \Sigma \times \mathbb{N}$ *such that:*

$$\begin{array}{rcl} E'_1 & = & \{\langle \mathsf{e}, 1\rangle \mid \mathsf{e} \in E_1\} \\ E'_i & = & \{\langle \mathsf{e}, j+1\rangle \mid \mathsf{e} \in E_i \wedge \mathsf{e}_{i-1} = \mathsf{e} \wedge \langle \mathsf{e}, j\rangle \in E'_{i-1}\} \cup \\ & & \{\langle \mathsf{e}, j+1\rangle \mid \mathsf{e} \in E_i \setminus E_{i-1} \wedge \exists k < i : \big(\langle \mathsf{e}, j\rangle \in E'_k \wedge \nexists m < i : \big(\langle \mathsf{e}, z\rangle \in E'_m \wedge z > j\big)\big)\} \cup \\ & & \{\langle \mathsf{e}, j\rangle \mid \mathsf{e} \in E_i \wedge \mathsf{e}_{i-1} \neq \mathsf{e} \wedge \langle \mathsf{e}, j\rangle \in E'_{i-1}\} \cup \\ & & \{\langle \mathsf{e}, 1\rangle \mid \mathsf{e} \in E_i \wedge \nexists j < i : \mathsf{e} \in E_j\} \end{array}$$

with $\mathsf{e}'_i = \langle \mathsf{e}_i, j\rangle$ *such that* $\langle \mathsf{e}_i, j\rangle \in E'_i$ *and* $\Sigma' = \bigcup_{\forall i} E'_i$*.* ■ 5.2

Although the formal definition is not very simple, the concept is trivial. As an example in Figure 5.5 a trace and its distinct trace are shown. It is easy to see that you can always obtain a distinct trace from a trace (by adding occurrence numbers to events) and vice versa (by simply removing them).

In this work, which focuses on the timing consistency of traces, we will use yet another *flavor* of trace which allows a more uniform treatment of the events of the trace in timing related problems.

DEFINITION 5.3 (STANDARDIZED TRACE)

Let $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots$ *be a trace over an alphabet of events* Σ*. The* standardized trace *of* θ *is a trace* $\theta' = E_0 \xrightarrow{\mathsf{e}_0} E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots$ *over an alphabet of events* $\Sigma' = \Sigma \cup \{\epsilon\}$ *where* $\mathsf{e}_0 = \epsilon$*,* $E_0 = \{\epsilon\}$ *and* $\epsilon \notin \Sigma$*.* ■ 5.3

A standardized trace is obtained from a trace by simply prepending the firing of a single special event ϵ. An example of a trace and its standardized trace can be found in Figure 5.6.

Some very useful relations between the events of a trace are given in Definition 5.4

DEFINITION 5.4 (TRIGGERING/DISABLING/FIRING ORDER IN A TRACE)

Let $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots$ *be a distinct trace. We define the trigger relation of* θ *as*

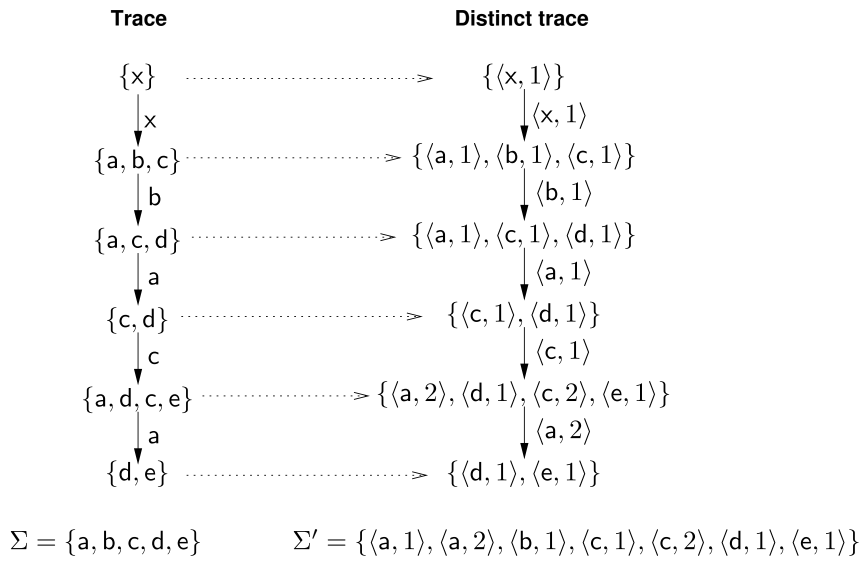


Figure 5.5. A trace with repeated events and its corresponding distinct trace.

$\mathrm{triggerRel}(\theta) = \{(\mathsf{a},\mathsf{b}) \mid \mathsf{a} = \mathsf{e}_i \wedge \mathsf{b} \in E_{i+1} \setminus E_i\}$. *If* $(\mathsf{a},\mathsf{b}) \in \mathrm{triggerRel}(\theta)$ *we will say that* a triggers b *in* θ *or that* b is triggered by a *in* θ. *In this latter case we will also write* $\mathrm{trigger}(\mathsf{b},\theta) = \mathsf{a}$ *(or just* $\mathrm{trigger}(\mathsf{b}) = \mathsf{a}$ *when* θ *is clear from context) as* $\mathrm{triggerRel}^{-1}$ *is a function* i.e. *an event can only have a single trigger in a distinct trace because it only becomes enabled once. The* triggerRel *relation is a pseudo-order (in particular a tree).*

Relation triggerRel *can be used to establish a strict partial order, noted as* $<_{\mathsf{tr}}$*, between the events of a trace by simply* $<_{\mathsf{tr}} \stackrel{\mathrm{def}}{=} \mathrm{triggerRel}^*$.

The disabling relation of θ *is defined as* $\mathrm{disablingRel}(\theta) = \{(\mathsf{a},\mathsf{b}) \mid \mathsf{a} = \mathsf{e}_i \wedge \mathsf{b} \in E_i \setminus E_{i+1} \wedge \mathsf{b} \neq \mathsf{e}_i\}$. *If* $(\mathsf{a},\mathsf{b}) \in \mathrm{disablingRel}(\theta)$ *we will say that* a disables b *in* θ *or that* b is disabled by a *in* θ *In this latter case we will also write* $\mathrm{disabler}(\mathsf{b},\theta) = \mathsf{a}$ *(or just* $\mathrm{disabler}(\mathsf{b}) = \mathsf{a}$ *when* θ *is clear from context) as* $\mathrm{disablingRel}^{-1}$ *is also a function.*

Finally, the firing order relation is defined as $\mathrm{firingRel}(\theta) = \{(\mathsf{a},\mathsf{b}) \mid \mathsf{a} = \mathsf{e}_i \wedge \mathsf{b} = \mathsf{e}_j \wedge i < j\}$ *and is a strict total order which we will usually note as* $<_{\mathsf{fo}}$. *If* $\mathsf{a} <_{\mathsf{fo}} \mathsf{b}$ *we will say that* a *fires before* b *in* θ. *Obviously,* $\mathsf{a} <_{\mathsf{tr}} \mathsf{b} \Rightarrow \mathsf{a} <_{\mathsf{fo}} \mathsf{b}$. ■ 5.4

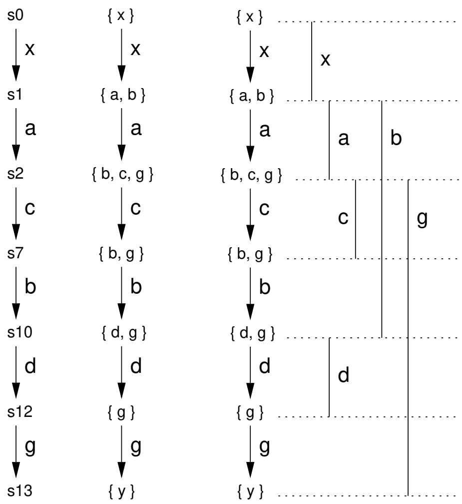
For instance in distinct trace of Figure 5.5 we have that $\mathrm{trigger}(\langle \mathsf{c},1\rangle) = \langle \mathsf{x},1\rangle$, $\mathrm{disabler}(\langle \mathsf{c},2\rangle) = \langle \mathsf{a},2\rangle$, $\langle \mathsf{x},1\rangle <_{\mathsf{tr}} \langle \mathsf{e},1\rangle$ and $\langle \mathsf{b},1\rangle <_{\mathsf{fo}} \langle \mathsf{c},1\rangle$. Note that standardized traces have the property that there is a lowest element in the $<_{\mathsf{tr}}$ and $<_{\mathsf{fo}}$ orders, which is precisely ϵ.

Although it is an abstract notion, a trace has a direct correspondence with the notion of run in transition systems. Since a TS is a particular case of LzTS, and a TTS is described in terms of a TS plus certain delays, the following definition also applies to those models.

Definition 5.5 (Traces in lazy transition systems)

Each run $\rho = \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \xrightarrow{\mathsf{e}_2} \cdots$ *of a* LzTS *defines a trace* $\theta_\rho = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots$ *where* E_i *is the set of events enabled at* s_i, i.e. $E_i = \mathcal{E}(\mathsf{s}_i)$. *As there are* valid runs *and* lazy runs *we will also talk about* valid *and* lazy traces. ■ 5.5

Given the importance that lazy traces will have in future sections we will denote by $\mathcal{L}_{\mathrm{lzy}}(A)$ the set of lazy traces of LzTS A.



(a) (b)

$$\delta^l(\mathsf{x}) \leq \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{x})$$
$$\delta^l(\mathsf{a}) \leq \mathrm{ft}(\mathsf{a}) - \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{a})$$
$$\delta^l(\mathsf{b}) \leq \mathrm{ft}(\mathsf{b}) - \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{b})$$
$$\delta^l(\mathsf{c}) \leq \mathrm{ft}(\mathsf{c}) - \mathrm{ft}(\mathsf{a}) \leq \delta^u(\mathsf{c})$$
$$\delta^l(\mathsf{g}) \leq \mathrm{ft}(\mathsf{g}) - \mathrm{ft}(\mathsf{a}) \leq \delta^u(\mathsf{g})$$
$$\delta^l(\mathsf{d}) \leq \mathrm{ft}(\mathsf{d}) - \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{d})$$

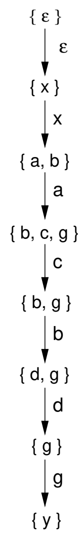
$$\mathrm{ft}(\mathsf{x}) \leq \mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{c}) \leq \mathrm{ft}(\mathsf{b}) \leq \mathrm{ft}(\mathsf{d}) \leq \mathrm{ft}(\mathsf{g})$$

(c)

$$\mathrm{ft}(\epsilon) = 0$$
$$\delta^l(\mathsf{x}) \leq \mathrm{ft}(\mathsf{x}) - \mathrm{ft}(\epsilon) \leq \delta^u(\mathsf{x})$$
$$\delta^l(\mathsf{a}) \leq \mathrm{ft}(\mathsf{a}) - \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{a})$$
$$\delta^l(\mathsf{b}) \leq \mathrm{ft}(\mathsf{b}) - \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{b})$$
$$\delta^l(\mathsf{c}) \leq \mathrm{ft}(\mathsf{c}) - \mathrm{ft}(\mathsf{a}) \leq \delta^u(\mathsf{c})$$
$$\delta^l(\mathsf{g}) \leq \mathrm{ft}(\mathsf{g}) - \mathrm{ft}(\mathsf{a}) \leq \delta^u(\mathsf{g})$$
$$\delta^l(\mathsf{d}) \leq \mathrm{ft}(\mathsf{d}) - \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{d})$$

$$\mathrm{ft}(\epsilon) \leq \mathrm{ft}(\mathsf{x}) \leq \mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{c}) \leq \mathrm{ft}(\mathsf{b}) \leq \mathrm{ft}(\mathsf{d}) \leq \mathrm{ft}(\mathsf{g})$$

(e)



(d)

Figure 5.6. Example of the trace associated to a run and the inequalities used to check its timing consistency: (a) a run taken from the TTS in Figure 5.2 (left) and its corresponding trace (right), (b) intervals in which the events of the trace remain enabled, (c) timing analysis of the trace, (d) standardized trace, and (e) timing analysis of the standardized trace.

Figure 5.6 (a) depicts a run, taken from the TTS in Figure 5.2, and its corresponding trace. A state of the run is substituted in the trace by the set of events enabled at such state in the transition system. The enabling intervals of the events in the trace are depicted as vertical lines in Figure 5.6 (b).

Note that a trace defines, at least, one run (and possibly more, see Figure 5.7 for an example). This, however, does not affect the timing analysis, as all the runs that have exactly the same trace will both exist or not in the TTS (timing consistency depends on the trace, not the particular run, as timing consistency relates to events, not the particular states visited).

Definition 5.6 (Runs generated by a trace)

Let $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR}\rangle$ *be a LzTS,* $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots$ *be one of its traces and* s *one state of* S*. The set of runs generated by* θ *in* A *starting from state* s *is defined as:*

$$\mathrm{runs}(A, \theta, \mathsf{s}) = \{\rho = \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \xrightarrow{\mathsf{e}_2} \cdots \mid \mathsf{s} = \mathsf{s}_1 \wedge \forall i \geq 1, \mathsf{s}_i \xrightarrow{\mathsf{e}_i} \mathsf{s}_{i+1} \in T \wedge \mathcal{E}(\mathsf{s}_i) = E_i\}.$$

When the LzTS is clear from the context we will simply write $\mathrm{runs}(\theta, \mathsf{s})$. ■ 5.6

Notions defined over the runs of a transition system can be naturally extended for their traces counterparts. Specially relevant for the verification problem is the notion of timing consistent run (Definition 2.6 and Lemma 5.2).

Next, the language of a system is defined by the set of valid traces that it can generate. In the case of a TTS only the traces defined by the runs that satisfy the delays associated to the events of the system are considered *i.e.* traces defined by the timing-consistent runs. That is:

Definition 5.7 (Languages)

The language $\mathcal{L}(A)$ *of a* LzTS A *is the set of traces defined by all valid runs of* A.

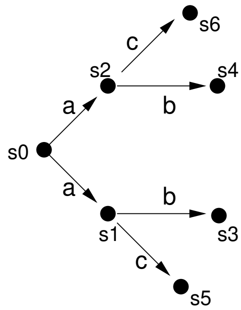


Figure 5.7. Example in which a trace generates many runs: in the TS trace $\theta = \{\mathsf{a}\} \xrightarrow{\mathsf{a}} \{\mathsf{b}, \mathsf{c}\} \xrightarrow{\mathsf{b}} \emptyset$ generates more than one run. In particular $\mathrm{runs}(\theta, \mathsf{s}_0)$ is formed by two runs, $\mathsf{s}_0 \xrightarrow{\mathsf{a}} \mathsf{s}_1 \xrightarrow{\mathsf{b}} \mathsf{s}_3$ and $\mathsf{s}_0 \xrightarrow{\mathsf{a}} \mathsf{s}_2 \xrightarrow{\mathsf{b}} \mathsf{s}_4$.

The language $\mathcal{L}(A)$ *of a* TTS $A = \langle A^-, \delta^l, \delta^u \rangle$ *is the set of traces defined by all timing consistent runs of* A^-. ■ 5.7

Lemma 5.1 (Language inclusion)

Let $A = \langle A^-, \delta^l, \delta^u \rangle$ *be a* TTS. *Then, its language is a subset of that of its underlying* TS, i.e. $\mathcal{L}(A) \subseteq \mathcal{L}(A^-)$. ■ 5.1

The proof of the lemma directly follows from Definition 2.6 and Definition 5.7.

Now the verification problem can be rewritten as a language containment problem: let $\mathcal{L}(P)$ be the language formed by all traces satisfying the set of safety properties P, then a TTS $A = \langle A^-, \delta^l, \delta^u \rangle$ is correct iff $\mathcal{L}(A) \subseteq \mathcal{L}(P)$ or, what is the same, $\mathcal{L}(A) \cap \overline{\mathcal{L}(P)} = \emptyset$.

5.3.2 Checking the timing consistency of traces

To solve the verification problem, the language of the TTS that models the system under verification must be computed or, at least, conservatively estimated. According to Definition 5.7 this requires a mechanism to check whether a trace of the underlying TS is timing consistent or not (see Definition 2.6). Checking the timing-consistency of a trace can be formulated quite simply by means of a set of expressions that bound the firing times of the events in the trace according to their delays.

We will derive these equations from Definition 2.6, but we first need to introduce some notation.

Definition 5.8 (Firing/enabling/disabling/active time)

Given a distinct trace $\theta' = E'_1 \xrightarrow{\mathsf{e}'_1} E'_2 \xrightarrow{\mathsf{e}'_2} \cdots$ *obtained from a run* $\rho = \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \xrightarrow{\mathsf{e}_2} \cdots$*:*

We denote by $\mathrm{ft}(\mathsf{e}'_i)$ *the* firing time *of event* $\mathsf{e}'_i = \langle \mathsf{e}, n \rangle$. *According to Definition 2.6, if* $\mathsf{s}_i \xrightarrow{\mathsf{e}} \mathsf{s}_{i+1}$ *is a part of* ρ, *then* $\mathrm{ft}(\mathsf{e}'_i) = \tau_{i+1}$. *That is the instant at which the event was fired.*

We denote by $\mathrm{et}(\mathsf{e}')$ *the* enabling time *of event* $\mathsf{e}' = \langle \mathsf{e}, n \rangle$. *If* $\mathsf{e}' \notin E'_{j-1}$ *and* $\mathsf{e}' \in E'_j$, *then according to Definition 2.6, if* $\mathsf{s}_{j-1} \xrightarrow{\mathsf{e}_{j-1}} \mathsf{s}_j$ *is a part of* ρ *and* $\mathsf{e} \notin \mathcal{E}(\mathsf{s}_{j-1})$ *and* $\mathsf{e} \in \mathcal{E}(\mathsf{s}_j)$, *then* $\mathrm{et}(\mathsf{e}') = \tau_j$. *That is the instant at which the event became enabled.*

We denote by $\mathrm{dt}(\mathsf{e}')$ *the* disabling time *of event* $\mathsf{e}' = \langle \mathsf{e}, n \rangle$. *If* $\mathsf{e}' \in E'_{k-1}$ *and* $\mathsf{e}' \notin E'_k$, *then according to Definition 2.6, if* $\mathsf{s}_{k-1} \xrightarrow{\mathsf{e}_{k-1}} \mathsf{s}_k$ *is a part of* ρ *and* $\mathsf{e} \in \mathcal{E}(\mathsf{s}_{k-1})$ *and* $\mathsf{e} \notin \mathcal{E}(\mathsf{s}_k)$, *then* $\mathrm{dt}(\mathsf{e}') = \tau_k$. *That is the instant at which the event became disabled.*

We denote by $\mathrm{at}(\mathsf{e}')$ *the* active time *of event* e'*. This is the amount of time that a particular event is enabled in the run or trace.* ■ 5.8

Rewriting Definition 2.6 using the previous notation for a distinct trace $\theta = E_1 \xrightarrow{\mathsf{e}_1} \cdots \xrightarrow{\mathsf{e}_n} E_{n+1}$ defined over the alphabet Σ, we have that for θ to be timing consistent we need that

(i) time stamps must increase monotonically. As there is a one to one correspondence between time stamps and firing times of fired events ($\mathrm{ft}(\mathsf{e}_i) = \tau_{i+1}$) this means that,

$$\forall i : 1 \leq i,\ \mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{e}_{i+1}) \tag{5.1}$$

(ii) for events e_i that fire in the trace, its active time must be greater than their minimum delay,

$$\forall i : 1 \leq i,\ \mathrm{at}(\mathsf{e}_i) \geq \delta^l(\mathsf{e}_i) \tag{5.2}$$

(iii) for all events e in the alphabet, their active time is less than their maximum delay,

$$\forall \mathsf{e} \in \Sigma,\ \mathrm{at}(\mathsf{e}) \leq \delta^u(\mathsf{e}) \tag{5.3}$$

as no event can be enabled more than this time.

We want these rules to be expressed in terms of the firing time of the events, thus we need to express $\mathrm{at}(\mathsf{e})$ in terms of ft variables. To do so we will use the following two observations:

(i) In a trace (or fragment), each event e must do one of these two things:

- Be initially enabled. Then $\mathrm{et}(\mathsf{e}) = 0$.
- Be enabled by some other event. Then $\mathrm{et}(\mathsf{e}) = \mathrm{ft}(\mathrm{trigger}(\mathsf{e}))$.

(ii) In a trace (or fragment), each event e must do one of these three things:

- Fire.
- Be disabled by some other event.
- Remain active (forever or until last state of trace depending on whether the trace is infinite or finite).

In all of these cases we can express $\mathrm{at}(\mathsf{e})$ in terms of the disbaling/firing/enabling times,

$$\mathrm{at}(\mathsf{e}) = \begin{cases} \mathrm{ft}(\mathsf{e}) - \mathrm{et}(\mathsf{e}) & \text{if } \mathsf{e} \text{ fires.} \\ \mathrm{dt}(\mathsf{e}) - \mathrm{et}(\mathsf{e}) & \text{if } \mathsf{e} \text{ is disabled.} \\ \mathrm{ft}(\mathsf{e}_n) - \mathrm{et}(\mathsf{e}) & \text{otherwise} \end{cases}$$

We call e_n the last fired event in the trace if it is finite (or is a fragment). In case of an infinite trace, for convenience, we define $\mathrm{ft}(\mathsf{e}_n) = \infty$ if trace is not Zeno. For Zeno runs $\mathrm{ft}(\mathsf{e}_n)$ is equal to the last timestamp in which time progresses.

Now we can build a table showing all possible combinations of the previous situations and giving the inequalities induced by the events. We suppose we are dealing with a finite trace (or fragment of trace), then each event e falls into one of the categories described in Table 5.1.

Note that standardized traces will only use the second column of the table because all events have a trigger with the sole exception of special event ϵ and for this event we simply define $\mathrm{ft}(\epsilon) = 0$.

Status of event $\mathtt{e}$	*Initially enabled*	*Enabled by the firing of some event*
Fired	$\delta^l(\mathtt{e}) \leq \mathrm{ft}(\mathtt{e}) \leq \delta^u(\mathtt{e})$	$\delta^l(\mathtt{e}) \leq \mathrm{ft}(\mathtt{e}) - \mathrm{ft}(\mathrm{trigger}(\mathtt{e})) \leq \delta^u(\mathtt{e})$
Disabled	$\mathrm{ft}(\mathrm{disabler}(\mathtt{e})) \leq \delta^u(\mathtt{e})$	$\mathrm{ft}(\mathrm{disabler}(\mathtt{e})) - \mathrm{ft}(\mathrm{trigger}(\mathtt{e})) \leq \delta^u(\mathtt{e})$
Remains after last event $\mathtt{e}_n$ has fired $^\alpha$	$\mathrm{ft}(\mathtt{e}_n) \leq \delta^u(\mathtt{e})$	$\mathrm{ft}(\mathtt{e}_n) - \mathrm{ft}(\mathrm{trigger}(\mathtt{e})) \leq \delta^u(\mathtt{e})$

$^\alpha$ for an infinite run/trace this category corresponds to: *Remains enabled forever.*

Table 5.1. Table of inequalities induced by an event $\mathtt{e}$ in the timing consistent checking problem of a trace.

LEMMA 5.2 (TIMING CONSISTENCY CHECKING OF A TRACE)
To decide whether a finite distinct trace $\theta = E_1 \xrightarrow{\mathtt{e}_1} E_2 \xrightarrow{\mathtt{e}_2} \cdots E_n \xrightarrow{\mathtt{e}_n} E_{n+1}$ defined over the event alphabet Σ is timing consistent or not it is enough to check if the following LP problem has some solution or not:

- *Each event has smaller or equal firing time than its successor in the trace:*

$$\forall i : 1 \leq i < n, \quad \mathrm{ft}(\mathtt{e}_i) \ \leq \ \mathrm{ft}(\mathtt{e}_{i+1})$$

  We will call this set of inequalities the sequence order equations.

- *For each event $\mathtt{e}$ in the alphabet Σ, add the corresponding equation from Table 5.1. We will refer to this set of inequalities as the* causal order equations. *For standardized traces this includes the special equation* $\mathrm{ft}(\epsilon) = 0$.

■ 5.2

Proof: *In Table 5.1 we already added the conditions for equation (5.2), thus using this table for each event in the alphabet of the trace we cover equations (5.2) and (5.3). If we add the condition for time progress (5.1) the equation system obtained will fulfill all the points from the rewriting of Definition 2.6, hence will be able to determine the timing consistency of the trace.*

This definition requires that θ is a distinct trace in order to make the firing time of each occurrence of an event independent from the other occurrence times. If such a "name-clash" of firing time variables is not possible (*i.e.* the trace has no repeated events) we will skip the occurrence number in order to make examples more clear.

We denote as $\mathrm{LP}(\theta)$ the system of equations obtained by the application of Lemma 5.2 to distinct trace θ. The inequalities of $\mathrm{LP}(\theta)$ are all of degree 2 and can be arranged as a subtraction of variables, thus, they can be represented using a DBM. Also they can be represented as a graph as seen in §4.4.1.0 when DBMs were introduced. Thereofore there is an equivalent graph-theoretic formulation of the timing consistency checking problem that we extensively use in §5.4. We also denote by $\mathrm{sol}(\mathrm{LP}(\theta))$ the set of solutions to the unknowns of $\mathrm{LP}(\theta)$. Given a distinct trace θ and its standardized trace θ' their set of solutions for their common alphabet (that is all events except ϵ) are the same, hence there is no problem in using standardized traces instead of the original ones in tasks where only their set of solutions is important *e.g.* timing consistency checking.

EXAMPLE 5.2 (CONT.) *Figure 5.6 (c) shows the set of constraints of the linear programming problem to check the timing-consistency of the trace of (b). Since no disabling situation appears in the trace, only expressions of the first row of Table 5.1 and equation (5.1) apply.*

In this case, the problem has no solution if the delays shown in Figure 5.2 (b) are considered. Therefore, the trace is not timing consistent and does not belong to the language of the system.

Notice that this result is coherent with that obtained in Example 5.2. The trace was removed from the LzTS *obtained after the first refinement of the verification approach (see Figure 5.3).* ■ 5.2

For finite distinct traces, timing consistency does not depend on what events appear in their last enabling (*i.e.* E_{n+1}) as next corollary states

COROLLARY 5.3 (REDUCED TIMING CONSISTENCY CHECKING OF A TRACE)
Given a finite distinct trace $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots E_n \xrightarrow{\mathsf{e}_n} E_{n+1}$*, its timing consistency does not depend on the events present in* E_{n+1}*. Hence we can talk simply about the timing consistency of* $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots E_n \xrightarrow{\mathsf{e}_n}$ *without having to specify* E_{n+1}. ■ 5.3

Proof: *Events in* $E_{n+1} \setminus E_n$ *are enabled by* e_n *(*i.e. $\forall \mathsf{e} \in E_{n+1} \setminus E_n, \mathrm{trigger}(\mathsf{e}) = \mathsf{e}_n$*) and, obviously, remain enabled until the end of the trace, thus according to Lemma 5.2 and Table 5.1 for any event* e *in such set, equation* $\mathrm{ft}(\mathsf{e}_n) - \mathrm{ft}(\mathsf{e}_n) \leq \delta^u(\mathsf{e})$ *is added to* $\mathrm{LP}(\theta)$*. As this equation is equal to* $0 \leq \delta^u(\mathsf{e})$*, which is trivially true, then can be safely removed from* $\mathrm{LP}(\theta)$*. On the other hand, events in* E_n *will either remain enabled until the end of trace (if they appear in* E_{n+1}*) or be disabled by last fired event (if they appear do not belong to* E_{n+1} *and are different than the last fired event). In any case the equations added to* $\mathrm{LP}(\theta)$ *are the same in both cases as disabler event for the disabled events is the last event (*i.e. $\mathrm{disabler}(\mathsf{e}) = \mathsf{e}_n$*).*

Conservative approximations

Before presenting a verification method based in the timing consistency checking of individual traces, let us pay attention to a small detail.

Given the following trace $\theta_f = \{\mathsf{a}, \mathsf{b}\} \xrightarrow{\mathsf{a}} \{\mathsf{b}\} \xrightarrow{\mathsf{b}}$ where events have the following delays: $\delta(\mathsf{a}) = [1, 3]$ and $\delta(\mathsf{b}) = [2, 4]$, we know by Lemma 5.2 and Corollary 5.3 that θ_f is timing consistent, if and only if the following system is satisfied:

$$\begin{array}{rcl} & \mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{b}) & \\ 1 & \leq \mathrm{ft}(\mathsf{a}) \leq & 3 \\ 2 & \leq \mathrm{ft}(\mathsf{b}) \leq & 4. \end{array}$$

However, by enforcing that $1 \leq \mathrm{ft}(\mathsf{a})$ and $2 \leq \mathrm{ft}(\mathsf{b})$ we are assuming that both events became enabled at exactly the same time. As they have no common trigger this seems rather a strong requirement. Imagine we have a circuit and we power it on. In this state the circuit can do two different transitions, lets say a and b, that affect two different parts of the system. Could we expect them to be totally synchronized? In this case we do assume that we must do a conservative approximation of the behavior of these events and consider $\delta^l(\mathsf{a}) = \delta^l(\mathsf{b}) = 0$. This allows to model uncertainty (see §B.13 for a discussion on this topic).

In general we will consider that any initially enabled event e in a trace has $\delta^l(\mathsf{e}) = 0$. Here *initially enabled* means that the event is enabled in the initial state. Formally, if the LzTS has a set of initial states S_{in}, then an event e is initially enabled in all states of the set $\mathrm{ER}(\mathsf{e}) \cap \mathsf{S}_{in}$ that appear as the first state of a run (*i.e.* states can be revisited afterwards but then events are not considered initially enabled). As events are all distinct in a trace (by renaming) there is no problem with the same event having two different delays. Alternatively we could have modified Table 5.1 with the same effect, but we will see later on that by modifying the delays of the events we can naturally reproduce this assumption when computing timing consistency by other means.

Anyway one can override this setting by introducing a *startup* event in the system, that will synchronize the previously initially enabled events. In our little example we could have chosen to add an event x such that it triggers both a and b. Note that this is easier than trying to do it the other way round, that is, assuming perfect synchronization between initially enabled events and trying to introduce uncertainty, because then you need one unbounded trigger event for each initially enabled event.

The assumption of uncertainty also arises in some other situations. In particular when analyzing the timing-consistency of fragments of traces. For instance imagine that θ_f is a fragment of a trace. We give two possible traces that contain it:

$$\begin{aligned}\theta_1 &= \{\mathsf{a},\mathsf{y}\}\xrightarrow{\mathsf{y}}\{\mathsf{a},\mathsf{b}\}\xrightarrow{\mathsf{a}}\{\mathsf{b}\}\xrightarrow{\mathsf{b}}\\ \theta_2 &= \{\mathsf{x},\mathsf{b}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b}\}\xrightarrow{\mathsf{a}}\{\mathsf{b}\}\xrightarrow{\mathsf{b}}\end{aligned}$$

In case the trace would be θ_1 then, considering only its fragment θ_f we could have used $2 \leq \mathrm{ft}(\mathsf{b})$ as we know that in state $\{\mathsf{a},\mathsf{b}\}$, event b has been now triggered. On the other hand, we must assume $\delta^l(\mathsf{a}) = 0$ as event a was already active and we do not know for how much time. We have just the opposite situation in case of θ_2.

To easily recognize which events of the traces are conservatively approximated we will use the following notation: if event e is conservatively approximated then we will write $\dot{\mathsf{e}}$. For instance if fragment θ_f would have been extracted from θ_1 then we will write $\theta_f = \{\dot{\mathsf{a}},\mathsf{b}\}\xrightarrow{\dot{\mathsf{a}}}\{\mathsf{b}\}\xrightarrow{\mathsf{b}}$. Had it been taken from θ_2 then we would have noted $\theta_f = \{\mathsf{a},\dot{\mathsf{b}}\}\xrightarrow{\mathsf{a}}\{\dot{\mathsf{b}}\}\xrightarrow{\dot{\mathsf{b}}}$.

In many cases conservative approximations will be even enforced. For instance when both events a and b are conservatively approximated as in trace $\theta'_f = \{\dot{\mathsf{a}},\dot{\mathsf{b}}\}\xrightarrow{\dot{\mathsf{a}}}\{\dot{\mathsf{b}}\}\xrightarrow{\dot{\mathsf{b}}}$. the solution set of $\mathrm{LP}(\theta'_f)$ is a superset of both $\mathrm{LP}(\theta_1)$ and $\mathrm{LP}(\theta_2)$. This does not happen if some of the events is not conservatively approximated and is an interesting feature as if θ'_f can be proved timing-inconsistent (which in this particular example cannot be proved) then we know that both θ_1 and θ_2 are also timing inconsistent. We will see later that this can increase the impact of a particular refinement, by making a composition potentially cover more traces.

Given that the events with a lower delay bound equal to 0 have some specific properties we define a function that helps to distinguish such events from the rest.

DEFINITION 5.9 (SET OF LOWER DELAY BOUNDED EVENTS)
The set of lower delay bounded events *of an enabling E, noted as $\mathrm{LB}(E)$, is defined as the set of events formed by all events of E such that their lower delay is greater than 0. Formally,* $\mathrm{LB}(E) = \{\mathsf{e} \in E \mid \delta^l(\mathsf{e}) > 0\}$. ■ 5.9

Conversely, the set of lower delay unbounded events of E is described in terms of $\mathrm{LB}(E)$ as $E \setminus \mathrm{LB}(E)$.

5.3.3 Trace-based verification

Provided the formulation developed above for the timing analysis on a trace, a trace-based method for the verification of timed systems can be devised. The method must consider all the traces leading from the initial state of the system up to the states where violations of the properties under verification occur. The system is correct if no failure trace exists when the delays are taken into account, *i.e.* if none of the failure traces is timing-consistent. On the contrary, a timing-consistent failure trace provides a counterexample that proves the incorrectness of the system.

The impossibility of this trace-based method for verification is obvious. The number of traces between two states of the system may be exponential with respect to the number of concurrent events in each conflict branch of the state space. Therefore, some strategy to alleviate this complexity is required.

Trace θ_1	**Trace θ_2**	**$\mathrm{gLP}(\theta_1)$**	**$\mathrm{gLP}(\theta_2)$**
$\{\mathsf{a},\mathsf{b}\} \overset{\mathsf{a}}{\downarrow} \emptyset$	$\{\mathsf{c},\mathsf{d}\} \overset{\mathsf{c}}{\downarrow} \{\mathsf{d}\}$	$\delta^l(\mathsf{a}) \leq \mathrm{ft}(\mathsf{a})$ $\mathrm{ft}(\mathsf{a}) \leq \delta^u(\mathsf{a})$ $\mathrm{ft}(\mathsf{a}) \leq \delta^u(\mathsf{b})$	$\delta^l(\mathsf{c}) \leq \mathrm{ft}(\mathsf{c})$ $\mathrm{ft}(\mathsf{c}) \leq \delta^u(\mathsf{c})$ $\mathrm{ft}(\mathsf{c}) \leq \delta^u(\mathsf{d})$
(a)	(b)	(c)	(d)

(e) (f)

Figure 5.8. Example of structural timing inclusion: trace θ_2 is structurally timing included in θ_1 if $\delta^l(\mathsf{c}) \geq \delta^l(\mathsf{a})$ and $\delta^u(\mathsf{d}) \leq \delta^u(\mathsf{b})$.

5.4 Timing inclusion

This section provides a fundamental result that helps addressing the complexity problem exposed by the trace-based verification method outlined in §5.3.3. In short, Theorem 5.4 states that the results obtained from the consistency checking of a given system of equations can be applied to all those traces that are *timing included* in that system.

DEFINITION 5.10 (SYSTEM OF PARAMETRIC TIMED EQUATIONS)

A System of Parametric Timed Equations *(SPTE) is a set of equations in which two kind of variables appear: firing time variables (of the form* $\mathrm{ft}(\mathsf{e})$*) and delay variables (of the form* $\delta^l(\mathsf{e})$ *or* $\delta^u(\mathsf{e})$*). All the equations in a SPTE are of the form* $\mathrm{ft}(\mathsf{e}_i) - \mathrm{ft}(\mathsf{e}_j) \leq k$, $\mathrm{ft}(\mathsf{e}_i) \leq k$ *or* $-\mathrm{ft}(\mathsf{e}_i) \leq k$*, where k can be a positive maximum delay variable (*i.e. $\delta^u(\mathsf{e})$*), a negative minimum delay variable (*i.e. $-\delta^l(\mathsf{e})$*) or the constant 0.*

Given an SPTE S, the alphabet of *S is the set of symbols* e *that appear in either a variable* $\mathrm{ft}(\mathsf{e})$, $\delta^l(\mathsf{e})$ *or* $\delta^u(\mathsf{e})$*.* ■ 5.10

In Figure 5.8 (c) we can see an SPTE named $\mathrm{gLP}(\theta_1)$. The alphabet of $\mathrm{gLP}(\theta_1)$ is $\{\mathsf{a},\mathsf{b}\}$.

DEFINITION 5.11 (GENERIC TIMING CONSISTENCY SYSTEM OF A TRACE)

The generic timing consistency system *of trace* θ*, noted as* $\mathrm{gLP}(\theta)$*, is the system of equations obtained by the application of Lemma 5.2 to the distinct trace of* θ *in which the delays of the events are kept as variables.* $\mathrm{gLP}(\theta)$ *is a SPTE.* ■ 5.11

In Figure 5.8 (c,d) we can see two SPTEs, named $\mathrm{gLP}(\theta_1)$ and $\mathrm{gLP}(\theta_2)$, corresponding to the application of Lemma 5.2 to the traces of subfigures (a) and (b) respectively.

The timing consistency problem can be translated into a DBM consistency check. DBMs can be graphically depicted as directed graphs and its consistency can be checked by finding negative cycles in that graph (see §4.4.1.0). In the same spirit, SPTEs can be translated into Parametric DBMs (PDBMs) [AAB00] using the delay variables as parameters and then they can be graphically represented also by a directed graph. For instance in Figure 5.8 (e,f) we can see the corresponding graphs associated to $\mathrm{gLP}(\theta_1)$ and $\mathrm{gLP}(\theta_2)$ respectively.

DEFINITION 5.12 (ASSOCIATED GRAPH OF A **SPTE**)
Let S be a SPTE. The associated graph *of S, noted as* $\mathrm{graph}(S)$, *is a directed labeled graph $\langle V, E, L\rangle$ in which the set of vertices V is formed by all the firing time variables of S plus a special node named* 0 *and for each equation* $\mathrm{ft}(\mathsf{e}_i) - \mathrm{ft}(\mathsf{e}_j) \leq k$ *the set of edges E contains an arc from node* $\mathrm{ft}(\mathsf{e}_j)$ *to node* $\mathrm{ft}(\mathsf{e}_i)$. *For equations of the type* $\mathrm{ft}(\mathsf{e}_i) \leq k$ *the source node is the special node* 0. *Conversely for equations of type* $-\mathrm{ft}(\mathsf{e}_i) \leq k$ *destination node is* 0. *Each edge is labeled with the k associated to the corresponding equation. In order to work more easily we will assume that all redundant arcs in E are removed.*

Given a cycle c of $\mathrm{graph}(S)$ *its weight is the sum of all labels of the edges of c. The set of possible negative cycles of* $\mathrm{graph}(S)$ *is formed by all the cycles in* $\mathrm{graph}(S)$ *such that:*

- *at least one edge has a negative label (*e.g. $-\delta^l(\mathsf{e})$*)* and
- *there is a at least one of the edges that has a negative label* $-\delta^l(\mathsf{e}_1)$ *for which the corresponding positive delay variable does not appear in the cycle.* i.e. *No edge labeled with* $\delta^u(\mathsf{e}_1)$ *appears in the cycle.*

■ 5.12

As delay variables can only be positive and the maximum delay bound is always greater or equal to the minimum delay bound (*i.e.* $0 \leq \delta^l(\mathsf{e}) \leq \delta^u(\mathsf{e})$) then any cycle without a negative minimum delay bound or in which both delay variables appear will have at least a weight of 0, so cannot be negative. For instance, in Figure 5.8 (e) $\mathrm{graph}(\mathrm{gLP}(\theta_1))$ is depicted. It has two simple cycles, namely:

$$\begin{aligned} c_1 &= 0 \xrightarrow{\delta^u(\mathsf{b})} \mathrm{ft}(\mathsf{a}) \xrightarrow{-\delta^l(\mathsf{a})} 0 \\ c_2 &= 0 \xrightarrow{\delta^u(\mathsf{a})} \mathrm{ft}(\mathsf{a}) \xrightarrow{-\delta^l(\mathsf{a})} 0 \end{aligned}$$

The cycle c_2 does not satisfy all the conditions to be a possible negative cycle because although it contains a negative label ($-\delta^l(\mathsf{a})$) it contains also its corresponding positive maximum delay bound ($\delta^u(\mathsf{a})$), so the only possible negative cycle is c_1.

The set of cycles of a graph is infinite if the graph contains at least one simple cycle, because simple cycles might be composed *ad libitum* to form bigger cycles. In our last example for instance the following is also a possible negative cycle: $0 \xrightarrow{\delta^u(\mathsf{b})} \mathrm{ft}(\mathsf{a}) \xrightarrow{-\delta^l(\mathsf{a})} 0 \xrightarrow{\delta^u(\mathsf{a})} \mathrm{ft}(\mathsf{a}) \xrightarrow{-\delta^l(\mathsf{a})} 0$. Our main interest in possible negative cycles is because if some of them is actually negative, then the trace is timing inconsistent. Although the set of possible negative cycles is infinite for most finite traces, it suffices to take into account only a finite subset of them in order to reason about the sign of their weight:

- If a compound cycle has a negative weight it means that at least one of the weights of the simple cycles is negative. Hence only simple cycles must be checked and there is a finite number of simple cycles in a SPTE coming from a trace timing consistency problem.
- If a simple cycle exists with a more negative weight than another, is clear that it is only necessary to check the one with minimum weight, as it will always be negative if the other is.

We call this reduced set of possible negative cycles the set of *representative possible negative cycles* and we formally define it as follows.

Definition 5.13 (Representative possible negative cycles)
Given a set of possible negative cycles C, the set R of representative negative cycles *of C is the subset of C formed by all simple cycles of C such that, for any simple cycle $c \in R$ there is no other simple cycle $c' \in R$ with* $\mathrm{weight}(c') \leq \mathrm{weight}(c)$.

■ 5.13

The only traces θ in which there are no possible negative cycles in $\mathrm{graph}(\mathrm{gLP}(\theta))$ are those that exhibit a sequential behavior. That is, each of its sets of enabled events E_i contains just a single event. Obviously these traces are always timing consistent, thus no possible negative cycle can appear in their associated graphs. As soon as concurrency appears, traces can become inconsistent, thus they must contain at least one possible negative cycle.

Definition 5.14 (Timing inclusion of **SPTE**s)
*Let S_1 and S_2 be **SPTE**s. We say that S_2 is timing included in S_1 if for each representative possible negative cycle c_1 in* $\mathrm{graph}(S_1)$ *of weight w_1 there is a representative possible negative cycle c_2 in* $\mathrm{graph}(S_2)$ *with weight w_2 such that $w_2 \leq w_1$.* ■ 5.14

In Figure 5.8 (a,b) we can see two traces, namely θ_1 and θ_2. Their associated graphs $\mathrm{gLP}(\theta_1)$ and $\mathrm{gLP}(\theta_2)$ are shown is subfigures (c) and (d) respectively. We will analyze in which situations $\mathrm{gLP}(\theta_2)$ is timing included in $\mathrm{gLP}(\theta_1)$. As already mentioned $\mathrm{graph}(\mathrm{gLP}(\theta_1))$ in subfigure (e) contains only one possible negative cycle:

$$c_1 = 0 \xrightarrow{\delta^u(\mathsf{b})} \mathrm{ft}(\mathsf{a}) \xrightarrow{-\delta^l(\mathsf{a})} 0 \text{ with weight } w_1 = \delta^u(\mathsf{b}) - \delta^l(\mathsf{a}).$$

On the other hand $\mathrm{graph}(\mathrm{gLP}(\theta_2))$ in subfigure (f) has also only one possible negative cycle:

$$c_2' = 0 \xrightarrow{\delta^u(\mathsf{d})} \mathrm{ft}(\mathsf{c}) \xrightarrow{-\delta^l(\mathsf{c})} 0 \text{ with weight } w_2' = \delta^u(\mathsf{d}) - \delta^l(\mathsf{c}).$$

The only possibility is to assign c_1 to c_2' and that $w_2' \leq w_1$. This means that $\delta^u(\mathsf{d}) - \delta^l(\mathsf{c}) \leq \delta^u(\mathsf{b}) - \delta^l(\mathsf{a})$ must be satisfied in order for $\mathrm{gLP}(\theta_2)$ to be timing included in $\mathrm{gLP}(\theta_1)$. For instance this equation is satisfied if $\delta^u(\mathsf{d}) \leq \delta^u(\mathsf{b})$ and $\delta^l(\mathsf{c}) \geq \delta^l(\mathsf{a})$.

Theorem 5.4 (Timing inclusion and timing consistency)
*Let S_1 and S_2 be two **SPTE**s. Let S_2 be timing included in S_1. Then,*

$$S_1 \textit{ is not consistent } \Longrightarrow S_2 \textit{ is not consistent}$$

■ 5.4

Proof: *If S_1 is not consistent, means that* $\mathrm{graph}(S_1)$ *contains a negative cycle. Thus one of its cycles in the set of representative possible negative cycles is actually negative, let name it c_n. As for any of the cycles in that set, we have a cycle in* $\mathrm{graph}(S_2)$ *that has a smaller weight, then we know that for c_n there is also a cycle with smaller weight in* $\mathrm{graph}(S_2)$*, and that this cycle must have negative weight, thus S_2 cannot be consistent.*

As an example we will apply the theorem to our running example of Figure 5.8. Imagine that θ_1 has the following delays: $\delta(\mathsf{a}) = [3, 4]$ and $\delta(\mathsf{b}) = [0, 2]$. Clearly $\mathrm{LP}(\theta_1)$ is timing inconsistent as it contains $3 \leq \mathrm{ft}(\mathsf{a})$ and $\mathrm{ft}(\mathsf{a}) \leq 2$, which is represented in $\mathrm{graph}(\mathrm{gLP}(\theta_1))$ by a negative cycle

$c_1 = 0 \xrightarrow{2} \mathrm{ft}(\mathsf{a}) \xrightarrow{-3} 0$ with weight $w_1 = 2 - 3 = -1$. Using the results we obtained from previous discussion, this means that any trace θ_2 with delays $3 \leq \delta^l(\mathsf{c})$ and $\delta^u(\mathsf{d}) \leq 2$ will also be timing inconsistent. Moreover any trace in which two events become enabled at the same time and the one that fires first has an upper bound that when the lower bound of the other event is subtracted we obtain something smaller or equal to -1, can be directly proved inconsistent.

Why did we proceed in a such cumbersome way to get that result? In our system we do not have any symbolic delay, so it seems much more easy and efficient to test timing consistency for each trace than to check by comparison with the timing consistency of another trace. However, in some cases, it is possible to translate the checking of the conditions that guarantee that a trace is timing included in another to the symbolic representation of the system, and then remove all these timing included traces with a single modification of our system. Obviously, our theorem is too much general to have this nice property. In fact, it is not difficult to see that, if used carefully, one should be able to remove all timing inconsistent traces in a single modification which is equivalent to constructing the symbolic region automata of our TTS. Precisely, the high cost of such construction is what makes interesting iterative refinement approaches, so we must reach an equilibrium in which the modification is little enough to be feasible, and also the set of removed traces is as big and interesting as possible (*i.e.* we are only interested in traces leading to failure states).

In §5.5 we present a much more restricted criterion that has a good balance between the cost of the modification and its impact.

5.5 Enabling compatibility

The main drawback of Theorem 5.4 is that comparisons between the weights of the representative possible negative cycles must be performed. For instance, in Figure 5.8, SPTE $\mathrm{gLP}(\theta_2)$ is only timing included in $\mathrm{gLP}(\theta_1)$ if equation $\delta^u(\mathsf{d}) - \delta^l(\mathsf{c}) \leq \delta^u(\mathsf{b}) - \delta^l(\mathsf{a})$ is satisfied. We can simplify things if both traces have the same events, as then the value of the delays can be compared in advance. As an example consider that we obtain a trace θ_3 from trace θ_2 by substituting event c by a and event d by b. That is $\theta_3 = \{\mathsf{a},\mathsf{b}\} \xrightarrow{\mathsf{a}} \{\mathsf{b}\}$. Now comparison between both traces gives that $\mathrm{gLP}(\theta_3)$ is timing included in $\mathrm{gLP}(\theta_1)$ if equation $\delta^u(\mathsf{b}) - \delta^l(\mathsf{a}) \leq \delta^u(\mathsf{b}) - \delta^l(\mathsf{a})$ is satisfied, which is trivially true. Hence, for a number of traces, we can guarantee timing inclusion irrespective of the actual values of their delays.

Definition 5.15 (Structural timing inclusion of **SPTE**s)
Let S_1 and S_2 be SPTEs. We say that S_2 is structurally timing included *in S_1 if S_2 is timing included in S_1 for any value of the delay bounds variables.* ■ 5.15

In our previous example $\mathrm{gLP}(\theta_2)$ is not structurally timing included in $\mathrm{gLP}(\theta_1)$ because with delay values $\delta^u(\mathsf{d}) = 4$, $\delta^l(\mathsf{c}) = 1$, $\delta^u(\mathsf{b}) = 2$ and $\delta^l(\mathsf{a}) = 3$ we do not satisfy $\delta^u(\mathsf{d}) - \delta^l(\mathsf{c}) \leq \delta^u(\mathsf{b}) - \delta^l(\mathsf{a})$. However $\mathrm{gLP}(\theta_3)$ is structurally timing included in $\mathrm{gLP}(\theta_1)$ because $\mathrm{gLP}(\theta_3)$ is timing included in $\mathrm{gLP}(\theta_1)$ no matter which value we do assign to $\delta^u(\mathsf{b})$ and $\delta^l(\mathsf{a})$.

We would like to have an easy criterion to decide, given two traces, if one of their associated SPTEs is structurally timing included in the other. The concept of *enabling compatible mapping* presented in Definition 5.16 guarantees structural inclusion.

Definition 5.16 (Enabling-compatible trace mapping)
Let $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots \xrightarrow{\mathsf{e}_{k-1}} E_k \xrightarrow{\mathsf{e}_k} E_{k+1} \xrightarrow{\mathsf{e}_{k+1}} E_{k+2} \xrightarrow{\mathsf{e}_{k+2}} \cdots E_{k+n} \xrightarrow{\mathsf{e}_{k+n}} E_{k+n+1} \longrightarrow \cdots$ be a trace over the alphabet of events Σ and let $\theta' = E'_1 \xrightarrow{\mathsf{e}'_1} E'_2 \xrightarrow{\mathsf{e}'_2} \cdots E'_m \xrightarrow{\mathsf{e}'_m} E'_{m+1}$ be a trace

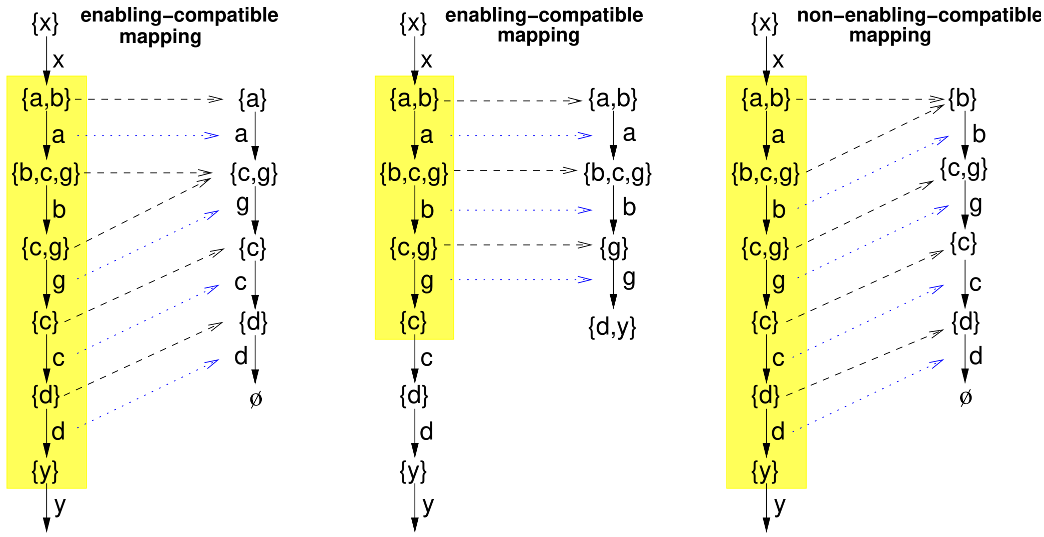


Figure 5.9. Enabling-compatible (left, center) and non-enabling-compatible mappings (right).

over the alphabet $\Sigma' \subseteq \Sigma$. *Let* $\theta_t = E_{k+1} \xrightarrow{\mathsf{e}_{k+1}} E_{k+2} \xrightarrow{\mathsf{e}_{k+2}} \cdots \xrightarrow{\mathsf{e}_{k+n}} E_{k+n+1}$ *be a fragment of* θ.

An enabling-compatible *mapping of* θ_t *onto* θ' *is a monotonically increasing function* $\mathsf{mp} : \mathbb{N} \mapsto \mathbb{N}$ *such that:*

a) $\mathsf{mp}(k+1) = 1$ (initialization)

$$\text{and} \begin{cases} E_k \cap \left(\mathrm{LB}(E'_1) \setminus \{\mathsf{e}_k\}\right) = \emptyset & \text{if } k \geq 1 \\ \mathrm{LB}(E'_1) = \emptyset & \text{if } k = 0 \end{cases}$$

b) $\forall i : k+1 \leq i \leq k+n,$ (firing)

$$\mathsf{mp}(i+1) = \begin{cases} \mathsf{mp}(i) + 1 & \text{if } m \geq \mathsf{mp}(i) \wedge \mathsf{e}_i = \mathsf{e}'_{\mathsf{map(i)}} \\ \mathsf{mp}(i) & \text{otherwise} \end{cases}$$

c) $\mathsf{mp}(k+n+1) = m+1$ (completeness)

d) $\forall i : \ k+1 \leq i \leq k+n, \ E_i \supseteq E'_{\mathsf{mp}(i)}$ (projection)

e) (enabling preservation)

$$\forall i : k+1 \leq i < k+n,$$
$$\mathsf{mp}(i+1) = \mathsf{mp}(i) + 1 \Longrightarrow E_{i+1} \setminus (E_i \setminus \{\mathsf{e}_i\}) \supseteq \mathrm{LB}\left(E'_{\mathsf{mp}(i+1)} \setminus (E'_{\mathsf{mp}(i)} \setminus \{\mathsf{e}'_{\mathsf{mp}(i)}\})\right)$$
$$\forall i : k+1 \leq i \leq k+n, \quad \mathsf{mp}(i+1) = \mathsf{mp}(i) \Longrightarrow \mathsf{e}_i \notin E'_{\mathsf{mp}(i)}$$

■ 5.16

The mapping of θ onto θ' is, broadly speaking, a function that preserves the enabledness of the events in Σ'. Initially, the events enabled in E'_1 must also be enabled in E_{k+1} (initialization & projection condition). Next, the events enabled along θ' must be also enabled in θ (enabling preservation condition) and must have the same trigger event (when $\mathsf{mp}(i+1) = \mathsf{mp}(i)+1$ in the firing condition). Moreover, θ may fire events that are not relevant to θ' (when $\mathsf{mp}(i+1) = \mathsf{mp}(i)$ in the firing condition). Since the firing time of an event only depends on its enabling time and its delay (see §2.2), this notion will allow us to apply the timing analysis of θ' to θ in the fragment θ_t.

There are several special situations that do not adhere perfectly to the above mentioned scenario: lower delay unbounded events are allowed to have a different trigger if this new trigger fires before the old one (hence this type of events can become enabled before than expected) or events that are not fired in θ' can be fired or disabled or remain enabled until the end of θ as long as the projection rule is satisfied. Some of these particularities will be discussed in Lemma 5.6.

Figure 5.9 shows three examples of trace mapping of the shadowed fragment. The black dashed arcs represent the relation between E_i and $E'_{\mathsf{mp}(i)}$. In these arcs the projection rule must be satisfied (*i.e.* $E_i \supseteq E'_{\mathsf{mp}(i)}$). The blue dotted arcs depict the relation between e_i and $\mathsf{e}'_{\mathsf{mp(i)}}$. It must be noted that neither the projection nor the enabling preservation rules are not checked in the last enabling of the mapping. This is because the timing consistency cannot be affected by the events appearing there, whether if newly enabled or not as exposed in Corollary 5.3.

The mappings on the left and center of Figure 5.9 are enabling-compatible. The mapping on the right in general would be not enabling-compatible since it violates the *enabling preservation* rule as c and g are enabled by a in one case, and by b in the other. However should $\delta^l(\mathsf{c}) = 0$ and $\delta^l(\mathsf{g}) = 0$, then the trace would be enabling compatible as lower delay unbounded events are allowed to become enabled *before* they would normally have to.

In [Peñ03] a more restricted enabling-compatible mapping definition was given that, for instance, qualified the central mapping of Figure 5.9 as not enabling-compatible (as well as the right-most mapping independently of the value of $\delta^l(\mathsf{c})$ and $\delta^l(\mathsf{g})$). For more details see Appendix A.3. Note that our definition allows to establish an enabling compatible mapping with a special type of degenerated trace formed by a single enabling. This allows a more uniform treatment in some of the material presented in later stages, notably in §6.4.

The existence of an enabling compatible mapping allows to define a binary relation between traces:

Definition 5.17 (Enabling-compatibility)
Given a set of traces Θ and two traces θ and θ' that belong to this set. We will say that trace θ is enabling compatible with trace θ' if there is a fragment θ_t of θ such that there exists an enabling compatible mapping of θ_t onto θ', in which case we will write $\theta \geq_{\mathsf{ec}} \theta'$. Additionally, if $\theta_t = \theta$ we will say that θ is fully enabling compatible with θ'. ■ 5.17

Lemma 5.5 (The Enabling-compatibility relation is a partial order)
The $\geq_{\mathsf{ec}}$ relation is a reflexive, antisymmetric and transitive relation, hence the enabling compatible relation is a partial order. ■ 5.5

Proof: *The reflexive property comes trivially if the identity mapping is used. Transitive property is shown using the composed mapping. If $\theta_1 \geq_{\mathsf{ec}} \theta_2$, it means that there is an enabling compatible mapping mp_1 between them. For the same reason there is an enabling compatible mapping mp_2 between θ_2 and θ_3. It is straightforward to see that mapping $\mathsf{mp}_2 \circ \mathsf{mp}_1$ is an enabling compatible mapping between θ_1 and θ_3, thus they are enabling compatible. The composed mapping is obviously only defined in the preimages of the domain of mp_2* i.e. *the fragment of θ_1 used as domain for mp_1 is restricted by the fragment of θ_2 used as domain for mp_2.*

Finally the antisymmetric property comes from the fact that both traces must have the same length due to the completeness rule in the enabling compatible mapping definition. This forces the mappings to be the identity mappings and, consequently, all fired events must be the same in both traces by the firing rule. As the set of enabled events must satisfy the projection rule in both sense this guarantees that both traces have exactly the same enablings as well. As they have the same enablings and they fire the same events, both traces are equal.

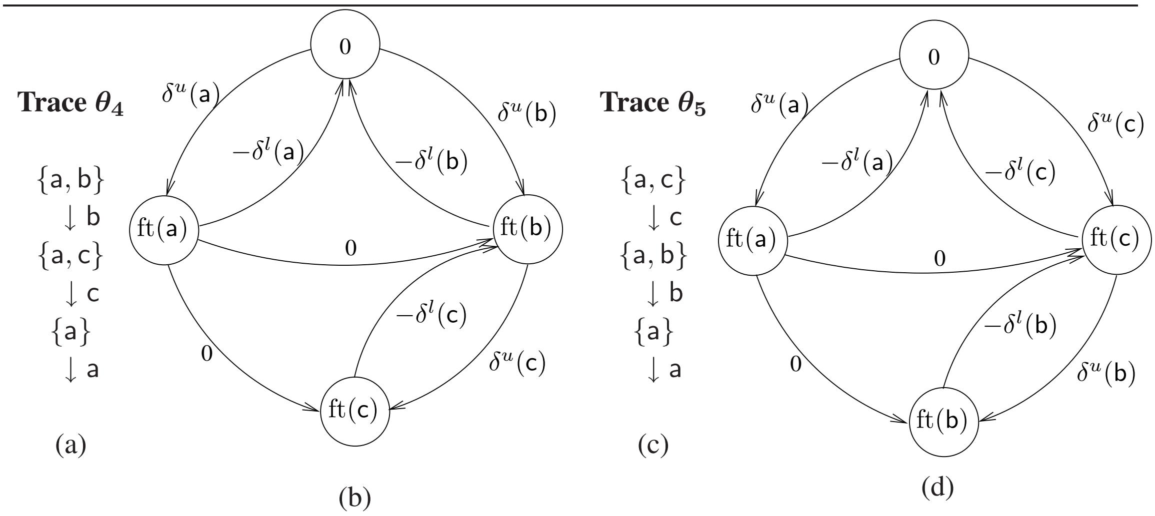


Figure 5.10. Example of two traces whose SPTEs satisfy structural timing inclusion (*i.e.* $\mathrm{gLP}(\theta_4)$ and $\mathrm{gLP}(\theta_5)$ are mutually structurally timing included) but not enabling compatible with each other.

Although we will later show that all enabling compatible traces have their corresponding SPTEs structurally timing included, it is not true that *all* traces whose SPTEs are structurally timing included are enabling compatible. As an example let look at traces θ_4 and θ_5 in Figure 5.10. There is a single representative possible negative cycle in $\mathrm{graph}(\mathrm{gLP}(\theta_4))$, namely:

$$c_1 = 0 \xrightarrow{\delta^u(\mathsf{a})} \mathrm{ft}(\mathsf{a}) \xrightarrow{0} \mathrm{ft}(\mathsf{c}) \xrightarrow{-\delta^l(\mathsf{c})} \mathrm{ft}(\mathsf{b}) \xrightarrow{-\delta^l(\mathsf{b})} 0$$

There is also a single representative possible negative cycle in $\mathrm{graph}(\mathrm{gLP}(\theta_5))$:

$$c_1' = 0 \xrightarrow{\delta^u(\mathsf{a})} \mathrm{ft}(\mathsf{a}) \xrightarrow{0} \mathrm{ft}(\mathsf{b}) \xrightarrow{-\delta^l(\mathsf{b})} \mathrm{ft}(\mathsf{c}) \xrightarrow{-\delta^l(\mathsf{c})} 0$$

Trace θ_4 is structurally timing included in θ_5 because $\mathrm{weight}(c_1) \leq \mathrm{weight}(c_1')$. Similarly θ_5 is structurally timing included in θ_4 because $\mathrm{weight}(c_1') \leq \mathrm{weight}(c_1)$.However it is easy to check that there is no possible enabling-compatible mapping between both traces (either from θ_5 to θ_4 or vice versa) because the first set of enabled events does not satisfy the projection rule, so the beginning of mapped fragment θ_t must be the second set of enabled events, but this leaves mapped fragment with a length of 2 while the other trace has length 3. Then it is not possible to satisfy the completeness rule.

Although enabling compatibility is not as "complete" as structural timing inclusion, it has the advantage that it is easier to modify the LzTS to remove enable compatible traces with a timing inconsistent trace. We explain the details of this procedure in Chapter 6.

To guarantee that enabling compatibility implies structural timing inclusion Definition 5.16 had to define some special initialization rules. We illustrate the reason for this using an example.

EXAMPLE 5.3 *In Figure 5.11, we can see an apparently enabling compatible mapping between traces θ_6 and θ_7 using $k = 1$, $\mathsf{mp}(2) = 1$, $\mathsf{mp}(3) = 2$ and $\mathsf{mp}(4) = 3$. However, by Definition 5.16, the traces are not enabling compatible because in the fragment of θ_6 that is mapped, event* a *is not enabled by* x.

Using the rules, as $k \geq 1$, then it must be satisfied that $E_1 \cap (\mathrm{LB}(E_1') \setminus \{\mathsf{e}_1\}) = \emptyset$ by the initialization rule. After substitution we have the condition $\{\mathsf{x}, \mathsf{a}\} \cap (\mathrm{LB}(\{\mathsf{a}, \mathsf{b}\}) \setminus \{\mathsf{x}\}) = \emptyset$.

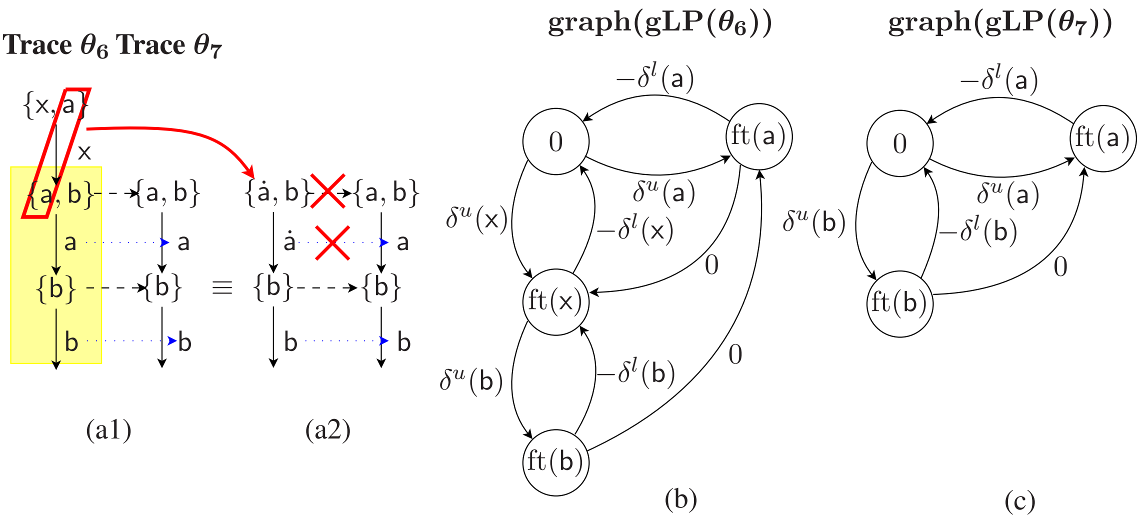


Figure 5.11. Example showing two traces that should not be enabling compatible in order to guarantee structural timing inclusion (if $\delta^l(\mathsf{a}) > 0$).

Assuming that lower delay bounds of events a *and* b *are greater than 0, we have that* $\mathrm{LB}(\{\mathsf{a},\mathsf{b}\}) = \{\mathsf{a},\mathsf{b}\}$ *and* $\{\mathsf{x},\mathsf{a}\} \cap \{\mathsf{a},\mathsf{b}\} = \{\mathsf{a}\} \neq \emptyset$ *which shows that the condition fails, hence both traces cannot be enabling compatible.*

If the initialization rule is not taken into account, then the mapping satisfies the rest of the rules to be an enabling compatible mapping. However, θ_6 *is not structural timing included in* θ_7 *if* $\delta^l(\mathsf{a}) > 0$ *as one can see by looking to their associated graphs. The associated graph of* θ_7 *has a single representative possible negative cycle with weight* $\delta^u(\mathsf{b}) - \delta^l(\mathsf{a})$*, while the associated graph of* θ_6 *has two such cycles with weights* $\delta^u(\mathsf{a}) - \delta^l(\mathsf{x})$ *and* $\delta^u(\mathsf{x}) + \delta^u(\mathsf{b}) - \delta^l(\mathsf{a})$*. None of these two weights is lower than* $\delta^u(\mathsf{b}) - \delta^l(\mathsf{a})$ *for any possible value of the delays, so structural timing inclusion does not exist between* $\mathrm{gLP}(\theta_6)$ *and* $\mathrm{gLP}(\theta_7)$*.* ■ 5.3

Note that in [Peñ03] the enabling compatibility definition failed to solve this problem and trace θ_6 could be defined as enabling compatible with θ_7 no matter which delays were assigned to the events.

Given two sets of traces L and L', we denote by $\mathsf{EnComp}(L, L')$ all the traces of L that are enabling compatible with any of the traces of L'. Formally, $\mathsf{EnComp}(L, L') = \{\theta \in L \mid \exists \theta' \in L' : \theta \geq_{\mathsf{ec}} \theta'\}$. For instance, using some of the traces that have appeared up to this point in Figures 5.8, 5.10 and 5.11, $\mathsf{EnComp}(\{\theta_4, \theta_5, \theta_6, \theta_7\}, \{\theta_1, \theta_2\}) = \{\theta_6, \theta_7\}$. Given that the enabling compatibility relation is a partial order so has the transitive property it is the case that redundant elements might be removed from the set L'. That is $\mathsf{EnComp}(L, L') = \mathsf{EnComp}(L, L'')$ where $L'' = \{\theta \in L' \mid \nexists \theta' \in L' : \theta \geq_{\mathsf{ec}} \theta' \wedge \theta \neq \theta'\}$.

5.6 The link between enabling compatibility and structural timing inclusion

In this section we formalize, in Theorem 5.7, the relationship that is established between two traces that are enabling compatible and the structural timing inclusion of their associated SPTE s. Briefly said, we will show that enabling compatibility implies structural timing inclusion. To achieve this goal, we must introduce several new elements to be used in the proof of the theorem. However, they will also prove useful in several other parts of this work, hence we present them now.

First of all, we define a new type of associated SPTE for traces. For less "aggressive" definitions of enabling compatible mapping, the generic timing-consistency system (*i.e.* $\mathrm{gLP}(\theta)$) is a suitable candidate for associated SPTE of a trace. However, with current definition it is no longer the case, as we will see.

Secondly, although the enabling compatible mapping was defined on traces that were not distinct, we also need a form of mapping that relates the events when the trace is converted into a distinct trace. This is because structural timing inclusion is tested in a setting where traces are considered to be distinct.

Finally, we describe several orderings between the events of two distinct traces that are enabling compatible.These relations allow to have a more intuitive idea of how traces should look like to be enabling compatible and are crucial in the proof of the theorem.

0-delay aware generic timing-consistency systems

The enabling compatible mapping definition uses some delay information. That means that two traces θ and θ' can be enabling compatible or not in function of the delays of their events. An example of this can be found in Figure 5.11, where $\theta_6 \geq_{\mathsf{ec}} \theta_7$ if $\delta^l(\mathsf{a}) = 0$, and are not enabling compatible otherwise. However, $\mathrm{gLP}(\theta_6)$ is not structurally timing included in $\mathrm{gLP}(\theta_7)$ because it considers any possible value for $\delta^l(\mathsf{a})$ and for values greater than 0, clearly it could be the case that θ_7 is timing inconsistent, while θ_6 is not.

To keep up with the enabling compatible mapping definition, the SPTE associated to each trace must include, at least, the same delay information. As Definition 5.16 distinguishes between events with delay bounds equal to 0 or greater than 0, this translates into converting to zeros all the delay variables that are known to have this value in the gLP.

DEFINITION 5.18 (0-SUBSTITUTED GENERIC TIMING-CONSISTENCY SYSTEM)
The 0-substituted generic timing consistency system *of trace θ, noted as $\mathrm{g_0LP}(\theta)$, is the system of equations obtained by the application of Lemma 5.2 to the distinct trace of θ in which the delays of the events are kept as variables except for delays equal to 0, which are kept as 0. $\mathrm{g_0LP}(\theta)$ is a SPTE.* ■ 5.18

Induced mapping on distinct traces

In our definition of enabling compatibility between traces we did not consider the traces to be distinct traces. However, to be able to reason in terms of the associated graph, which uses distinct traces to generate the nodes of the graph, it would be useful to derive a mapping between distinct events from the enabling compatible mapping.

For instance, in Figure 5.12, we can see two traces, namely θ_8 and θ_9, together with their respective distinct traces. A fragment of the distinct trace of θ_9 is shown in subfigure (e). In Figure 5.13 (a) we can see that trace θ_8 is enabling compatible with that fragment. Note that in the fragment of θ_9 there are two conservatively approximated events (namely $\dot{\mathsf{b}}$ and $\dot{\mathsf{c}}$) and that events b and c of θ_8 synchronize with them. That the traces are enabling compatible means that there exists a mapping mp that satisfies Definition 5.16. If we apply the same mapping to the distinct traces of θ_8 and θ_9, we obtain subfigure (b). Let us use Σ_1^d and Σ_2^d to identify the alphabets of the distinct traces of θ_8 and θ_9 respectively. It is straightforward to see that a bijective partial function $\mathsf{mp}_e : \Sigma_1^d \to \Sigma_2^d$ can be derived from the mapping mp when applied to the distinct traces. Given an

Trace θ_8	**Trace θ_9**	**Distinct trace of θ_8**	**Distinct trace of θ_9**	**Fragment of distinct trace of θ_9**
$\{\mathsf{z},\mathsf{b}\}$		$\{\langle\mathsf{z},1\rangle,\langle\mathsf{b},1\rangle\}$		
$\downarrow \mathsf{z}$	$\{\mathsf{z},\mathsf{b}\}$	$\downarrow \langle\mathsf{z},1\rangle$	$\{\langle\mathsf{z},1\rangle,\langle\mathsf{b},1\rangle\}$	
$\{\mathsf{x},\mathsf{b}\}$	$\downarrow \mathsf{b}$	$\{\langle\mathsf{x},1\rangle,\langle\mathsf{b},1\rangle\}$	$\downarrow \langle\mathsf{b},1\rangle$	
$\downarrow \mathsf{x}$	$\{\mathsf{y},\mathsf{b},\mathsf{c}\}$	$\downarrow \langle\mathsf{x},1\rangle$	$\{\langle\mathsf{y},1\rangle,\langle\mathsf{b},2\rangle,\langle\mathsf{c},1\rangle\}$	
$\{\mathsf{b},\mathsf{c}\}$	$\downarrow \mathsf{y}$	$\{\langle\mathsf{b},1\rangle,\langle\mathsf{c},1\rangle\}$	$\downarrow \langle\mathsf{y},1\rangle$	$\vdots$
$\downarrow \mathsf{b}$	$\{\mathsf{b},\mathsf{c}\}$	$\downarrow \langle\mathsf{b},1\rangle$	$\{\langle\mathsf{b},2\rangle,\langle\mathsf{c},1\rangle\}$	$\{\langle\dot{\mathsf{b}},2\rangle,\langle\dot{\mathsf{c}},1\rangle\}$
$\{\mathsf{a},\mathsf{c}\}$	$\downarrow \mathsf{b}$	$\{\langle\mathsf{a},1\rangle,\langle\mathsf{c},1\rangle\}$	$\downarrow \langle\mathsf{b},2\rangle$	$\downarrow \langle\dot{\mathsf{b}},2\rangle$
$\downarrow \mathsf{a}$	$\{\mathsf{c}\}$	$\downarrow \langle\mathsf{a},1\rangle$	$\{\langle\mathsf{c},1\rangle\}$	$\{\langle\dot{\mathsf{c}},1\rangle\}$
$\{\mathsf{d},\mathsf{c}\}$	$\downarrow \mathsf{c}$	$\{\langle\mathsf{d},1\rangle,\langle\mathsf{c},1\rangle\}$	$\downarrow \langle\mathsf{c},1\rangle$	$\downarrow \langle\dot{\mathsf{c}},1\rangle$
$\downarrow \mathsf{c}$	$\{\mathsf{a},\mathsf{b}\}$	$\downarrow \langle\mathsf{c},1\rangle$	$\{\langle\mathsf{a},1\rangle,\langle\mathsf{b},3\rangle\}$	$\{\langle\mathsf{a},1\rangle,\langle\mathsf{b},3\rangle\}$
$\{\mathsf{a},\mathsf{b},\mathsf{d}\}$	$\downarrow \mathsf{a}$	$\{\langle\mathsf{a},2\rangle,\langle\mathsf{b},2\rangle,\langle\mathsf{d},1\rangle\}$	$\downarrow \langle\mathsf{a},1\rangle$	$\downarrow \langle\mathsf{a},1\rangle$
$\downarrow \mathsf{a}$	$\{\mathsf{g}\}$	$\downarrow \langle\mathsf{a},2\rangle$	$\{\langle\mathsf{g},1\rangle\}$	$\{\langle\mathsf{g},1\rangle\}$
$\{\mathsf{f},\mathsf{b}\}$		$\{\langle\mathsf{f},1\rangle,\langle\mathsf{b},2\rangle\}$		
(a)	(b)	(c)	(d)	(e)

Figure 5.12. Example of induced event mapping on enabling compatible distinct traces –first figure: (a,b) Two traces. (c,d) Its corresponding distinct traces. (e) A fragment of distinct trace shown in (d). Note that both events $\langle\mathsf{b},2\rangle$ and $\langle\mathsf{c},1\rangle$ are conservatively approximated because in the complete distinct trace they were enabled before the starting point of the fragment.

event in the distinct trace of θ_8, function mp_e simply gives the corresponding event in the distinct trace of θ_9.

Definition 5.19 (Induced event mapping on distinct traces)
Given a two traces θ_1 and θ_2 such that $\theta_1 \geq_{\mathsf{ec}} \theta_2$ with an enabling compatible mapping mp *such that* $\mathsf{mp}(k+1)=1$ *and* $\mathsf{mp}(k+n+1)=m+1$. *Let $\theta_1^d = E_1 \xrightarrow{\mathsf{e}_1} E_2 \ldots$ and $\theta_2^d = E_1' \xrightarrow{\mathsf{e}_1'} E_2' \ldots$ their corresponding distinct traces with alphabets Σ_1^d and Σ_2^d respectively. Then function $\mathsf{mp}_e : \Sigma_1^d \to \Sigma_2^d$ is defined as:*

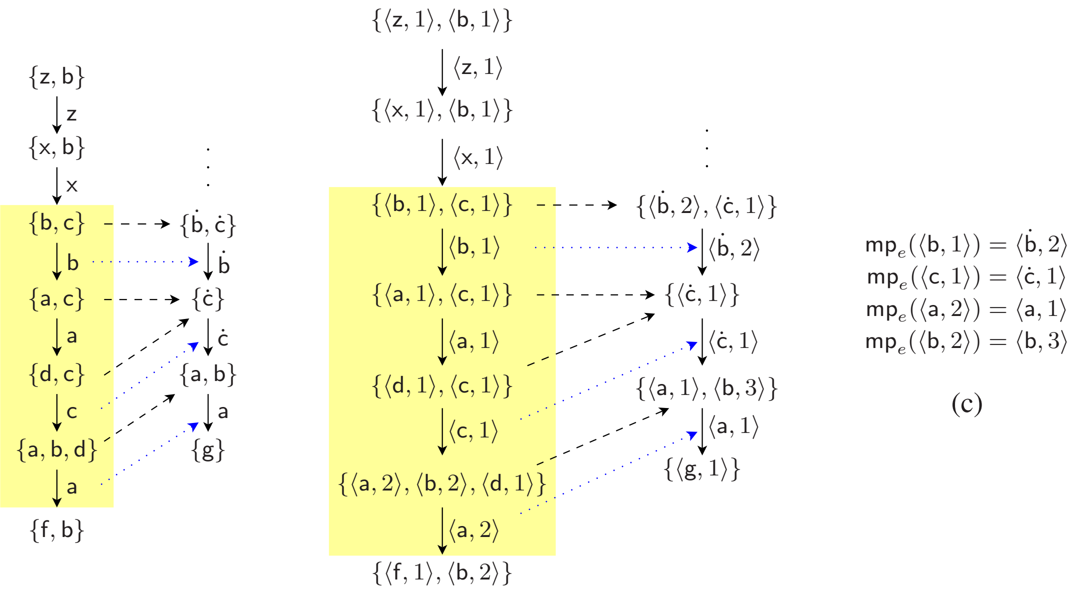
$$\forall i : k+1 \leq i \leq k+n+1,\ \forall \langle \mathsf{e}, \alpha\rangle \in E_i,\ \langle \mathsf{e}, \beta\rangle \in E'_{\mathsf{mp}(i)} \Rightarrow \mathsf{mp}_e(\langle \mathsf{e}, \alpha\rangle) = \langle \mathsf{e}, \beta\rangle$$

Mapping mp_e can be trivially extended to standardized distinct traces by simply adding $\mathsf{mp}_e(\mathsf{e}_k) = \epsilon$. ■ 5.19

For instance, in our example of Figure 5.12, $\mathsf{mp}_e(\langle \mathsf{a},2\rangle) = \langle \mathsf{a},1\rangle$. The complete definition of the function is shown in Figure 5.13 (c). Function mp_e is partial because not all elements of Σ_1^d have a corresponding element in Σ_2^d, for instance it is the case of event $\langle \mathsf{d},1\rangle$. As the function is bijective the inverse function mp_e^{-1} exists. Should have we been working with standardized traces then, for instance, $\mathsf{mp}_e^{-1}(\epsilon) = \langle \mathsf{x},1\rangle$.

Properties of enabling compatible distinct traces

Lemma 5.6 (Properties of enabling compatible distinct traces)
Let θ_1, θ_2 be two traces such that $\theta_1 \geq_{\mathsf{ec}} \theta_2$. Let mp *be the enabling compatible mapping established between both traces and mp_e the event mapping induced by* mp *that is established*



(a) (b)

Figure 5.13. Example of induced event mapping on enabling compatible distinct traces - second figure: (a) Trace θ_8 of Figure 5.12 is enabling compatible with a fragment of θ_9. (b) Using the same mapping but on the distinct traces of θ_8 and θ_9. (c) The bijective mapping that can be established between both distinct traces.

between their corresponding standardized distinct traces $\theta_1^{s,d}$ and $\theta_2^{s,d}$. We will use the terms θ_1^d and θ_2^d to refer to the non-standardized distinct traces. Then the following properties are satisfied:

- *For all events* e *of* θ_2^d *triggered by some event in* θ_2^d $(\mathrm{trigger}(\mathsf{e}, \theta_2^{s,d}) \neq \epsilon)$*,*

$$\textit{if } \delta^l(\mathsf{e}) > 0, \mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_1^d) = \mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}, \theta_2^d)) \tag{5.4}$$

$$\textit{if } \delta^l(\mathsf{e}) = 0, \mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_1^{s,d}) \leq_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}, \theta_2^d)) \tag{5.5}$$

- *For all events* e *of* θ_2^d *not triggered by some event in* θ_2^d $(\mathrm{trigger}(\mathsf{e}, \theta_2^{s,d}) = \epsilon)$*,*

$$\textit{if } \delta^l(\mathsf{e}) > 0, \mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_1^{s,d}) = \mathsf{mp}_e^{-1}(\epsilon) \tag{5.6}$$

$$\textit{if } \delta^l(\mathsf{e}) = 0, \mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_1^{s,d}) \leq_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\epsilon) \tag{5.7}$$

- *Given two fired events* e_i *and* e_j *of* θ_2 *then*

$$\mathsf{e}_i <_{\mathsf{fo}} \mathsf{e}_j \Rightarrow \mathsf{mp}_e^{-1}(\mathsf{e}_i) <_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\mathsf{e}_j) \tag{5.8}$$

- *For all disabled events* e *of* θ_2^d*,*

$$\textit{if } \mathsf{mp}_e^{-1}(\mathsf{e}) \textit{ is disabled in } \theta_1^d, \mathsf{mp}_e^{-1}(\mathrm{disabler}(\mathsf{e}, \theta_2^d)) \leq_{\mathsf{fo}} \mathrm{disabler}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_1^d) \tag{5.9}$$

$$\textit{if } \mathsf{mp}_e^{-1}(\mathsf{e}) \textit{ fires in } \theta_1^d, \mathsf{mp}_e^{-1}(\mathrm{disabler}(\mathsf{e}, \theta_2^d)) <_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\mathsf{e}) \tag{5.10}$$

■ 5.6

Proof: *See Appendix B.1.*

For instance, in Figure 5.12, we have that $\langle \mathsf{b}, 3\rangle$ is an event of distinct trace θ_9^d triggered by $\langle \dot{\mathsf{c}}, 1\rangle$, thus $\mathrm{trigger}(\langle \mathsf{b}, 3\rangle, \theta_9^d) = \langle \dot{\mathsf{c}}, 1\rangle$. Now $\mathsf{mp}_e^{-1}(\mathrm{trigger}(\langle \mathsf{b}, 3\rangle, \theta_9^d)) = \mathsf{mp}_e^{-1}(\langle \dot{\mathsf{c}}, 1\rangle) = \langle \mathsf{c}, 1\rangle$. This satisfies the property (5.4) as $\mathsf{mp}_e^{-1}(\langle \mathsf{b}, 3\rangle) = \langle \mathsf{b}, 2\rangle$ whose trigger is precisely $\langle \mathsf{c}, 1\rangle$. In this particular case $\delta^l(\mathsf{c})$ could be 0 or greater than 0 (property (5.4) is stronger than (5.5)). As an example for property (5.6) we will use events $\langle \dot{\mathsf{b}}, 2\rangle$ and $\langle \dot{\mathsf{c}}, 1\rangle$. For $\langle \dot{\mathsf{c}}, 1\rangle$ we have that $\mathrm{trigger}(\mathsf{mp}_e^{-1}(\langle \dot{\mathsf{c}}, 1\rangle), \theta_1^{s,d}) = \langle \mathsf{x}, 1\rangle$ and that $\mathsf{mp}_e^{-1}(\epsilon) = \langle \mathsf{x}, 1\rangle$. On the other hand $\mathrm{trigger}(\mathsf{mp}_e^{-1}(\langle \dot{\mathsf{b}}, 2\rangle), \theta_1^{s,d}) = \epsilon$ as $\langle \mathsf{b}, 1\rangle$ is already enabled at the beginning of θ_1^d so it is triggered by special event ϵ in the standardized trace. As ϵ is considered the first event fired in the standardized trace, it is verified that $\epsilon <_{\mathsf{fo}} \langle \mathsf{x}, 1\rangle$.

Enabling-compatibility implies structural timing inclusion

Theorem 5.7 (Enabling-compatibility implies str. timing inclusion)
Let θ, θ' be two traces. If θ is enabling-compatible with θ', then $\mathrm{g_0LP}(\theta)$ is structurally timing included in $\mathrm{g_0LP}(\theta')$. ■ 5.7

Proof: *See Appendix B.2.*

This theorem is one of the cornerstones of this work because, once a trace has been shown timing inconsistent, then all the enabling compatible traces can be removed safely from our system, as we know they are also timing inconsistent.

5.7 An alternative view of enabling compatibility

In Definition 5.16 we used a mapping between traces to define enabling compatibility. However, in some cases, it is more convenient to define enabling compatibility in terms of transformations on a trace. Let θ' be a trace obtained from θ by applying any number of the following transformations:

- Prefix removal.
- Modification of last enabling.
- Promotion of events to conservative events.
- Advance the disabling of an event (this can lead to the complete removal from the trace of the disabled event).
- Delay the enabling of events with 0-valued lower delay bound.
- Supression of the firing of a *sink* event. In this context a sink event is an event that does not trigger nor disable any other event.

Then $\theta \geq_{\mathsf{ec}} \theta'$. Because enabling compatibility is transitive, these transformation can be iteratively applied to obtain traces to whom θ is enabling compatible. Moreover, we prove in next lemma that any trace θ' such that $\theta \geq_{\mathsf{ec}} \theta'$ can be obtained by combining these transformations. This provides a useful alternative definition of the enabling compatibility.

Lemma 5.8
A trace θ is enabling compatible with trace θ', if and only if, θ' can be derived from θ by applying any number of the following transformations:

Prefix removal. *Let* θ_1 *be a non-empty distinct trace* $E_1 \xrightarrow{\mathsf{e_1}} E_2 \xrightarrow{\mathsf{e_2}} \dots \xrightarrow{\mathsf{e_n}} E_{n+1}$*. Then, after prefix removal on* θ_1*, we have that* $\theta_1' = E_2' \xrightarrow{\mathsf{e_2'}} \dots \xrightarrow{\mathsf{e_n'}} E_{n+1}'$*, where*

$$\forall i : 2 \le i \le n+1, \;\; E_i' = \{\mathsf{e} \mid \mathsf{e} \in E_i \setminus E_1\} \cup \{\dot{\mathsf{e}} \mid \mathsf{e} \in E_i \cap E_1\}$$

$$\forall i : 2 \le i \le n, \;\; \mathsf{e}_i' = \begin{cases} \dot{\mathsf{e}}_\mathsf{i} & \textit{if } \mathsf{e}_i \in E_1 \\ \mathsf{e}_i & \textit{otherwise} \end{cases} .$$

Modification of last enabling. *Let* $\theta_1 = E_1 \xrightarrow{\mathsf{e_1}} \dots \xrightarrow{\mathsf{e_n}} E_{n+1}$ *be a trace. Then, after the modification of the last enabling we have that* $\theta_1' = E_1 \xrightarrow{\mathsf{e_1}} \dots \xrightarrow{\mathsf{e_n}} E_{n+1}'$ *where* E_{n+1}' *can be any set of events.*

Promotion of events to conservative events. *Let* θ_1 *be a trace. Then, after promotion of event* e *to conservative event* $\dot{\mathsf{e}}$*, we have that* $\theta_1' = \theta_1[\mathsf{e} \leftarrow \dot{\mathsf{e}}]$*, that is, all occurrences of* e *in the trace have been replaced by* $\dot{\mathsf{e}}$*.*

Advance the disabling of an event. *Let* $\theta_1 = E_1 \xrightarrow{\mathsf{e_1}} \dots \xrightarrow{\mathsf{e_{i-1}}} E_i \xrightarrow{\mathsf{e_i}} E_{i+1} \xrightarrow{\mathsf{e_{i+1}}} \dots E_{n+1}$ *be a trace and* e *an event disabled by* e_i*, then* $\theta_1 = E_1 \xrightarrow{\mathsf{e_1}} \dots \xrightarrow{\mathsf{e_{i-1}}} E_i' \xrightarrow{\mathsf{e_i}} E_{i+1} \xrightarrow{\mathsf{e_{i+1}}} \dots E_{n+1}$ *where* $E_i' = E_i \setminus \{\mathsf{e}\}$*.*

Delay the enabling of events with 0-valued lower delay bound. *Let* θ_1 *be a trace such that* $\theta_1 = E_1 \xrightarrow{\mathsf{e_1}} \dots \xrightarrow{\mathsf{e_{i-1}}} E_i \xrightarrow{\mathsf{e_i}} E_{i+1} \xrightarrow{\mathsf{e_{i+1}}} \dots E_{n+1}$ *and* e *an event triggered by* e_{i-1} *such that* $\delta^l(\mathsf{e}) = 0$ *and* $\mathsf{e} \neq \mathsf{e}_i$*. Then* $\theta_1 = E_1 \xrightarrow{\mathsf{e_1}} \dots \xrightarrow{\mathsf{e_{i-1}}} E_i' \xrightarrow{\mathsf{e_i}} E_{i+1} \xrightarrow{\mathsf{e_{i+1}}} \dots E_{n+1}$ *where* $E_i' = E_i \setminus \{\mathsf{e}\}$*.*

Supression of the firing of a sink event. *Let* $\theta_1 = E_1 \dots \xrightarrow{\mathsf{e_{i-1}}} E_i \xrightarrow{\mathsf{e_i}} E_{i+1} \xrightarrow{\mathsf{e_{i+1}}} \dots E_{n+1}$ *be a trace and* e_i *a fired event of* θ_1 *such that* $\nexists \mathsf{e} : \mathrm{trigger}(\mathsf{e}) = \mathsf{e}_i \vee \mathrm{disabler}(\mathsf{e}) = \mathsf{e}_i$*. Then* $\theta_1' = E_1 \xrightarrow{\mathsf{e_1}} \dots \xrightarrow{\mathsf{e_{i-1}}} E_{i+1} \xrightarrow{\mathsf{e_{i+1}}} \dots E_{n+1}$*.*

■ 5.8

Proof: *See Appendix B.3.*

Chapter 6

REMOVING ENABLING COMPATIBLE TRACES

Summary

A given timing inconsistent trace is timing consistent up to the point where an event is fired in the wrong order (*i.e.* another event should have fired before it). The firing of this "badly ordered" event should be disallowed. We will show how the LzTS of the system under analysis can be modified in a way that all enabling compatible traces with the timing inconsistent one are removed together with the original one.

We will build an *enabling compatible trace identifier automaton* (ECTIA) which accepts the original failure trace (or a fragment of it which still contains the timing inconsistency) as well as all enabling compatible traces with it. Then we will see how we can compute the product of the original LzTS and the trace identifier automaton in order to produce another LzTS in which we prune, not only the original trace from which we derived the trace identifier automaton, but also all enabling compatible traces with that trace.

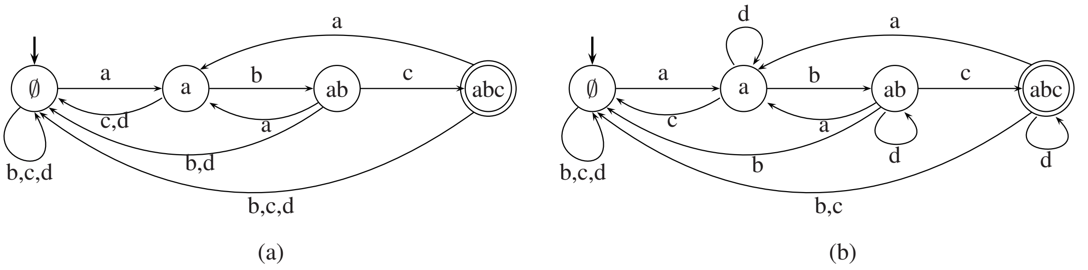


Figure 6.1. Example relating enabling compatible trace detection with word recognition using automata: (a) Automaton accepting words over the alphabet $\{a, b, c, d\}$ that end in abc. (b) Automaton accepting words over the alphabet $\{a, b, c, d\}$ that end with a suffix satisfying the regular expression $a(d^*)b(d^*)c(d^*)$.

6.1 Introduction

Given a trace θ, the construction of an automaton that recognizes the enabling compatible trace with θ is a similar problem to the recognition of a word. For instance imagine we want to build an automaton which accepts any word which ends with abc over the alphabet $\Sigma = \{a, b, c, d\}$. The corresponding automaton is depicted in Figure 6.1 (a). The name of each state represents the amount of sequence that has been identified up to this point. When an unexpected symbol appears we must go back to state $\emptyset$, except if the unexpected symbol coincides with the first symbol in the word (symbol a in the example). In that case we must go to the second state of the automaton, as the first symbol has been already parsed.

Now imagine that instead of the suffix abc we want to identify any sequence that ends in abc but can have an arbitrary number of d symbols in any position inside the suffix. This corresponds to the regular expression $a(d^*)b(d^*)c(d^*)$. The corresponding automaton is depicted in Figure 6.1 (b). We can see that the symbol d does not change the state of the automaton. To recognise an enabling compatible trace we have more or less the same problem: we must ensure that all fired events in the recognised trace θ are always present but a number of concurrent events can be fired in between these obligatory events. Obviously our problem is harder because other elements besides the symbols must be checked in order to guarantee the enabling compatibility, but the general idea of the solution will be the same.

We selected first to introduce the problem in terms of word recognition in order to be able to point out some subtleties that otherwise can be hidden by the additional complexity of the enabling compatibility detection. For instance imagine that you want to recognise all words that end with the regular expression $a(d^*)b(d^*)a(d^*)b(d^*)c(d^*)$. If we apply the same construction schema as in Figure 6.1 (b) we end up with the automaton shown in Figure 6.2 (a). However this automaton fails to recognise the sequence $abababc$ although it clearly satisfies the regular expression. The problem is located in state ab. At that point when an a is encountered we proceed to state aba, however we should also consider the case that this a can be the first one in a $ababac$ sequence for instance. As a result we must introduce an extra arc which can also go to state a from state ab when an a is encountered. The resulting automaton is non deterministic and can be seen in Figure 6.2 (b). If the automaton is determinized using the usual subset construction we obtain the deterministic automaton of Figure 6.2 (c).

In our example, because of its simplicity, we could have directly derived the deterministic automaton using the Aho-Corasick construction method [AC75]. However when applied to our particular problem, which is harder, we should use the same procedure as in our example: add

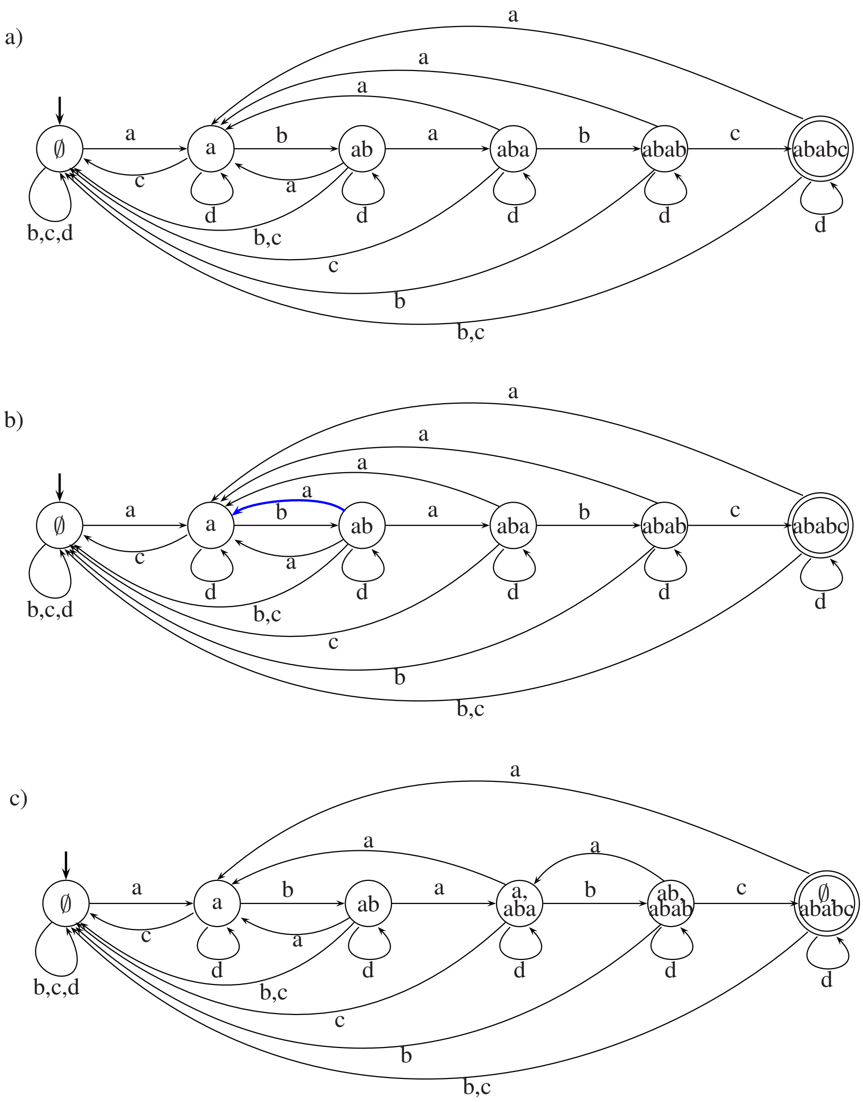


Figure 6.2. Example of determinization issues arising in automata accepting words: (a) Deterministic automaton trying to accept words over the alphabet $\{a, b, c, d\}$ that end with a suffix satisfying the regular expression $a(d^*)b(d^*)a(d^*)b(d^*)c(d^*)$. It only works partially: it fails for instance with sequence $abababc$. (b) Non-deterministic automaton which effectively recognises the desired language, the only difference is the addition of the blue arc which renders the automaton non-deterministic. (c) Deterministic version obtained after subset construction.

the indeterministic arcs that are needed, then determinize and finally possibly minimize. We will later see that the product between the LzTS and the automaton can be performed with a non-deterministic automaton. However there is a small technical problem in doing so: in the product automaton we will have the two possible behaviors, using our simple example, one trace accepting the string $abababc$ and one not accepting it. In our real problem we will have states in which there will be arcs that will not be allowed to fire, and we want them not to be fireable in any of the two traces. However if the automaton is not determinized it is not possible to render an event not fireable for the two traces without removing some valid traces. We will come back to this point in Chapter 10.

6.2 Construction of the Enabling Compatible Trace Identifier Automaton

In our case the automaton we will build will not check for symbols in a word. Instead it will check for arcs in a trace. An arc of a LzTS can be uniquely identified when three elements are given: the source state, the event that labels the arc and the destination state. To identify a set of arcs we can also use three elements: the set of source states, the set of events that can appear in the labels of the arcs and the set of destination states. Given a set of arcs denoted as $\langle \mathsf{S}_s, \Sigma', \mathsf{S}_d \rangle$ this expression represents the following set: $\{\mathsf{s}_s \xrightarrow{\mathsf{e}} \mathsf{s}_d \mid \mathsf{s}_s \in \mathsf{S}_s \wedge \mathsf{s}_d \in \mathsf{S}_d \wedge \mathsf{e} \in \Sigma'\}$.

We want to construct an automaton that identifies all enabling compatible traces with trace $\theta' = E'_1 \xrightarrow{\mathsf{e}'_1} E'_2 \xrightarrow{\mathsf{e}'_2} \cdots E'_m \xrightarrow{\mathsf{e}'_m} E'_{m+1}$. Our automaton will have nodes that will identify the current position of the enabling compatible mapping. For this reason we will have a special node $\perp$ that represents the fact that the mapping has not started. Then for each set of enabled events in θ' such that the projection rule must be satisfied there will be a node called *configuration* and denoted $\mathcal{C}_i$ (that is for sets $E'_1 \ldots E'_m$ we will have configurations $\mathcal{C}_1 \ldots \mathcal{C}_m$). For instance assume we want to build the automaton for trace $\theta'_1 = \{\mathsf{c}, \mathsf{g}\} \xrightarrow{\mathsf{c}} \{\mathsf{g}\} \xrightarrow{\mathsf{g}}$. Assume that we obtained trace θ'_1 from the system of Figure 5.2 (a), although there is no such trace in the system. In this case we will use a synthetic trace to simplify the example. Anyway although θ'_1 does not belong to the system, the system contains enabling compatible traces with θ'_1, so it is still valuable to use it as example. Using trace θ'_1 the nodes of our automaton will be $\{\perp, \mathcal{C}_1, \mathcal{C}_2\}$. We assign each set of enabled events in $E'_1 \ldots E'_m$ to its corresponding configuration by defining $\mathcal{E}(\mathcal{C}_i) = E'_i$. By convention $\mathcal{E}(\perp) = \emptyset$. Now we will add arcs between these nodes, labeled with sets of arcs. Only the arcs of traces satisfying these labels will be allowed to take the corresponding arcs in the automaton.

Formally, an ECTIA obtained from a trace $\theta' = E'_1 \xrightarrow{\mathsf{e}'_1} E'_2 \xrightarrow{\mathsf{e}'_2} \cdots E'_m \xrightarrow{\mathsf{e}'_m}$ in LzTS $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR} \rangle$ is a TS $E = \langle S', \Sigma', T', \mathsf{s}'_0 \rangle$ such that $S' = \{\perp\} \cup \{\mathcal{C}_i \mid \forall i : 1 \leq i \leq m\}$, $\Sigma' = T$ and $\mathsf{s}'_0 = \perp$. To ease the description of the transitions in T' we must first define some auxiliary functions:

- Given a set of events E the function SE returns all the states which belong to the enabling region of all the events contained in E. Formally:

$$\mathrm{SE}(E) = \bigcap_{\forall \mathsf{e} \in E} \mathrm{ER}(\mathsf{e})$$

  We define this set to be S (S is the complete states space of LzTS A) if $E = \emptyset$, that is $\mathrm{SE}(\emptyset) = S$.

- Given a set of events E the function SD returns all the states which do not belong to any of the enabling regions of the events contained in E. Formally:

$$\mathrm{SD}(E) = \bigcap_{\forall \mathsf{e} \in E} \overline{\mathrm{ER}(\mathsf{e})}$$

  We define this set to be S if $E = \emptyset$, that is $\mathrm{SD}(\emptyset) = S$.

Now we can define all the transitions in T' (the transitions of the ECTIA):

Transitions starting the mapping. Remeber we are working with a trace θ' that has been found in a LzTS $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR} \rangle$. These transitions must check that all events that are not conservative (that is all events in E'_1 whose lower delay bound is not 0) are not enabled previously. To satisfy the projection rule, all events in E'_1 must be enabled after taking the transition. The general case corresponds to transitions in set $\langle \mathrm{SD}(\mathrm{LB}(\mathcal{E}(\mathcal{C}_1))), \Sigma \setminus \mathrm{LB}(\mathcal{E}(\mathcal{C}_1)), \mathrm{SE}(\mathcal{E}(\mathcal{C}_1)) \rangle$. Additionally we must consider the special case in which an event is not selfdisabling (*i.e.* an event fires and, immediately after that, is enabled again). In that circumstance the different occurrences of the same event must be considered as distinct entities and we should not enforce that the event that is fired to enter $\mathcal{C}_1$ is disabled previously, even if it is not conservative.

Transition	Conditions
$\perp \xrightarrow{\textbf{enter}} \mathcal{C}_1$	$\textbf{enter} = \langle \mathrm{SD}(\mathrm{LB}(\mathcal{E}(\mathcal{C}_1))), \Sigma \setminus \mathrm{LB}(\mathcal{E}(\mathcal{C}_1)), \mathrm{SE}(\mathcal{E}(\mathcal{C}_1)) \rangle \cup$ $\cup \ \{\langle \mathrm{SD}(\mathrm{LB}(\mathcal{E}(\mathcal{C}_1)) \setminus \{\mathsf{e}\}) \cap \mathrm{SE}(\{\mathsf{e}\}), \{\mathsf{e}\}, \mathrm{SE}(\mathcal{E}(\mathcal{C}_1)) \rangle \mid \forall \mathsf{e} \in \mathrm{LB}(\mathcal{E}(\mathcal{C}_1))\}$

There is also another special case when all events in E'_1 have a minimum delay value of 0 (either because they are conservative or because of the original delay bounds), as then it can be that an initial state satisfies the projection rule in E'_1, but has no incoming transition, thus it would never be mapped to E'_1 although it should. We would treat this case when performing the composition of the automaton with the LzTS (see §6.6) but, basically, the idea is that some traces will not start in recognition state equal to configuration $\perp$, instead they will directly start at configuration $\mathcal{C}_1$.

Transitions advancing the mapping. In node $\mathcal{C}_i$, when there is a transition with the same label as the current fired event in θ', then we can advance to next node $\mathcal{C}_{i+1}$ given that all events newly enabled in $\mathcal{C}_{i+1}$ which have a lower delay bound greater than 0 are disabled in $\mathcal{C}_i$. We must deal with the possibility that event e'_i is not selfdisabling, hence the set of newly enabled events in $\mathcal{C}_{i+1}$ cannot be simply computed as $\mathcal{E}(\mathcal{C}_{i+1}) \setminus \mathcal{E}(\mathcal{C}_i)$. Instead the expression $\mathcal{E}(\mathcal{C}_{i+1}) \setminus (\mathcal{E}(\mathcal{C}_i) \setminus \{\mathsf{e}'_i\})$ must be used.

Transition	Conditions
$\mathcal{C}_i \xrightarrow{\textbf{advance}_\mathbf{i}} \mathcal{C}_{i+1}$	$\textbf{advance}_\mathbf{i} = \langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)) \cap \mathrm{SD}(\mathrm{LB}(\mathcal{E}(\mathcal{C}_{i+1}) \setminus (\mathcal{E}(\mathcal{C}_i) \setminus \{\mathsf{e}'_i\}))), \{\mathsf{e}'_i\}, \mathrm{SE}(\mathcal{E}(\mathcal{C}_{i+1})) \rangle$
$\mathcal{C}_m \xrightarrow{\textbf{advance}_\mathbf{m}} \perp$	$\textbf{advance}_\mathbf{m} = \langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_m)), \{\mathsf{e}'_m\}, S \rangle \ \cap \ \overline{\textbf{enter}}$

There is a special case for configuration $\mathcal{C}_m$ because in that configuration to *advance* in the mapping also means to go to $\perp$. As we are only interested in marking as lazy arcs the timing inconsistent arcs, once the last arc in the mapping is seen, there is no advantage in creating a final configuration and then from that configuration go again to the initial configuration. Hence we avoid this useless configuration by using directly $\perp$ as the destination. Note that this still allows the last arc to be declared as lazy if the timing analysis requires it. The reason to consider only the transitions that do not start the mapping (*i.e.* we have an intersection with $\overline{\textbf{enter}}$ in $\textbf{advance}_\mathbf{m}$) is that we only want to send to the "nothing recognized" configuration the arcs that cannot be used to start again the mapping (see the *Transitions restarting the mapping* section).

Transitions keeping the mapping. In node $\mathcal{C}_i$, when there is a transition with a different label than the ones in E'_i, then we can keep the mapping in the same node $\mathcal{C}_i$ if the enablings are still preserved. Even if the enablings are preserved, in the case where the event fired is in E'_i the mapping cannot be kept, because we would have a violation of the enabling preservation rule of the enabling compatible mapping.

Transition	Conditions
$\mathcal{C}_i \xrightarrow{\textbf{keep}_\mathbf{i}} \mathcal{C}_i$	$\textbf{keep}_\mathbf{i} = \langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)), \Sigma \setminus \mathcal{E}(\mathcal{C}_i), \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)) \rangle$
$\perp \xrightarrow{\textbf{keep}_\mathbf{0}} \perp$	$\textbf{keep}_\mathbf{0} = \overline{\textbf{enter}}$

Transitions restarting the mapping. From a node $\mathcal{C}_i$ we can restart the mapping if the conditions for starting the mapping are also satisfied. When there are arcs in this set who also belong to the transitions that keep or advance the mapping then we have indeterminacy. Although is very frequent that such indeterminacy can arise when the ECTIA is considered alone, fortunately, while

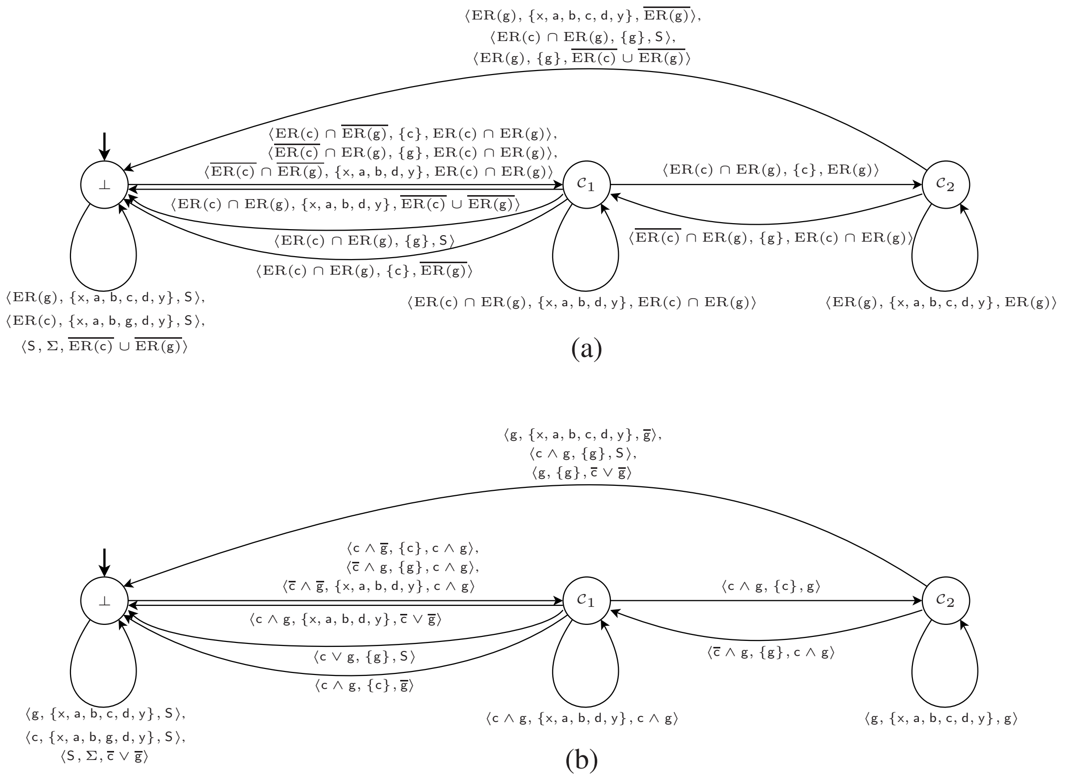


Figure 6.3. Example of an ECTIA: (a) An ECTIA obtained from trace $\theta'_1 = \{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}\{\mathsf{g}\}\xrightarrow{\mathsf{g}}$ (in which $\delta^l(\mathsf{c}) > 0$ and $\delta^l(\mathsf{g}) > 0$). (b) Representation of the same ECTIA using the simplified notation for sets of transitions.

composing with LzTS A the indeterministic arcs are very seldom taken. Indeterminacy will be discussed in Chapter 10.

Transition	**Conditions**
$\mathcal{C}_i \xrightarrow{\mathbf{reenter_i}} \mathcal{C}_1$	$\mathbf{reenter_i} = \langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)), \Sigma, S\rangle \cap \mathbf{enter}$

Transitions breaking the mapping. All transitions from node $\mathcal{C}_i$ such that they do not fall in any of the preceeding cathegories simply break the mapping and lead to node $\perp$. We do not use $\mathbf{reenter_i}$ arcs because they are already included in $\mathbf{enter}$ arcs (*i.e.* $\forall i, \mathbf{reenter_i} \subseteq \mathbf{enter}$).

Transition	**Conditions**
$\mathcal{C}_i \xrightarrow{\mathbf{break_i}} \perp$	$\mathbf{break_i} = \langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)), \Sigma, S\rangle \cap \overline{(\mathbf{enter} \cup \mathbf{keep_i} \cup \mathbf{advance_i})}$

6.3 Drawing an ECTIA

We can see an example of an ECTIA in Figure 6.3. To compute some of the arcs of the automaton we need to obtain the complementation of a set of arcs $T = \langle \mathsf{S}_s, \Sigma', \mathsf{S}_d\rangle$ which can be computed as the set $\overline{T} = \langle \overline{\mathsf{S}_s}, \Sigma, S\rangle \cup \langle S, \overline{\Sigma'}, S\rangle \cup \langle S, \Sigma, \overline{\mathsf{S}_d}\rangle$. Recall that here S represents the full state space and Σ all the alphabet of events. For example we will show how to compute the arc $\mathcal{C}_2 \xrightarrow{\mathbf{advance_2}} \perp$ in Figure 6.3. This corresponds to the last transition advancing the mapping, hence from table we know that it is computed as $\langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_2)), \{\mathsf{e}'_2\}, S\rangle \cap \overline{\mathbf{enter}}$. So first of all we need to find out which transitions satisfy $\overline{\mathbf{enter}}$. $\mathbf{enter}$ can be directly deduced from table as $\mathrm{LB}(\mathcal{E}(\mathcal{C}_1)) = \{\mathsf{c},\mathsf{g}\}$

in our case, so

$$\mathbf{enter} = \langle \overline{\mathrm{ER}(\mathsf{c})} \cap \overline{\mathrm{ER}(\mathsf{g})}, \{\mathsf{x},\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{y}\}, \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g})\rangle \cup$$
$$\langle \mathrm{ER}(\mathsf{c}) \cap \overline{\mathrm{ER}(\mathsf{g})}, \{\mathsf{c}\}, \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g})\rangle \cup \langle \overline{\mathrm{ER}(\mathsf{c})} \cap \mathrm{ER}(\mathsf{g}), \{\mathsf{g}\}, \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g})\rangle.$$

To obtain $\overline{\mathbf{enter}}$ we have to compute

$$\overline{\langle \overline{\mathrm{ER}(\mathsf{c})} \cap \overline{\mathrm{ER}(\mathsf{g})}, \{\mathsf{x},\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{y}\}, \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g})\rangle \cup}$$
$$\overline{\langle \mathrm{ER}(\mathsf{c}) \cap \overline{\mathrm{ER}(\mathsf{g})}, \{\mathsf{c}\}, \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g})\rangle \cup}$$
$$\overline{\langle \overline{\mathrm{ER}(\mathsf{c})} \cap \mathrm{ER}(\mathsf{g}), \{\mathsf{g}\}, \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g})\rangle}$$

which we can transform into

$$\overline{\langle \overline{\mathrm{ER}(\mathsf{c})} \cap \overline{\mathrm{ER}(\mathsf{g})}, \{\mathsf{x},\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{y}\}, \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g})\rangle} \cap$$
$$\overline{\langle \mathrm{ER}(\mathsf{c}) \cap \overline{\mathrm{ER}(\mathsf{g})}, \{\mathsf{c}\}, \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g})\rangle} \cap$$
$$\overline{\langle \overline{\mathrm{ER}(\mathsf{c})} \cap \mathrm{ER}(\mathsf{g}), \{\mathsf{g}\}, \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g})\rangle}$$

by applying De Morgan laws. Now we apply the definition of the complement of a set of transitions and we obtain:

$$(\langle \mathrm{ER}(\mathsf{c}) \cup \mathrm{ER}(\mathsf{g}), \Sigma, S\rangle \cup \langle S, \{\mathsf{c},\mathsf{g}\}, S\rangle \cup \langle S, \Sigma, \overline{\mathrm{ER}(\mathsf{c})} \cup \overline{\mathrm{ER}(\mathsf{g})}\rangle) \cap$$
$$(\langle \overline{\mathrm{ER}(\mathsf{c})} \cup \mathrm{ER}(\mathsf{g}), \Sigma, S\rangle \cup \langle S, \{\mathsf{x},\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{g},\mathsf{y}\}, S\rangle \cup \langle S, \Sigma, \overline{\mathrm{ER}(\mathsf{c})} \cup \overline{\mathrm{ER}(\mathsf{g})}\rangle) \cap$$
$$(\langle \mathrm{ER}(\mathsf{c}) \cup \overline{\mathrm{ER}(\mathsf{g})}, \Sigma, S\rangle \cup \langle S, \{\mathsf{x},\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{y}\}, S\rangle \cup \langle S, \Sigma, \overline{\mathrm{ER}(\mathsf{c})} \cup \overline{\mathrm{ER}(\mathsf{g})}\rangle)$$

Finally, after applying distributivity and simplifying, we get this result

$$\langle \mathrm{ER}(\mathsf{g}), \{\mathsf{x},\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{y}\}, S\rangle \cup \langle \mathrm{ER}(\mathsf{c}), \{\mathsf{x},\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{g},\mathsf{y}\}, S\rangle \cup \langle S, \Sigma, \overline{\mathrm{ER}(\mathsf{c})} \cup \overline{\mathrm{ER}(\mathsf{g})}\rangle$$

Last step to obtain $\mathbf{advance_2}$ is simply to intersect this last result with set $\langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_2)), \{\mathsf{e}'_m\}, S\rangle$. As $\mathrm{SE}(\mathcal{E}(\mathcal{C}_2)) = \mathrm{ER}(\mathsf{g})$ and $\mathsf{e}'_2 = \mathsf{g}$ the set is $\langle \mathrm{ER}(\mathsf{g}), \{\mathsf{g}\}, S\rangle$ and the intersection gives $\langle \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g}), \{\mathsf{g}\}, S\rangle \cup \langle \mathrm{ER}(\mathsf{g}), \{\mathsf{g}\}, \overline{\mathrm{ER}(\mathsf{c})} \cup \overline{\mathrm{ER}(\mathsf{g})}\rangle$.

In order to ease the reading we introduce an alternative notation to express sets of transitions. As most of the sets of states are described in terms of the enabling regions of the events but the ER syntax is very cumbersome we will simply omit them. Hence for instance the set of transitions $\langle \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g}), \{\mathsf{g}\}, S\rangle \cup \langle \mathrm{ER}(\mathsf{g}), \{\mathsf{g}\}, \overline{\mathrm{ER}(\mathsf{c})} \cup \overline{\mathrm{ER}(\mathsf{g})}\rangle$ will be written as $\langle \mathsf{c} \wedge \mathsf{g}, \{\mathsf{g}\}, S\rangle \cup \langle \mathsf{g}, \{\mathsf{g}\}, \overline{\mathsf{c}} \vee \overline{\mathsf{g}}\rangle$. In Figure 6.3 (b) we can see the same ECTIA of subfigure (a) but using this simplified notation.

Usually if several arcs have the same source and destination configurations but different labels, they will be drawn as a single arc with all the labels separated by commas following standard automaton notation. For instance this is the case of the arc going from configuration $\perp$ to $\mathcal{C}_1$ in Figure 6.3 which represents in fact three different sets of arcs. However in many cases we will not "compress" arcs because we need to be able to distinguish between them in order to apply the results of the timing analysis. For instance, all events that break the mapping but are enabled in the configuration are drawn independently (*e.g.* arc $\mathcal{C}_1 \longrightarrow \perp$ with label $\langle \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{g}), \{\mathsf{g}\}, \mathsf{S}\rangle$ in Figure 6.3). These arcs must be drawn separatedly because it can be the case that some of them must be defined as lazy after the timing analysis while some others are still valid. We provide an example of this in §6.5.

To specify an ECTIA, even for a very simple trace, is very cumbersome. For this reason we will usually work with the traces in this document, as it is frequently easier to check if two traces are enabling compatible than to build the automaton. However, the ECTIA formalism is needed to perform the removal of the enabling compatible traces, as we will see in the following sections. Furthermore, it allows to reason safely when complex cases like indeterminacy arise. In [Peñ03] the ECTIA construction overhead was avoided by defining a much more complex product operation, however this had the sideeffect of making the treatment of possible indeterminizations much more difficult. Actually, indeterminacy was not handled. See §A.5 for further details.

At this point we do not yet provide an example to show how indeterminacy is handled by our methodology. We will delay our discussion on the subject until the basic verification flow has presented. Just as a short introduction to the problem we will say that the ECTIA definition we have given usually produces indeterministic automatons except for the most simple cases (actually the ECTIA of Figure 6.3 is very simple). However in original systems such indeterminacy hardly ever appears, hence it seems reasonable to design a procedure to test if the determinization effort is really needed in order to avoid an unnecessary overhead. Indeterminacy will be discussed in depth in Chapter 10.

6.4 The link between enabling compatibility and ECTIA state

In this section we formalize the relationship that is established between a trace that has been recognised by an ECTIA up to a particular configuration and the trace (or sets of traces) used to build the ECTIA. Although the ECTIA construction method presented in §6.2 only considers the situation where we have a single trace to generate the ECTIA, in later stages of this work a set of traces will be used instead (§7.3.1) hence we will formalize the link between the enabling compatibility of traces and the recognition state of the ECTIA considering this more general context to avoid redoing the work once the extended ECTIA construction method has been presented.

Before presenting the corresponding theorem some auxiliary definitions must be introduced. For instance we will assign a set of traces to each configuration and each combination of configuration and event of the ECTIA. The idea is to define the traces to whom the trace checked by the ECTIA is enabling compatible in terms of these auxiliary sets.

We will note the set of traces assigned to configuration $\mathcal{C}_i$ as $\mathrm{traces}(\mathcal{C}_i)$ and we will recursively define it in terms of the sets of traces of its predecessor configurations except for configurations $\perp$ and $\mathcal{C}_1$ (this last configuration is noted as $\top$ when the ECTIA is generated using the extended method). For these two configurations we have that $\mathrm{traces}(\perp) = \emptyset$ and $\mathrm{traces}(\mathcal{C}_1) = \{\mathcal{E}(\mathcal{C}_1)\}$. The definition for $\mathrm{traces}(\mathcal{C}_1)$ is not exactly a set of traces because it only contains an enabling and no fired event, but this irregularity allows to treat more uniformly the rest of configurations $\mathcal{C}_i$. In this case

$$\begin{aligned}\mathrm{traces}(\mathcal{C}_i) = \{\theta \xrightarrow{\mathsf{e}} \mathcal{E}(\mathcal{C}_i) \mid \forall \mathcal{C}_j : \mathcal{C}_j \xrightarrow{\langle \mathsf{S}_s, \Sigma, \mathsf{S}_d \rangle} \mathcal{C}_i \in T \wedge \mathcal{C}_i \notin \{\perp, \mathcal{C}_1, \mathcal{C}_j\},\\ \forall \theta \in \mathrm{traces}(\mathcal{C}_j), \forall \mathsf{e} \in \Sigma \cap \mathcal{E}(\mathcal{C}_j)\}\end{aligned}$$

The set of traces associated to a combination of configuration and event $(\mathcal{C}_i, \mathsf{e})$ is defined as

$$\mathrm{traces}(\mathcal{C}_i, \mathsf{e}) = \begin{cases} \mathrm{traces}(\mathcal{C}_i) & \text{if } \mathsf{e} \notin \mathcal{E}(\mathcal{C}_i) \\ \{\theta' \xrightarrow{\mathsf{e}} \mid \forall \theta' \in \mathrm{traces}(\mathcal{C}_i)\} & \text{if } \mathsf{e} \in \mathcal{E}(\mathcal{C}_i) \end{cases}$$

THEOREM 6.1 (ECTIA RECOGNITION STATE AND ENABLING COMPATIBILITY)
Let $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \ldots E_n \xrightarrow{\mathsf{e}_n} E_{n+1}$ be a trace of a LzTS A generated by a run starting at some initial state of A and $\mathrm{EC} = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$ be an ECTIA. Let configuration $\mathcal{C}$ be the recognition state of EC after having checked the trace fragment $E_1 \xrightarrow{\mathsf{e}_1} E_2 \ldots E_n$.

If configuration $\mathcal{C}$ is different than $\perp$, then there exists some trace in $\mathrm{traces}(\mathcal{C}, \mathsf{e}_n)$ *such that* θ *is enabling compatible with it. Formally,* $\exists\theta' \in \mathrm{traces}(\mathcal{C}, \mathsf{e}_n), \theta \geq_{\mathsf{ec}} \theta'$. ■ 6.1

Proof: *See Appendix B.4.*

To illustrate the theorem let us check that it works on the example of Figure 6.3. For instance the values of auxiliary functions are as follows:

$$\mathrm{traces}(\perp) = \emptyset, \mathrm{traces}(\mathcal{C}_1) = \{\{\mathsf{c},\mathsf{g}\}\}, \mathrm{traces}(\mathcal{C}_2) = \{\{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}\{\mathsf{g}\}\}$$

and

$$\begin{aligned}
&\forall \mathsf{e} \in \Sigma, \quad \mathrm{traces}(\perp, \mathsf{e}) = \emptyset \\
&\forall \mathsf{e} \in \Sigma \setminus \{\mathsf{c},\mathsf{g}\}, \quad \mathrm{traces}(\mathcal{C}_1, \mathsf{e}) = \{\{\mathsf{c},\mathsf{g}\}\} \\
&\mathrm{traces}(\mathcal{C}_1, \mathsf{c}) = \{\{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}\} \\
&\mathrm{traces}(\mathcal{C}_1, \mathsf{g}) = \{\{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{g}}\} \\
&\forall \mathsf{e} \in \Sigma \setminus \{\mathsf{g}\}, \quad \mathrm{traces}(\mathcal{C}_2, \mathsf{e}) = \{\{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}\{\mathsf{g}\}\} \\
&\mathrm{traces}(\mathcal{C}_2, \mathsf{g}) = \{\{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}\{\mathsf{g}\}\xrightarrow{\mathsf{g}}\}
\end{aligned}$$

According to the theorem, trace $\{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b}\}\xrightarrow{\mathsf{a}}\{\mathsf{b},\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}\{\mathsf{g},\mathsf{b}\}\xrightarrow{\mathsf{b}}\{\mathsf{g}\}$ is enabling compatible with $\{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}\{\mathsf{g}\}$ because ECTIA starts in $\perp$ (both events c and g are not conservative nor they have a 0 lower delay bound) and remains in that configuration after the firing of x. Then ECTIA goes to $\mathcal{C}_1$ by the transition $\{\mathsf{a},\mathsf{b}\}\xrightarrow{\mathsf{a}}\{\mathsf{b},\mathsf{c},\mathsf{g}\}$, then to $\mathcal{C}_2$ by the firing of c that keeps event g enabled. This means that the configuration reached before firing the last event in the trace is configuration $\mathcal{C}_2$ that has an assigned set of traces $\mathrm{traces}(\mathcal{C}_2) = \{\{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}\{\mathsf{g}\}\}$. As $\mathsf{b} \notin \mathcal{E}(\mathcal{C}_2)$ we know that $\mathrm{traces}(\mathcal{C}_2, \mathsf{b}) = \mathrm{traces}(\mathcal{C}_2)$. Hence the given trace is enabling compatible with $\{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}\{\mathsf{g}\}$ as expected.

THEOREM 6.2 (ENABLING COMPATIBILITY ENFORCES **ECTIA** STATE)
Let $\theta = E_1\xrightarrow{\mathsf{e}_1}E_2 \dots E_n\xrightarrow{\mathsf{e}_n}E_{n+1}$ *be a trace of a LzTS* A *generated by a run starting at some initial state of* A *and* EC *be an ECTIA. If exists* θ_s *such that* θ_s *is a suffix of* θ *and* θ_s *is fully enabling compatible with some trace* θ' *that belongs to* $\mathrm{traces}(\mathcal{C}_i)$*, then when* θ *is checked by* EC *its recognition state must include* $\mathcal{C}_i$. ■ 6.2

Proof: *See Appendix B.5.*

6.5 Timing analysis on an ECTIA

We will try to determine which arcs of the ECTIA correspond to traces that are timing inconsistent. The main idea is to view the ECTIA as a LzTS and then distinguish between the arc that are timing consistent from the ones that are not by marking the inconsistent ones as lazy arcs. Formally, given an ECTIA $E = \langle S, \Sigma, T, \mathsf{s}_0\rangle$ we consider the LzTS $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR}\rangle$ that we call *Enabling Compatible Trace Remover Automaton* (ECTRA) in which the firing regions of the transitions are defined so as to render the timing inconsistent arcs lazy. This will allow to remove traces that are not consistent when the ECTRA is composed with the original system (see §6.6).

To perform the timing analysis on an ECTIA in order to obtain an ECTRA there are two possibilities. The easiest one is simply to consider the timing consistency of the trace that originated the ECTIA. Obviously the whole trace is timing inconsistent (otherwise it is a counterexample!) but usually not all its prefixes are, so we must discover at which point the firing of an event renders the trace inconsistent in order to mark the corresponding arc of the ECTIA as lazy. This approach

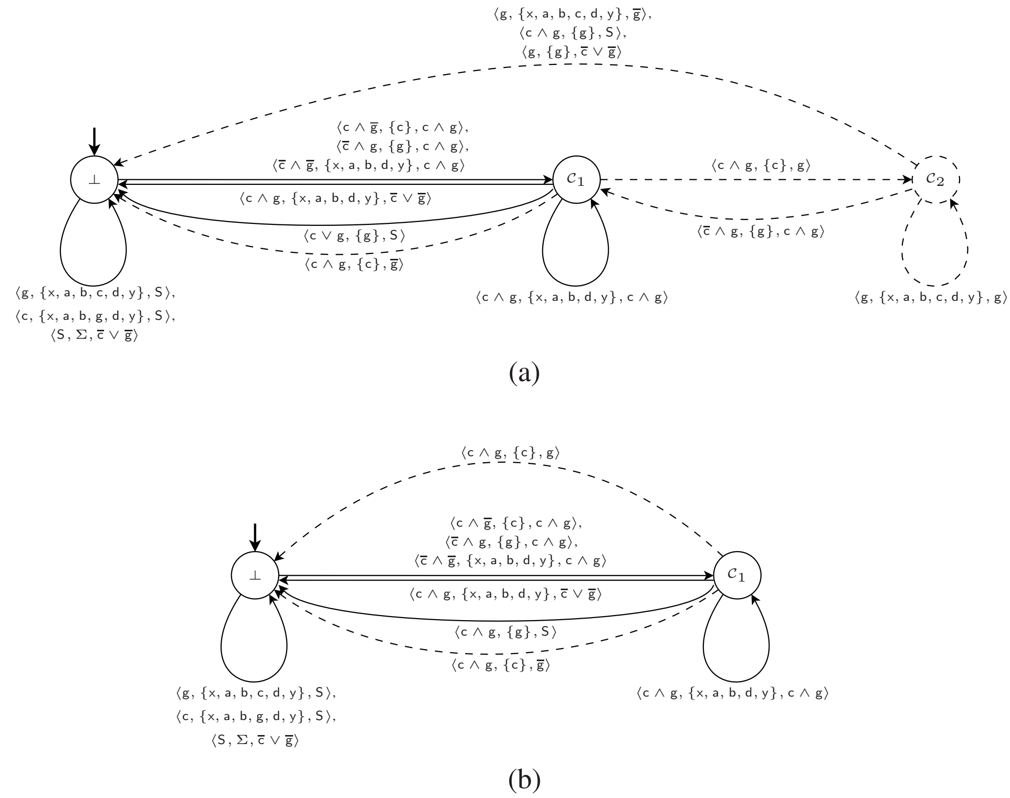


Figure 6.4. Example of ECTRAs using single trace analysis: (a) An ECTRA corresponding to the ECTIA of Figure 6.3 after the trace $\theta'_1 = \{\mathsf{c}, \mathsf{g}\}\xrightarrow{\mathsf{c}}\{\mathsf{g}\}\xrightarrow{\mathsf{g}}$ has been analyzed and it has been shown that the firing of event c renders the trace timing inconsistent. (b) An ECTRA obtained from trace $\theta'_2 = \{\mathsf{c}, \mathsf{g}\}\xrightarrow{\mathsf{c}}$.

will be discussed in §6.5.1. Another possibility is also to take into account the timing consistency of traces formed by firing events enabled in the trace. This requires additional timing consistency checkings because more traces are considered but can prune more timing inconsistent behaviours. This approach is discussed in §6.5.2.

6.5.1 Single trace analysis

A given timing inconsistent trace when analyzed in prefixes of increasing size is always timing consistent until a point is reached where an event is fired in the wrong order (some other event concurrently enabled with it should have fired before it). Hence all enabling compatible traces should be allowed to execute until this point is reached and then the firing of the badly ordered event should be forbidden. We can simulate this behavior by converting our ECTIA in a LzTS in which the timing considerations have been incorporated (*i.e.* using an ECTRA).

Formally, the result of the timing analysis on a trace $\theta' = E'_1\xrightarrow{\mathsf{e}'_1}E'_2\xrightarrow{\mathsf{e}'_2}\cdots E'_m\xrightarrow{\mathsf{e}'_m}E'_{m+1}$ is an ECTRA $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR}\rangle$. The ECTRA A is equal to the ECTIA derived from the trace with the only difference that the transitions are allowed to fire until the point where the timing inconsistent firing in θ' takes places. For instance if the firing of e'_j in θ' is what makes this trace timing inconsistent, then the ECTRA will not allow to fire the transitions from $\mathcal{C}_j$ to $\mathcal{C}_{j+1}$ (or any other transition labeled with event e'_j originating in $\mathcal{C}_j$) nor any of the transistions originating in any configuration $\mathcal{C}_k$ with $k > j$.

In Figure 6.4 (a) we can see the ECTRA that corresponds to the ECTIA of Figure 6.3 in which we have incorporated the timing analysis of the trace $\theta'_1 = \{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}\{\mathsf{g}\}\xrightarrow{\mathsf{g}}$, which becomes timing inconsistent at the moment in which event c is fired before event g. Hence, in Figure 6.4, the firing of event c is marked as lazy as well as all arcs after that point.

We commented in §6.3 that separated outgoing arcs from configurations might be needed because some of them could be declared lazy due to timing analysis while some others not. We can see an example of this phenomenon in configuration $\mathcal{C}_1$ of Figure 6.4 (a). There are two arcs braking the mapping, one corresponding to event c and one for event g. As already mentioned, the firing of c is what makes the trace timing inconsistent so the corresponding arc is declared lazy, while the arc associated to g is still valid. Consequently both arcs must be drawn independently.

The fact that, once the firing of the event that renders a trace timing inconsistent is found, subsequent configurations of the ECTRA are only reachable through lazy arcs makes it possible to use smaller ECTRAs to achieve the same results. For instance, in Figure 6.4 (b), we use the trace $\theta'_2 = \{\mathsf{c},\mathsf{g}\}\xrightarrow{\mathsf{c}}$ instead of θ'_1 to construct the ECTRA. Trace θ'_2 is a prefix of θ'_1 and its ECTRA is smaller than the one of θ'_1. However we will see in §6.6 that the "pruning" capability of both ECTRA s is the same. This observation will be the basis for some of the optimisations we present later on (§13.5), once the basic verification methodology has been established.

6.5.2 Multiple trace analysis

It is possible to use an ECTIA not only to prune the timing inconsistent suffix of the single trace that originated the ECTIA. A number of additional traces that share its prefix can also be checked. For instance in Figure 6.5 (a) we can see a TTS with a failure state. Assume that we find the following run to failure state: $\rho_f = \mathsf{s}_1\xrightarrow{\mathsf{x}}\mathsf{s}_2\xrightarrow{\mathsf{b}}\mathsf{s}_5\xrightarrow{\mathsf{a}}\mathsf{s}_7$. The corresponding trace θ_{ρ_f} is $\{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{b}}\{\mathsf{a},\mathsf{c}\}\xrightarrow{\mathsf{a}}\{\mathsf{c}\}$. The ECTIA obtained from that trace is shown in subfigure (c). In order to make the figure readable most of the reenter and break arcs have not been drawn with the exception of two arcs that we need so as to show the advantage of the multiple trace analysis over the single trace approach. There is another particularity with the ECTIA, that is that both configuration $\perp$ and $\mathcal{C}_1$ are initial configurations because in this example event x is unbounded so the set $\mathrm{LB}(\mathcal{E}(\mathcal{C}_1))$ is empty.

Using the single trace timing analysis presented in §6.5.1 we would analyze the consistency of trace $\{\mathsf{x}\}\xrightarrow{\mathsf{x}}$ which is timing consistent, and then trace $\{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{b}}$ which is not. We would mark as lazy all arcs firing event b from configuration $\mathcal{C}_2$ in the ECTRA (see Figure 6.6 (a)). Although the composition of the ECTRA will not be formally defined until §6.6, intuitively marking this arc as lazy will render arc $\mathsf{s}_2\xrightarrow{\mathsf{b}}\mathsf{s}_5$ also lazy in the original LzTS. However a quick inspection of the delays of the events reveals that not only event a should fire before event b, but also before event c. Hence also some additional arcs can be marked as lazy, in particular all the arcs in which event c is enabled in the configuration and fires before event a.

The set of traces that are considered in this kind of analysis given a trace $\theta = E_1\xrightarrow{\mathsf{e}_1}\dots E_n\xrightarrow{\mathsf{e}_n}$ can be described by the following set: $\{\theta' = E_1\xrightarrow{\mathsf{e}_1}\dots E_i\xrightarrow{\mathsf{e}} \mid \forall i : 1 \le i \le n \land \forall \mathsf{e} \in E_i\}$. For instance, in our example, this set is

$$\begin{array}{ll}
\{\mathsf{x}\}\xrightarrow{\mathsf{x}} & \{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{c}} \\
\{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{a}} & \{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{b}}\{\mathsf{a},\mathsf{c}\}\xrightarrow{\mathsf{a}} \\
\{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{b}} & \{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{b}}\{\mathsf{a},\mathsf{c}\}\xrightarrow{\mathsf{c}}
\end{array}$$

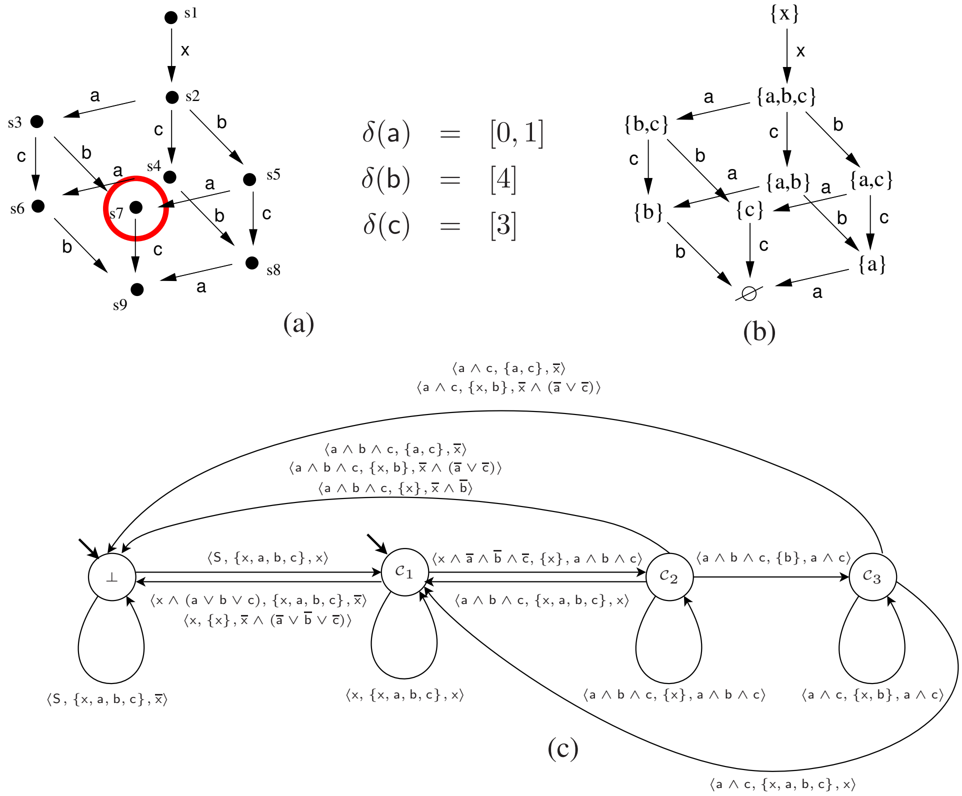


Figure 6.5. Example to illustrate the differences between single trace analysis and multiple trace analysis – first figure: (a) TTS in which state s_7 is declared a failure state. (b) Compact description of the traces of the system. (c) ECTIA obtained from trace $\theta = \{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{b}}\{\mathsf{a},\mathsf{c}\}\xrightarrow{\mathsf{a}}$.

Note that timing analysis can never affect the *keep* arcs because they do not belong to the set of considered traces.

6.6 Composition of an ECTRA with the original LzTS

Given a LzTS $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR}\rangle$ and an ECTRA $E = \langle S', T, T', \perp, \mathrm{FR}'\rangle$ their composition, noted as $A \odot E$, is simply a LzTS $A_c = \langle S_c, \Sigma, T_c, \mathsf{s}''_0, \mathrm{FR}_c\rangle$, where $S_c = S \times S'$ and $T_c = \{(\mathsf{s}_i, \mathsf{s}'_j)\xrightarrow{\mathsf{e}}(\mathsf{s}_k, \mathsf{s}'_l) \mid \mathsf{s}_j \xrightarrow{Set} \mathsf{s}_l \in T' \wedge \mathsf{s}_i \xrightarrow{\mathsf{e}} \mathsf{s}_k \in T \cap Set\}$. The firing region of events FR_c is defined so as to fire an event only if both arcs involved are fireable in A and E *and* all states in the firing region are reachable through at least one valid trace[1]. The initial state is $\mathsf{s}''_0 = (\mathsf{s}_0, \perp)$ except in the case where all the events in $\mathcal{E}(\mathcal{C}_1)$ are conservatively approximated. In that case the initial state is $\mathsf{s}''_0 = (\mathsf{s}_0, \mathcal{C}_1)$ if $\mathcal{E}(\mathsf{s}_0) \supseteq \mathcal{E}(\mathcal{C}_1)$, $\mathsf{s}''_0 = (\mathsf{s}_0, \perp)$ otherwise. If the LzTS A is defined with a set of initial states S_{in} instead of a single initial state s_0, then A_c is also defined with an initial set of states $\mathsf{S}'_{in} = \{(\mathsf{s}, \perp) \mid \mathsf{s} \in \mathsf{S}_{in}\}$ if there is some event in $\mathcal{E}(\mathcal{C}_1)$ that is not conservatively approximated. If all events are conservatively approximated then $\mathsf{S}'_{in} = \{(\mathsf{s}, \mathcal{C}_1) \mid \mathsf{s} \in \mathsf{S}_{in} \cap \mathrm{SE}(\mathcal{E}(\mathcal{C}_1))\} \cup \{(\mathsf{s}, \perp) \mid \mathsf{s} \in \mathsf{S}_{in} \cap \overline{\mathrm{SE}(\mathcal{E}(\mathcal{C}_1))}\}$. The condition $\mathsf{s} \in \mathsf{S}_{in} \cap \mathrm{SE}(\mathcal{E}(\mathcal{C}_1))$ is equivalent to $\mathsf{s} \in \mathsf{S}_{in} \wedge \mathcal{E}(\mathsf{s}) \supseteq \mathcal{E}(\mathcal{C}_1)$.

[1]This last condition is not necessary but it gives a much more intuitive result in the examples. For instance should this condition not be enforced, then in Figure 6.7 (b) the arcs $(\mathsf{s}_7, \perp)\xrightarrow{\mathsf{g}}(\mathsf{s}_8, \perp)$ and $(\mathsf{s}_7, \perp)\xrightarrow{\mathsf{b}}(\mathsf{s}_{10}, \perp)$ would become fireable. This would not change the fact that state $(\mathsf{s}_7, \perp)$ is not reachable but it would make it less evident.

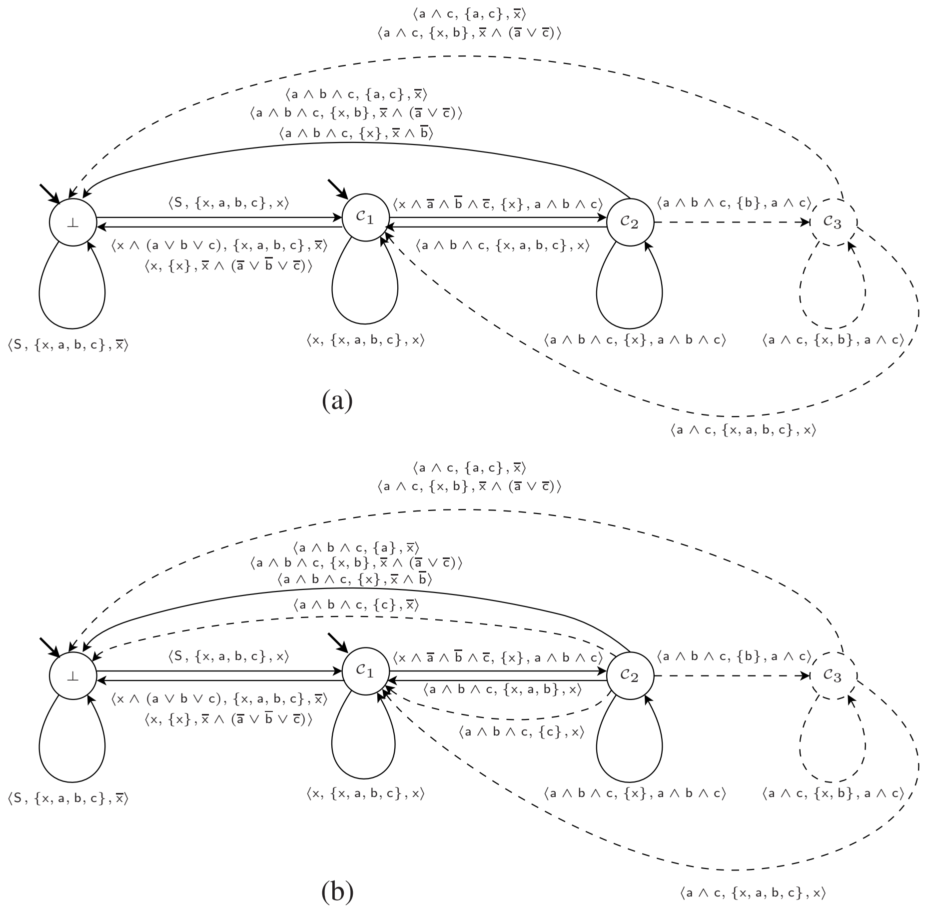


Figure 6.6. Example to illustrate the differences between single trace analysis and multiple trace analysis – second figure: (a) ECTRA obtained from ECTIA of Figure 6.5 (c) after single trace timing analysis has been conducted. (b) Same ECTRA but in which multiple trace analysis has been conducted.

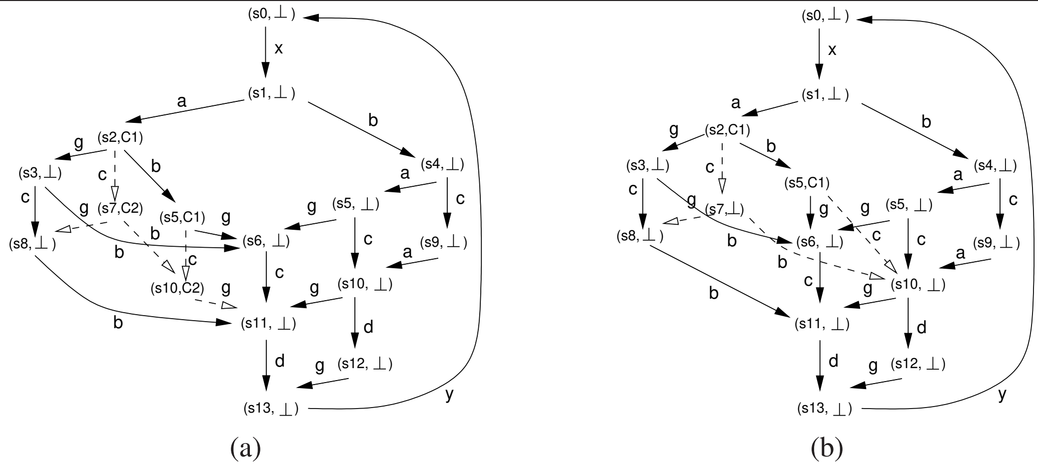


Figure 6.7. Example of composition of a LzTS with an ECTRA: (a) Composition of the ECTRA of Figure 6.4 (a) with a LzTS of Figure 5.2 (a). (b) Composition of the ECTRA of Figure 6.4 (b) with the same LzTS.

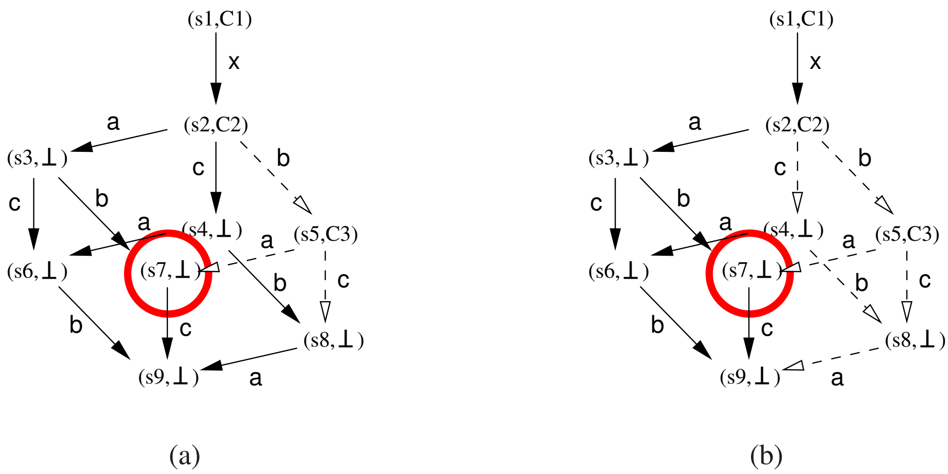


Figure 6.8. Example to illustrate the limitations of not considering possible concurrent traces in the timing analysis: (a) Composition of the ECTRA of Figure 6.6 (a) with a LzTS of Figure 6.5 (a). (b) Composition of the ECTRA of Figure 6.6 (b) with the same LzTS.

In Figure 6.7 (a) we can see the resulting LzTS of the compostion of the ECTRA of Figure 6.4 with the system of Figure 5.2 (a). We can see that it removes all runs either containing the sequence $s_2 \xrightarrow{c} s_7$ or the sequence $s_2 \xrightarrow{b} s_5 \xrightarrow{c} s_{10}$. They are removed as expected because both traces are enabling compatible with trace $\theta'_1 = \{c, g\} \xrightarrow{c} \{g\} \xrightarrow{g}$ used to generate the ECTRA of Figure 6.4 (a). We also give in Figure 6.7 (b) the resulting LzTS of the composition of the same system with the ECTRA of Figure 6.4 (b). Although the resulting system is smaller the same traces as before are removed. In general it is sufficient to build the ECTRA using the prefix of a trace instead of the whole trace. The only requirement is, obviously, that the prefix must contain the first timing inconsistent arc.

We can also increase the chances of removing timing inconsistent traces by using the *multiple trace analysis* approach explained in §6.5.2. For instance, in Figure 6.8, we can see two compositions of TTS of Figure 6.5 with the ECTRAs of Figure 6.6. Subfigure (a) is the result of the composition with the ECTRA obtained by the *single trace analysis*. Only the arcs enabling compatible with the original trace can be declared as lazy. However in subfigure (b) more traces were considered and more arcs could be disabled, so the composition has less valid traces and the number of rechable failure states can be reduced. For instance imagine that state s_8 was considered to be a failure state. With the single trace analysis we would require an additional composition while with the multiple trace analysis the state is already not reachable after the first composition.

We have seen that we can remove enabling compatible traces. And not only the enabling compatible traces with the original trace but also enabling compatible with traces that share a suffix with the original trace (*i.e.* with the *multiple trace analysis* approach). This might help in reducing the amount of iterations needed to verify a system if compared with the single trace based verification. However the enabling compatibility does not cope with different interleavings of events, thus the scope of the composition is still very limited, specially in systems with a high degree of concurrency. For instance in our previous example runs containing sequence $s_1 \xrightarrow{x} s_2 \xrightarrow{a} s_3 \xrightarrow{b}$ "escape" from the analysis, although the events involved are the same ones as in our original trace with the only difference that the firing order is changed. This is an inherent problem of the enabling compatibility concept that requires equivalent enabling and firing order of the events.

To solve this problem we will perform a timing analysis not only for a single trace, but also for all related interleaving behaviors that appear in our system. To do so we could just have increased the set of traces for which timing consistency is checked. However this is impractical due to the huge amount of interleaving behaviors that can appear in a concurrent system. To be able to complete such a task we need a mathematical entity that represents a set of interleaving behaviors and in which timing analysis can be performed efficiently. Fortunately there exist such kind of mathematical entities which are called *Causal Event Structures* (CES) and will be treated in the next section.

Chapter 7

EVENT STRUCTURES

Summary

This chapter presents the basic theory on *causal event structures* (CESs). A CES describes all the possible sequential and concurrent executions of a set of events. Thus, it allows to capture a set of enabling-compatible traces under a single mathematical object.

We start from the simplest model of event structure and we enrich it every time we find it contains an inherent limitation for our purposes. This way, an asymmetric conflict relation is first introduced and then a restriction relation is also added. The final CES model obtained, called *causal event structure with asymmetric conflict and restrictions* (CESAR), allows to model precisely sets of traces arising in arbitrary state spaces. We show how we can convert this model into an ECTIA. In Chapter 8 we will discuss their timing analysis, which would allow to convert an ECTIA into an ECTRA. Once the ECTRA is available, we can remove all the timing inconsistent traces represented by the CESAR using the composition method shown in Chapter 6.

Although the obtention of a CESAR that perfectly fits a part of the state space is a complex problem (Chapter 11), and their timing analysis is not as simple as the timing analysis of simpler CES models (Chapter 8), CESARs are vital in our framework because, otherwise, the methodology can only provide conservative results for many complex systems.

7.1 Elementary event structures

The simplest possible model of event structure was proposed in [NPW81], with the denomination of *elementary event structure*. However, we will use the much more common term of *causal event structure* to refer to it.

DEFINITION 7.1 (CAUSAL EVENT STRUCTURE [NPW81])

A causal event structure (CES)*,* $\mathrm{CS} = \langle \Sigma, \prec \rangle$*, is a finite set* Σ *of* events *and a* precedence relation $\prec \subset \Sigma \times \Sigma$ (*irreflexive, antisymmetric and transitive*) *called the* causality relation.

■ 7.1

A CES can then be depicted as a Hasse diagram, by showing the transitivity-irredundant $\prec$ relations (*i.e.* the covering relation of $\prec$, denoted as $\prec_\mu$) in form of solid arcs with triangular black head. We usually refer to the $\prec_\mu$ relation as the *direct causality* relation. *Root* events are events that have no predecessor. In case of having more than one root event, for the sake of clarity, we add a dummy root event, named ϵ[1]. In this way all graphical representations always have a single root, making all the graph connected. In Figure 12.3 (page 189) we can see several examples of this. Other examples of causal event structures can be seen in Figure 5.3 (b) and Figure 5.4 (b).

Given a CES $\mathrm{CS} = \langle \Sigma, \prec \rangle$ and a set of events $X \subseteq \Sigma$, the following sets can be defined [RE88]:

$$({}^{\rightarrow}X)_{\prec} \stackrel{\text{def}}{=} \{\mathsf{e}_1 \in \Sigma \mid \exists\, \mathsf{e}_2 \in X \ : \ \mathsf{e}_1 \prec \mathsf{e}_2\} \qquad (X^{\rightarrow})_{\prec} \stackrel{\text{def}}{=} \{\mathsf{e}_1 \in \Sigma \mid \exists\, \mathsf{e}_2 \in X \ : \ \mathsf{e}_2 \prec \mathsf{e}_1\}$$

$$({}^{\circ}X)_{\prec} \stackrel{\text{def}}{=} \{\mathsf{e}_1 \in X \mid \nexists\, \mathsf{e}_2 \in X \ : \ \mathsf{e}_2 \prec \mathsf{e}_1\} \qquad (X^{\circ})_{\prec} \stackrel{\text{def}}{=} \{\mathsf{e}_1 \in X \mid \nexists\, \mathsf{e}_2 \in X \ : \ \mathsf{e}_1 \prec \mathsf{e}_2\}$$

$$(\downarrow\! X)_{\prec} \stackrel{\text{def}}{=} ({}^{\rightarrow}X)_{\prec} \cup X \qquad (X\!\uparrow)_{\prec} \stackrel{\text{def}}{=} (X^{\rightarrow})_{\prec} \cup X$$

When it is clear from the context, we will just write ${}^{\rightarrow}X$, $X^{\rightarrow}$, ${}^{\circ}X$, X° and $\downarrow\! X$. Intuitively, ${}^{\rightarrow}X$ is the part of CS *before* X, $X^{\rightarrow}$ is the part of CS *after* X, ${}^{\circ}X$ are the *root* events of CS, and X° are the *sink* (with no successors) events of CS.

DEFINITION 7.2 (WORDS AND PREFIXES OF A **CES**)

A word *of the events of* $\mathrm{CS} = \langle \Sigma, \prec \rangle$ *is a sequence* $\omega = \mathsf{e}_1 \cdots \mathsf{e}_n \in \Sigma^n$ $(n \leq |\Sigma|)$*, such that all the events are distinct and* $\forall i : 1 \leq i \leq n, (\forall \mathsf{e} : \mathsf{e} \prec \mathsf{e}_i, \exists j : \mathsf{e}_j = \mathsf{e} \wedge j < i)$ i.e. *events are ordered in* ω *according to* $\prec$ *relation.*

Given a word $\omega = \mathsf{e}_1 \cdots \mathsf{e}_i \mathsf{e}_{i+1} \cdots \mathsf{e}_n$*, the i-th prefix of* ω *is denoted by* $\omega_i = \mathsf{e}_1 \cdots \mathsf{e}_i$*. The empty prefix is denoted by* ω_0*. The whole set of words of* CS *is noted as* words(CS).

■ 7.2

In Figure 7.1 we can see an example of CES in subfigure (a). The following sequences of events are words of the depicted CES: xagcbd, xbacdg, x, xab,... Consider the word $\omega =$ xbacd, then $\omega_1 =$ x is the first prefix of ω, $\omega_2 =$ xb is the second prefix of ω, etc.

A convenient way in which the words of a CES can be graphically depicted in a compact way is to use a *Graph of Reachable Configurations* (GRC). It is simply a directed acyclic graph in which states are called *configurations* and represent the set of events fired so far. Arcs between configurations are labeled with events and have the following semantic: the arc $\mathcal{C}_i \xrightarrow{\mathsf{e}} \mathcal{C}_j$ means that $\mathcal{C}_j = \mathcal{C}_i \cup \{\mathsf{e}\}$, *i.e.* from all the words that lead to $\mathcal{C}_i$ we can create another valid word of the CES by simply appending event e to them. The set of possible paths in that graph (identified by the events in the arcs) is exactly the set of valid words of the corresponding CES.

[1] A CES can always be interpreted as generating standardized traces, in which all traces begin with special event ϵ and there is always a unique root event.

Definition 7.3 (Graph of Reachable Configurations)
Given a set of words W over alphabet Σ such that satisfies the following properties:

- $\forall\omega \in W, \exists\omega' \in W \wedge \mathsf{e} \in \Sigma : \omega = \omega'\mathsf{e}$ i.e. *Any word in W can be obtained from another word in W by adding an event of the alphabet (the empty word is always considered to be part of W).*
- $\forall\omega_a, \omega_b \in W : \omega_a$ *is a permutation of* $\omega_b, \forall\omega_c \in W : \omega_c = \omega_b\mathsf{e}, \exists\omega_d \in W : \omega_d = \omega_a\mathsf{e}$ i.e. *For any word created from another word ω by appending an event, all the permutations of ω in the set W must be able to generate a new word by appending the same event.*

Then the set W can be represented succinctly as a directed acyclic graph called the Graph of Reachable Configurations *(GRC). Formally, given a set of words W its GRC is defined as a TS $G = \langle C, \Sigma, T, \top\rangle$ where C is the set of configurations*

$$C \stackrel{\text{def}}{=} \{\mathcal{C} = \{\mathsf{e}_1, \dots, \mathsf{e}_n\} \mid \exists\omega \in W : \omega = \mathsf{e}_1 \dots \mathsf{e}_n\}$$

Σ is the alphabet of W and T is the transition relation defined as

$$\mathcal{C}_i \xrightarrow{\mathsf{e}} \mathcal{C}_j \in T \Leftrightarrow \mathcal{C}_i, \mathcal{C}_j \in C \wedge \mathcal{C}_j = \mathcal{C}_i \cup \{\mathsf{e}\}$$

The special symbol $\top$ is used to identify the configuration corresponding to the empty word ($\top = \emptyset$) and is the starting configuration. ■ 7.3

Given a CES CS, the set words(CS) satisfies all the conditions in Definition 7.3 to represent it using a GRC. Hence the words of a CES can always be represented with a GRC. We preferred this definition based on properties of the set of words instead of the event structure because several types of event structures are presented in this chapter. All of them generate a set of words which satisfies the above mentioned conditions, thus we can use GRC s for any of them without having to adapt Definition 7.3 for each new type of event structure. Also, for all the event structures presented, we have the additional property that there is always a single longest word in the set. The corresponding configuration, which can be viewed as the "last" because has no outgoing edges in the graph, is depicted using the symbol $\perp$, for symmetry with respect to the initial configuration $\top$.

In Figure 7.1 (b) we can see the corresponding GRC of the CES of subfigure (a). Configurations have been labeled $\{\top, \mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_{11}, \perp\}$ and each one represents the set of events that one needs to fire to get to the configuration. For instance $\top = \emptyset$, $\mathcal{C}_1 = \{\mathsf{x}\}$, $\mathcal{C}_2 = \{\mathsf{x}, \mathsf{a}\}$, $\dots$, $\perp = \{\mathsf{x}, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{g}\}$.

The successive firing of the event of a word will enable other events, as soon as their causal predecessors have been fired. This notion is formally captured by the following definition:

Definition 7.4 (Events enabled by a prefix in a **CES**)
Let $\text{CS} = \langle\Sigma, \prec\rangle$ *be a* CES *and let* ω *be a word of* CS. *The set of events enabled by a prefix* ω_i *in* CS *is defined as* $\mathcal{E}_{\text{CS}}(\omega_i) = \{\mathsf{e}_k \notin \omega_i \mid \forall\mathsf{e}_j \in \Sigma, (\mathsf{e}_j \prec \mathsf{e}_k \Rightarrow \mathsf{e}_j \in \omega_i)\}$. *We will simply write $\mathcal{E}(\omega_i)$ when* CS *is clear from context.* ■ 7.4

That is, an event e_k is enabled by a prefix ω_i if *all* the causal predecessor events (by $\prec$) are in ω_i. For instance in Figure 7.1 the events enabled by $\omega = \mathsf{xacb}$ are $\{\mathsf{g}, \mathsf{d}\}$, while the events enabled by $\omega = \mathsf{xb}$ are $\{\mathsf{a}\}$. For GRC s derived from CES s (for other types of event structures this cannot be true) one can easily check that by inspecting the labels of the outgoing arcs of the configuration

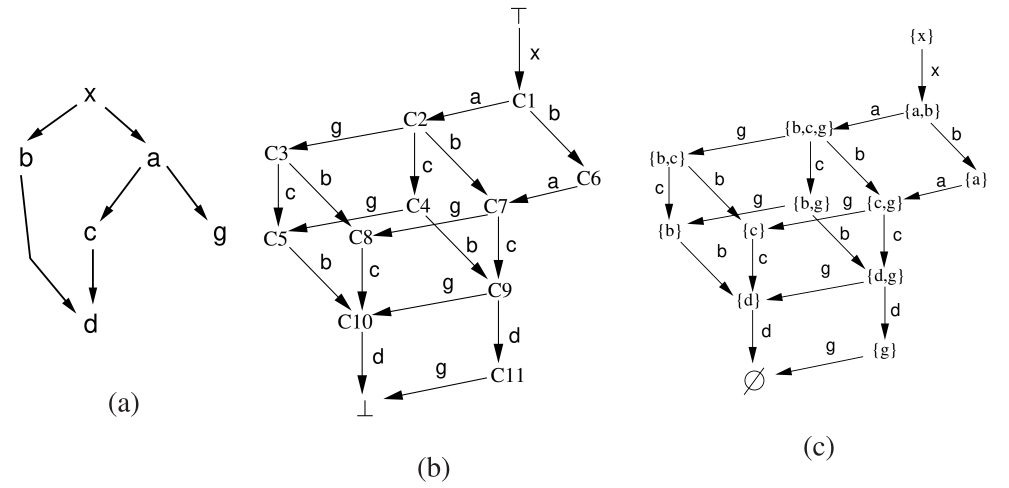


Figure 7.1. (a) A CES taken from example of Figure 5.3. (b) Its corresponding GRC. (c) Its corresponding GRE.

reached by the word in the GRC. For instance word $\omega = \mathsf{xb}$ leads to configuration $\mathcal{C}_6$ which has only one outgoing arc with label a.

Words in a CES can be used to generate traces. This fact is expressed by the following definition:

Definition 7.5 (Traces generated by words)

Let $CS = \langle \Sigma, \prec \rangle$ be a CES *and let $\omega = \mathsf{e}_1\mathsf{e}_2 \cdots \mathsf{e}_n$ be a word of* CS. *The trace generated by ω in* CS *is defined as:* $\theta_{\omega,\mathrm{CS}} = \mathcal{E}(\omega_0) \xrightarrow{\mathsf{e}_1} \mathcal{E}(\omega_1) \xrightarrow{\mathsf{e}_2} \cdots \xrightarrow{\mathsf{e}_{n-1}} \mathcal{E}(\omega_{n-1}) \xrightarrow{\mathsf{e}_n} \mathcal{E}(\omega_m)$ *. When no confusion is possible about the event structure we will simply write θ_ω instead of $\theta_{\omega,\mathrm{CS}}$.*

The set of all traces generated by the words of CS *is denoted as* traces(CS)*. Formally,* $\mathrm{traces(CS)} = \{\theta_\omega \mid \omega \in \mathrm{words(CS)}\}$. ■ 7.5

In the same way that a GRC represents compactly the set of words of a CES, we can also compactly represent the set of traces they generate by means of a *Graph of Reachable Enablings* (GRE). The GRE is isomorphic to the GRC, but each node no longer represents the event fired to reach that point but the events that are enabled at that point.

Definition 7.6 (Graph of Reachable Enablings)

Given a GRC $\mathrm{GC} = \langle C, \Sigma, T_G, \top \rangle$ *derived from an event structure* CS *satisfying the property that for all* $\omega \in \mathrm{words(CS)}$ *any permutation ω' of ω enables the same events (*i.e. $\mathcal{E}(\omega') = \mathcal{E}(\omega)$*), we define the set of events enabled by the configuration $\mathcal{C}_i$ in the GRC as the set of events enabled by any of the words reaching that configuration in the GRC. We denote this set as $\mathcal{E}(\mathcal{C}_i)$. By the previous property of the event structure it is ensured that $\mathcal{E}(\mathcal{C}_i)$ is independent from the path taken to reach $\mathcal{C}_i$.*

The Graph of Reachable Enablings *(GRE) is a TS* $\langle E, \Sigma, T_E, \mathcal{E}(\top) \rangle$ *such that:*

- *E is the set of enablings defined as $E = \{\mathcal{E}(\mathcal{C}) \mid \mathcal{C} \in C\}$,*
- *Σ is the alphabet of the event structure, and*
- *T_e is the transition relation defined as $\mathcal{E}(\mathcal{C}_i) \xrightarrow{\mathsf{e}} \mathcal{E}(\mathcal{C}_j) \in T_E \Leftrightarrow \mathcal{C}_i \xrightarrow{\mathsf{e}} \mathcal{C}_j \in T_G$.* ■ 7.6

In Figure 7.1 (c) we can see the corresponding GRE of subfigure (b).

The problem with the type of CES presented in Definition 7.1 is that it cannot generate all type of traces. In particular traces with disablings cannot be obtained from words of a CES as defined in [NPW81]. For instance trace $\{a, b\} \xrightarrow{a} \{b, c, g\} \xrightarrow{b} \{g\} \xrightarrow{g}$ contains a disabling, as the firing of event b disables event c. Since in the CES model the enabledness of an event only depends on the events fired before and once it becomes enabled it will continue enabled unless it is fired, then obviously CES are not suitable for traces with disablings. An example of the undesirable consequences that might appear if this model is used in the verification methodology can be found in Appendix C.1.

Definition 7.1 was used in [PCKP00] and extensions were introduced in the CES model in to cope with disablings in [Peñ03]. The reason to develop a new kind of event structure instead of reusing one of the multiple already available types was that preexistent models did not have all the properties desired. The key feature is that the event structure must be capable of stating that some event can disable another by specifying formally this kind of conflict between their occurrence.

7.2 Event structures with conflict

From its inception *event structures* have been modified and extended in many ways. Now there is a wide range of *event structures* to chose from. For instance:

> *stable event structures* [Win87], *free event structures* [DD90], *prioritized event structures* [DGV93], *local event structures* [HKT93], *flow event structures* [BC94], *bundle event structures* [Lan93], *extended bundle event structures* [Lan92], *asymmetric event structures* [BCM98, BCM01], *event structures for resolvable conflict* [vGP04], *event automaton* [PP95], *inhibitor event structures* [Bal00], *real-time event structures* [Dub05], *timed event structures* [MV01], etc.

We will briefly review the ones that are able to specify some kind of conflict between events. A first attempt to introduce such concept was made as early as in [NPW81] where a class of event structures with conflict relations, usually called *prime event structures*, was proposed. However, the conflict relation used was symmetric which corresponds to mutual disabling of competing events. The reason for a symmetric disabling relation was that event structures originated closely related to Petri nets [Pet62]. In such a context, events are usually mutually exclusive hence a symmetric disabling seemed the most natural option.

It is possible to "simulate" an asymmetric conflict relation with a *prime event structure* by duplicating events, but this introduces unnecessary complexity in the timing analysis of the event structure that must be conducted afterwards. First of all, events that must be duplicated (a process which is also called *label splitting* in the literature) must be detected. Then the event structure represents both the behavior in which the disabling appears, and the one in which the disabling does not occur. The increase in the number of events considered also increases the cost of the timing analysis of the event structure. As the timing analysis is one of the core operations of the proposed method it would be impractical.

A more general relation is required for our purposes that models the asymmetry of event disabling, such as it appears in digital circuits, for example. In [Peñ03] another reason to require an asymmetric conflict relation was that event structures were derived from single traces, and a particular trace can only capture a single branch of a conflict relation.

Some of the event structure types previously mentioned consider an asymmetric disabling relation, in particular *asymmetric event structures*, *extended bundle event structures* and *event automaton*. However, all of them have a small inconvenient when generating their GRC. We will show this with an example.

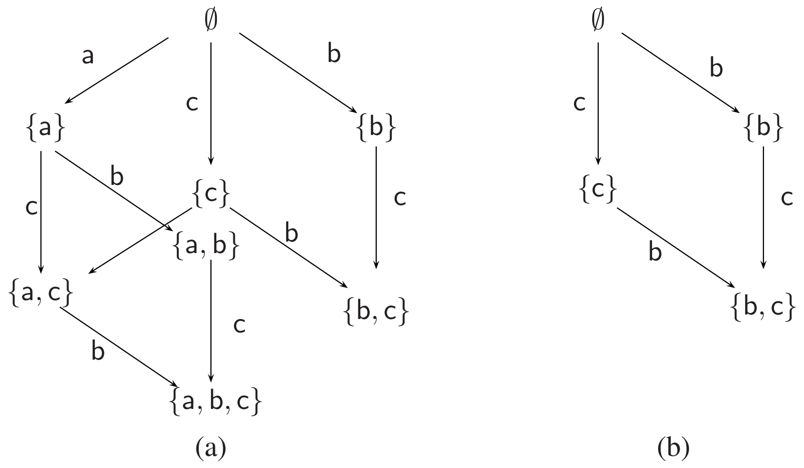


Figure 7.2. Comparion between ES models with asymmetric conflict: (a) GRC using the configuration definition of the ES models in the literature with an asymmetric conflict relation. The firing of event b disables event a, but also the configuration in which a is fired before b are also present. (b) GRC obtained from the CES with asymmetric disabling relation of [Peñ03]. Only the configurations related to the disabling are considered.

EXAMPLE 7.1 *Imagine we want to express that we have a system with three concurrent events* a, b *and* c, *and that the firing of* b *disables* a. *Using the semantics of the three models we will end up with a GRC like the one in Figure 7.2 (a). However, we would like to have a GRC in which only the part in which the disabling really appears is depicted, as in subfigure (b).* ■ 7.1

We cannot yet enter into details at this point to explain why we prefer (b) instead of (a). Just say by the moment that the accuracy of the timing analysis and the effort to perform it are affected by the selection (see §8.6.1 for a discussion on the topic). Hence, in [Peñ03], CES s where equipped with a disabling relation $\triangleright$ that models the asymmetric conflict and a new definition for the valid words. From a practical point of view, these CESs are the *asymmetric event structures* of [BCM01] in which the set of valid words is redefined to forbid all behaviors in which the disabling does not appear. A more detailed information can be found in §A.4.

DEFINITION 7.7 (**CES** WITH ASYMMETRIC CONFLICT)

A causal event structure with asymmetric conflict (CESA), $\mathrm{CS} = \langle \Sigma, \prec, \triangleright \rangle$, *is a CES* $\mathrm{CS}^- = \langle \Sigma, \prec \rangle$ *equipped with a* conflict relation $\triangleright \subset \Sigma \times \Sigma$ (*irreflexive and antisymmetric*) *called the* disabling relation.

Relations $\prec$ *and* $\triangleright$ *satisfy the following properties:*

- $\forall e_1, e_2, e_3 \in \Sigma,\ e_1 \triangleright e_2 \wedge e_1 \prec e_3 \Rightarrow e_3 \triangleright e_2$ i.e. *Relation* $\triangleright$ *is inherited via* $\prec$.
- $\forall e_1, e_2, e_3 \in \Sigma,\ e_1 \triangleright e_2 \wedge e_1 \not\prec e_3 \Rightarrow e_3 \not\triangleright e_2$ i.e. *No event can be disabled by two or more concurrent events.*
- $\prec \cap \triangleright = \emptyset,\ \prec^{-1} \cap \triangleright = \emptyset$ i.e. *The causality and the conflict relation are disjoint (no event can disable its own causal predecessor or successors).*
- $\forall e_1, e_2 \in \Sigma,\ e_1 \triangleright e_2 \ \Rightarrow\ \nexists e_3 \in \Sigma\ :\ e_2 \prec e_3 \vee e_2 \triangleright e_3$ i.e. *Disabled events cannot be causal predecessors of the other events in the CESA nor disable any of them.*

■ 7.7

The disabling relation will be graphically depicted as a dashed white head arrow between the involved events. As before, to make pictures more clear we will only represent $\triangleright_\mu$ which is defined as the covering relation (under $\prec$) of $\triangleright$. Formally, $\mathsf{e} \triangleright_\mu \mathsf{d} \Rightarrow \mathsf{e} \triangleright \mathsf{d} \wedge \nexists \mathsf{e}' \in \Sigma : \mathsf{e}' \prec \mathsf{e} \wedge \mathsf{e}' \triangleright \mathsf{d}$.

Most of the properties that relations $\prec$ and $\triangleright$ must satisfy in Definition 7.7 are straightforward. However the property that forbids an event to be disabled by two or more concurrent events needs some explanation. However the reasons for that constraint are related to the timing analysis of event structures, hence we delay our explanation until §8.6.2.

With the addition of disablings, now the set of valid words of a CESA must take into account that disabled events can never appear in a word.

Definition 7.8 (Words of a **CESA**)

A word *of the events of a CESA* $\mathrm{CS} = \langle \Sigma, \prec, \triangleright \rangle$ *is a sequence* $\omega = \mathsf{e}_1 \cdots \mathsf{e}_n \in \Sigma^n$ *(with* $n \leq |\Sigma|$*), such that* ω *is a word of the CES* $\mathrm{CS}^- = \langle \Sigma, \prec \rangle$ *and* $\forall \mathsf{e}, \mathsf{d} \in \Sigma,\ \mathsf{e} \triangleright \mathsf{d} \Rightarrow \mathsf{d} \notin \omega$. ■ 7.8

And then the set of enabled events by a word must accommodate to include the disabling phenomenon.

Definition 7.9 (Events enabled/disabled by a prefix in a **CESA**)

Let $\mathrm{CS} = \langle \Sigma, \prec, \triangleright \rangle$ *be a* CESA *and let* ω *be a word of* CS. *The set of events enabled by a prefix* ω_i *is defined as* $\mathcal{E}(\omega_i) = \{\mathsf{e}_k \notin \omega_i \mid \forall \mathsf{e}_j \in \Sigma, (\mathsf{e}_j \prec \mathsf{e}_k \Rightarrow \mathsf{e}_j \in \omega_i) \wedge (\mathsf{e}_j \triangleright_\mu \mathsf{e}_k \Rightarrow \mathsf{e}_j \notin \omega_i)\}$ *. Similarly, the set of events disabled by a prefix* ω_i *is defined as* $\mathcal{D}(\omega_i) = \{\mathsf{e}_k \notin \omega_i \mid \exists \mathsf{e}_j \in \Sigma\ :\ \mathsf{e}_j \triangleright \mathsf{e}_k \wedge \mathsf{e}_j \in \omega_i\}$. ■ 7.9

Thanks to the introduction of disablings, now CESAs are able to generate any imaginable trace.

Given a trace θ', one can derive a CES CS such that θ' is a trace generated by one word ω belonging to the words of CS. That is $\exists \omega \in \mathrm{words(CS)} : \theta_\omega = \theta'$. There exist many CESAs that satisfy this condition. An infinite number, in fact. In next definition we give the conditions for a CESA to be able to generate the trace.

Definition 7.10 (**CESA** that can generate a trace)

Let $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots \xrightarrow{\mathsf{e}_{n-1}} E_n \xrightarrow{\mathsf{e}_n} E_{n+1}$ *be a finite trace with* $D(\theta)$ *as the set of disabled events along it and* Σ_θ *its alphabet (*i.e. $\Sigma_\theta = \bigcup_{i=1}^{n+1} E_i$*). The CESA* $\mathrm{CS}_\theta = \langle \Sigma, \prec, \triangleright \rangle$ *can generate* θ *iff satisfies the following conditions:*

- $\Sigma \supseteq \Sigma_\theta$. i.e. *Not only the firing events are included, but all those enabled along the trace, including the disabled ones, and possibly many more that do not belong to* Σ_θ.
- *Relation* $\prec$ *is defined as follows:*

$$\forall i : 1 < i \leq n+1,\ \forall \mathsf{e} \in E_i : \mathsf{e} \notin E_{i-1},\ \mathsf{e}_{i-1} \prec \mathsf{e}$$

  these are the minimally required relations (the trigger of an event must *be one of its causal predecessors), but additional relations may be added from the following set (and* only *from that set)*

$$\{\mathsf{e}_j \prec \mathsf{e} \mid \mathsf{e}_i \prec \mathsf{e} \wedge 1 \leq j < i-1\}$$

  i.e. *any of the previously fired events* can be *a causal predecessor. The only condition is that the selection must be done in such a way that relation* $\prec$ *finally satisfies the transitive property (irreflexivity and antisymmetry are already guaranteed by the definition itself for the events in* Σ_θ*).*

- *Relation $\triangleright$ between the events in Σ_θ is defined as follows:*

$$\forall i : 1 \leq i < n+1,\ \forall \mathsf{d} \in E_i : \mathsf{d} \notin E_{i+1} \wedge \mathsf{d} \neq \mathsf{e}_i,\ \mathsf{e}_i \triangleright_\mu \mathsf{d}$$

these are the minimally required relations, but additional relations may be added from the following set (and only *from that set)*

$$\{\mathsf{e} \triangleright \mathsf{d} \mid \mathsf{e}, \mathsf{d} \in E_{n+1}\}$$

- *For events in the alphabet Σ but not in the alphabet of the trace (*i.e. $\Sigma_P = \Sigma \setminus \Sigma_\theta$*) either:*
  - *At least one of their causal predecessors must be some of the events enabled until the end of trace (*i.e. *in E_{n+1}) or other events in Σ_P, or*
  - $\mathsf{d} \in \Sigma_P$ *and* $\exists i,\ \mathsf{e}_i \triangleright \mathsf{d} \wedge \exists \mathsf{e} \in \Sigma : \mathsf{e} \prec \mathsf{d} \wedge (\mathsf{e} = \mathsf{e}_j \wedge i < j \vee \mathsf{e} \in E_{n+1} \cup \Sigma_P)$. i.e. *event d is disabled by some event e_i fired in the trace and is enabled by either an event not fired in the trace ($E_{n+1} \cup \Sigma_P$) or by an event e_j fired later than its disabler ($j > i$).*

  Between the events in Σ_P we can add an arbitrary number of $\triangleright$ and/or $\prec$ relations given that the properties of Definition 7.7 are respected. Also events in Σ_P may arbitrarily disable events of E_{n+1}.

We denote the infinite set of CESA*s that satisfy this definition for a trace θ as* CESA(θ).

■ 7.10

This definition contains a set of minimally "obliged" relations that can be arbitrarily enlarged. Although already stated, let us remark that this degree of freedom is restricted up to the point where the relations satisfy the expected properties of a CESA. For instance, we said that events in Σ_P may arbitrarily disable events of E_{n+1}, but CESA definition does not allow to disable own causal predecessors.

We give some examples of CESAs that can generate trace $\{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{f}\}\xrightarrow{\mathsf{a}}\{\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{b}}\{\mathsf{c},\mathsf{d}\}$ in Figure 7.3.

LEMMA 7.1
Given a finite distinct trace $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots \xrightarrow{\mathsf{e}_{n-1}} E_n \xrightarrow{\mathsf{e}_n} E_{n+1}$ and a CESA CS. CS *can generate θ if and only if* CS $\in$ CESA(θ). *Formally,*

$$\theta \in \mathrm{traces(CS)} \Longleftrightarrow \mathrm{CS} \in \mathit{CESA}(\theta)$$

■ 7.1

Proof: *See Appendix B.6.*

CESAs can generate any possible trace, but this is not enough for our purposes. As we will see in Chapter 11, any time that the set of traces generated by a CESA contains a trace that does not really exist in our system, there is the danger that our analysis becomes conservative (*i.e.* the verification is not able to show the correctness of the system although it is correct). State spaces of systems can be arbitrarily complex so, in general, there is no guarantee that the full set of traces that a CESA represents will also be present as a possible behavior of the system (see Figure 7.4). Hence we do need to introduce a mechanism to "prune" the non-existent traces from the set of traces of a CESA. By doing so we remove a possible source of conservativeness in the methodology.

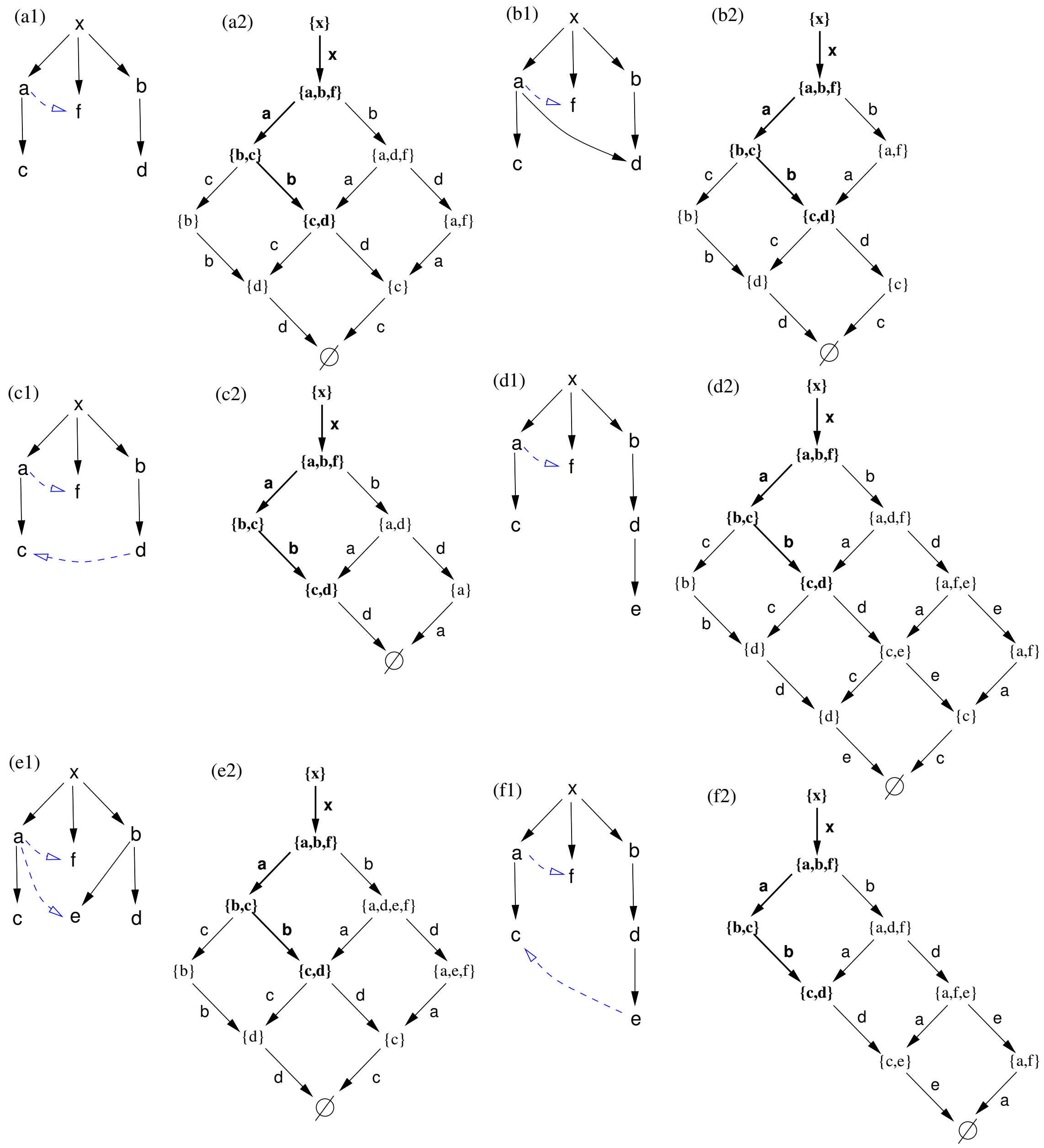


Figure 7.3. Examples of CESAs that can generate a given trace: (a1,b1,c1,d1,e1,f1) CESAs that can generate trace $\{x\}\xrightarrow{x}\{a,b,f\}\xrightarrow{a}\{b,c\}\xrightarrow{b}\{c,d\}$. (a2,b2,c2,d2,e2,f2) Their corresponding GRE s in which the trace has been marked. Each one of the CESAs illustrates some aspect of Definition 7.10: (a1) Is the CESA with only the "obliged" relations, hence the most concurrent one. (b1) Events of trace can be causal predecessor of other events if the former are fired before the trigger of the latter. (c1) Disabling between events in E_{n+1} (in this case $E_4 = \{c, d\}$ is allowed. (d1) Extra events not appearing in the trace may be added. In that case added event e is causal successor of an event of E_{n+1}. (e1) An extra event might have a causal predecessor in the set of fired events of the trace given that it is also disabled by a previously fired event. (f1) Events not fired in the trace might disable events in E_{n+1}.

7.3 Event structures with asymmetric conflict and restrictions

The mechanism we introduce is to add a *restriction relation* in the CESA, which will constrain the set of valid words without changing the enablings (the CESA disabling relation works in part as a restriction but has also an effect in the enablings of the traces).

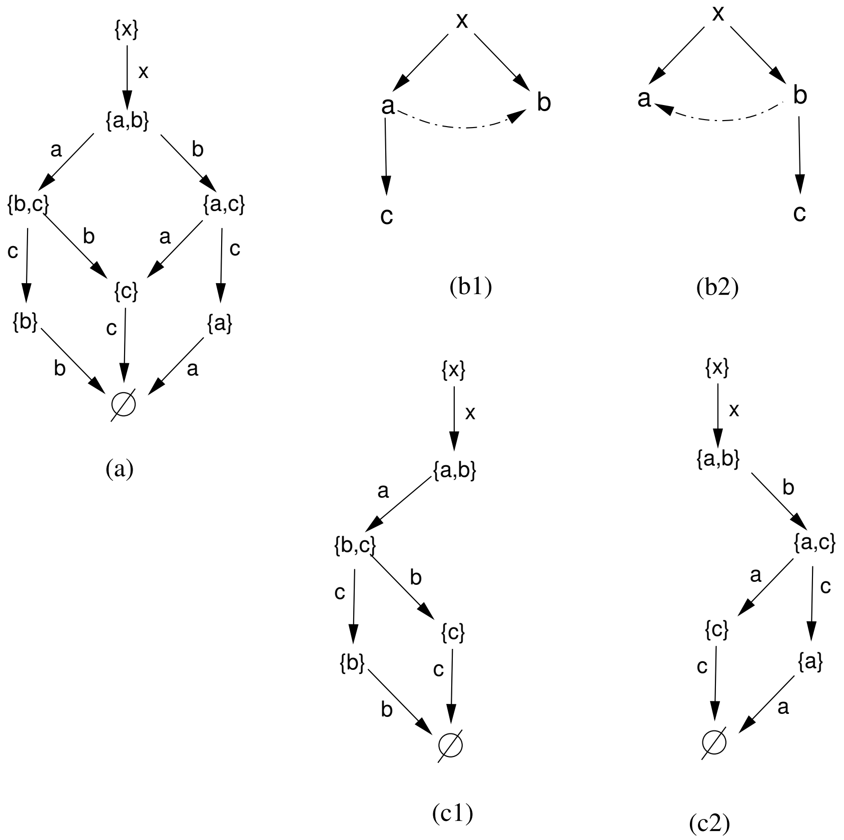


Figure 7.4. Example to show that restrictions are needed in event structure definition to be able to model arbitray state spaces: (a) GRE that we would like to obtain. As it contains OR causality (event c can either be triggered by a or b) it cannot be generated by a CESA. Instead two CESARs (b1,b2) must be used to cover all possible traces. Dashed-dotted arcs represent restrictions. For instance in (b1) the restriction depicted is $a < b$. (c1,c2) Corresponding GREs of the CESARs. They completely cover all traces of (a) without adding any other trace. Using multiple CESs or CESAs one can also cover all traces of (a) but cannot avoid to consider also some other traces not in (a). This might lead to conservative results after timing analysis.

Definition 7.11 (**CES** with Asymmetric conflict and Restrictions) *A* causal event structure with asymmetric conflict and restrictions (CESAR)*,* $\mathrm{CS} = \langle \Sigma, \prec, \triangleright, < \rangle$*, is a* ***CESA*** $\mathrm{CS}^- = \langle \Sigma, \prec, \triangleright \rangle$ *equipped with an additional* precedence relation $<$ (*irreflexive, antisymmetric and transitive*)*, called the* restriction relation*, that satisfies the following properties:*

- *If we define* $<_{min} = (\prec \cup \triangleright_\mu)^*$ *then,* $< \supseteq <_{min}$ i.e. *The restriction relation must respect at least the order in which events are fired due to the disabling and precedence relation.*
- $a \triangleright b \Rightarrow \nexists c \in \Sigma : b < c$ i.e. *A disabled event cannot be forced to fire before another.*
- $a \triangleright b \Rightarrow \nexists c \in \Sigma : c \not<_{min} a \ \wedge \ c < b$ i.e. *No additional ordering can be established with an event which is disabled.*

■ 7.11

The restriction relation establishes the partial order in which events can be fired, hence it must include the causality and the disabling relations. Additional restrictions may be added to prune undesired parts of the GRC and are graphically depicted (its covering relation, as usual) as dashed-dotted arcs as in Figure 7.4 (b1,b2). As states the definition, these additional restrictions cannot be established with an event which is disabled. The reason for this constraint is the same one that justifies the prohibition that two concurrent events disable the same event in a CESA. This is a timing analysis related problem, hence we delay our explanation until §8.6.2 once timing analysis on event structures has been presented.

To allow the restriction relation to determine the valid firing order of the events now we must define the set of valid words of a CESAR to be the ones that satisfy this restriction relation.

DEFINITION 7.12 (WORDS OF A **CESAR**)
A word *of the events of a CESAR* $\mathrm{CS} = \langle \Sigma, \prec, \rhd, <_{cs} \rangle$ *is a sequence* $\omega = \mathsf{e}_1 \cdots \mathsf{e}_n \in \Sigma^n$ *(with* $n \leq |\Sigma|$*), such that* $\forall i : 1 \leq i \leq n, (\forall \mathsf{e} : \mathsf{e} <_{cs} \mathsf{e}_i, \exists j : \mathsf{e}_j = \mathsf{e} \wedge j < i) \wedge \nexists \mathsf{e} : \mathsf{e} \rhd \mathsf{e}_i$ i.e. *events are ordered in* ω *according to the restriction relation.* ■ 7.12

The set of enabled/disabled events by a word is still defined as in Definition 7.4 because restrictions do not affect the enablings.

In Definition 7.10 we described the infinite set of CESAs that can generate a trace. Initially, at the beginning of each iteration of our verification methodology, we have a single failure trace and we will not be interested in CESAs that define additional events in their alphabets, hence we restrict to the ones which satisfy $\Sigma = \bigcup_{i=1}^{n+1} E_i$. Using this restriction, the set of possible CESAs becomes finite, as the definition enforces a minimum set of relations that can be arbitrary enlarged up to a certain threshold. If the size of Σ is fixed, then there is a limit to the addition of relations as no other elements can be added to the relation when it already describes a total order. As our objective is to cover as much concurrent behavior as we can, we will use the CESAs which have the smaller $\prec$ relation. That is, we will use for $\prec$ only the relations that the definition obliges. By doing so we take an aggressive approach: the obtained CESA we know it will cover a maximum number of traces, but precisely for this very same reason, the probabilities that some of this set of traces does not really appears in the system is bigger. As we explain in Chapter 11, we will then restrict the CESA trying to obtain a CESAR that has the biggest number of traces and all them appearing in the system *i.e.* that represents the maximum real concurrency in the system.

LEMMA 7.2 (**CESAR** CAN STILL GENERATE TRACE FROM A **CESA**)
Given a finite distinct trace θ *and a CESA* $\mathrm{CS}^- = \langle \Sigma, \prec, \rhd \rangle$ *such that* CS^- *can generate* θ *(*i.e. $\mathrm{CS}^- \in$ *CESA*(θ)*), then CESAR* $\mathrm{CS} = \langle \Sigma, \prec, \rhd, < \rangle$ *can generate* θ*, if and only if, the partial order* $<$ *does not contradict the order in which events are fired in* θ*.* ■ 7.2

Proof: *If the order* $<$ *allows the firing order of* θ*, then* CS *can generate the trace because* CS^- *was able to do it. If the order* $<$ *contradicts the firing order of* θ*, then the words that would generate* θ *does no belong to* words(CS)*, hence the trace cannot be generated.*

We use this CESAR (derived from failure traces and then restricted to fit the actual traces of our system) to perform timing analysis. This analysis unveils relative timing relations among the events in the CESAR can be found that help to prove the timing-consistency or timing-inconsistency of the traces represented by it. Moreover, thanks to Theorem 5.7, the timing relations derived from the analysis also apply to the set of traces enabling-compatible with the traces covered by the event structure. However, before we can incorporate this timing information back into the system we need to describe how we can obtain an ECTIA for the traces covered by the CESAR.

7.3.1 Generating an ECTIA from a CESAR

In §6.2 we have seen the procedure to build an ECTIA of a given a trace. We would like to extend this procedure to be able to obtain an automata that identifies all the enabling compatible traces with any of the traces covered by the CESAR. To be consistent with the notation previously used we will use the GRC of the CESAR, instead of its GRE, as the basis to generate the ECTIA. The only minor notational difference is that in §6.2 we use $\mathcal{C}_1$ as the first configuration of the ECTIA which indicates that the enabling compatible mapping is starting, while here we use $\top$ to maintain the notation introduced in the GRC definition.

Given a GRC $G = \langle C, \Sigma, T_G, \top \rangle$ that describes some traces in a LzTS $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR} \rangle$, we can obtain an ECTIA $E = \langle C, T, T_E, \perp \rangle$. Remember set of configurations C of GRC already contains special configurations $\top$ and $\perp$ to represent the beginning of the enabling compatible mapping and the non-synchronizing state, respectively. Transitions in T_E are defined as follows (this definition is a straightforward extension of the transitions defined in §6.2 to cope with several advance arcs and the introduction of the $\top$ notation):

Transitions starting the mapping.

Transition	Conditions
$\perp \xrightarrow{\mathbf{enter}} \top$	$\mathbf{enter} = \langle \mathrm{SD}(\mathrm{LB}(\mathcal{E}(\top))), \Sigma \setminus \mathrm{LB}(\mathcal{E}(\top)), \mathrm{SE}(\mathcal{E}(\top)) \rangle \cup$ $\cup \; \{ \langle \mathrm{SD}(\mathrm{LB}(\mathcal{E}(\top)) \setminus \{\mathsf{e}\}) \cap \mathrm{SE}(\{\mathsf{e}\}), \{\mathsf{e}\}, \mathrm{SE}(\mathcal{E}(\top)) \rangle \mid \forall \mathsf{e} \in \mathrm{LB}(\mathcal{E}(\top)) \}$

Transitions advancing the mapping.

Transition	Conditions
$\top \xrightarrow{\mathbf{advance_{in,j}}} \mathcal{C}_j$	$\top \xrightarrow{\mathsf{e}} \mathcal{C}_j \in T_G \;\wedge$ $\mathbf{advance_{in,j}} = \langle \mathrm{SE}(\mathcal{E}(\top)) \cap \mathrm{SD}(\mathrm{LB}(\mathcal{E}(\mathcal{C}_j) \setminus (\mathcal{E}(\top) \setminus \{\mathsf{e}\}))), \{\mathsf{e}\}, \mathrm{SE}(\mathcal{E}(\mathcal{C}_j)) \rangle$
$\mathcal{C}_i \xrightarrow{\mathbf{advance_{i,j}}} \mathcal{C}_j$	$\mathcal{C}_i \xrightarrow{\mathsf{e}} \mathcal{C}_j \in T_G \;\wedge$ $\mathbf{advance_{i,j}} = \langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)) \cap \mathrm{SD}(\mathrm{LB}(\mathcal{E}(\mathcal{C}_j) \setminus (\mathcal{E}(\mathcal{C}_i) \setminus \{\mathsf{e}\}))), \{\mathsf{e}\}, \mathrm{SE}(\mathcal{E}(\mathcal{C}_j)) \rangle$
$\mathcal{C}_i \xrightarrow{\mathbf{advance_{i,out}}} \perp$	$\mathcal{C}_i \xrightarrow{\mathsf{e}} \perp \in T_G \;\wedge\; \mathbf{advance_{i,out}} = \langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)), \{\mathsf{e}\}, S \rangle \;\cap\; \overline{\mathbf{enter}}$

Transitions keeping the mapping.

Transition	Conditions
$\top \xrightarrow{\mathbf{keep_{in}}} \top$	$\mathbf{keep_{in}} = \langle \mathrm{SE}(\mathcal{E}(\top)), \Sigma \setminus \mathcal{E}(\top), \mathrm{SE}(\mathcal{E}(\top)) \rangle$
$\mathcal{C}_i \xrightarrow{\mathbf{keep_i}} \mathcal{C}_i$	$\mathbf{keep_i} = \langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)), \Sigma \setminus \mathcal{E}(\mathcal{C}_i), \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)) \rangle$
$\perp \xrightarrow{\mathbf{keep_{out}}} \perp$	$\mathbf{keep_{out}} = \overline{\mathbf{enter}}$

Transitions restarting the mapping.

Transition	Conditions
$\mathcal{C}_i \xrightarrow{\mathbf{reenter_i}} \top$	$\mathbf{reenter_i} = \langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)), \Sigma, S \rangle \;\cap\; \mathbf{enter}$

Transitions breaking the mapping.

Transition	Conditions
$\top \xrightarrow{\mathbf{break_{in}}} \perp$	$\mathbf{break_{in}} = \langle \mathrm{SE}(\mathcal{E}(\top)), \Sigma, S \rangle \;\cap\; \overline{(\mathbf{enter} \cup \mathbf{keep_{in}} \cup \bigcup_{\top \xrightarrow{\mathsf{e}} \mathcal{C}_j \in T_G} \mathbf{advance_{in,j}})}$
$\mathcal{C}_i \xrightarrow{\mathbf{break_i}} \perp$	$\mathbf{break_i} = \langle \mathrm{SE}(\mathcal{E}(\mathcal{C}_i)), \Sigma, S \rangle \;\cap\; \overline{(\mathbf{enter} \cup \mathbf{keep_i} \cup \bigcup_{\mathcal{C}_i \xrightarrow{\mathsf{e}} \mathcal{C}_j \in T_G} \mathbf{advance_{i,j}})}$

7.3.2 Enabling compatibility and ECTIAs generated by CESARs

In §6.4 we presented several theorems that relate the enabling compatibility and the current configuration of an ECTIA. These theorems apply for ECTIAs generated from traces as well as the ones generated by CESARs. There are only some minor considerations. First one is that now we use $\top$ notation instead of $\mathcal{C}_1$ to indicate the first synchronizing configuration (one must be aware of this when reading the proofs as they were done using the $\mathcal{C}_1$ notation). The second one is that the auxiliary function $\mathrm{traces}(\mathcal{C}_i)$ is much more easily defined now as the set of traces of the CESAR reaching configuration $\mathcal{C}_i$.

Chapter 8

TIMING ANALYSIS ON EVENT STRUCTURES

Summary

Instead of performing a trace by trace timing consistency checking, CESARs [1] can be used to obtain a similar analysis with less effort. The main idea is to use the available efficient algorithms that compute the maximal separation time between any two events e_1 and e_2.

However, when the effect of CESAR restrictions is considered, this translation is more complex than previously thought. We prove this with some examples and we give an algorithm to perform a correct timing analysis in the presence of restrictions.

The timing analysis allows to deduce the relative firing order of the events imposed by their timings. This information can be incorporated into a CESAR, producing a new structure called *Lazy CESAR* (LzCES). From this new structure we can construct an ECTRA which will add the timing information back into the system once composed with it.

Finally, we justify some of the constraints that appear in the CESAR definition that have been introduced to avoid inconsistencies in the timing analysis.

[1] Although CESARs in particular, and events structures in general, do not have timing information (*i.e.* delay bounds) formally associated to their events, we will consider that the delay bounds are inherent to the events and we will not enlarge the definitions for the sake of clarity. Strictly speaking, a subtype of event structures called *Timed Event Structures* [AV05, VG04, MV01], in which delay bounds are assigned to the events, should have been used.

8.1 Maximal separation time between events

The maximal separation time between two events in a set of traces is the maximum difference, allowed by the given delays, between the firing times of the two events in the traces of the set.

DEFINITION 8.1 (MAX. SEP. TIME BETWEEN EVENTS IN A SET OF TRACES)
Given a distinct trace θ, we define the maximal separation time *of* e_1 *and* e_2 *in* θ*, denoted as* $\mathrm{mst}_\theta(\mathsf{e}_1, \mathsf{e}_2)$*, as the maximum difference between their firing times (from* e_1 *to* e_2*) in* θ*. This value is only defined if both events are fired in* θ *and* θ *is timing consistent. Formally,*

$$\mathrm{mst}_\theta(\mathsf{e}_1, \mathsf{e}_2) = \max\{\mathrm{ft}(\mathsf{e}_2) - \mathrm{ft}(\mathsf{e}_1) \mid \xrightarrow{\mathsf{e}_1}, \xrightarrow{\mathsf{e}_2} \in \theta \wedge \mathrm{sol}(\mathrm{LP}(\theta)) \neq \emptyset\}$$

We can extend this definition to a set of distinct traces Θ *as follows: the* maximal separation time *of* e_1 *and* e_2 *in* Θ*, denoted as* $\mathrm{mst}_\Theta(\mathsf{e}_1, \mathsf{e}_2)$*, is the maximum value of* $\mathrm{mst}_\theta(\mathsf{e}_1, \mathsf{e}_2)$ *for any* θ *in* Θ*. Formally,* $\mathrm{mst}_\Theta(\mathsf{e}_1, \mathsf{e}_2) = \max\{\mathrm{mst}_\theta(\mathsf{e}_1, \mathsf{e}_2) \mid \forall\theta \in \Theta\}$ ■ 8.1

If $\mathrm{mst}_\Theta(\mathsf{e}_1, \mathsf{e}_2) < 0$, then, in all traces of Θ, $\mathrm{ft}(\mathsf{e}_1) - \mathrm{ft}(\mathsf{e}_2) < 0$, which means that the latest firing time of e_1 is smaller than the earliest firing time of e_2, thus e_1 happens before e_2. This precedence between the events due to their delays can be expressed with a relative timing constraint of the type $\mathsf{e}_1 < \mathsf{e}_2$. Any trace in the set that does not satisfy this relative ordering is known to be timing inconsistent and can be pruned.

However, determining which traces are timing inconsistent using the maximal separation times is a tricky problem. Consider the trace $\{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{a}}\emptyset$ with $\delta(\mathsf{a}) = [2]$ and $\delta(\mathsf{b}) = [1]$. Clearly, the trace is timing inconsistent but $\mathrm{mst}_\theta(\mathsf{a}, \mathsf{b})$ is undefined, because b does not fire in the trace and $\mathrm{sol}(\mathrm{LP}(\theta)) = \emptyset$ as well. This contradicts with the fact that, intuitively, one knows that b has to happen one time unit before a and, consequently, $\mathrm{mst}_\theta(\mathsf{a}, \mathsf{b})$ should value -1. To obtain such result, it is not difficult to see that intuition focuses on the delays and not the actual firing order of the events. We can mimick this analysis by testing all possible interleavings of the events that appear in the enablings of the trace, instead of just looking to the trace. We formalize this idea in next definition.

DEFINITION 8.2 (SET OF TIMING COMPUTATION TRACES OF A TRACE)
Given a distinct trace $\theta = E_1\xrightarrow{\mathsf{e}_1}E_2 \ldots E_n\xrightarrow{\mathsf{e}_n}E_{n+1}$*, let* $\mathrm{expand}(\theta)$ *be the* set of expanded traces of θ*, defined as the set of traces obtained from* θ *by firing the events enabled in* E_{n+1} *in all possible orders. Then, the* set of timing computation traces of θ*, denoted as* $\mathrm{tct}(\theta)$*, is the union of the expanded traces of all the prefixes of* θ*. Formally,*

$$\mathrm{tct}(\theta) = \bigcup_{\forall\theta' \in \mathrm{prefix}(\theta)} \mathrm{expand}(\theta), \quad \textit{where } \mathrm{prefix}(\theta) \textit{ is the set of all prefixes of } \theta.$$

■ 8.2

For instance, for our example trace $\theta = \{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{a}}\emptyset$, we have that

$$\begin{aligned}
\mathrm{prefix}(\theta) &= \{\{\mathsf{a}, \mathsf{b}\}, \{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{a}}\emptyset\} \\
\mathrm{expand}(\{\mathsf{a}, \mathsf{b}\}) &= \{\{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{a}}\{\mathsf{b}\}\xrightarrow{\mathsf{b}}\emptyset, \{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{b}}\{\mathsf{a}\}\xrightarrow{\mathsf{a}}\emptyset\} \\
\mathrm{expand}(\{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{a}}\emptyset) &= \{\{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{a}}\emptyset\} \\
\mathrm{tct}(\theta) &= \{\{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{a}}\{\mathsf{b}\}\xrightarrow{\mathsf{b}}\emptyset, \{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{b}}\{\mathsf{a}\}\xrightarrow{\mathsf{a}}\emptyset, \{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{a}}\emptyset\}
\end{aligned}$$

Now $\mathrm{mst}_{\mathrm{tct}(\theta)}(\mathsf{b}, \mathsf{a}) = -1$ as expected. It is straightforward to extend previous definition to a set of traces Θ by defining $\mathrm{tct}(\Theta) = \bigcup_{\forall\theta\in\Theta} \mathrm{tct}(\theta)$ and $\mathrm{expand}(\Theta) = \bigcup_{\forall\theta\in\Theta} \mathrm{expand}(\theta)$. Given the relevance of the set of timing computation traces of all the traces of a CESAR CS, we will write it as $\mathrm{tct}(\mathrm{CS})$, instead of the longer $\mathrm{tct}(\mathrm{traces}(\mathrm{CS}))$.

If determining the timing consistency of traces using maximal separation times is so tricky, why do we care about this option? The answer is that it worths the effort, because efficient algorithms exist to determine the maximal separation times in event structures. However, the previous formalization is required to understand that what algorithms provide is not always what we require as we will see in §8.3.

8.2 The final objective

We want to use maximal separation times to prune the GRE of a CESAR. That is, given an arc $\mathcal{E}(\mathcal{C}_i)\xrightarrow{\mathsf{e}}\mathcal{E}(\mathcal{C}_j)$ of the GRE we want to know if all traces of the GRE that go through that arc are timing inconsistent or not. If $\Theta_{\mathcal{C}_i\xrightarrow{\mathsf{e}}}$ is the set of such traces, we can answer the question by computing $\mathrm{mst}_{\mathrm{tct}(\Theta_{\mathcal{C}_i\xrightarrow{\mathsf{e}}})}(\mathsf{a},\mathsf{b})$ for all events fired in the traces of the set. Negative maximal separation times indicate relative precedences between events that all these traces must respect. If this is not the case, we know they are all timing inconsistent.

This procedure is not efficient since lots of timing computation traces are shared between the different sets we have to compute. It is a better option to use an inductive approach that takes profit from analysis done in previous arcs of the GRE. For each enabling $\mathcal{E}(\mathcal{C})$, we denote by $\Theta_{\mathcal{C}}$ all the traces reaching it. It is enough to consider the traces in $\mathrm{expand}(\Theta_{\mathcal{C}})$ and compute $\mathrm{mst}_{\mathrm{expand}(\Theta_{\mathcal{C}})}(\mathsf{a},\mathsf{b})$ for all events in $\mathcal{E}(\mathcal{C})$ to determine which outgoing arcs from the enabling are timing consistent or not. This way, all the outgoing traces can be proved timing inconsistent by some of these precedences or if the enabling is not reachable through a sequence of timing consistent arcs.

Further improvements are possible now. Instead of computing $\mathrm{mst}_{\mathrm{expand}(\Theta_{\mathcal{C}})}(\mathsf{a},\mathsf{b})$ for each enabling in the GRE, we can group all these results in a single computation, that is calculating only $\mathrm{mst}_{\Theta'}(\mathsf{a},\mathsf{b})$ for some set of traces Θ', given that a number of conditions are satisfied. We will examinate these conditions in next section, after we have presented the standard algorithms to compute the maximal separation times that directly work on event structures.

8.3 Maximal separation time algorithms on event structures

Maximal separation time algorithms on event structures (or similar structures) is a research field on its own [McM92, Tod93, HB94, EG94, Wal95, YWCI94, CDY99]. In this section we try to present the subject as briefly as possible, but some technical details are required to explain why some previous timing verification approaches using event structures are correct [PCKP00] while others have some flaws [Peñ03].

Maximal separation time algorithms are a family of algorithms which can be classified depending on whether they work on acyclic specifications (for instance the event structures used in this work) or cyclic ones. The problems related to cyclic specifications are harder than the acyclic counterparts, but in our case we only use the algorithms that solve the latter problems as our event structures are always acyclic.

Another possible classification is considering whether computed solutions are exact or aproximated. As usual exactness is sacrificed to obtain a faster algorithm. In our case, taking into account the type of problem we generate, we only use exact algorithms because the cost of analysis is not high compared to the time needed to perform some of the other steps in the verification flow. Moreover approximated algorithms can introduce a source of conservativeness in the methodology, something that we want to avoid as much as possible.

Finally, the last classification that is relevant to our exposition is done according to the type of timing constraints present in the specification. There are three possible types of constraints:

Max constraints Max constraints arise when one wants to express some relationship between events in which one event depends on some others to happen, but it can only occur after *all* of them have fired. This is precisely the semantics of the causality relation in the event structures. These contraints can be translated into linear inequalities of the type:

$$\max_{\forall \mathsf{e}_p : \mathsf{e}_p \prec_\mu \mathsf{e}_i} (\mathrm{ft}(\mathsf{e}_p)) + \delta^l(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{e}_i) \leq \max_{\forall \mathsf{e}_p : \mathsf{e}_p \prec_\mu \mathsf{e}_i} (\mathrm{ft}(\mathsf{e}_p)) + \delta^u(\mathsf{e}_i)$$

Following the notation we introduced for CESARs, we use the term $\mathsf{e}_j \prec_\mu \mathsf{e}_i$ to express that e_j is a direct causal predecessors of e_i, where $\prec_\mu$ is the covering relation of the $\prec$ relation.

Min constraints Min constraints are the opposite of the Max constraints: an event can happen as long as *some* of its causal precedessors has happened. These contraints can be translated into linear inequalities of the type:

$$\min_{\forall \mathsf{e}_p : \mathsf{e}_p \prec_\mu \mathsf{e}_i} (\mathrm{ft}(\mathsf{e}_p)) + \delta^l(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{e}_i) \leq \min_{\forall \mathsf{e}_p : \mathsf{e}_p \prec_\mu \mathsf{e}_i} (\mathrm{ft}(\mathsf{e}_p)) + \delta^u(\mathsf{e}_i)$$

Min constraints appear when OR causality is allowed in event structures, which is not the case in the models we have presented.

Linear constraints Linear constraints are used to express the fact that some event e_i has to occur a number of time units before some other event e_j. Let α be this number of time units, then linear contraints can be translated into linear inequalities of the type:

$$\mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{e}_j) - \alpha$$

Depending on the types of constraints that a problem contains, different algorithms (with different worst case running times) are used. For instance in [McM92] several algorithms were presented: a polynomial algorithm for the timing analysis with *max* constraints only; an exponential, but feasible in practice, algorithm for the case with *max* and linear constraints; and a branch and bound approach for the general case including *min/max* and linear constraints. Later, [Wal95, YWCI94] gave conjectured polynomial algorithms for the case with *max* and linear constraints.

For a CESs, only *max* constraints are needed and the polynomial algorithm of [McM92] can be used. This algorithm computes the maximal separation times that can be obtained from the following system of equations, that represents all the possible firing times that render valid at least one trace of a given CES.

DEFINITION 8.3 (VALID TIMESTAMPS FOR THE TRACES OF A **CES**)
*Given a **CES** $\mathrm{CS} = \langle \Sigma, \prec \rangle$, the system of equations that provides the firing times that render valid at least one of the traces of* CS *is:*

$$\forall \mathsf{e}_i, \mathsf{e}_j \in \Sigma : \mathsf{e}_j \prec_\mu \mathsf{e}_i, \quad \mathrm{ft}(\mathsf{e}_j) \leq \mathrm{ft}(\mathsf{e}_i) - \delta^l(\mathsf{e}_i) \tag{8.1}$$

$$\forall \mathsf{e}_i \in \Sigma, \quad \mathrm{ft}(\mathsf{e}_i) \leq \max_{\forall \mathsf{e}_j \prec_\mu \mathsf{e}_i} (\mathrm{ft}(\mathsf{e}_j)) + \delta^u(\mathsf{e}_i) \tag{8.2}$$

Equations of the type (8.1) and (8.2) are max contraints. *We denote the whole system of equations as* $\mathrm{LP}(\mathrm{CS})$. ■ 8.3

For a CES CS, the system $\mathrm{LP}(\mathrm{CS})$ has always at least one solution. This is easy to see because one can assign the firing times using a BFS schema, following the $\prec$ relation. This system of equations is equivalent to the system obtained by computing $\bigvee_{\forall\theta\in\Theta} \mathrm{LP}(\theta)$, where Θ is the set of full traces of CS.

LEMMA 8.1
Given a CES $\mathrm{CS} = \langle\Sigma, \prec\rangle$, *let G be its GRE. If a timing consistent trace θ reaches an enabling $\mathcal{E}(\mathcal{C})$ of G, then:*

$$\forall \mathsf{a}, \mathsf{b} \in \mathcal{E}(\mathcal{C}), \quad \mathrm{mst}_{\mathrm{expand}(\Theta_{\mathcal{C}})}(\mathsf{a}, \mathsf{b}) < 0 \Leftrightarrow \mathrm{mst}_{\mathrm{traces(CS)}}(\mathsf{a}, \mathsf{b}) < 0$$

■ 8.1

Proof: *See Appendix B.7.*

As a result it is enough to consider the complete traces in a CES to obtain all maximal separation times between all pairs of events. This is the underlying reason why the approach presented in [PCKP00] was correct, although it was never explicitely stated.

To obtain the maximal separation times of a CESAR, the system in Definition 8.3 is not sufficient. First obvious problem is that there is no direct translation for disablings, although restrictions (either induced or not by disablings) can be modeled using linear constraints of the type $\mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{b})$, if event a must happen before event b. Bad news is that the maximal separation times obtained by the algorithms that work with *max* and linear constraints [McM92, Wal95, YWCI94] do not fully correspond to the maximal separation times required to prune the GRE. Instead their outcome are the maximal separation times over the set of *full traces without disablings*, which is formally defined as follows:

DEFINITION 8.4 (SET OF FULL TRACES WITHOUT DISABLINGS OF A **CESAR**)
Given a CESAR $\mathrm{CS} = \langle\Sigma, \prec, \triangleright, <\rangle$, *its set of* full traces without disablings*, denoted as* $\mathrm{twd(CS)}$ *is the set of traces of maximal length of CESAR* $\mathrm{CS}' = \langle\Sigma, \prec, \emptyset, <\rangle$. *The event structure* CS' *is a copy of* CS *but in which disablings are removed, allowing all events to fire. Formally,*

$$\mathrm{twd(CS)} = \{\theta_{\omega,\mathrm{CS}'} \mid \forall\omega \in \mathrm{words}(\mathrm{CS}') \wedge \mathrm{CS}' = \langle\Sigma, \prec, \emptyset, <\rangle \wedge |\omega| {=} |\Sigma|\}$$

■ 8.4

To obtain the maximal separation times over the set of full traces without disablings, algorithms compute the maximum difference between all the events in the following system of equations:

DEFINITION 8.5 (VALID TIMESTAMPS FOR THE TRACES OF A **CESAR**)
Given a CESAR $\mathrm{CS} = \langle\Sigma, \prec, \triangleright, <\rangle$, *the system of equations that provides the firing times that render valid at least one of the full traces without disablings of* CS *is formed by all the equations of* $\mathrm{LP}(\mathrm{CS}^-)$, *where* CS^- *is the underlying CES of* CS, i.e. $\mathrm{CS}^- = \langle\Sigma, \prec\rangle$, *and*

$$\forall \mathsf{e}_i, \mathsf{e}_j \in \Sigma : \mathsf{e}_j <_\mu \mathsf{e}_i \wedge \mathsf{e}_j \not\prec \mathsf{e}_i, \quad \mathrm{ft}(\mathsf{e}_j) \leq \mathrm{ft}(\mathsf{e}_i) \tag{8.3}$$

The relation $<_\mu$ denotes the covering relation of $<$. Equations of the type (8.3) are linear constraints[2]. *We denote the whole system of equations as* $\mathrm{LP(CS)}$. ■ 8.5

The algorithm of [McM92] for *max* and linear constraints tailored for the computation of maximal separation times over the set of full traces without disablings of a CESAR is shown in Figure 8.1. A matrix s is used to store the currently computed maximal separation times between events *e.g.* s_{ij}

[2]A discussion why linear constraints are translated as $\mathrm{ft}(\mathsf{e}_j) \leq \mathrm{ft}(\mathsf{e}_i)$ and not $\mathrm{ft}(\mathsf{e}_j) < \mathrm{ft}(\mathsf{e}_i)$ can be found in Appendix B.14.

```
// CS is a CESAR
maxsep (CS = {Σ = {e_1, e_2 ... e_n}, ≺, ▷, <}) {
    ∀e_i, e_j ∈ Σ,  s_ij ← ∞;
    ∀(e_i, e_j) ∈< \ ≺,  s_ij ← 0;
    ∀e_i ≺_μ e_j,  s_ji ← min(s_ji, −δ^l(e_j));
    return mst_fixpoint({Σ, ≺}, s);
}

// CS is a CES, s is a n × n matrix
mst_fixpoint (CS = {Σ = {e_1, e_2 ... e_n}, ≺}, s) {
    repeat
        ∀e_i, e_j, e_k ∈ Σ,  s_ij ← min(s_ij, s_ik + s_kj);
        ∀e_i, e_j ∈ Σ,  s_ij ← min(s_ij, max_{∀e_k ≺_μ e_j} s_ik + δ^u(e_j));
    until s_ii < 0 or there is no change in any s_ij
    return s
}
```

Figure 8.1. McMillan and Dill maximum separation time algorithm for *max* and linear constraints adapted to analyze CESARs. Matrix position s_{ij} contains the value for $\mathrm{mst}_{\mathrm{twd(CS)}}(\mathsf{e}_i, \mathsf{e}_j)$ when algorithm returns. If for some i, s_{ii} becomes less than 0, then there is no solution for the system of equations.

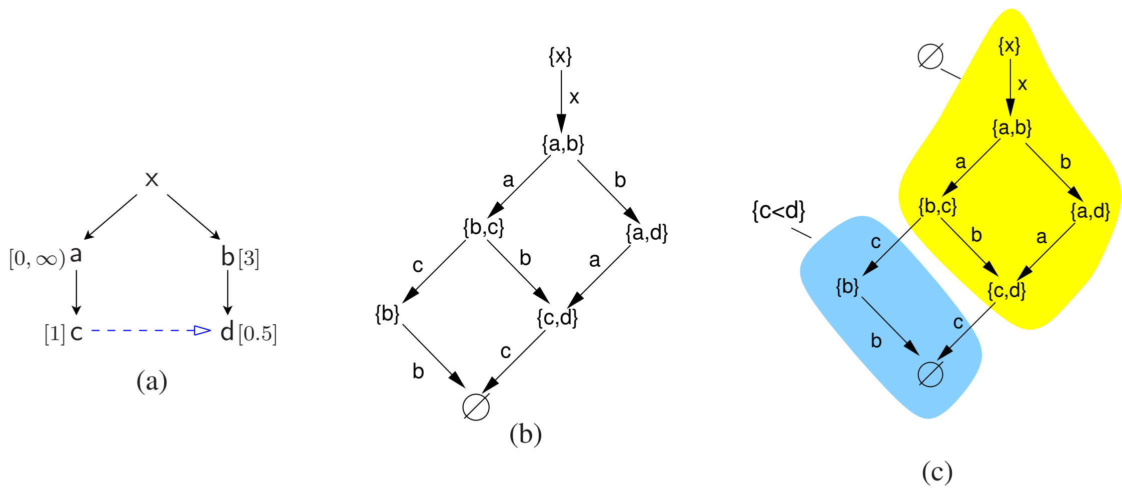


Figure 8.2. Example of timing analysis of a CESAR: (a) CESAR with a disabling. (b) Its GRE. (c) The two timing zones present in the GRE: $\emptyset$ and $\{\mathsf{c} < \mathsf{d}\}$.

is the currently computed maximal separation time from e_i to e_j. Algorithm works by "tightening" these separation times until all equations are satisfied (*i.e.* in a complete iteration no s_{ij} changed) or the system is found inconsistent (some s_{ii} is less than 0, which is a contradiction because the maximal separation time of an event with respect to itself can only be 0).

Contrary to what happens in CESs, we cannot compute the required maximal separation times of a CESAR by just using the full traces (even if we remove the disablings). For instance in Figure 8.2 (a) we can see a CESAR in which the firing of c disables d. Let us name this CESAR as CS_d. Now $\mathrm{LP}(\mathrm{CS}_d)$ is:

$$\mathrm{ft}(\mathsf{x}) \leq \mathrm{ft}(\mathsf{a}) \qquad \mathrm{ft}(\mathsf{d}) = \mathrm{ft}(\mathsf{b}) + 0.5$$
$$\mathrm{ft}(\mathsf{b}) = \mathrm{ft}(\mathsf{x}) + 3 \qquad \mathrm{ft}(\mathsf{c}) \leq \mathrm{ft}(\mathsf{d})$$
$$\mathrm{ft}(\mathsf{c}) = \mathrm{ft}(\mathsf{a}) + 1$$

Then $\mathrm{ft}(\mathsf{c}) \leq \mathrm{ft}(\mathsf{b}) + 0.5$ so $\mathrm{ft}(\mathsf{a}) + 1 \leq \mathrm{ft}(\mathsf{b}) + 0.5$ which means that $\mathrm{ft}(\mathsf{a}) - \mathrm{ft}(\mathsf{b}) \leq -0.5$ and consequently $\mathrm{mst}_{\mathrm{twd}(\mathrm{CS}_d)}(\mathsf{b}, \mathsf{a}) = -0.5$. With this information one could erroneously deduce that $\mathsf{a} < \mathsf{b}$[3]. However, take the set of traces $\Theta_{\{\mathsf{a},\mathsf{b}\}}$ reaching enabling $\{\mathsf{a}, \mathsf{b}\}$ in the corresponding GRE of CS_d (Figure 8.2 (b)). The set contains only a single trace, namely $\{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a}, \mathsf{b}\}$. The set of expanded traces $\mathrm{expand}(\Theta_{\{\mathsf{a},\mathsf{b}\}})$, contains two traces, in particular trace $\{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{b}}\{\mathsf{a}\}\xrightarrow{\mathsf{a}}\emptyset$. Maximal separation time from event b to event a in that case is bounded by equation $\mathrm{ft}(\mathsf{a}) - \mathrm{ft}(\mathsf{b}) \leq 0.5$. Consequently, $\mathrm{mst}_{\mathrm{expand}(\Theta_{\{\mathsf{a},\mathsf{b}\}})}(\mathsf{b}, \mathsf{a}) = 0.5$ showing that, for some enablings, it is possible that the maximal separation times obtained using the full traces without disablings are smaller than the ones obtained using the set of expanded traces. The consequences in the methodology of using this result instead of the correct one can be seen in Appendix C.2.

The cause of this phenomenon is that linear constraints remove from consideration some of the traces, hence it is possible that the maximum separation time found by the algorithms is less than the real one, simply because it computes the maximum out of a subset, and the maximum of a subset is always less or equal than the maximum of the full set.

Fortunately, it is possible to use the algorithm to compute the maximal separation times over the set of timing computation traces by performing several timing analysis, one for each zone of the GRE in which a different set of restrictions are currently satisfied.

For instance in CS_d there is a single restriction $\mathsf{c} < \mathsf{d}$, induced by the disabling $\mathsf{c} \rhd \mathsf{d}$. For this reason the GRE of Figure 8.2 (b) must be split in two different timing zones, one in which $\mathsf{c} < \mathsf{d}$ is satisfied and one in which it is not. As this depends on whether event c has fired or not, the resulting division is shown in subfigure (c) [4].

Another example is shown in Figure 8.3 (a). In this case two disablings are present: $\mathsf{a} \rhd \mathsf{f}$ and $\mathsf{d} \rhd \mathsf{c}$. The corresponding GRE in subfigure (b) is then divided in four zones, depending on whether events a and d have fired or not. In §8.4 we will prove that working with zones allows to obtain an equivalent information (in terms of which arcs must be declared as lazy) given by $\mathrm{mst}_{\Theta_{\mathcal{C}}}(\mathsf{e}_1, \mathsf{e}_2)$ in each enabling $\mathcal{E}(\mathcal{C})$, but using $\mathrm{mst}_{\mathrm{twd}(\mathrm{CS})}(\mathsf{e}_1, \mathsf{e}_2)$ instead.

8.4 Using timing zones to compute timing consistency

We define as a timing zone in a GRE, the set of enablings of the GRE for which the same set of linear constraints (from the CESAR that produced the GRE) are satisfied. This set of linear constraints identifies uniquely the timing zone.

Given a timing zone in a GRE derived from a CESAR CS, the set of traces of that zone that are timing consistent can be determined using the maximal separation times over the full traces without disablings of a CESAR CS' that is obtained from CS by removing all restrictions not related to the timing zone.

[3] Actually from $\mathrm{mst}_{\mathrm{twd}(\mathrm{CS}_d)}(\mathsf{b}, \mathsf{a}) = -0.5$ it is deduced that $\mathsf{a} < \mathsf{b}$ *but only* for traces in which $\mathsf{c} < \mathsf{d}$, hence it would be wrong to mark arc $\{\mathsf{a}, \mathsf{b}\}\xrightarrow{\mathsf{b}}\{\mathsf{a}, \mathsf{d}\}$ as lazy in the GRE, because there are traces of the system going through that arc in which condition $\mathsf{c} < \mathsf{d}$ is not satisfied.

[4] In this particular example, however, last timing zone (*i.e.* $\mathsf{c} < \mathsf{d}$) is useless because there is no more than one event enabled in each enabling of this zone. Should any concurrent event be added to the CESAR, then the timing zone would be required.

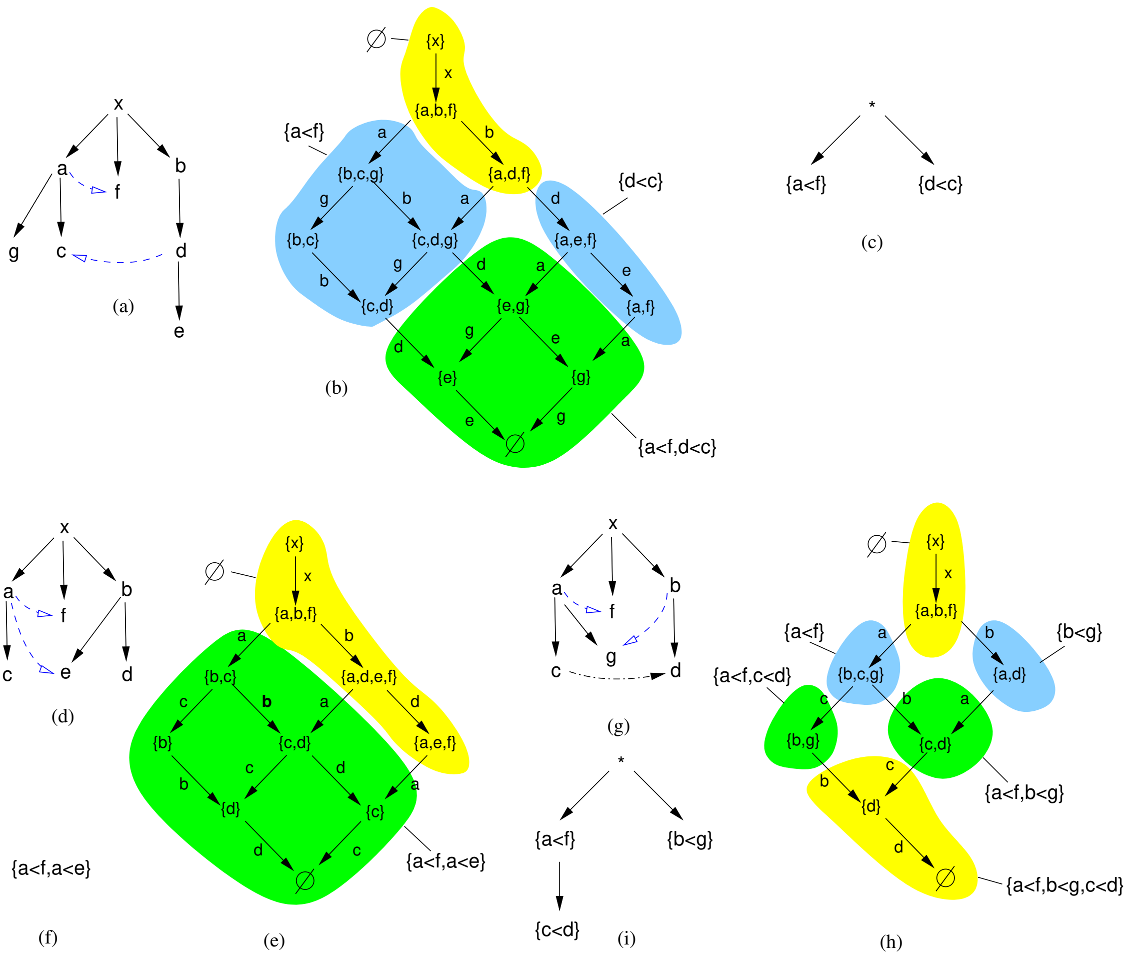


Figure 8.3. Example illustrating timing zones and dependency between restrictions: (a) CESAR with two independent disablings. (b) Its GRE. Four timing zones present: $\emptyset$, $\{a<f\}$, $\{d<c\}$ and $\{a<f, d<c\}$. (c) Restriction dependency graph of the previous GRE. The dummy event "*" is depicted whenever there are multiple initial events, as in this case, to make the graph connected. (d) CESAR with two mutually dependant disablings. (e) Its GRE. Only two timing zones present in the GRE: $\emptyset$ and $\{a<f, a<e\}$. (f) Restriction dependency graph of the previous GRE, formed by a single restriction. (g) CESAR with two independent disablings and a restriction that depends on one of the disablings. (h) Its GRE. Restriction $c<d$ depends on $a<f$ because in all timing zones in which $c<d$ appears, restriction $a<f$ also appears. (i) Restriction dependency graph of the previous GRE.

DEFINITION 8.6 (**CESAR** ASSOCIATED TO A TIMING ZONE)

Given a CES $\mathrm{CS} = \langle \Sigma, \prec \rangle$ *and set of precedences* TZ *which identify a timing zone in a GRE derived from* CS*, we define the associated CESAR to the timing zone* TZ *as the CESAR* $\mathrm{CS}_{\mathrm{TZ}} = \langle \Sigma, \prec, \emptyset, (\mathrm{TZ} \cup \prec)^* \rangle$. ■ 8.6

Now we can obtain the equivalent to Lemma 8.1 but for CESARs.

LEMMA 8.2

Given a CESAR $\mathrm{CS} = \langle \Sigma, \prec, \triangleright, < \rangle$*, let* G *be its GRE. If a timing consistent trace* θ *reaches an enabling* $\mathcal{E}(\mathcal{C})$ *in timing zone* TZ *of* G*, then:*

$$\forall a, b \in \mathcal{E}(\mathcal{C}), \quad \mathrm{mst}_{\mathrm{expand}(\Theta_{\mathcal{C}})}(a, b) < 0 \Leftrightarrow \mathrm{mst}_{\mathrm{twd}(\mathrm{CS}_{\mathrm{TZ}})}(a, b) < 0$$

■ 8.2

Proof: *See Appendix B.8.*

The relationship between the relative order of the events in timing consistent traces and the maximal separation time on the set of traces without disablings is summarized in the following theorem.

THEOREM 8.3
Given a CESAR $\mathrm{CS} = \langle \Sigma, \prec, \triangleright, < \rangle$ *and an enabling* $\mathcal{E}(\mathcal{C})$ *of its GRE that belongs to a timing zone* TZ*, all timing consistent traces departing from enabling* $\mathcal{E}(\mathcal{C})$ *(that is traces reaching* $\mathcal{E}(\mathcal{C})$ *and then firing one of the enabled events) satisfy all relative orderings* $<_{\mathrm{lzy}}$ *where* $<_{\mathrm{lzy}} = \{(\mathsf{a}, \mathsf{b}) \mid \mathrm{mst}_{\mathrm{twd}(\mathrm{CS_{TZ}})}(\mathsf{b}, \mathsf{a}) < 0\}$*. In case all maximal separation times are undefined because all traces in* $\mathrm{twd}(\mathrm{CS_{TZ}})$ *are timing inconsistent, then no departing trace from* $\mathcal{E}(\mathcal{C})$ *is timing consistent.* ■ 8.3

Proof: *See Appendix B.9.*

This theorem allows us to use the standard maximal separation time algorithms to determine which arcs should be declared as lazy in each timing zone.

8.4.1 An efficient algorithm to obtain the $\mathrm{mst_{twd}}$ in all timing zones

It might seem that an exponential number of applications of the algorithm are always required to complete the analysis (exponential respect to the number r of restrictions) as there are 2^r possible different sets of restrictions, hence 2^r possible different timing zones in the GRE. However this is not usually true for two reasons: first of all not all restrictions are independent one from the other. For instance in Figure 8.3 (d) there are also two disablings but the corresponding GRE has only two timing zones. The reason is that both restrictions depend on the same event (in this case event a) so either both are not yet satisfied or both are satisfied. We will more carefully treat this subject in the following. The second reason why exponentiality does not arise in practice is that timing analysis is conducted in a way that prevents the algorithm from analyzing timing zones we know will be avoided (the results of the previously analyzed timing zones provide this information).

We have applied the terms *independent* and *dependent* to restrictions intuitively, but some formal meaning must be attached to these adjectives. We will say that a restriction r_1 depends on another restriction r_2 whenever if r_1 is true, then also r_2 is satisfied. The conditions under which the dependency between restriction arises in a CESAR are described in the next lemma.

LEMMA 8.4 (DEPENDENCY BETWEEN RESTRICTIONS)
Given a CESAR $\mathrm{CS} = \langle \Sigma, \prec, \triangleright, < \rangle$*, a restriction* $r_1 = \mathsf{a} < \mathsf{b}$ *depends on* $r_2 = \mathsf{c} < \mathsf{d}$*, if and only if,* $\mathsf{c} < \mathsf{a}$ *or* $\mathsf{c} = \mathsf{a}$ (i.e. $\mathsf{c} \in (\downarrow\{\mathsf{a}\})_<$*).*
In the case of $\mathsf{c} = \mathsf{a}$ *we say that they are mutually dependant.* ■ 8.4

Proof: *A restriction* $\mathsf{a} < \mathsf{b}$ *in the context of a GRE defined by a CESAR can only have two possible states: either restriction is not yet satisfied or it is satisfied. The change between these two states is produced by the firing of event* a*. If restriction* $\mathsf{c} < \mathsf{d}$ *must be satisfied whenever event* a *has fired, then event* c *has to be fired always before event* a *or be the same event* (i.e. $\mathsf{c} = \mathsf{a}$*). Given an event* e_2 *in a CESAR, all the events* e_1 *always fired before it must satisfy* $\mathsf{e}_1 < \mathsf{e}_2$*, hence in our case* $\mathsf{c} < \mathsf{a}$.

Once dependency has been established, given a set of restrictions one can create a dependency graph among them. We will use a CES as the mathematical formalism for this task. To do so one has to treat carefully mutually dependant restrictions: they must be considered as a unity, otherwise the precedence relation violates the antisymmetry property.

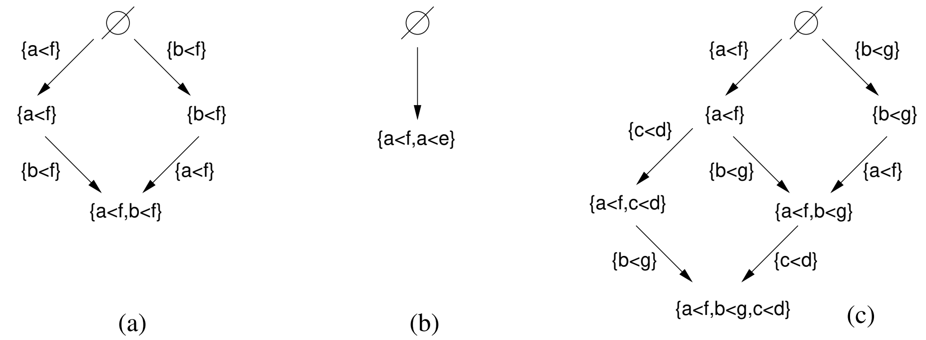


Figure 8.4. Examples of Graph or Reachable Timing Zones: (a,b,c) GRTZs of the dependency graphs of Figures 8.3 (c,f,i) respectively.

Definition 8.7 (Restriction dependency graph)
Given a CESAR $\mathrm{CS} = \langle \Sigma, \prec, \rhd, < \rangle$ *we define its* restriction dependency graph *as a CES* $\mathrm{DG} = \langle \Sigma_D, \prec_D \rangle$ *such that:*

- *The set of events is formed by the set of restrictions which have the same predecessor event. Formally,* $\Sigma_D = \{R \subseteq <_\mu \mid \forall \mathsf{a} < \mathsf{b} \in R, \mathsf{c} < \mathsf{d} \in R \Rightarrow \mathsf{c} = \mathsf{a} \wedge \forall \mathsf{a} < \mathsf{d} \in <_\mu \Rightarrow \mathsf{a} < \mathsf{d} \in R\}$. *This simply creates a singleton* $\{\mathsf{a} < \mathsf{b}\}$ *if there is no other restriction in* $<$ *of the type* $\mathsf{a} < \mathsf{d}$*, otherwise creates a set containing all mutually dependant restrictions. This allows to avoid problems when defining the precedence relation* $\prec_D$*. Note that each event* e *in* Σ_D *can be uniquely identified by the event that appears as predecessor in* $<$ *in all the restrictions of the set. We will note this event as* $\mathrm{pred}(\mathsf{e})$*. For instance* $\mathrm{pred}(\{\mathsf{c} < \mathsf{d}, \mathsf{c} < \mathsf{e}\}) = \mathsf{c}$.
- $\forall \mathsf{e}_1, \mathsf{e}_2 \in \Sigma_D, e_1 \prec_D e_2 \Leftrightarrow \mathrm{pred}(\mathsf{e}_1) < \mathrm{pred}(\mathsf{e}_2)$. *That is, we establish a precedence relation in* DG *between* e_1 *and* e_2*, if and only if, all the restrictions in set* e_2 *imply all the restrictions in the set* e_1.

■ 8.7

For instance, in Figure 8.3 (a), there are two restrictions ($\mathsf{a} < \mathsf{f}$ and $\mathsf{d} < \mathsf{c}$) induced by two disablings. The restriction dependency graph of this CESAR is $\mathrm{DG}_1 = \langle \{\{\mathsf{a} < \mathsf{f}\}, \{\mathsf{d} < \mathsf{c}\}\}, \emptyset \rangle$ shown in subfigure (c). In this case there is no dependency between the two restrictions. For the CESAR of Figure 8.3 (d) we obtain $\mathrm{DG}_2 = \langle \{\{\mathsf{a} < \mathsf{f}, \mathsf{a} < \mathsf{e}\}\}, \emptyset \rangle$, depicted in subfigure (f), in which there is only one "event" formed by both restrictions as they are mutually dependant. Finally, the dependency graph of the CESAR of subfigure (g) is $\mathrm{DG}_3 = \langle \{\{\mathsf{a} < \mathsf{f}\}, \{\mathsf{b} < \mathsf{g}\}, \{\mathsf{c} < \mathsf{d}\}\}, \{\mathsf{a} < \mathsf{f}\} \prec \{\mathsf{c} < \mathsf{d}\} \rangle$ represented in subfigure (i).

The dependency graph is useful because by building its GRC we can know exactly which timing zones will appear in the GRE we are analyzing. To avoid confusions, we will call the GRC of a dependency graph a *Graph of Reachable Timing Zones* (GRTZ). As an example the GRTZs of the dependency graphs of Figure 8.3 (c,f,i) are depicted in Figure 8.4 (a,b,c) respectively. Each node in the GRTZ represents a timing zone that appears in our GRE, so we will compute the *maximal separation times* for each one of this timing zones, by applying one of the algorithms for *max* and *linear* constraints. In each timing zone the only linear constraints considered are the restrictions that define the timing zone.

The algorithm of [McM92] proceeds iteratively *tightening* the separation times by checking that all equations are satisfied *i.e.* it reduces the separation times until a fixpoint is reached in which all equations hold. As all max equations are common to all timing zones in the GRE, we do avoid unnecessary recomputation by applying the algorithms in a BFS manner to the GRTZ. That is, we start by timing zone $\emptyset$ in which only *max* constraints are present and we run the algorithm of Figure 8.1. Then we proceed to the timing zones in the next level by adding the corresponding linear equations. Then we *copy* the separation times obtained in timing zone $\emptyset$ and use them as the starting point for the computation of this new timing zone in which additional linear equations have been added. This allows us to avoid repeated iterations. We can do so because when linear equations are added the only thing they can do is restrict the firing times of the events, hence they might only decrease their separation times as next lemma formalizes.

LEMMA 8.5
Given two ***CESAR****s,* $\mathrm{CS1} = \{\Sigma, \prec, \rhd, <_1\}$ *and* $\mathrm{CS2} = \{\Sigma, \prec, \rhd, <_2\}$*, if* $<_1 \subseteq <_2$ *then* $\forall \mathsf{e}_i, \mathsf{e}_j \in \Sigma,\ \mathrm{mst}_{\mathrm{twd(CS1)}}(\mathsf{e}_i, \mathsf{e}_j) \geq \mathrm{mst}_{\mathrm{twd(CS2)}}(\mathsf{e}_i, \mathsf{e}_j)$. ■ 8.5

Using this lemma one can devise an efficient algorithm to compute the maximal separation times of each timing zone as the one given in Algorithm 8.1.

Algorithm 8.1: maxsepAllTZ

input : $G = \langle V = \{\mathrm{TZ}_1, \mathrm{TZ}_2 \dots \mathrm{TZ}_m\}, E\rangle$ is a GRTZ. Subindexes of timing zones of G (*i.e.* TZ_i) indicate the order in which the timing zone is visited in a BFS traversal.
input : $\mathrm{CS} = \langle \Sigma = \{\mathsf{e}_1, \mathsf{e}_2 \dots \mathsf{e}_n\}, \prec, \rhd, <\rangle$ is a CESAR
output : s is a vector of matrices, containing the maximal separation times for each zone. $s_{xy}[\mathrm{TZ}_i]$ indicates the maximal separation time from event x to y in timing zone TZ_i.

1 $s[\mathrm{TZ}_1] \leftarrow \texttt{maxsep}(\langle \Sigma, \prec, \emptyset, \prec\rangle)$ ▷ Compute max. sep. times of context $\emptyset$
2 consistent $[\mathrm{TZ}_1] \leftarrow true$ ▷ Initial context is always consistent
3 **for** $i \leftarrow 2$ **to** n **do**
4   **forall** $j, k : 1 \leq j, k \leq |\Sigma|$ **do** $s_{jk}[\mathrm{TZ}_i] \leftarrow \min_{\forall p : \mathrm{TZ}_p \longrightarrow \mathrm{TZ}_i \in E}(s_{jk}[\mathrm{TZ}_p])$
5   **forall** $(\mathsf{e}_x < \mathsf{e}_y) \in \mathrm{TZ}_i$ **do** $s_{yx}[\mathrm{TZ}_i] \leftarrow \min(0, s_{yx}[\mathrm{TZ}_i])$
6   consistent $[\mathrm{TZ}_i] \leftarrow \forall(\mathrm{TZ}_p \longrightarrow \mathrm{TZ}_i) \in E$, consistent $[\mathrm{TZ}_p] \wedge \nexists (\mathsf{e}_x < \mathsf{e}_y) \in \mathrm{TZ}_i : s_{xy}[\mathrm{TZ}_p] < 0$
7   **if** consistent $[\mathrm{TZ}_i]$ **then** $s[\mathrm{TZ}_i] \leftarrow \texttt{mst\_fixpoint}(\{\Sigma, \prec\}, s[\mathrm{TZ}_i])$
8 **return** s

The algorithm proceeds as follows: first of all the maximal separation times of the timing zone $\emptyset$ are computed (line 1). As this system of equations does not contain any linear equation it has always a solution hence it is marked as consistent (line 2). Then we proceed by traversing the timing zones in a BFS manner. For each timing zone we take the minimum of all separation times of all predecessor timing zones. As current timing zone will contain all equations in any of the predecessor timing zones, it is obvious that the separation times will be at least the minimum ones (line 3). Then we incorporate all linear equations that define the timing zone by taking the minimum between 0 and the previously computed value (line 4). The consistency of the timing zone can be deduced by the values of the previous zones: if all predecessor zones were consistent and in none of them we found a precedence between events that violates any of the precedences that define the current timing zone, then the zone is consistent (line 6). Finally only if the timing zone was found to be consistent, the algorithm for tightening the equation system is called (line 7).

8.4.2 Retrieving relative time information

The maximal separation times computed by the timing analysis algorithm can be incorporated into the CESAR CS in the form of relative timing constraints. Given a timing zone TZ, if $\mathrm{mst}_{\mathrm{twd(CS_{TZ})}}(\mathsf{e}_2, \mathsf{e}_1) < 0$, then e_1 precedes e_2 in the domain of TZ and all subsequent timing zones. In this case we can also say that e_2 is lazy to fire until e_1 has fired. We refer to the new relations obtained from timing information as *lazy relations*, because of the similarity with the notion of laziness in LzTSs (§2.3).

In [PCKP00, Peñ03] these constraints were added directly to the CES as extra causal relations, *i.e.* added to the $\prec$ relation. However, this modifies the set of enabled events in each configuration, causing undesired sideeffects (see Appendix A.4.1). To correct this problem, lazy relations and causal relations must be kept separated. For this reason we define a new CESAR derivative that incorporates lazy relations as an independent element.

DEFINITION 8.8 (LAZY **CESAR** GENENERATED BY A TRACE)
Let $\text{CS} = \langle \Sigma, \prec, \triangleright, <_{\text{CS}} \rangle$ *be a CESAR. The tuple* $\text{LCS} = \langle \text{CS}, <_{\text{lzy}}, \text{context} \rangle$ *is called a* lazy causal event structure *(LzCES), where* $< \subseteq <_{\text{lzy}} \subseteq \Sigma \times \Sigma$ *is a set of* lazy relations *such that* $<_{\text{lzy}} = \{(e_i, e_j) \in \Sigma \times \Sigma \mid \exists \text{TZ} \in \text{TZS} : \text{mst}_{\text{twd}(\text{CS}_{\text{TZ}})}(e_j, e_i) < 0 \wedge e_j, e_i \notin (\downarrow\{\text{pred(TZ)}\})_{<_{\text{CS}}} \wedge \nexists d \in (\downarrow\{\text{pred(TZ)}\})_{<_{\text{CS}}} \cup \text{fbe}(b, <_{\text{CS}}) : d \triangleright e_i \vee d \triangleright e_j\}$ *where:*

- TZS *is the set of timing zones of* CS*,*
- context *is a function that, given a precedence in* $<_{\text{lzy}}$ *returns the set of smaller timing zones for which the precedence holds, and*
- $\text{fbe}(b, <_{\text{CS}})$ *is the set of events than fire before the enabling of event* b*,* i.e. $\text{fbe}(b, <_{\text{CS}}) = (\downarrow\{e \mid e \prec_\mu b\})_{<_{\text{CS}}}$.

■ 8.8

One could think of adding the relative timing relations into the $<_{\text{CS}}$, but this is not possible either. The $<_{\text{CS}}$ relation identifies which traces will be analyzed and which not, while $<_{\text{lzy}}$ relation expresses which traces are timing consistent and which not. It is perfectly possible that traces "falling out" from GRE (thus not satisfying $<_{\text{CS}}$ relation) are found timing inconsistent (thus not satisfying $<_{\text{lzy}}$ relation) or timing consistent (hence satisfying $<_{\text{lzy}}$ relation).

We use the outcome of Algorithm 8.1 to compute the relation $<_{\text{lzy}}$. When the algorithm finishes we have a GRTZ in which maximal separation times have been computed for each one of the reachable timing zones. Due to Lemma 8.5, once a relative ordering between two events is established in one timing zone, it is satisfied in all timing zones that are reachable from this one.

Relative orderings found in a timing zone have effect only if several conditions are fulfilled. First, the events of the precedence should not have been already fired prior to entering that timing zone (like it happens with $a < b$ in Figure 8.2, where $a < b$ can only be derived in timing zone $\{c < d\}$, but event a has been already fired when this timing zone is reached). This is the reason for the existence of the "$e_j, e_i \notin (\downarrow \{\text{pred(TZ)}\})_{<_{\text{CS}}}$" check in the definition of $<_{\text{lzy}}$. Second condition is that none of the events in $(\downarrow \{\text{pred(TZ)}\})_{<_{\text{CS}}}$ or in $\text{fbe}(b, <_{\text{CS}})$ disables any of the events in the precedence. Given a precedence $a < b$, it starts to have a "pruning" efect only when event b is enabled and a has not yet fired. If previously either one of the two events is disabled, then the precedence does not prune any path and is also useless.

Figures 5.3 (c) and 5.4 (c), on page 66, show LzCESs obtained after the timing analysis on the CESAR of Figures 5.3 (b) and 5.4 (b), respectively. The lazy relations corresponding to the relative timing information are depicted as dotted arcs. As usual we only draw the covering relation of the lazy relation $<_{\text{lzy}}$ (*i.e.* redundant arcs by transitivity are removed) and this only if there is no previous arc in the CESAR (whatever it is a causality/disabling/restriction arc). An important information is the context in which the lazy relations are valid, thus in the drawings we sometimes annotate lazy arcs with their context. However, we can guarantee that the context of a precedence is always satisfied when the point where it would start having some effect on the GRE is reached.

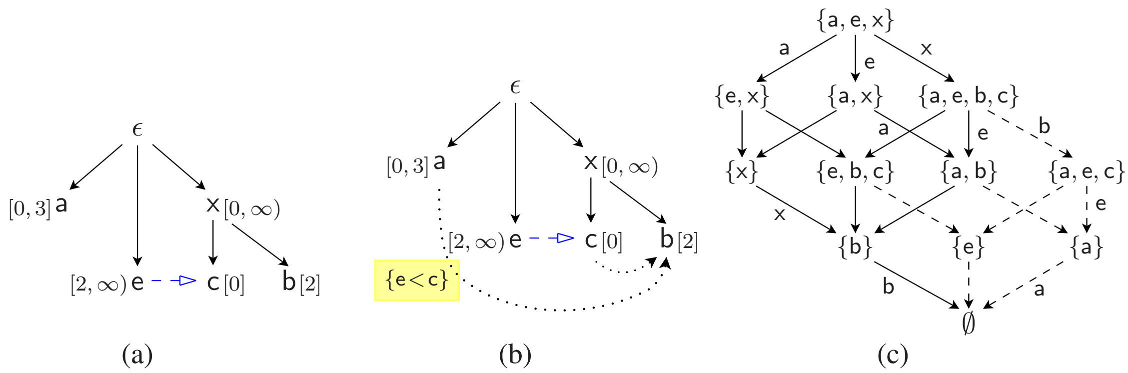


Figure 8.5. Examples of a LzCES: (a) CESAR with a disabling. (b) Its LzCES. In timing zone $\emptyset$ we find precedence $\mathsf{c} <_{\text{lzy}} \mathsf{b}$, while in timing zone $\{\mathsf{e}<\mathsf{c}\}$ we have $\mathsf{a} <_{\text{lzy}} \mathsf{b}$. Since enablings in which a and b are enabled can be reached without satisfying $\mathsf{e}<\mathsf{c}$ the arc is annotated with the context. (c) Pruned GRE using the $<_{\text{lzy}}$ relation. The arcs $\{\mathsf{a},\mathsf{e},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{b}}$ and $\{\mathsf{e},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{b}}$ are found inconsistent due to $\mathsf{c} <_{\text{lzy}} \mathsf{b}$, while arc $\{\mathsf{a},\mathsf{b}\}\xrightarrow{\mathsf{b}}$ is found inconsistent due to $\mathsf{a} <_{\text{lzy}} \mathsf{b}$. Note that this enabling belongs to timing zone $\{\mathsf{e}<\mathsf{c}\}$.

The reason for that is simple: an event of an enabling can only be declared lazy if it is not ready to fire when another enabled event can no longer wait. Consequently the arcs of an event can be declared lazy at the beginning, but at some point the event must become ready to fire. After this point, it will fire (except for disabled events, which can be disabled either when they are still not ready to fire, or when they are). This means that the arcs of an event can be lazy and become "non-lazy" afterwards in the GRE but that is impossible that, if an arc of an event is not lazy, it becomes lazy by the firing of some other event. Since arcs cannot become lazy, and because context of lazy relations would not be guaranteed to be satisfied only in such situation, then we must deduce that context will always be satisfied. We illustrate this with an example.

EXAMPLE 8.1 *In Figure 8.5 (a) we can see a* CESAR *that contains a disabling. In subfigure (b) we show the* LzCES *obtained after timing analysis on the previous* CESAR*. Two lazy relations are present:* $\mathsf{c} <_{\text{lzy}} \mathsf{b}$ *and* $\mathsf{a} <_{\text{lzy}} \mathsf{b}$*. These precedences have different context, namely:* $\text{context}(\mathsf{c} <_{\text{lzy}} \mathsf{b}) = \emptyset$ *and* $\text{context}(\mathsf{a} <_{\text{lzy}} \mathsf{b}) = \{\mathsf{e}<\mathsf{c}\}$*. We have annotated this latter context in the corresponding lazy arc of the figure. Note that* $\mathsf{a} <_{\text{lzy}} \mathsf{b}$ *is not true if* e *does not fire before* c*. However, when the first enabling in which* a *and* b *are enabled is reached, event* b *cannot fire because* $\mathsf{c} <_{\text{lzy}} \mathsf{b}$*, and when event* e *is fired, reaching enabling* $\{\mathsf{a},\mathsf{b}\}$*, then* $\mathsf{a} <_{\text{lzy}} \mathsf{b}$ *has its context satisfied.*

Another interesting element of this example is that it illustrates that lazy relations involving disabled events, like $\mathsf{c} <_{\text{lzy}} \mathsf{b}$*, must only be satisfied as long as the event is not disabled. In this case, there is an arc in which event* b *is not lazy. Any trace through that arc, does not fire event* c*. However there are timing consistent traces crossing it. For instance* $\{\mathsf{a},\mathsf{e},\mathsf{x}\}\xrightarrow{\mathsf{a}}\{\mathsf{e},\mathsf{x}\}\xrightarrow{\mathsf{e}}\{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{b}\}\xrightarrow{\mathsf{b}}$ *with timestamps* $(0, 2, 3, 5)$. ■ 8.1

Although context is always satisfied, it is still a valuable information to determine which events in a CESAR are relevant for a particular lazy precedence. For instance, this information is used in §13.3.1 to obtain LzCES that include only the relevant events to avoid a particular failure.

A CESAR is a succint form to express a set of traces. In the same spirit a LzCES is a succint form to express a set of lazy/valid traces. The valid ones will be the traces satisfying all relative ordering of the $<_{\text{lzy}}$ relation. Otherwise they will be defined as lazy because we know they are timing inconsistent by Theorem 8.3. For instance if a lazy relation is found (*e.g.* $\mathsf{a} <_{\text{lzy}} \mathsf{b}$) by performing timing analysis on a CESAR, this means that all traces in which b happened before a are timing inconsistent (except if event a is disabled before the firing of b). Conversely, if no such lazy relation is present (*i.e.* $\mathsf{a} \not<_{\text{lzy}} \mathsf{b}$) means that *at least* there is one trace in which b happens before a that is timing consistent.

This link between timing consistency checking of traces and timing analysis of CESARs is fundamental to understand how we can relate results obtained in both areas. For instance it can be the case that a particular trace that satifies all the relative orderings of $<_{\text{lzy}}$ relation is timing inconsistent (see an example in Figure 11.4). However methodology can be adapted to solve the problem as we explain Chapter 13.

8.5 Derivation of an ECTRA from a LzCES

Once timing analysis on a CESAR has been successfully conducted and we have a LzCES, we would like to incorporate into our system all the timing information found. To do so we build an ECTIA from the original CESAR (following the procedure of §7.3.1) and then convert it into an ECTRA. The conversion reduces to identifying the arcs of the ECTIA that are timing inconsistent and declaring them lazy. This way we obtain the ECTRA that, when composed with the original system, removes all the traces that are enabling compatible with the timing inconsistent ones.

To identify the timing inconsistent arcs we use the $<_{\text{lzy}}$ of the LzCES. An arc is declared lazy in the ECTRA if:

- either the source configuration is not reachable through any timing consistent arc or,
- if configuration is reachable, the firing of that particular event violates some of the relative orderings in the $<_{\text{lzy}}$ relation, unless relative ordering is of the type $\mathsf{a} <_{\text{lzy}} \mathsf{b}$ and event a has been already disabled when the configuration is reached (like in Example 8.1).

As the $<_{\text{lzy}}$ relation comes from the timing analysis performed on the CESAR, which only considers the events appearing in the configurations, the only arcs that can be declared lazy are the arcs that fire events enabled in the configurations. Thus, only the advance, reenter, or break arcs that correspond to the events enabled in a configuration can be declared lazy. Keep arcs can only be declared lazy if source configuration is not reachable through any timing consistent arc, as by definition they never correspond to events enabled in a configuration.

8.6 Additional remarks

We delayed the presentation of some relevant related topics in order not to disturb the general flow of the chapter. This section treats them briefly.

8.6.1 Limitations of other event structures for timing analysis

Now we can explain the reason why we prefer event structures with asymmetric conflict (*i.e.* CESAs) that have a more reduced state space than the already available in the literature. In Figure 8.6 we show the GREs of the GRCs shown in Figure 7.2. Let us say that arc $\{\mathsf{a},\mathsf{b},\mathsf{c}\} \xrightarrow{\mathsf{b}} \{\mathsf{c}\}$ is considered to be a failure arc and we want to know if it is timing consistent or not. Why should we want to analyze the behavior of traces not directly related to that failure like all the traces in the timing zone $\{\mathsf{a} < \mathsf{b}\}$? It is true that, potentially, other failures could be analyzed because of the extra traces considered but this is not interesting to do so for several reasons:

- It is not sure that other failures are affected by the analysis on the extra traces.

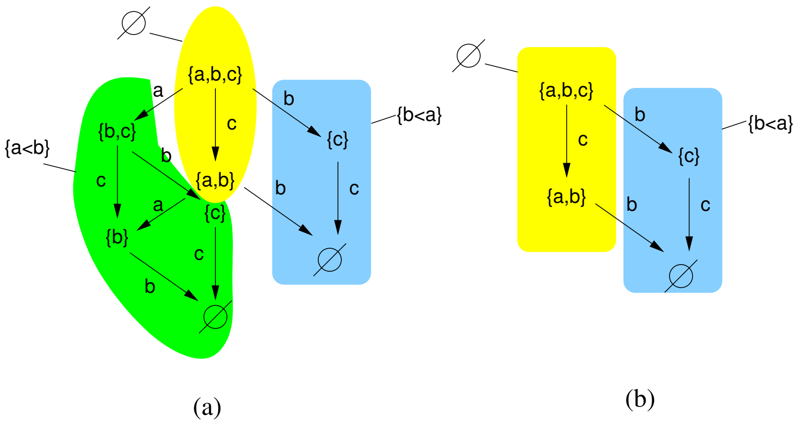


Figure 8.6. Example showing timing analysis differences between two event structure models with asymmetric conflict: (a) GRE of an event structure with asymmetric conflict in which $\mathsf{b} \triangleright \mathsf{a}$ using the same model as in Figure 7.2 (a). (b) GRE of the equivalent CESA.

- The number of timing zones to consider greatly increases.

- As we will see later, timing analysis is only exact when the behavior represented by the GRE really exists in the state space of the system. This implies that an exhaustive checking of the CESAR to ensure that it represents accurately enough our system is done. The cost of such a checking can become prohibitive should all conflict branches be taken into account.

For all these reasons we decided to work with a more reduced model, as it can more easily be analyzed and adapted. However, it must be noted that all the theory developed can be directly applied to asymmetric event structures too.

8.6.2 Limitations of CESAs and CESARs due to timing analysis

In Definitions 7.7 and 7.11 we defined the properties that CESAs and CESARs should satisfy. We explained at that point that some of the properties were added because otherwise there would be problems in the timing analysis. We will now show with an example which kind of problems we do find if such properties are not enforced in the definitions.

For instance in Figure 8.7 (a) we can see a CESA which violates the property that no concurrent event should disable the same event. In principle there seems to be no inherent reason for such restriction as the words of the CESA are still well defined and one can construct the GRC and GRE without problems if this property is ignored. In fact the resulting GRE is shown in subfigure (b). However, if $\mathsf{a} \triangleright \mathsf{f}$ and $\mathsf{b} \triangleright \mathsf{f}$ are present at the same time in the CESA, then the timing zones do not follow the expected behavior.

The timing zones we have defined previously are state-related zones (*i.e.* given a particular state in the GRE one can tell to which zone it belongs). If we allow the two disablings, then the timing zone does no longer depend on the state we are, but on the path followed to reach that point. For instance configuration $\{\mathsf{c}, \mathsf{d}\}$ can be reached either by one of these two traces:

$$\theta_1 = \{\mathsf{x}\} \xrightarrow{\mathsf{x}} \{\mathsf{a}, \mathsf{b}, \mathsf{f}\} \xrightarrow{\mathsf{a}} \{\mathsf{b}, \mathsf{c}\} \xrightarrow{\mathsf{b}} \{\mathsf{c}, \mathsf{d}\}$$

$$\theta_2 = \{\mathsf{x}\} \xrightarrow{\mathsf{x}} \{\mathsf{a}, \mathsf{b}, \mathsf{f}\} \xrightarrow{\mathsf{b}} \{\mathsf{a}, \mathsf{d}\} \xrightarrow{\mathsf{a}} \{\mathsf{c}, \mathsf{d}\}$$

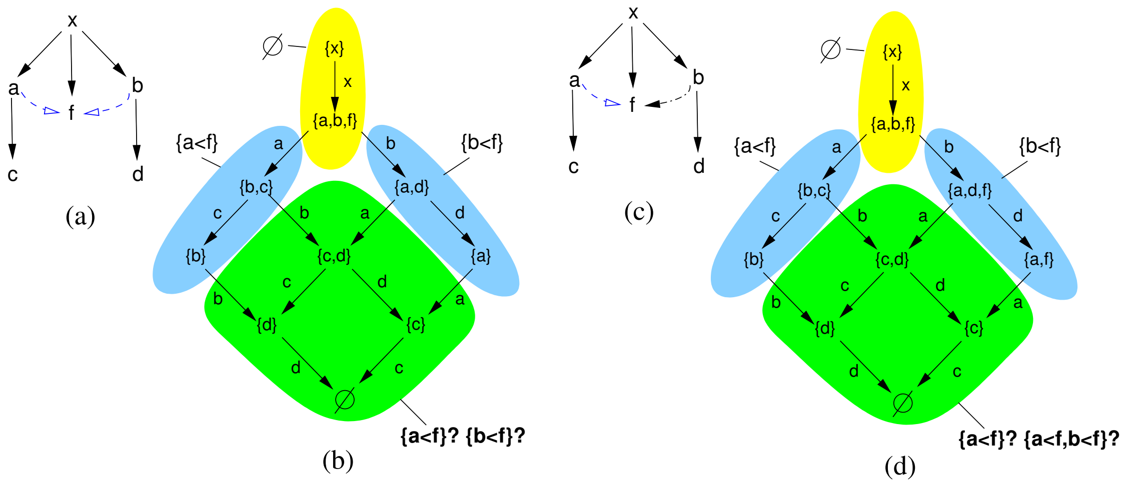


Figure 8.7. Example of badly constructed CESARs that cause problems to the timing analysis: (a) CESA in which two concurrent events disable the same event. (b) Its GRE in which timing zones have been marked. Problem arises because the timing zone reached depends on the relative order of the firing of the events a and b. (c) CESAR in which a restriction affects a disabled event causing the same problem as before. (d) Its corresponding GRE.

However in θ_1, after the firing of a, we are cleary in timing zone $\{\mathsf{a}<\mathsf{f}\}$, but then, when event b fires, we cannot say that event b has fired before event f, because f is already disabled! As a consequence the restriction $\mathsf{b}<\mathsf{f}$ does not apply any more and the timing zone is still $\{\mathsf{a}<\mathsf{f}\}$. Doing the same simulation with θ_2, we obtain that configuration $\{\mathsf{c},\mathsf{d}\}$ is in timing zone $\{\mathsf{b}<\mathsf{f}\}$. Hence, depending on the path taken to reach it, this configuration can be in timing zone $\{\mathsf{a}<\mathsf{f}\}$ or $\{\mathsf{b}<\mathsf{f}\}$.

A similar thing happens in Figure 8.7 (c). The CESAR that appears in the picture is equal to the CESA of subfigure (a) but the disabling $\mathsf{b} \triangleright \mathsf{f}$ has been changed for the restriction $\mathsf{b}<\mathsf{f}$. As before configuration $\{\mathsf{c},\mathsf{d}\}$ can be reached by two traces:

$$\theta_1 = \{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{f}\}\xrightarrow{\mathsf{a}}\{\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{b}}\{\mathsf{c},\mathsf{d}\} \text{ (as in previous example)}$$
$$\theta_3 = \{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{a},\mathsf{b},\mathsf{f}\}\xrightarrow{\mathsf{b}}\{\mathsf{a},\mathsf{d},\mathsf{f}\}\xrightarrow{\mathsf{a}}\{\mathsf{c},\mathsf{d}\}.$$

Note that θ_3 is different from θ_2 because now the firing of b does not disable f. Hence in this trace the timing zone of configuration $\{\mathsf{c},\mathsf{d}\}$ is $\{\mathsf{a}<\mathsf{f},\mathsf{b}<\mathsf{f}\}$, because event a effectively disables f. However we have seen before that if using θ_1 then the timing zone reached was simply $\{\mathsf{a}<\mathsf{f}\}$ and again the timing zone depends on the path instead of the configuration. To avoid such path dependant behavior the restriction that *no additional ordering can be established with an event which is disabled* was added to the definition of a CESAR (Definition 7.11).

8.6.3 Conservative approximations

The delays assigned to the events in the CESAR for timing analysis play a crucial role in the context of our verification approach. Failure traces and/or CESARs may come from a variety of sources: a designer, whose knowledge about the possible sources of failures in a system can be useful to guide the verification; a CESAR derived not from a whole trace that starts from the initial state, but from a portion of trace that ensures a localized failure analysis that involves less events; etc. In these cases, the prehistory of the enabledness of some events involved in the analysis may be unknown to the verification algorithm. As a consequence, given a CESAR $\mathrm{CS} = \langle \Sigma, \prec, \triangleright, < \rangle$, the minimum delay bound for all root events $((^{\circ}\Sigma)_{\prec})$ *i.e.* those for which no causal predecessor exists, for which we do not know their previous enabledness is conservatively set to 0. That is, mimicking an infinitely early enabling time. Note that this is completely consistent with what was done in §5.3.2 when checking the timing consistency of fragments of traces. With this strategy, timing analysis is still exact if the prehistory of enabledness of all events is known or in case the CESAR has only one root event, even if conservative, since the relative firing order of all other events does not depend on the enabling time of their common predecessor.

Chapter 9

A preliminary verification methodology

Summary

In previous sections we have presented several elements which can be combined to build the skeleton of an iterative verification methodology. In this section we describe this verification methodology, although some parts of it have not been fully explained up to this point. We proceed this way because we believe it helps to understand each one of the steps if we see how it fits in the overall process. Moreover in Part III we will deal with some problems that require to understand this basic methodology because decisions taken in some step may have an effect several steps later. Hence the described verification flow may not be seen as a final result, it is rather an intermediate step useful to understand the flow described in Chapter 13. This is why we use the term *preliminary verification methodology* in this section.

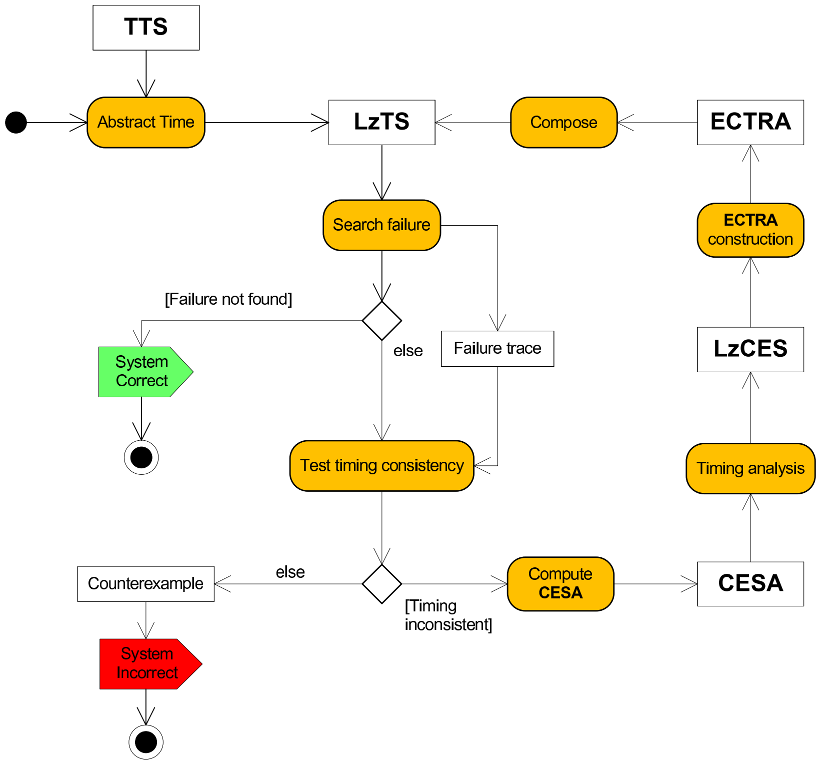


Figure 9.1. Flow of the preliminary verification methodology as a UML 2.0 activity diagram.

9.1 Putting all the parts together

The preliminary verification flow is graphically depicted in Figure 9.1 as a UML 2.0 activity diagram. The verification starts by abstracting the timing information from the TTS that models the system to verify. This gives a TS which is a particular case of LzTS in which the enabling and the firing information of all the events coincide.

Given a safety property, a trace is identified that leads to some state in which it is violated. If the trace is timing consistent (§5.3.3) then the system does not satisfy the required property and the trace provides a counter-example. On the contrary, if the trace is not timing consistent, it is used to refine the untimed state space and remove other timing inconsistent traces. Causality information between the events in the trace is extracted and a CESA is built from it. This part has not been fully discussed in previous section and it will be covered in depth in Chapter 11. Assume by now that in this phase we select the CESA that has the least number of events and relations between events (hence it is the most concurrent possible CESA) among the family of CESAs (§7.2) that contains the failure trace.

Timing analysis on the CESA is performed by using the algorithm presented in §8.4. The extracted temporal information is used to obtain a LzCES (§8.4.2) which, in turn, is used to build an automaton that is capable of removing all the timing inconsistent related traces as well as their enabling compatible traces, *i.e.* an ECTRA. The ECTRA, which can be described as a LzTS, is composed with the original LzTS, thus including the temporal information necessary to prove that some of the traces in the system are not valid. In particular, at least the failure trace found in the initial step is removed. The process is repeated until no violation of the property exists or a timing consistent failure trace is found.

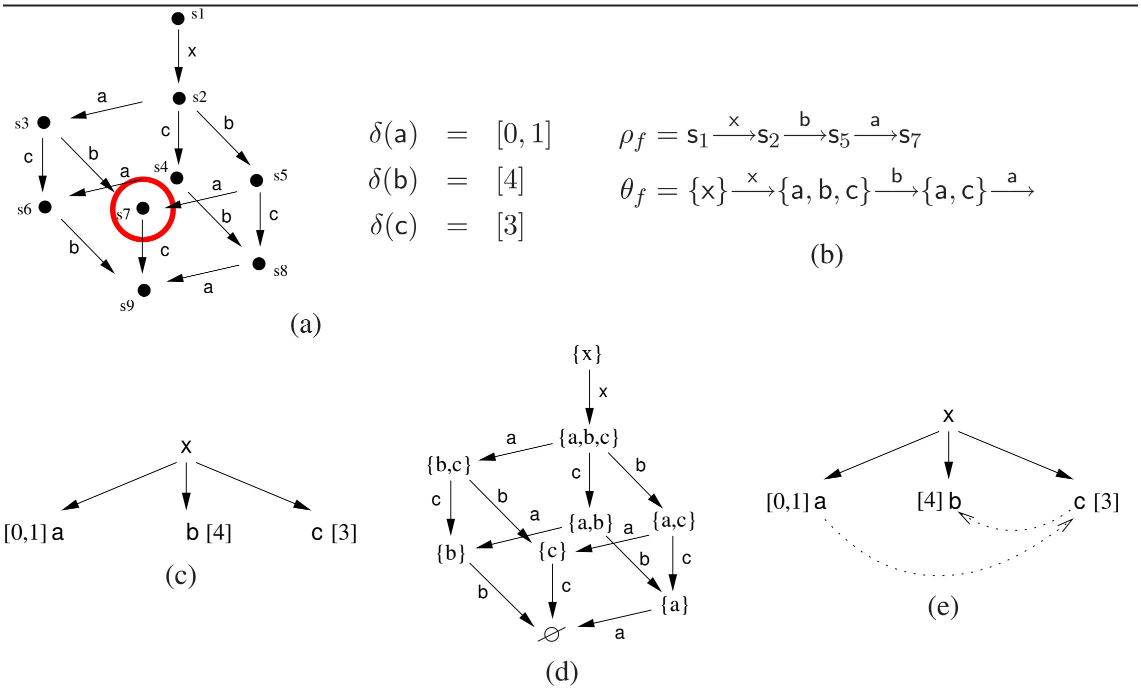


Figure 9.2. Example of application of the preliminary verification methodology – first figure: (a) TTS in which state s_7 is declared a failure state. (b) Run to failure state and its corresponding trace. (c) CESA derived from failure trace. It is the smallest CESA in the family of CESAs that contain θ_f *i.e.* the smallest element in CESA(θ_f). (d) Its corresponding GRE. In this case the GRE coincides completely with the traces present in the system. (e) LzCES obtained from CESA after timing analysis has been conducted on the event structure.

Along the series of refinements each LzCES is reported. At the end of the process, the resulting set of LzCESs constitute a set of sufficient relative timing constraints that prove the correctness of the system. They can also be used as valuable back-annotation information to help the designer improve his/her knowledge of the parts of the system which are critical for its correct operation.

9.2 An example of application

In Figure 9.2 (a) we can see a TTS. Verification flow starts by abstracting delay information (*i.e.* considering the underlying TS of the TTS) and computing its reachable set of states. The system is the same that we presented before, in Figure 6.5, to compare the single trace timing analysis against the multiple trace timing analysis. In none of both previous approaches, CESAR s were used so at least two refinements were needed. We will see that with the proposed methodology a single refinement is needed.

After the reachable set is computed we do check if there is some failure state present. In this case, state s_7 is defined as a failure state by the designer, so a counterexample run/trace reaching it is found. Suppose that the particular run and trace found are the ones shown in subfigure (b). The timing consistent checking of the trace (using $\mathrm{LP}(\theta_f)$ described in Lemma 5.2) shows that it is not timing consistent. Then be build the CESA that includes θ_f in the set of traces it represents.

In this preliminary verification flow we simply select the smallest CESA CS in the family of CESAs that include θ_f *i.e.* CS $\in$ CESA(θ_f), as defined in Definition 7.10. The selected CESA CS is shown in subfigure (c). It can be checked that trace θ_f belongs to the set of traces that CS represents by inspecting its GRE depicted in subfigure (d).

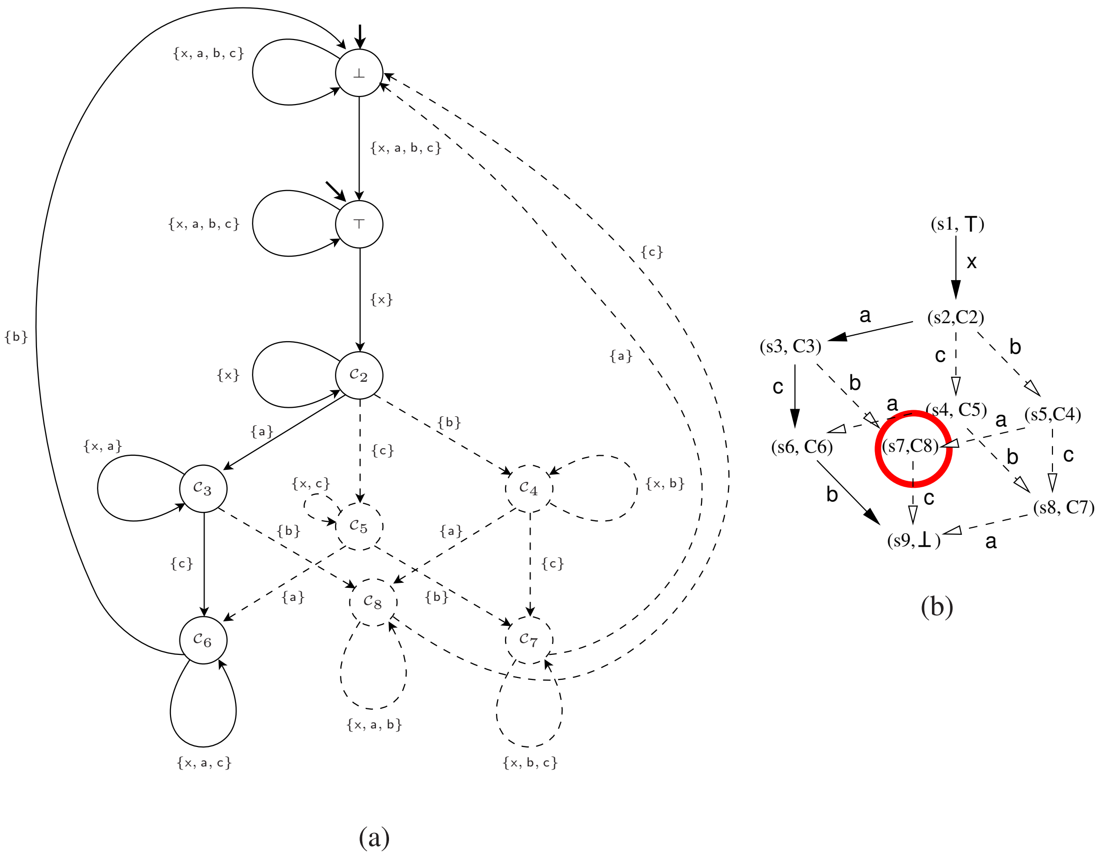


Figure 9.3. Example of application of the preliminary verification methodology – second figure: (a) ECTRA obtained from the LzCES of Figure 9.2 (e). In order to make the graph readable the reenter and break arcs are not drawn and only sets of events (instead of the full transition description) are used as arc labels. (b) LzTS obtained after the composition of the former ECTRA with the original system. Failure state is no longer reachable through a valid trace.

Once the CESA has been selected we consider it a CESAR and timing analysis is conducted on it as explained in Chapter 8 using Algorithm 8.1. In this example there are no restrictions or disablings in the CESAR so only one timing zone is required. From the results of the analysis and the CESAR we build a LzCES in which the detected relative orderings in the time domain are represented as additional precedences. We show a graphical representation of the LzCES in subfigure (d), in which only transitivity irredundant arcs are depicted, as usual. From the LzCES an ECTRA is constructed which is able to “identify” all enabling compatible traces with some of the traces represented by the CESAR and in which timing inconsistent arcs (according to the precedences expressed in the LzCES) are defined as lazy. A simplified version of the ECTRA is shown in Figure 9.3 (a).

Finally, once the ECTRA is available we can compose it with the original LzTS (a TS is a particular type of LzTS) to incorporate the timing analysis back into the original system (see §6.6). The result of the composition is another LzTS (shown in subfigure (b)) that is used as the starting point of the next iteration of the verification flow. In this case the reachability analysis will not find any failure state reachable through a valid trace, hence the system will be declared as correct.

9.3 What we have and what we lack

Part II has presented the theoretical foundations of relative timing verification. This chapter, which concludes the part, shows how to combine these theoretical elements to create a preliminary verification methodology for safety properties in timed systems.

The methodology combines relative timing with conventional methods based on symbolic reachability analysis. Two fundamental facts are at the basis of the approach: the set of traces of a transition system can be covered by a set of event structures, and the use of relative timing allows to represent the timed domain of a system in an efficient way.

Rather than calculating the exact timed state space, the verification approach performs an *off-line*[1] timing analysis on a set of event structures that covers the traces leading to failure states. This timing analysis is efficiently performed by using a tailored algorithm that uses the McMillan and Dill's algorithm [MD92] in each timing zone. The resulting timing constraints are incorporated to the system in the form of relative timing information along a series of iterative refinements of the original untimed state space. Finally, if some of the traces leading to failure situations cannot be proved to be timing inconsistent, then the system is incorrect and the failure trace is a counterexample.

The approach presented here not only verifies the correctness of the system with respect to a set of given safety properties, but also provides as back-annotation a set of timing constraints sufficient to prove correctness. This information is crucial in frameworks in which synthesis and verification are iteratively invoked to design systems that must meet functional and non-functional constraints.

The key features of the verification approach can be summarized by:

- Relative timing allows to avoid the computation of the exact timed state space of the system. Instead, the timed behavior of events is captured by means of partial orders that represent simple facts, such as if an event happens before another.

- The timing analysis is performed locally for a set of failure traces that are covered by an event structure. Therefore, only a subset of the events is involved and the timing analysis can be carried out efficiently.

- The methodology is lazy: timing information is only considered in an *on-demand* basis, as long as it is required to prove the infeasibility in the timed domain of a set of failure traces.

- The verification not only proves or disproves the correctness of the system with respect to a set of safety properties. In case the system is correct the algorithm provides the set of relative timing relations used for the proof, which can be used as valuable back-annotation information. In case the system is incorrect, a counterexample failure trace is provided.

Despite all the benefits, there are many cases in which the methodology might fail. First of all the ECTRAs we build are usually indeterministic automata, and in some cases this has undesired consequences in composition.

Secondly, the results of the analysis can be conservative. The event structures used have to fulfill some specific conditions to guarantee that any timing inconsistent trace found can be removed from the system. The methodology does not check if the CESAs used satisfy them, hence is possible that we are not capable of removing some timing inconsistent traces.

[1] *offline* in this context means that no timing information is considered explicitly while performing the reachability analysis.

Finally, for most systems the methodology never ends and produces refinements until the computer resources are exhausted. The reason is related to the amount of information used to prove that a trace is timing inconsistent. Here we use all the trace, but usually less information is enough to prove timing inconsistency.

All these three questions –indeterminism, conservativeness and convergence– are treated in depth in Part III.

PART III

TIMING VERIFICATION - ADVANCED TOPICS

Summary of Part III

In Part II we introduced the basic theory of verification with relative timing together with the unifying framework of trace semantics. The basic theory extends and corrects some of the previous relative timing approaches [PCKP00, Peñ03] and leads to a preliminary verification methodology (Chapter 9). However, the presented methodology is still limited. In this part we point out such limitations and we provide an extension of the methodology that is able to overcome them.

The first problem we treat is ECTRA indeterminism. The construction procedure usually yields a non deterministic automaton. If this automaton is used in composition, it can be that some traces that are enabling compatible a lazy trace of the ECTRA are not removed. Clearly this violates one of the assumptions of the methodology: all enabling compatible traces with a lazy trace should be removed. To avoid this undesired sideeffect a determinization procedure is provided. However determinization is only theoretical and never actually appears in the system. Thus it would be a wasted effort to use the expensive determinization methods unnecessaily. Consequently we also investigate the possible alternatives to detect when this procedure is really needed.

A second weak point in the preliminary verification methodology is the CESARs it uses. In Chapter 11 we show that all the relevant traces to failures represented by the CESAR have to be present in the state space of the system. Otherwise it can be that timing inconsistent traces found cannot be removed, another violation of the iterative refinement principles. To solve this situation we provide an algorithm that adjusts the CESAR to the real state space.

Another interesting question is to consider whether the methods based on ECTRAs are equally poerful than the exact timed state space generation. The answer is that, contrary to other iterative refinement techniques, they do not always converge (at least if loops are not taken into account). In Chapter 12 we describe several classes of TTSs for which convergence is guaranteed, and we organize them in a hierarchy.

Finally, taking into account all the previous questions, we can modify the preliminary verification methodology to obtain our proposed methodology (Chapter 13), that has been implemented in the tool OOTRANSYT. Some relevant practical considerations on the implementation are introduced in Chapter 14 as well as some optimization techniques. To test the validity of our approach we have conducted several experiments detailed in Chapter 15. The results show that our method outperforms previous approaches based on the same principles and in many cases has a comparable or better performance than state-of-the-art tools using a variety of different approaches.

Chapter 10

INDETERMINACY

Summary

In §6.5 and §8.5 we have seen different procedures to create ECTRAs. However, the resulting ECTRAs are indeterministic except for very simple cases. This can be a problem since the composition of an indeterministic ECTRA does not guarantee that all enabling compatible traces are removed from the system. The only solution to prevent this problem is to determinize them. Determinization of an automaton usually does not yield the smallest possible equivalent deterministic automaton (a desirable property in composition), hence they have to be minimized afterwards.

In this chapter we present tailored determinization and minimization algorithms for ECTRAs. The creation of tailored algorithms is justified because ECTRAs have specific properties that make them not very suited for standard algorithms. In particular:

- They have lots of symbols labelling their arcs.
- They distinguish between lazy and valid traces.
- They are partially specified.

Basically the presented algorithms are modifications of the standard procedures to adapt to the previous characteristics.

However, the presented procedures are computationally expensive and most of the times have a negligeable effect in the composition. The reason is that indeterminism is not very usual in the type of systems analyzed. Consequently, we also present several strategies to decide if determination is required and, if it is, where it is exactly needed. This way useless work can be avoided.

Finally, once indeterminacy has been discussed, we can present the theorem that defines the language accepted by the composed LzTS in terms of traces that are enabling compatible with the lazy traces of the ECTRA.

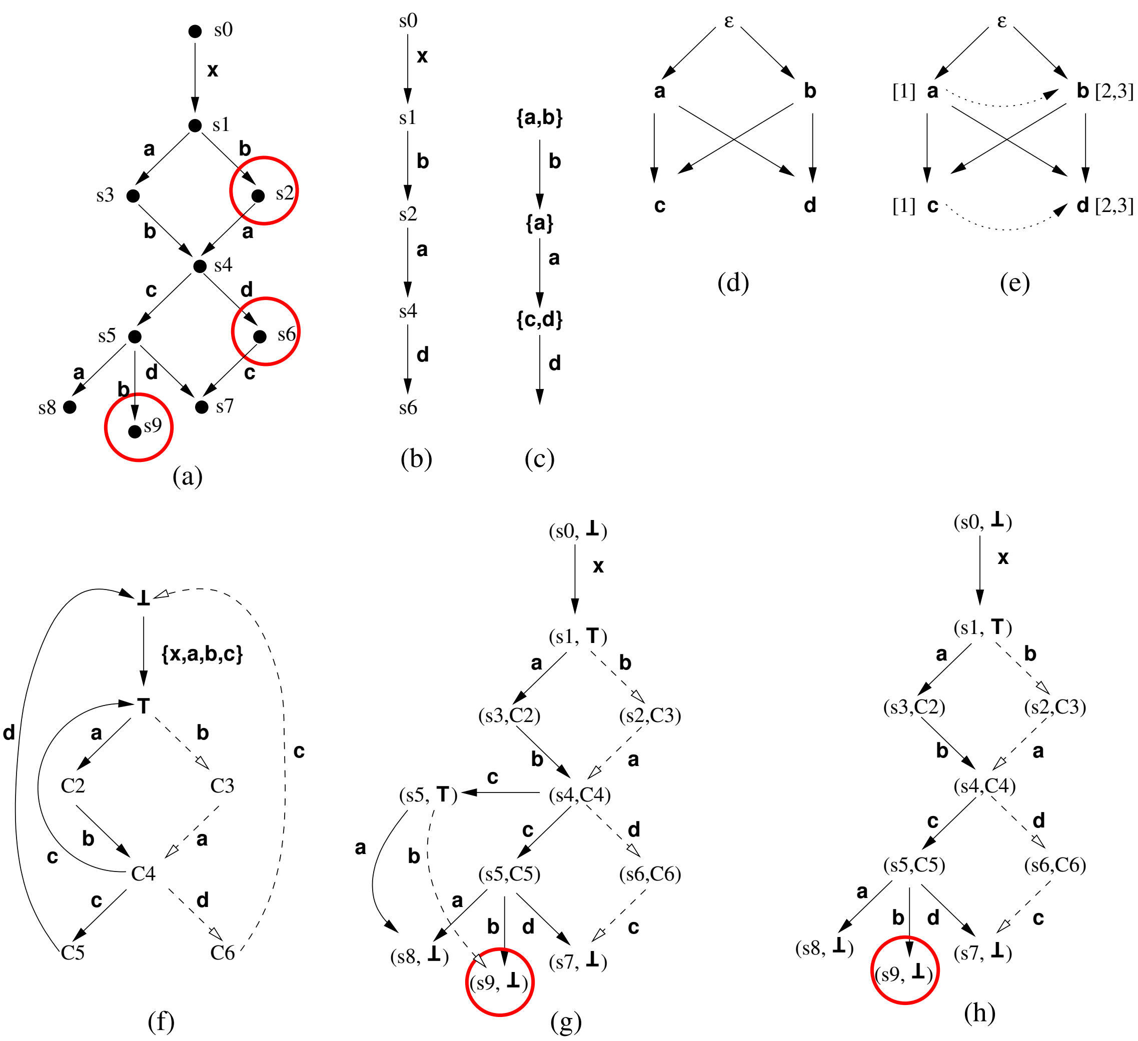


Figure 10.1. Example illustrating indeterminacy problems in ECTRAs: (a) TTS in which states s_2, s_6 and s_9 are defined as failure states. (b) Run to failure. (c) Trace fragment of the run. (d) A possible CESAR containing the trace. (e) LzCES obtained after timing analysis has been performed. (f) ECTRA derived from the LzCES. Most of the reenter, break and keep are omitted for readability with the exception of a reenter arc from $\mathcal{C}_4$ which is relevant for this particular example. Also instead of the full description of the set of transitions that labels each arc, only the corresponding set of events is depicted. (g) Resulting LzTS after composition with the indeterministic ECTRA. (h) Resulting LzTS after composition with the ECTRA in which indeterminacy has been solved using the *resolve conflict* policy. Advance and keep arcs have been given priority over reenter arcs.

10.1 Introduction

The construction of ECTRAs usually yields a non deterministic automaton. For instance in Figure 10.1 (f) we can see a simplified representation of an ECTRA obtained from the LzCES of subfigure (e). Apparently there is no indeterminism, but this is just because of the simplified representation. In fact the ECTRA contains a lot of indeterministic transitions. As an example Table 10.1 lists all conflicting transitions that arise between non-lazy arcs in this ECTRA. Conflicts involving lazy arcs are omitted because lazy arcs are given automatically preference over the other types of arcs. The reason to proceed this way is that we want to prune as many traces as possible.

Arc	Type	Complete label of arc
$\mathcal{C}_2 \xrightarrow{\mathsf{b}} \mathcal{C}_4$	Advance	$\langle \mathsf{b} \wedge \overline{\mathsf{c}} \wedge \overline{\mathsf{d}}, \{\mathsf{b}\}, \mathsf{c} \wedge \mathsf{d} \rangle$
$\mathcal{C}_2 \xrightarrow{\mathsf{b}} \top$	Reenter	$\langle \mathsf{b} \wedge \overline{\mathsf{a}}, \{\mathsf{b}\}, \mathsf{a} \wedge \mathsf{b} \rangle$
Shared transitions between $\mathcal{C}_2 \xrightarrow{\mathsf{b}} \mathcal{C}_4$ and $\mathcal{C}_2 \longrightarrow \top$		
$\langle \mathsf{b} \wedge \overline{\mathsf{a}} \wedge \overline{\mathsf{c}} \wedge \overline{\mathsf{d}}, \{\mathsf{b}\}, \mathsf{a} \wedge \mathsf{b} \wedge \mathsf{c} \wedge \mathsf{d} \rangle$		

Arc	Type	Complete label of arc
$\mathcal{C}_4 \xrightarrow{\mathsf{a,b}} \mathcal{C}_4$	Keep	$\langle \mathsf{c} \wedge \mathsf{d}, \{\mathsf{a}, \mathsf{b}\}, \mathsf{c} \wedge \mathsf{d} \rangle$
$\mathcal{C}_4 \xrightarrow{\mathsf{a,b}} \top$	Reenter	$\langle \mathsf{a} \wedge \overline{\mathsf{b}} \wedge \mathsf{c} \wedge \mathsf{d}, \{\mathsf{a}\}, \mathsf{a} \wedge \mathsf{b} \rangle \cup \langle \overline{\mathsf{a}} \wedge \mathsf{b} \wedge \mathsf{c} \wedge \mathsf{d}, \{\mathsf{b}\}, \mathsf{a} \wedge \mathsf{b} \rangle$
Shared transitions between $\mathcal{C}_4 \xrightarrow{\mathsf{a,b}} \mathcal{C}_4$ and $\mathcal{C}_4 \xrightarrow{\mathsf{a,b}} \top$		
$\langle \mathsf{a} \wedge \overline{\mathsf{b}} \wedge \mathsf{c} \wedge \mathsf{d}, \{\mathsf{a}\}, \mathsf{a} \wedge \mathsf{b} \wedge \mathsf{c} \wedge \mathsf{d} \rangle \cup \langle \overline{\mathsf{a}} \wedge \mathsf{b} \wedge \mathsf{c} \wedge \mathsf{d}, \{\mathsf{b}\}, \mathsf{a} \wedge \mathsf{b} \wedge \mathsf{c} \wedge \mathsf{d} \rangle$		

Arc	Type	Complete label of arc
$\mathcal{C}_4 \xrightarrow{\mathsf{c}} \mathcal{C}_5$	Advance	$\langle \mathsf{c} \wedge \mathsf{d}, \{\mathsf{c}\}, \mathsf{d} \rangle$
$\mathcal{C}_4 \xrightarrow{\mathsf{c}} \top$	Reenter	$\langle \overline{\mathsf{a}} \wedge \overline{\mathsf{b}} \wedge \mathsf{c} \wedge \mathsf{d}, \{\mathsf{c}\}, \mathsf{a} \wedge \mathsf{b} \rangle$
Shared transitions between $\mathcal{C}_4 \xrightarrow{\mathsf{c}} \mathcal{C}_5$ and $\mathcal{C}_4 \xrightarrow{\mathsf{c}} \top$		
$\langle \overline{\mathsf{a}} \wedge \overline{\mathsf{b}} \wedge \mathsf{c} \wedge \mathsf{d}, \{\mathsf{a}\}, \mathsf{a} \wedge \mathsf{b} \wedge \mathsf{d} \rangle$		

Arc	Type	Complete label of arc
$\mathcal{C}_5 \xrightarrow{\mathsf{a,b,c}} \mathcal{C}_5$	Keep	$\langle \mathsf{d}, \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}, \mathsf{d} \rangle$
$\mathcal{C}_5 \xrightarrow{\mathsf{a,b,c}} \top$	Reenter	$\langle \overline{\mathsf{a}} \wedge \overline{\mathsf{b}} \wedge \mathsf{d}, \{\mathsf{c}\}, \mathsf{a} \wedge \mathsf{b} \rangle \cup \langle \mathsf{a} \wedge \overline{\mathsf{b}} \wedge \mathsf{d}, \{\mathsf{a}\}, \mathsf{a} \wedge \mathsf{b} \rangle \cup \langle \overline{\mathsf{a}} \wedge \mathsf{b} \wedge \mathsf{d}, \{\mathsf{b}\}, \mathsf{a} \wedge \mathsf{b} \rangle$
Shared transitions between $\mathcal{C}_5 \xrightarrow{\mathsf{a,b,c}} \mathcal{C}_5$ and $\mathcal{C}_5 \xrightarrow{\mathsf{a,b,c}} \top$		
$\langle \overline{\mathsf{a}} \wedge \overline{\mathsf{b}} \wedge \mathsf{d}, \{\mathsf{c}\}, \mathsf{a} \wedge \mathsf{b} \wedge \mathsf{d} \rangle \cup \langle \mathsf{a} \wedge \overline{\mathsf{b}} \wedge \mathsf{d}, \{\mathsf{a}\}, \mathsf{a} \wedge \mathsf{b} \wedge \mathsf{d} \rangle \cup \langle \overline{\mathsf{a}} \wedge \mathsf{b} \wedge \mathsf{d}, \{\mathsf{b}\}, \mathsf{a} \wedge \mathsf{b} \wedge \mathsf{d} \rangle$		

Table 10.1. List of conflicting transitions of the ECTRA of Figure 10.1 (f).

A quick inspection of the table reveals a rather high number of conflicts. This is the standard scenario for any non-trivial ECTRA and is a situation that can have many undesired consequences. Let us delve into this subject a little bit more.

10.2 Dealing with indeterminacy

To deal with non determinism there are three main possibilities:

- Do not resolve indeterminism *i.e.* to compose the original system directly with the non-deterministic automaton without any further processing.
- To ignore (or give preference to) some kind of arc. For instance we can give priority to *advance* arcs over the *reenter* arcs. Should some priority be established, clearly *advance* arcs will be given the highest priority, otherwise you take the risk of not identifying the original trace which is the only one that should be identified for sure (this is required by any iterative refinement methodology as explained in §4.5). For the similarities that this solution has with the compilers mechanism of resolution of ambiguities, we will call this solution the *resolve conflict* solution.
- To determinize (and simplify) before composing.

In particular for the example of Figure 10.1 we can see the effect of the first two policies in subfigures (g) and (h). In both cases the run leading to failure state s_9 is still valid after composition although the trace is enabling compatible with a timing inconsistent fragment of the initial trace (depicted in subfigure (c)) so it should have been declared as lazy. From the two policies the least interesting is the first one (do nothing and compose with the indeterministic automaton) because it

Arc	Type	Complete label of arc	Shared transitions
$\top \xrightarrow{x,a,b,c} \mathcal{C}_2$	Advance	$\langle x \wedge \overline{a} \wedge \overline{b} \wedge \overline{c}, \{x, a, b, c\}, a \wedge b \wedge c\rangle$	$\langle x \wedge \overline{a} \wedge \overline{b}, \{x\}, x \wedge a \wedge b \wedge c\rangle$
$\top \xrightarrow{x} \top$	Reenter	$\langle x, \{x\}, x\rangle$	
$\mathcal{C}_2 \xrightarrow{x} \mathcal{C}_2$	Keep	$\langle a \wedge b \wedge c, \{x\}, a \wedge b \wedge c\rangle$	$\langle a \wedge b \wedge c, \{x\}, x \wedge a \wedge b \wedge c\rangle$
$\mathcal{C}_2 \xrightarrow{x} \top$	Reenter	$\langle a \wedge b \wedge c, \{x\}, x\rangle$	
$\mathcal{C}_2 \xrightarrow{a} \mathcal{C}_3$	Advance	$\langle a \wedge b \wedge c, \{a\}, b \wedge c\rangle$	$\langle a \wedge b \wedge c, \{a\}, x \wedge b \wedge c\rangle$
$\mathcal{C}_2 \xrightarrow{a} \top$	Reenter	$\langle a \wedge b \wedge c, \{a\}, x\rangle$	
$\mathcal{C}_3 \xrightarrow{x,a} \mathcal{C}_3$	Keep	$\langle b \wedge c, \{x, a\}, b \wedge c\rangle$	$\langle b \wedge c, \{x, a\}, x \wedge b \wedge c\rangle$
$\mathcal{C}_3 \xrightarrow{x,a} \top$	Reenter	$\langle b \wedge c, \{x, a\}, x\rangle$	
$\mathcal{C}_3 \xrightarrow{c} \mathcal{C}_6$	Advance	$\langle b \wedge c, \{c\}, b\rangle$	$\langle b \wedge c, \{c\}, x \wedge b\rangle$
$\mathcal{C}_3 \xrightarrow{c} \top$	Reenter	$\langle b \wedge c, \{c\}, x\rangle$	
$\mathcal{C}_6 \xrightarrow{x,a,c} \mathcal{C}_6$	Keep	$\langle b, \{x, a, c\}, b\rangle$	$\langle b, \{x, a, c\}, x \wedge b\rangle$
$\mathcal{C}_6 \xrightarrow{x,a,c} \top$	Reenter	$\langle b, \{x, a, c\}, x\rangle$	

Table 10.2. List of conflicting transitions of the ECTRA of Figure 9.3.

duplicates unnecessarily some of the reachable states of the system[1]. The *resolve conflict* policy was the one taken in [Peñ03] although determinism issues were never explicitly addressed (see §A.5 for more details). The determinization option will be discussed in §10.4.

We provide an additional example of non determinism to give an idea of how frequent this phenomenon is. Later on we will use this example to illustrate the fact that sometimes indeterminacy is only apparent but not real. Our example is the ECTRA that appears in the example of application of the preliminary verification methodology (see Figure 9.3) that is not deterministic (although, again, it is difficult to notice due to the arcs suppressed to ease the reading). In Table 10.2 we give a list of non-lazy arcs of the ECTRA that share some transitions (remember the labels of the arcs are transitions of the original system) and cause the non determinism.

10.3 Consequences of the *Resolve conflict* policy

What happens if one simply ignores the reenter arcs whenever there is a conflict with a break or advance arc? Consequences cannot be qualified as dramatic. In fact it is hard to find non synthetic examples in which there are transitions in the system that can use a non deterministic arc of the ECTRA. In the worst scenario some of traces that are enabling compatible with some timing inconsistent trace of the CESAR will not become lazy and additional iterations of the verification flow will be needed in order to remove them. At some point the non-removed traces will become the original failure trace of an iteration, hence they will be removed for sure because iterative refinement methods guarantee the removal, at least, of the original trace. For instance in our example of Figure 10.1, failure state s_9 is still reachable after composition with the ECTRA using the resolve conflict policy. This should not happen if all enabling compatible traces with any timing inconsistent trace of LzCES of subfigure (e) were pruned by the composition. However it will only take an additional iteration to find a trace to state $(s_9, \perp)$ and then remove it.

Although it is possible to build a practical tool using the resolve conflict policy it must be noted that this has an important theoretical consequence: one cannot longer state that the compo-

[1] Note that it would be an error to define as lazy the firing of event b in configuration $\mathcal{C}_5$, because this could remove timing consistent traces. For instance should not event a be fireable in state s_5, then the firing of event b in $(s_5, \mathcal{C}_5)$ would have been declared lazy, but now trace leading to state s_9 would be timing consistent.

sition of the ECTRA with the original system will remove all enabling compatible traces. Hence determinization is needed if one wants to hold this claim.

10.4 Determinization procedure

The determinization procedure for ECTRAs is fundamentally the same as subset construction for normal automata [Les95, Sip96] with some particularities. Subsets in which configuration $\perp$ appears with other configurations should be considered as formed by these other configurations *e.g.* $\{\perp, \mathcal{C}_2\} = \{\mathcal{C}_2\}$, $\{\perp, \mathcal{C}_2, \mathcal{C}_5\} = \{\mathcal{C}_2, \mathcal{C}_5\}$. This is because being in some recognition state has preference over being in the *nothing-recognised* state (anyway from each configuration already exist reenter arcs so one can alternatively imagine that original configurations also contain $\perp$ configuration). Another important thing is how lazyness of arcs must be treated. The rule is simple: if in any of the configurations of the subset an outgoing event is lazy, then outgoing arcs from the subset related to the same event will also be lazy. The reason for this is that if an event is lazy in some of the configurations, then there is a way in which the trace containing such an event can be enabling compatible with a timing inconsistent fragment of some of the traces of the CESAR, hence we can safely remove it from consideration.

We give an informal description of the algorithm (along with an example to illustrate it):

1. For all basic configurations (*i.e.* the original configurations of the ECTRA) check if there is indeterminacy. This is done by intersecting reenter arcs with advance and keep (due to the construction there cannot be any common transition between advance and keep transitions). In example of Figure 10.1 these configurations are $\mathcal{C}_2, \mathcal{C}_4$ and $\mathcal{C}_5$. Compute the compound configuration for these shared transitions as the union of all reachable configurations through the transitions. For instance in our example there are shared transitions between the advance arc of $\mathcal{C}_2 \xrightarrow{\mathsf{b}} \mathcal{C}_4$ and the reenter arc $\mathcal{C}_2 \xrightarrow{\mathsf{b}} \top$. Shared transitions are $\langle \mathrm{ER}(\mathsf{b}) \cap \overline{\mathrm{ER}(\mathsf{a})} \cap \overline{\mathrm{ER}(\mathsf{c})} \cap \overline{\mathrm{ER}(\mathsf{d})}, \{\mathsf{b}\}, \mathrm{ER}(\mathsf{a}) \cap \mathrm{ER}(\mathsf{b}) \cap \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{d})\rangle$. Their compound configuration destination will be $\{\top, \mathcal{C}_4\}$.

2. Repeat until all compound configurations are processed:

   - For all basic configurations present in the compound configuration, take one of their outgoing arcs. Compute the intersection of all the labels. If result is the empty set, continue. Otherwise, build the destination compound configuration the same way as before (this might yield new configurations that must be processed afterwards). If the compound configuration includes other configurations besides $\perp$, remove $\perp$ from it.
   - Repeat this process for all possible combinations of outgoing arcs. In our example we would process configuration $\{\top, \mathcal{C}_4\}$ as follows: configuration $\top$ has three outgoing arcs *e.g.* $\top \xrightarrow{\mathsf{c,d}} \top, \top \xrightarrow{\mathsf{a}} \mathcal{C}_2$ and $\top \longrightarrow \perp$, while configuration $\mathcal{C}_4$ has four outgoing arcs *e.g.* $\mathcal{C}_4 \xrightarrow{\mathsf{ab}} \mathcal{C}_4, \mathcal{C}_4 \xrightarrow{\mathsf{c}} \mathcal{C}_5, \mathcal{C}_4 \longrightarrow \top$ and $\mathcal{C}_4 \longrightarrow \perp$. This give a total of twelve checkings, one for each possible combination of arcs. For instance $\top \xrightarrow{\mathsf{c,d}} \top \cap \mathcal{C}_4 \xrightarrow{\mathsf{a,b}} \mathcal{C}_4 = \emptyset$, but $\top \xrightarrow{\mathsf{c,d}} \top \cap \mathcal{C}_4 \xrightarrow{\mathsf{c}} \mathcal{C}_5 = \langle \mathrm{ER}(\mathsf{a}) \cap \mathrm{ER}(\mathsf{b}), \{\mathsf{c}, \mathsf{d}\}, \mathrm{ER}(\mathsf{a}) \cap \mathrm{ER}(\mathsf{b})\rangle \cap \langle \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{d}), \{\mathsf{c}\}, \mathrm{ER}(\mathsf{d})\rangle = \langle \mathrm{ER}(\mathsf{a}) \cap \mathrm{ER}(\mathsf{b}) \cap \mathrm{ER}(\mathsf{c}) \cap \mathrm{ER}(\mathsf{d}), \{\mathsf{c}\}, \mathrm{ER}(\mathsf{a}) \cap \mathrm{ER}(\mathsf{b}) \cap \mathrm{ER}(\mathsf{d})\rangle$. These transitions go to configuration $\{\top, \mathcal{C}_5\}$ which will be added to the list of configurations to process.

3. Mark as lazy all arcs in which the corresponding event is lazy in any of the configurations that form the compound configuration. In our example, all outgoing arcs from configuration $(\top, \mathcal{C}_4)$ which involve events b or d are declared as lazy, because the firing of b in configuration $\top$ and the firing of d in configuration $\mathcal{C}_4$ were already defined as lazy.

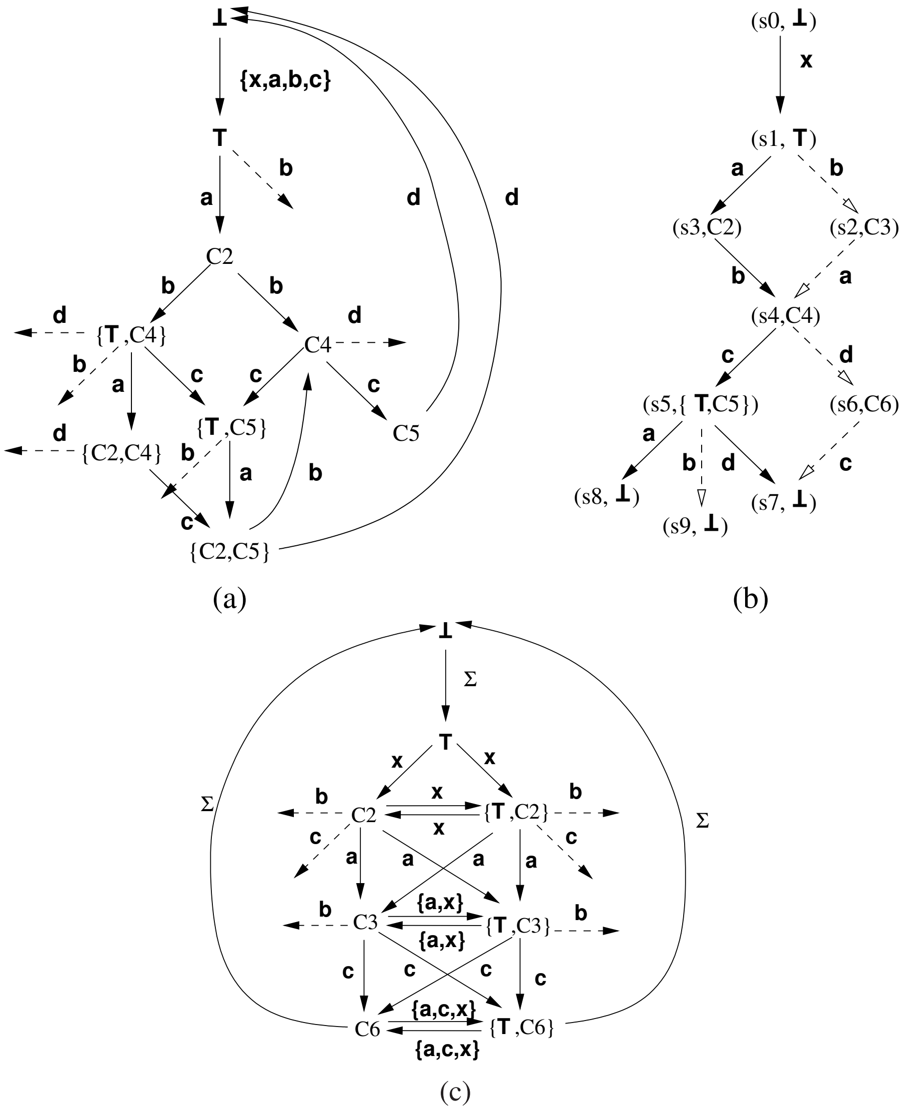


Figure 10.2. Examples of determinized ECTRAs: (a) Determinized ECTRA of Figure 10.1 (f). Due to its size and the number of arcs, we do use classical LzTS representation (thus not showing the behavior of lazy arcs). Most of the reenter, break and keep arcs are omitted for readability. Also only events are shown as label arcs instead of the full set of transitions. This can make the impression that the ECTRA is still non-deterministic, but this is because of the simplified representation. For instance complete labels for arcs $\mathcal{C}_2 \xrightarrow{\mathsf{b}} \{\top, \mathcal{C}_4\}$ and $\mathcal{C}_2 \xrightarrow{\mathsf{b}} \mathcal{C}_4$ are $\langle \overline{\mathsf{a}} \wedge \mathsf{b} \wedge \overline{\mathsf{c}} \wedge \overline{\mathsf{d}}, \{\mathsf{b}\}, \mathsf{a} \wedge \mathsf{b} \wedge \mathsf{c} \wedge \mathsf{d}\rangle$ and $\langle \mathsf{a} \wedge \mathsf{b} \wedge \overline{\mathsf{c}} \wedge \overline{\mathsf{d}}, \{\mathsf{b}\}, \mathsf{c} \wedge \mathsf{d}\rangle \cup \langle \mathsf{b} \wedge \overline{\mathsf{c}} \wedge \overline{\mathsf{d}}, \{\mathsf{b}\}, (\overline{\mathsf{a}} \vee \overline{\mathsf{b}}) \wedge \mathsf{c} \wedge \mathsf{d}\rangle$ respectively, whose intersection is the empty set, so there is no indeterminacy in configuration $\top$. (b) Resulting LzTS after composition with the determinized ECTRA. (c) Simplified depiction of determinized ECTRA of Figure 9.3 using classical LzTS representation. Again most of the reenter, break and keep arcs are omitted for readability.

Steps 1 and 2 of the algorithm are simply an adaptation of regular subset construction for automata. Justification for step 3 of this algorithm is as follows. By Theorems 6.1 and 6.2 we know that if a suffix of a trace θ is fully enabling compatible with some trace in $\mathrm{traces}(\mathcal{C})$ then the recognition state of the ECTRA will include $\mathcal{C}$. Consequently a compound configuration in the determinized ECTRA will only be reached by a trace such that its suffix is fully enabling compatible with some trace in $\mathrm{traces}(\mathcal{C})$ for all configuration $\mathcal{C}$ in the compound configuration. Similarly if the

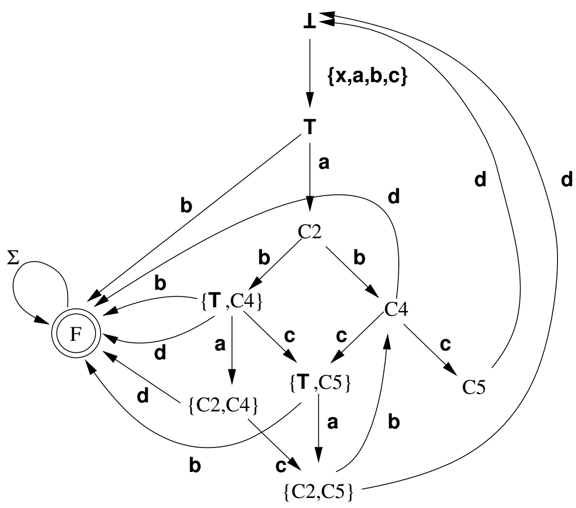


Figure 10.3. An alternative view of the determinized ECTRA of Figure 10.2 (a) that allows an easier usage of minimization algorithms.

ECTRA state is compound configuration C before parsing the last event e_n of θ, we have that $\forall \mathcal{C} \in C, \exists \theta' \in \text{traces}(\mathcal{C}, \mathsf{e}_n) \wedge \theta \geq_{\mathsf{ec}} \theta'$.

An arc $\mathcal{C} \xrightarrow{\langle \mathsf{S}_s, \{\mathsf{e}_n\}, \mathsf{S}_d \rangle}$ of an ECTRA is declared lazy if $\forall \theta' \in \text{traces}(\mathcal{C}, \mathsf{e}_n)$, θ' is timing inconsistent (this is implied by Theorem 8.3). We know that θ is enabling compatible with at least one trace of each set of traces, and that one these set of traces is composed only by timing inconsistent traces (because the arc was declared lazy in some of the basic configurations), thus θ will be timing inconsistent also by Theorem 5.4. Consequently, the firing of event e_n in compound configuration C can be declared lazy.

In Figure 10.2 (a) we can see a simplified drawing of the deterministic version of the ECTRA of Figure 10.1 (f). Using it the composition with the original LzTS yields the expected result (shown in subfigure (b)). However, note that event d is not declared lazy in state $(\mathsf{s}_5, \{\top, C5\})$, although from delays it is clear the event a should fire before it. The reason is that timing analysis of CESAR did not take into account this relationship between both events *i.e.* in this state both become enabled at the same time, while in CESAR event a is a causal predecessor of d. As a result, no timing information is available for the new context.

We give an additional example in Figure 10.2 (c) that contains the deterministic version of the ECTRA of Figure 9.3.

10.5 Minimization

One could apply to the determinized ECTRA a standard minimization algorithm to reduce its size [Hop71, HU79, ASU86] (see [Wat93] for an overview) with the only difference that the initial partitioning (using Hopcroft terminology) would not be between final and non-final states, instead, as our relevant information is related to the laziness of transitions, we would initially partition in terms of configurations that have the same outgoing lazy events or not. Conversely, one could add a final sink configuration to whom all lazy arcs will point as in Figure 10.3, in which the new configuration is marked as F. With this procedure an ECTRA can be viewed as an automaton accepting timing inconsistent traces.

Given an ECTRA EC derived from a LzTS $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR}\rangle$, EC is a partially specified automaton because from each configuration not all possible transitions in set $S \times \Sigma \times S$ are considered, only a subset of them. In particular in configuration $\mathcal{C}_i$ only transitions in set $\mathrm{SE}(\mathcal{E}(\mathcal{C}_i)) \times \Sigma \times S$ are defined. Another particularity is that the alphabet of the ECTRA is huge because labels are sets of transitions in A. This causes many problems to minimization algorithms because most of them rely on an exhaustive search for each pair of state and symbol of the alphabet.

For these reasons it is easier to use a minimization algorithm which computes the distinguishability relation between configurations (this solves the problem of being a partially specified automaton) that does not rely on an exhaustive search for each pair of state and symbol of the alphabet (which solves the problem of having a huge alphabet). Later on, the main ideas is that all configurations that are not distinguishable between them will form an equivalence class and will be joined into a single configuration in the minimized automaton. The procedure used is shown as Algorithm 10.1.

Algorithm 10.1: minimizeECTRA

input : $\mathrm{EC} = \langle C, \Sigma, T, \perp, \mathrm{FR}\rangle$ an ECTRA in which lazy arcs have been changed for arcs pointing to final configuration F.

output : M a map of unordered pairs of configurations in C representing their distinguishability relation.

forall $\mathcal{C}_i, \mathcal{C}_j \in C$ **do**
    if $\mathcal{C}_i = F$ *xor* $\mathcal{C}_j = F$ **then**
        $M(\{\mathcal{C}_i, \mathcal{C}_j\}) \leftarrow 1$;
    else
        $M(\{\mathcal{C}_i, \mathcal{C}_j\}) \leftarrow 0$;

repeat
    forall $\mathcal{C}_i, \mathcal{C}_j \in C$ **do**
        if $M(\{\mathcal{C}_i, \mathcal{C}_j\}) = 0$ **then**
            forall $\mathcal{C}_i \xrightarrow{A} \mathcal{C}_{i'}, \mathcal{C}_j \xrightarrow{B} \mathcal{C}_{j'} \in T$ **do**
                if $A \cap B \neq \emptyset$ *and* $M(\mathcal{C}_{i'}, \mathcal{C}_{j'}) = 1$ **then** $M(\{\mathcal{C}_i, \mathcal{C}_j\}) \leftarrow 1$;

until *no change in* M;
return M;

EXAMPLE 10.1 *In Figure 10.4 we can see the minimized versions of the determinized ECTRAs of Figures 10.2 (a) and (c). For the example of Figure 10.2 (a), equivalence classes are (we do only list equivalence classes with more than one element):* $\{\perp, \mathcal{C}_5\}$*,* $\{\top, \{\top, \mathcal{C}_5\}\}$ *and* $\{\mathcal{C}_2, \{\mathcal{C}_2, \mathcal{C}_5\}\}$*. For the example of subfigure (c), equivalence classes are:* $\{\top, \mathcal{C}_6, \{\top, \mathcal{C}_6\}\}$*,* $\{\mathcal{C}_2, \{\top, \mathcal{C}_2\}\}$ *and* $\{\mathcal{C}_3, \{\top, \mathcal{C}_3\}\}$*. Note that, in this last example, there are alternative valid ways in which the equivalence classes can be arranged. In particular the distinguishability relation tells us that configuration* $\top$ *and* $\perp$ *are distinguishable. However, configurations* $\mathcal{C}_6$ *and* $\{\top, \mathcal{C}_6\}\}$ *are equivalent with* $\top$ *and* $\perp$*, hence they could be merged with any of them. In Figure 10.4 they were merged with* $\top$ *so the equivalent class is* $\{\top, \mathcal{C}_6, \{\top, \mathcal{C}_6\}\}$*.* ■ 10.1

The lack of transitivity of the equivalence relation that appears in previous example is related to the fact that the automaton is partially specified. Configurations $\top$ and $\perp$ are distinguishable because some transition defined for both configurations goes to different distinguishable configurations, while for all transitions defined in configurations $\mathcal{C}_6$ and $\{\top, \mathcal{C}_6\}$ the behavior of $\top$ and $\perp$ is equivalent. To compute the equivalence classes one can translate the problem into a graph problem: represent each configuration of the ECTRA as a vertex of a graph and assume two vertices are connected if they are not distinguishable. Then the problem of finding a set of equivalence classes is the same as finding a set of disjoint cliques (*i.e.* find cliques that do not share vertex) in the graph that do cover all vertex. To obtain the minimal ECTRA the set of equivalence classes must be the smallest one. The equivalent graph problem is known as the *minimum clique partition* problem which is an NP-hard problem [CK98][2].

Worst case execution for algorithms solving NP problems is exponential. Thus, the cost of the minimization algorithm can be quite high because, after determinization, the number of configurations can be exponentially higher than in the original ECTRA. As a last resort one could use a less efficient algorithm that does not give the minimum ECTRA, but with better running times.

Although we have seen how to perform the determinization and minimization of an ECTRA, the cost of using these procedures is not interesting: there are lots of non-deterministic arcs, so determinization will usually yield big ECTRAs and then an additional effort must be done afterwards to try to reduce its size. In many cases resulting in a similar ECTRA than the one that the *resolve conflict* strategy would have derived! An example of this is the ECTRA of Figure 10.4. In many more cases the ECTRA will preserve some of the new configurations due to the determinization, but they will be very seldom used in practice when the ECTRA is composed with the original system.

Due to all these reasons seems reasonable to do some previous checking to ensure that the determinization is really needed.

10.6 Avoiding useless determinization

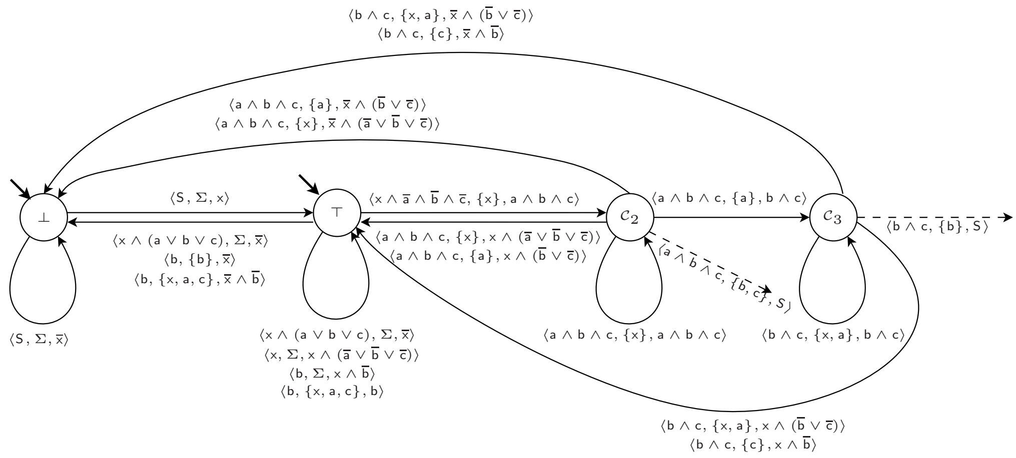
Given an ECTRA derived from a LzTS $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR} \rangle$ there are several ways in which the necessity of determinization can be checked (or at least reduce it):

- Restrict ECTRA arcs using the information of the transitions T in the original system A.
- Further reduce the allowed transitions in T by using the set of reachable states of the LzTS (this might have a much greater impact than the previous alternative in the context where the LzTS is stored symbolically).
- Do a “projection” of which transitions will be taken in each configuration.
- Compose and then check in the composed LzTS.

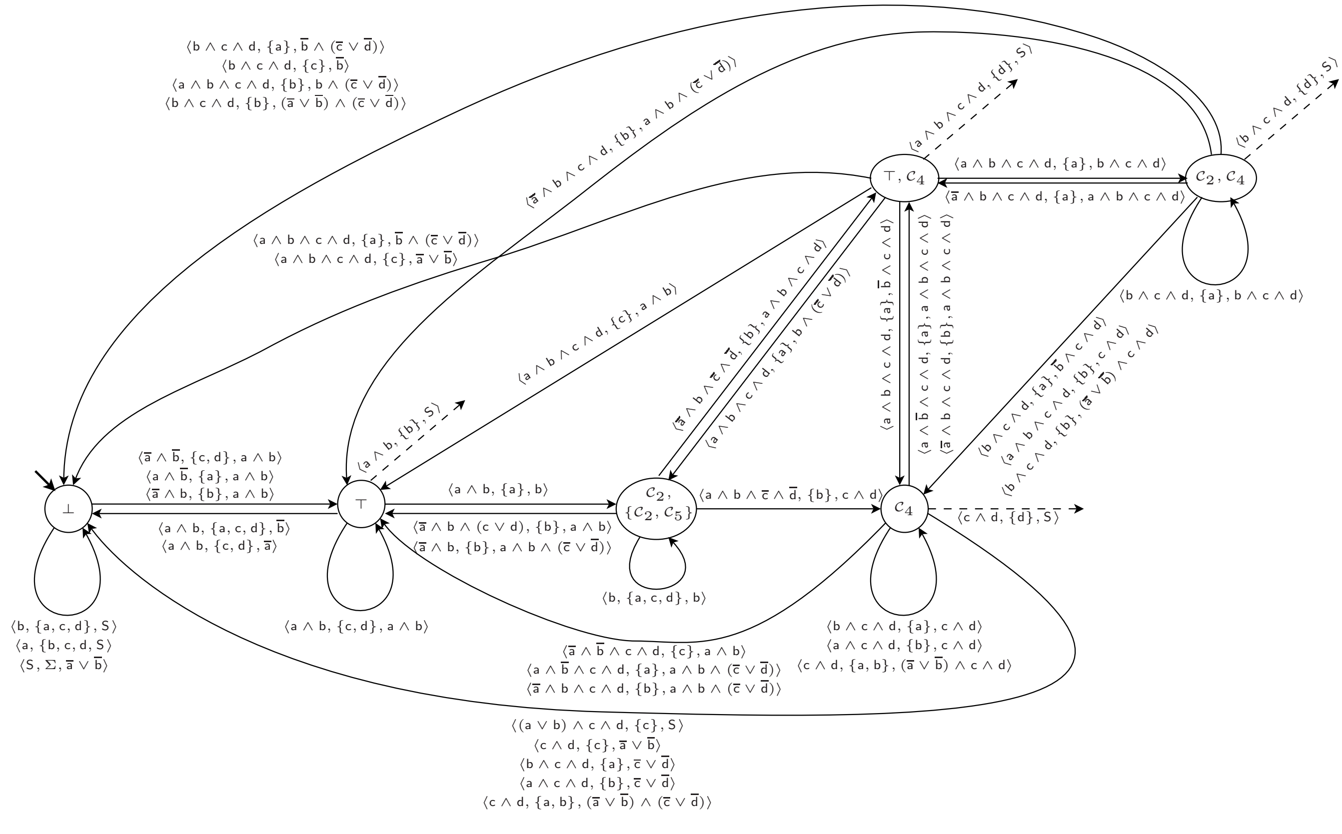
We ordered options in terms of its capacity to reveal inexistent indeterminacy (from less capable to the most capable). As usual the effort to apply them will be proportional to their capacity. However last two options, besides being the most powerful procedures, have additional advantages when the composition is performed symbolically (see §14.1). In our work we will use the *projection* method in our verification flow because is the one with the best balance between efficiency and the computational effort required.

We will illustrate the different efficiency of each approach by means of examples in the following sections.

[2] In 1996 a controversial paper [Plo96] was published in a journal (which no longer exists) by ukranian engineer doctor Anatoly Plotnikov in which a polynomial time algorithm for this problem was given (with asymptotic cost $O(n^6)$, where n is the number of vertex in the graph). As the problem is known to be NP-hard, this would imply that NP=P! The paper contained some errors which were meant to be corrected in a second version of the paper [Plo00] which was released (but not published in any journal or conference proceedings) in year 2000. Commonly held opinion by the scientific community is that this latter paper still contains flaws.



(a)



(b)

Figure 10.4. Examples of minimized ECTRAs: (a) Minimized ECTRA obtained from deterministic ECTRA of Figure 10.2 (c). In the picture configuration $\mathcal{C}_2$ corresponds to equivalence class $\{\mathcal{C}_2, \{\top, \mathcal{C}_2\}\}$, configuration $\mathcal{C}_3$ to class $\{\mathcal{C}_3, \{\top, \mathcal{C}_3\}\}$ and $\top$ to $\{\top, \mathcal{C}_6, \{\top, \mathcal{C}_6\}\}$. (b) Minimized ECTRA obtained from deterministic ECTRA of Figure 10.2 (a). All arcs are depicted except the arc going from configuration of equivalence class $\{\mathcal{C}_2, \{\mathcal{C}_2, \mathcal{C}_5\}\}$ to $\bot$.

10.6.1 Using the information in the transition relation to avoid determinization

Assume for a while that the LzTS A of Figure 10.5 (a) (this is the LzTS that corresponds to the TTS of Figure 9.2 (a) in which timing information has been completely abstracted) is represented

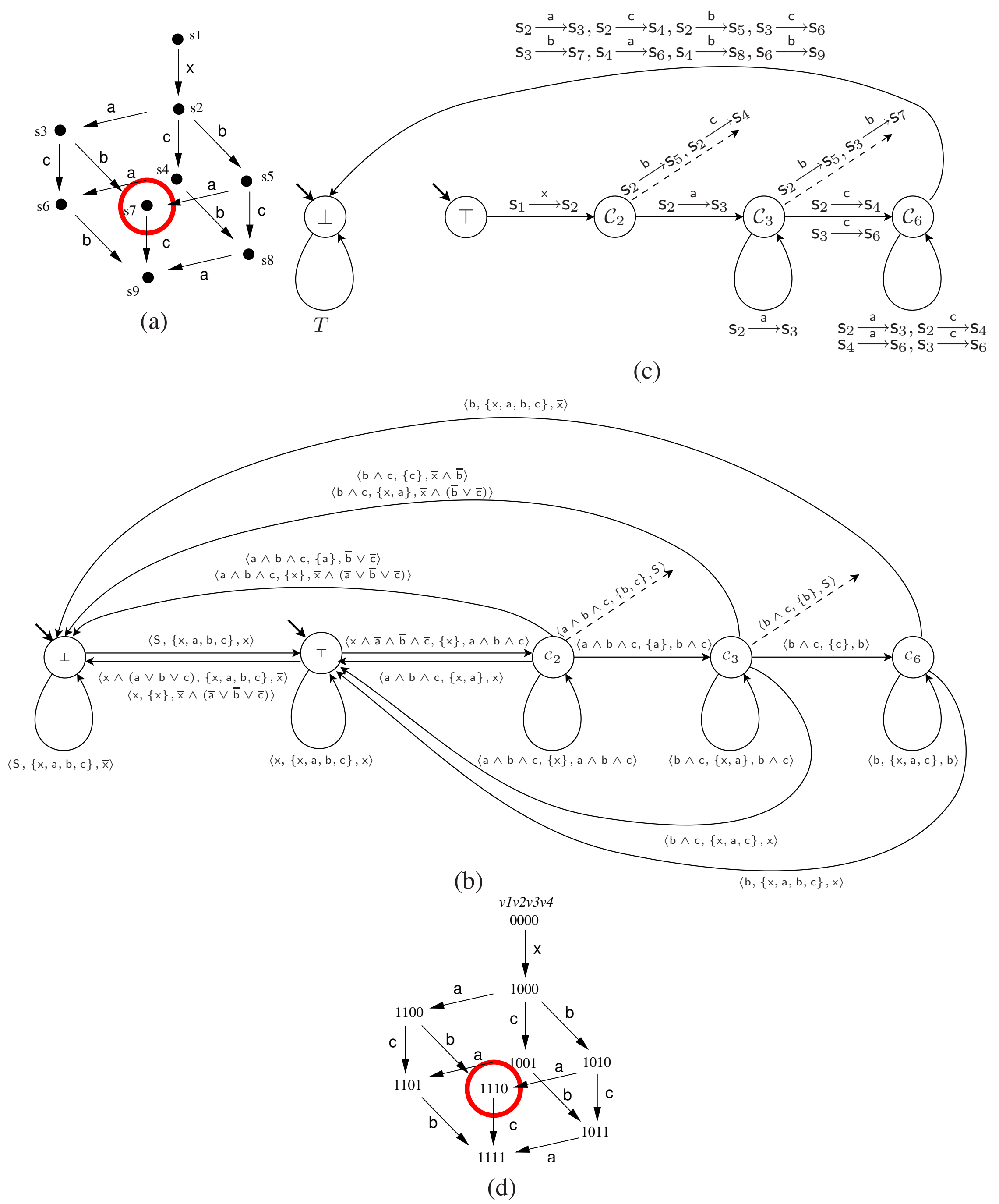


Figure 10.5. Example of the use of the transition relation to avoid unnecessary determinization: (a) LzTS that we want to compose (b) ECTRA of Figure 9.3 (a) fully represented as a classical LzTS (*i.e.* behavior after a lazy arc is not shown) using simplified notation for sets of transitions. (c) The same ECTRA once intersected with transition relation T of the LzTS of subfigure (a), assuming T is stored explicitly. It no longer exhibits indeterminacy. (d) Reachable state space of a LzTS encoded symbolically that is isomorphic with the state space of the LzTS of subfigure (a).

using an explicit representation *i.e.* not using symbolic methods. In this case the transition relation of A that we will denote as T is formed by the set:

$$T = \{s_1 \xrightarrow{x} s_2, s_2 \xrightarrow{a} s_3, s_2 \xrightarrow{b} s_5, s_2 \xrightarrow{c} s_4, s_3 \xrightarrow{c} s_6, s_3 \xrightarrow{b} s_7, s_4 \xrightarrow{a} s_6,$$
$$s_4 \xrightarrow{b} s_8, s_5 \xrightarrow{a} s_7, s_5 \xrightarrow{c} s_8, s_6 \xrightarrow{b} s_9, s_7 \xrightarrow{c} s_9, s_8 \xrightarrow{a} s_9\}$$

Given an indeterministic ECTRA like the one in Figure 9.3 (a) instead of determinizing the ECTRA to obtain the ECTRA of Figure 10.2 (c), which is an ECTRA that can be used to be composed with any LzTS in which the relative timing of the events x, a, b and c is the same as in A, we could previously check if the LzTS contains a transition that appears in an indeterministic arc of the ECTRA. To do so we will intersect T with all the labels appearing in the arcs of the ECTRA. For instance label of arc $\perp \longrightarrow \top$ is $\langle S, \{x, a, b, c\}, x\rangle$ which represents the set of transitions $\{s_1, \ldots, s_9\} \xrightarrow{x,a,b,c} \{s_1\}$ because $\mathrm{ER}(x) = \{s_1\}$. As there is no such arc in T this arc of the ECTRA will never be used in the composition, thus can be safely discarded. In fact all reenter arcs of the ECTRA (*i.e.* all arcs going to the $\top$ configuration) require that destination state of the transition belongs to $\mathrm{ER}(x)$ but it is easy to check that there is no such transition in T. As a result all reenter arcs can be removed. The resulting ECTRA after the intersection of the labels with T is shown in subfigure (c). It contains no indeterminacy so the expensive determinization and minimization steps can be skipped.

However, when the information of the transition relation T is not explicitly stored but symbolically represented, things do not behave so well. For instance, imagine that the LzTS is stored symbolically using Boolean algebra as explained in §2.5 and that its description is as follows[3]: $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$, $\mathrm{TR}(x) = \overline{v_1}v_1'(v_2 = v_2')(v_3 = v_3')(v_4 = v_4')$, $TR(a) = v_1\overline{v_2}v_1'v_2'(v_3 = v_3')(v_4 = v_4')$, $\mathrm{TR}(b) = (v_1\overline{v_3}v_1'v_3' + \overline{v_1}v_3\overline{v_1'}\,\overline{v_3'})(v_2 = v_2')(v_4 = v_4')$, $\mathrm{TR}(c) = (v_1\overline{v_4}v_1'v_4' + \overline{v_1}v_4\overline{v_1'}\,\overline{v_4'})(v_2 = v_2')(v_3 = v_3')$. With this transition relation definition we have that $\mathrm{EF}(x) = \overline{v_1}$, $\mathrm{EF}(a) = v_1\overline{v_2}$, $\mathrm{EF}(b) = v_1\overline{v_3} + \overline{v_1}v_3$ and $\mathrm{EF}(c) = v_1\overline{v_4} + \overline{v_1}v_4$. The reachable state space of this system is isomorphic to the one shown in subfigure (a) if the initial state is encoded as $\overline{v_1}\,\overline{v_2}\,\overline{v_3}\,\overline{v_4}$. Now, when we repeat the same process as before, that is, we intersect the indeterministic ECTRA with the transition relation T, we do obtain very different results. For instance ECTRA arc $\mathcal{C}_3 \longrightarrow \top$ with label $\langle b \wedge c, \{x, a, c\}, x\rangle$ when intersected with T gives:

$$\begin{aligned}
&\mathrm{EF}(b) \cdot \mathrm{EF}(c) \cdot \big(\mathrm{TR}(x) + \mathrm{TR}(a) + \mathrm{TR}(c)\big) \cdot \mathrm{EF}'(x) = \\
&= (v_1\overline{v_3} + \overline{v_1}v_3) \cdot (v_1\overline{v_4} + \overline{v_1}v_4) \cdot \big(\overline{v_1}v_1'(v_2 = v_2')(v_3 = v_3')(v_4 = v_4') + \\
&\vee v_1\overline{v_2}v_1'v_2'(v_3 = v_3')(v_4 = v_4') + (v_1\overline{v_4}v_1'v_4' + \overline{v_1}v_4\overline{v_1'}\,\overline{v_4'})(v_2 = v_2')(v_3 = v_3')\big) \cdot v_1' = \\
&= \overline{v_1}v_3v_4\overline{v_1'}v_3'\overline{v_4'}(v_2 = v_2')
\end{aligned}$$

and for ECTRA arc $\mathcal{C}_3 \longrightarrow \mathcal{C}_4$ with label $\langle b \wedge c, \{c\}, b\rangle$ we obtain:

$$\begin{aligned}
&\mathrm{EF}(b) \cdot \mathrm{EF}(c) \cdot \big(\mathrm{TR}(x) + \mathrm{TR}(a) + \mathrm{TR}(c)\big) \cdot \mathrm{EF}'(x) = \\
&= (v_1\overline{v_3} + \overline{v_1}v_3) \cdot (v_1\overline{v_4} + \overline{v_1}v_4) \cdot (v_1\overline{v_4}v_1'v_4' + \overline{v_1}v_4\overline{v_1'}\,\overline{v_4'})(v_2 = v_2')(v_3 = v_3') \cdot v_1' = \\
&= \overline{v_1}v_3v_4\overline{v_1'}v_3'\overline{v_4'}(v_2 = v_2')
\end{aligned}$$

Consequently, both ECTRA arcs share all their labels yielding an non-deterministic automaton (although it is clear from previous discussion that in fact there is no real indeterminacy). Although

[3]In order to avoid increasing unnecessarily the complexity of the example we avoid using partial TRs to represent the transition relations of the events.

it might be the case that intersecting with an implicitly defined transition relation can remove some arcs from consideration, it is very likely that some indeterminacy persists because in implicit definitions usually transition relations are defined with an enormous quantity of transitions from which only a small fraction is really used in the reachable part of the system. A similar problem can appear using an explicit representation for T if arcs that do not originate in a reachable state of the system are present. For this reason the next logical step would be to use the information of the reachable set of states to further reduce the probabilities of preserving indeterministic transitions.

10.6.2 Combining the transition relation and the set of reachable states to avoid determinization

To obtain the same results as when T does only contain transitions really appearing in the system (given a particular initial state), either in the context of an overdefined explicit transition relation or a implicitly defined one, we can use the reachable set of states. It is clear that given a LzTS $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR} \rangle$ the set $T \cap \mathrm{Reach}(\mathsf{s}_0, T)$ contains only the transitions that really appear in the system. Using this approach the ECTRA of Figure 10.5 (b) applied to the LzTS of subfigure (d) can be shown that contains no real indeterminacy although uses a symbolic encoding. For instance, transition in set $\overline{v_1} v_3 v_4 \overline{v_1'} v_3' \overline{v_4'} (v_2 = v_2')$ were found to cause indeterminacy, but simple inspection of the figure reveals that states 0011 and 0111 are not reachable, hence the set is, in fact, empty.

However, even if we restrict to the relevant part of T, there might be cases in which false indeterminacy can arise. For instance in Figure 10.6 (a) we can see a LzTS that can be composed with the ECTRA of Figure 10.5 (b) because delay bounds of the events are the same. It is easy to see that it is the same LzTS as Figure 10.5 (a) with some additional arcs. If the general version of the ECTRA is restricted taking into account the transition relation as well as the reachable set of states, we obtain the ECTRA of Figure 10.6 (b). There is an indeterministic transition ($\mathsf{s}_5 \xrightarrow{\mathsf{c}} \mathsf{s}_8$) that from configuration $\mathcal{C}_3$ can either go to $\top$ or $\mathcal{C}_6$, although when the ECTRA is composed with the LzTS such an arc will never cause troubles because to reach configuration $\mathcal{C}_3$ trace must contain transition $\mathsf{s}_2 \xrightarrow{\mathsf{a}} \mathsf{s}_3$ and no trace containing such transition can also fire transition $\mathsf{s}_5 \xrightarrow{\mathsf{c}} \mathsf{s}_8$, hence the conflicting transition will never be taken.

A possible solution to greatly reduce "false" indeterminacy is to perform a preliminary composition and check if some conflicting transition will ever happen.

10.6.3 Projecting the composed states to avoid false indeterminacy

The idea of performing a preliminary composition is simply to determine which states of the system will be mapped to which configuration. This technique is not exact because it does not deal with the fact that some state may become unreachable due to the laziness of some of the arcs of the ECTRA, but it is by far the most powerful approach in terms of being able to avoid determinization of the three options presented so far. Yet it has an additional advantage: the mapping of the states can be used later on to minimize the transition relation representation in a symbolic setting as we explain in §14.1.

The projection process is quite simple:

- Let RS be the set of reachable states of the LzTS that we want to compose with. We map RS to configuration $\perp$. We denote this as $\mathrm{states}(\perp) = \mathrm{RS}$. In our example of Figure 10.6 (b), $\mathrm{states}(\perp) = \{\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_{12}\}$.
- If all the initially enabled events are conservative or have a minimal delay bound of 0 we will also map to $\top$ the states that satisfy that all initially enabled events are enabled *i.e.* $\mathrm{states}(\top) = \mathrm{RS} \cap \mathrm{SE}(\mathcal{E}(\top))$. Now from $\perp$ we check which transitions that appear in the arc to configuration

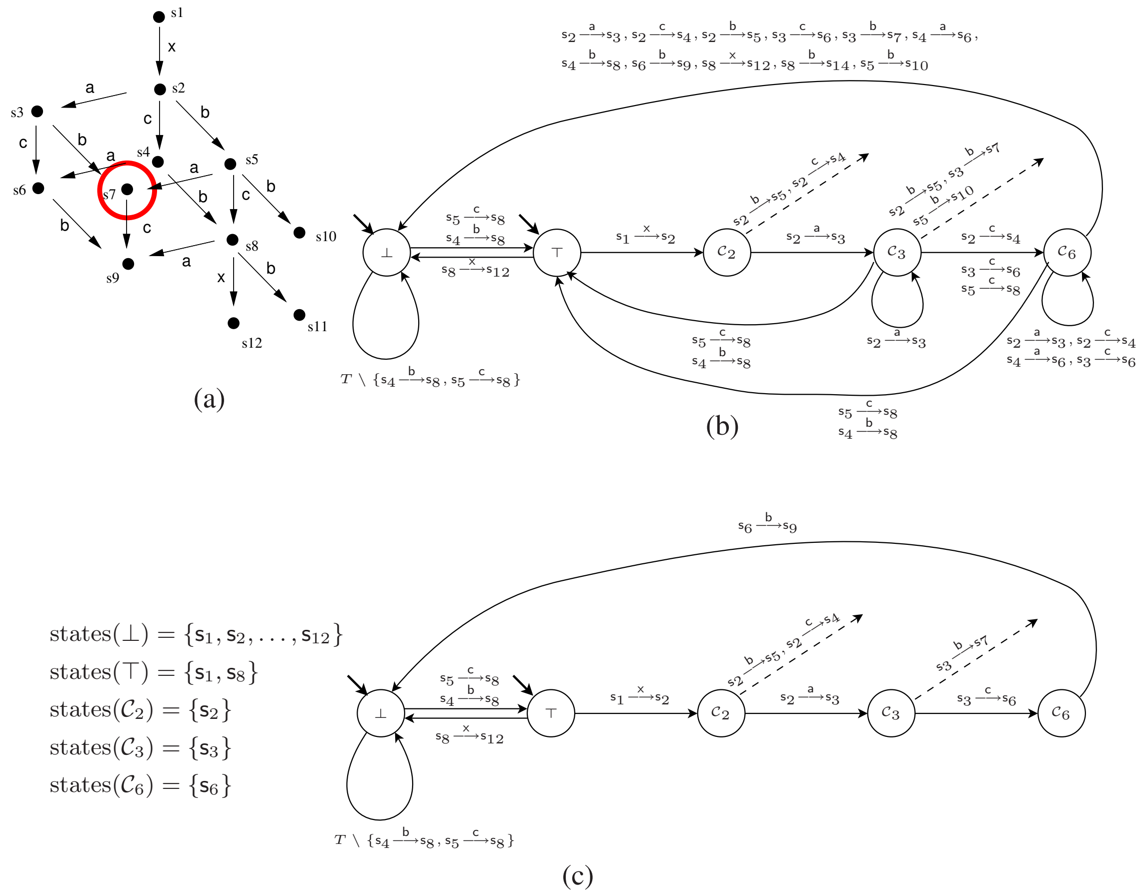


Figure 10.6. Example of the use of the transition relation and the reachable set of states to avoid unnecessary determinization: (a) LzTS that we want to compose. (b) ECTRA of Figure 10.5 (b) fully represented as a classical LzTS (*i.e.* behavior after a lazy arc is not shown) once intersected with transition relation T of the LzTS of subfigure (a) and its reachable set. It still exhibits indeterminacy. (c) The same ECTRA further refined using the mapped states to each configuration computed by Algorithm 10.2 (shown on its left). It no longer exhibits indeterminacy.

$\top$ are fireable from the mapped states. We obtain the destination states of all these transitions and add them to the mapped states of $\top$. In our example the events initially enabled are just event x. As it is unbounded then $\mathrm{states}(\top) = \{\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_{12}\} \cap \mathrm{SE}(\{\mathsf{x}\}) = \{\mathsf{s}_1, \mathsf{s}_8\}$.

- Once this setup step has been finished we proceed in a BFS way through all the configurations ignoring all the break arcs (as RS is already $\mathrm{states}(\bot)$ there is no possibility that a new state is ever mapped to this configuration), reenter arcs (for similar reasons as the break arcs) as well as all lazy arcs (indeterminacy arising in a lazy arc is not relevant).
  - First of all in current configuration we compute a fixpoint using the transitions labeling keep arcs. It is a fixpoint because we accumulate into the set of mapped states the destination states of the *keep* transitions until no new state is mapped.
  - Then for all outgoing non lazy arcs we compute their corresponding destination states and add them to the mapped states of the destination configuration of the outgoing arc. For instance from configuration $\top$ with $\mathrm{states}(\top) = \{\mathsf{s}_1, \mathsf{s}_8\}$ we have a single outgoing arc to $\mathcal{C}_2$ labeled $\mathsf{s}_1 \xrightarrow{\mathsf{x}} \mathsf{s}_2$, hence $\mathrm{states}(\mathcal{C}_2) = \{\mathsf{s}_2\}$.

A more formal description can be found in Algorithm 10.2. Note that the input ECTRA has some special "topological" requirements regarding the BFS order of the destination configurations. This is to guarantee that the computation is complete in a single pass for all configurations (because all incoming arcs of any configuration will have been processed before the configuration itself is processed). However the algorithm can be used on ECTRAs that do not satisfy this requirement if several iterations are performed until the mapping reaches its fixpoint.

Algorithm 10.2: projectStatesToECTRA

input : $\mathrm{EC} = \langle C, \Sigma, T, \perp, \mathrm{FR}\rangle$ an ECTRA in which lazy arcs have been changed for arcs pointing to final configuration F and in which outgoing arcs from configurations do not go to nodes in the same or lower BFS level (counting the BFS level from configuration $\perp$) except from the arcs going to F, $\top$ or $\perp$.

input : RS is the set of reachable states of a LzTS A.

output : A mapping states that for each configuration contains which states from the reachable set of states can potentially appear with the configuration in a compound state after composition of EC with A.

forall $\mathcal{C}_i \in C$ **do** $\mathrm{states}(\mathcal{C}_i) = \emptyset$
$\mathrm{states}(\perp) \leftarrow \mathrm{RS}$
if $\mathrm{LB}(\mathcal{E}(\top)) = \emptyset$ **then** $\mathrm{states}(\top) = \mathrm{RS} \cap \mathrm{SE}(\mathcal{E}(\top))$
forall $\mathcal{C}_i \in C \setminus \{\perp, F\}$ *s.t. they are visited in BFS order from* $\perp$ **do**
  repeat
    oldStates $\leftarrow \mathrm{states}(\mathcal{C}_i)$
    forall $\mathcal{C}_i \xrightarrow{\langle \mathsf{S}_a, \Sigma', \mathsf{S}_b\rangle} \mathcal{C}_i \in T$ **do** $\mathrm{states}(\mathcal{C}_i) \leftarrow \mathrm{states}(\mathcal{C}_i) \cup \bigcup_{\forall \mathsf{e} \in \Sigma'} \mathrm{Image}(\mathrm{states}(\mathcal{C}_i) \cap \mathsf{S}_a, \mathsf{e}) \cap \mathsf{S}_b$
  until (*oldStates* $= \mathrm{states}(\mathcal{C}_i)$)
  forall $\mathcal{C}_i \xrightarrow{\langle \mathsf{S}_a, \Sigma', \mathsf{S}_b\rangle} \mathcal{C}_j \in T$ **do**
    if $(\mathcal{C}_j \notin \{\mathcal{C}_i, \perp, \top, F\})$ **then** $\mathrm{states}(\mathcal{C}_j) \leftarrow \mathrm{states}(\mathcal{C}_j) \cup \bigcup_{\forall \mathsf{e} \in \Sigma'} \mathrm{Image}(\mathrm{states}(\mathcal{C}_i) \cap \mathsf{S}_a, \mathsf{e}) \cap \mathsf{S}_b$
return states

Once the mapping is available, transitions not starting in a mapped state can be pruned. In our example $\mathrm{states}(\mathcal{C}_3) = \{\mathsf{s}_3\}$, thus transitions $\mathsf{s}_5 \xrightarrow{\mathsf{c}} \mathsf{s}_8$ and $\mathsf{s}_4 \xrightarrow{\mathsf{c}} \mathsf{s}_8$ can be removed from the label of arc $\mathcal{C}_3 \longrightarrow \top$. As no transitions remain in the label, the arc itself can be removed. Consequently indeterminacy caused by transition $\mathsf{s}_5 \xrightarrow{\mathsf{c}} \mathsf{s}_8$ disappears as can be seen in Figure 10.6 (c).

As commented before this method still might not remove all false indeterminism because takes the current reachable set of states as the starting point. Consequently states that might become unreachable due to the laziness of some of the arcs of the ECTRA can be unnecessarily taken into account. For instance, in Figure 10.7, we can see a LzTS that we will compose also with the ECTRA of Figure 10.5 (b). In this example there is indeterminacy because of transition $\mathsf{s}_{12} \xrightarrow{\mathsf{c}} \mathsf{s}_{14}$. This transition can go from configuration $\mathcal{C}_3$ to either $\mathcal{C}_6$ or $\top$ (as in s_{12} event x is also enabled). Applying Algorithm 10.2 to this example we would not get rid of the conflicting transition because $\mathrm{states}(\mathcal{C}_3) = \{\mathsf{s}_3, \mathsf{s}_{12}\}$. However, after composition, all the zone in which state s_{12} is included is shown to be timing inconsistent, hence determinization was useless. Consequently, the only possibility to remove all false indeterminacy is to perform the composition. This can be done by modifying Algorithm 10.2 as explained in §10.6.4.

10.6.4 Composing to avoid false indeterminacy

Using this procedure will ensure that no false indeterminacy will remain in our ECTRA, however it has a computational cost that is similar to performing a reachability analysis. The idea is to use a variation of Algorithm 10.2 until a fixpoint in the mapped states is reached. Instead of the full set of reachable states, only the initial states are given to the algorithm. Once the mapping is computed,

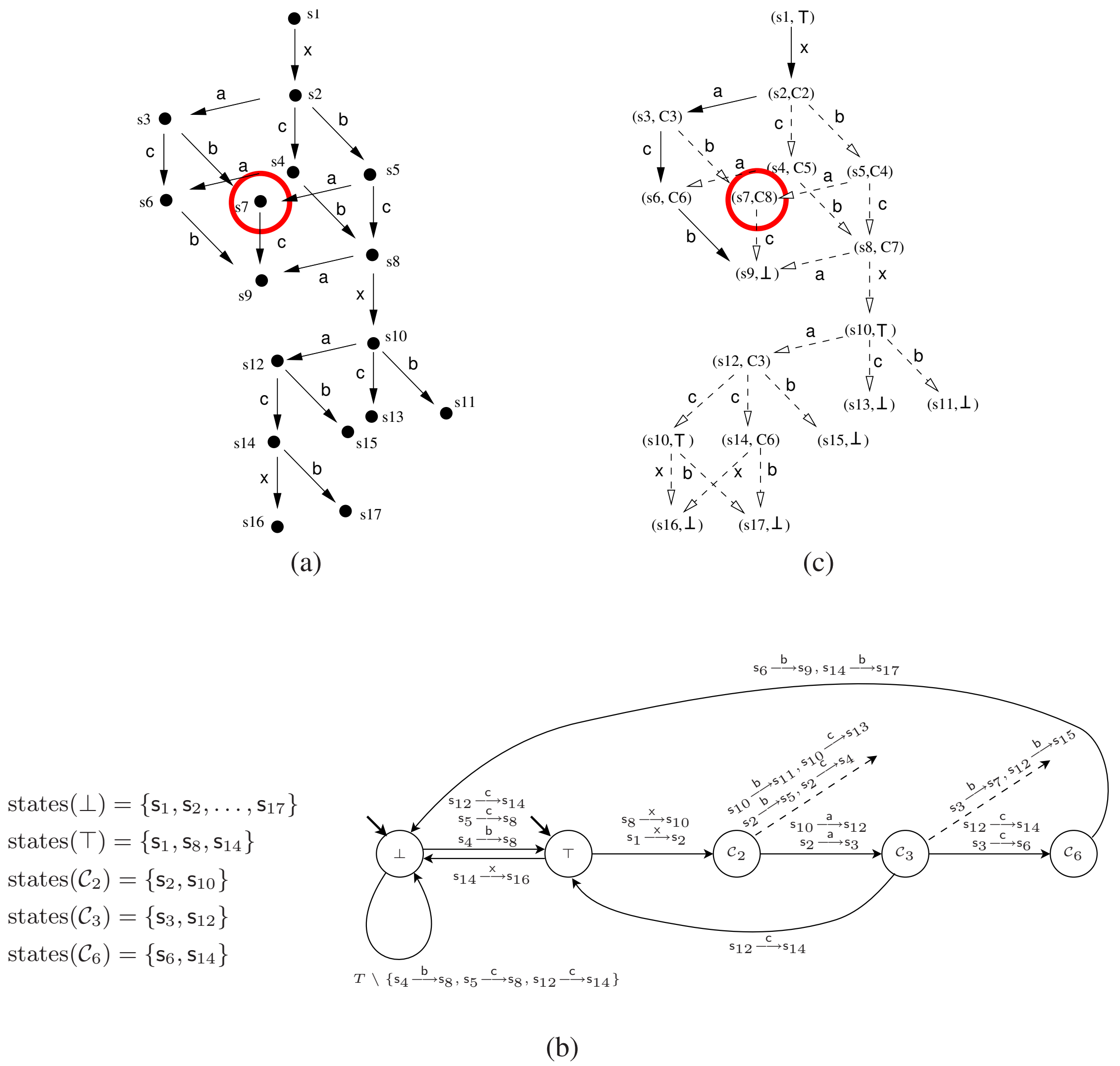


Figure 10.7. Example of reachable states projection to avoid unnecessary determinization: (a) LzTS that we want to compose. The determinization of the ECTRA of Figure 10.5 (b) cannot be avoided even with the projection method as shown in subfigure (b). After composition of the LzTS with the "indeterministic" ECTRA it is clear that the indeterminacy found was completely irrelevant because it happened precisely in the pruned part of the state space (subfigure (c)). However the projection method cannot be aware of this fact because it uses the reachable set of states available prior to the composition.

the mapped states to $\perp$ are used as the starting point for the next iteration. This process is repeated until no further change in the mapping is observed.

The modifications that must be done on Algorithm 10.2 simply affect lines 4 and 10. In the original version the mapped states for configurations $\top$ and $\perp$ were never updated as no new state could ever be generated that was not already there. However, this is not longer true, so the condition of the **forall** clause must be changed to $\mathcal{C}_i \in C \setminus \{\perp, F\}$ and the condition for the **if** clause simplifies to $\mathcal{C}_j \notin \{\mathcal{C}_i, F\}$.

Although using this procedure gives the best results in our new methodology we will not use it because of its cost. Instead we will use the projection method presented in §10.6.3 which has a good balance between efficiency and computational cost. Additionally if an indeterminacy is found using that method it is possible to use afterwards the procedure presented in this section to check if the indeterminacy is real or not. The number of variables in the system, the size of its reachable state space, the size of the ECTRA and the number of found non-deterministic arcs can be used to evaluate which option is better:

- check if indeterminacy really exists by composing (which might reveal that the determinization is unavoidable), or
- directly do the determinization and minimization of the ECTRA.

10.7 The language accepted by a system composed with an ECTRA

Once the determinization issue has been discussed we can safely present the theorem that formally relates enabling compatibility of traces with the language accepted by the LzTS resulting from the composition of an ECTRA with a LzTS.

THEOREM 10.1 (LANGUAGE ACCEPTED BY COMPOSITION)
Given an ECTRA EC *and a LzTS* A*, let* Θ *be the set of lazy traces of* EC*. Then the language accepted by the composition of* EC *and* A *(that is* $\mathcal{L}(A \odot \mathrm{EC})$*) is* $\mathcal{L}(A) \setminus \mathsf{EnComp}(\mathcal{L}(A), \Theta)$ *if the ECTRA is deterministic or no indeterministic arc of* EC *is ever taken in the composition. Otherwise* $\mathcal{L}(A) \supseteq \mathcal{L}(A \odot \mathrm{EC}) \supseteq \mathcal{L}(A) \setminus \mathsf{EnComp}(\mathcal{L}(A), \Theta)$. ■ 10.1

Proof: *See §B.10.*

In this definition the set of lazy traces of EC is used. This set Θ is formally defined as the set of all traces that go through a lazy arc when checked by EC. This set is infinite if EC has lazy arcs, however, due to the transitivity of the enabling compatible relation (see §5.5 in Chapter 5) this set can be reduced to a finite one (call it Θ') such that $\mathsf{EnComp}(\mathcal{L}(A), \Theta) = \mathsf{EnComp}(\mathcal{L}(A), \Theta')$ by using the following definition: if $\mathrm{EC} = \langle C, \Sigma, T, \mathsf{s}_0, \mathrm{FR} \rangle$ then

$$\Theta' = \bigcup_{\forall \mathcal{C} \in C, \forall \mathsf{e} \in \mathcal{E}(\mathcal{C}) : \mathcal{C} \notin \mathrm{FR}(\mathsf{e})} \mathrm{traces}(\mathcal{C}, \mathsf{e}) \tag{10.1}$$

10.8 Final remarks

Even if we are able to remove all enabling compatible traces by composing with a suitable ECTRA, the preliminary verification flow presented in Chapter 5 has two big problems. It might:

- Produce a so conservative timing analysis that is unable to remove a known timing inconsistent trace (see Chapter 11), thus violating one of the properties that iterative refinement methods must fulfill.
- Never terminate because it will perform an infinite number of compositions (see Chapter 12).

Moreover, in the former case, the overall verification process might run overly slow, because traces are removed almost individually from the system due to the lack of appropriate representativeness of the CESARs used.

Chapter 11

CONSERVATIVENESS

Summary

One of the factors that hampers the efficiency of the preliminary verification methodology is that it always uses the most concurrent CESAR that contains the failure trace. In practice this is not a good policy as the refinements of the state space in many cases are more local and many extra iterations are required. Moreover it can be the case that a timing inconsistent trace cannot be removed, violating one of the principles of iterative refinement methods. In this chapter we investigate what requirements needs to fulfill a CESAR to avoid these problems, and how we can obtain such CESAR.

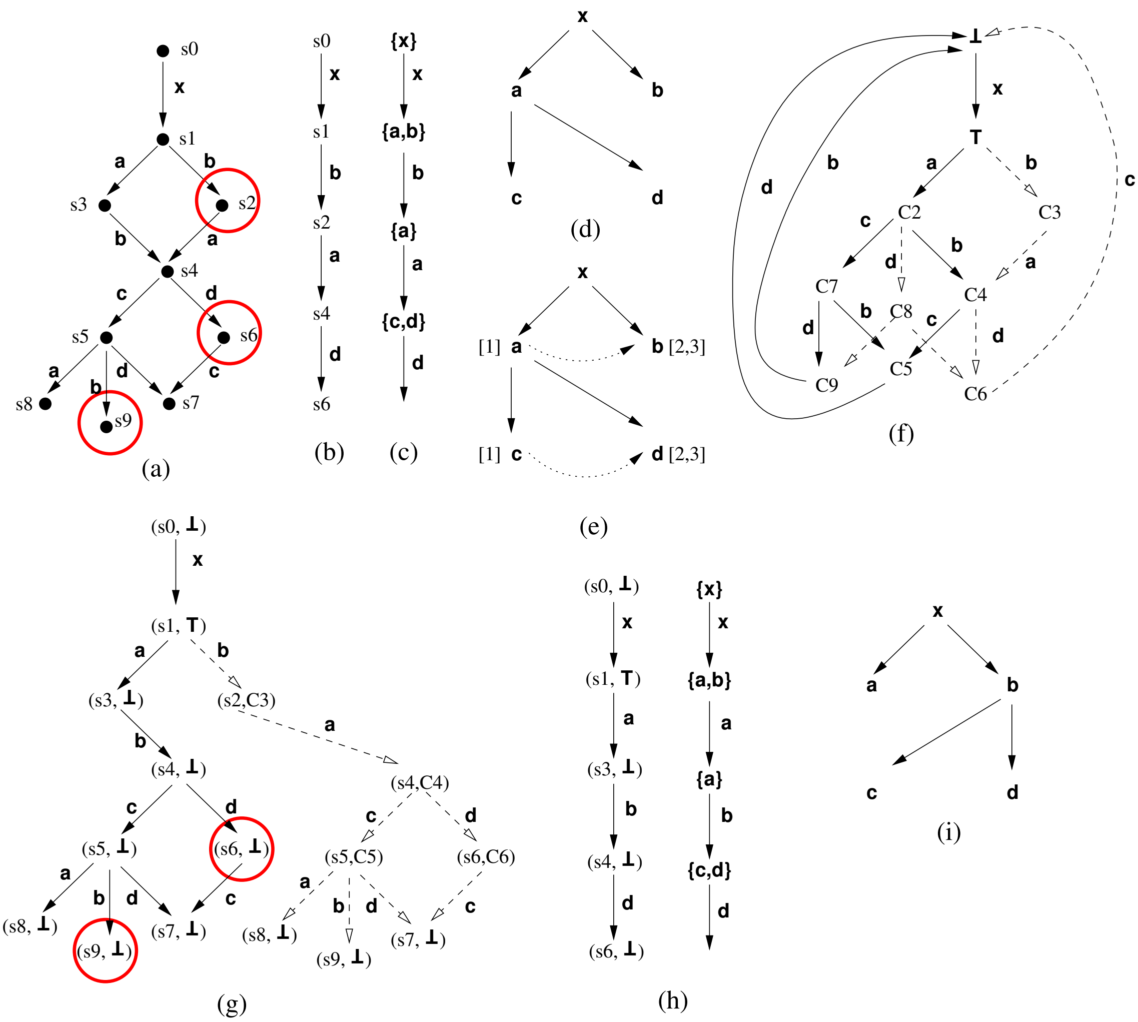


Figure 11.1. Example of local refinements due to inaccurate CESAR: (a) LzTS of Figure 10.1 (a). (b) Run to failure. (c) Trace of the run. (d) CESAR used by the preliminary verification flow. (e) LzCES obtained after timing analysis has been performed. (f) ECTRA derived from the LzCES. Most of the reenter, break and keep are omitted for readability. Also instead of the full description of the set of transitions that labels each arc, only the corresponding set of events is depicted. (g) Resulting LzTS after composition with the ECTRA. State s_6 is not removed because the ECTRA derived from the CESAR expected events b and c to be enabled after the firing of event a in configuration $\top$, but this does not correspond to the actual enablings in the system. (h) Run and trace to failure found in second iteration. (i) CESAR computed in second iteration which will finally remove failure state s_6.

11.1 When things go wrong

We start with an example of slow convergence due to a “bad” CESAR. In Figure 11.1 we can see the first iteration of the preliminary verification flow on the example of Figure 10.1 (a). The CESAR used this time is the most concurrent one (Figure 11.1 (d)) also called the *trigger graph*, in contrast of the one used in Figure 10.1 (d). Consequently the configurations of the ECTRA do not match so well the enablings of the traces of the system and some of the traces to failure state s_6 (all traces in which events c and d are enabled by b) are not removed in the first iteration as opposed to what happens in Figure 10.1. A similar problem appears if CESAR contains additional causality arcs that do not match the enabling behavior of the traces of the system.

There is an even more important reason why CESARs should be adjusted to the real state space of the system: when a CESAR is used that represents traces not appearing in the system, its timing analysis might be conservative. We illustrate this fact with an example.

EXAMPLE 11.1 *In Figure 11.2 (a) we can see a TTS. Event* x *has an unbounded delay and the delays of the remaining events are shown in subfigure (c). State* s_7 *is declared as a failure state. Subfigure (b) shows the unique failure trace present in the system. The trace is timing inconsistent because if event* b *happens before* a *then event* d *will occur before event* c*. However this information cannot be derived from the trigger graph (subfigure (d)).*

In this case the problem is related to a disabling, however the same outcome can be obtained in systems in which the only problem is that concurrent paths do not converge (see Figure C.3).

■ 11.1

This is very bad news because iterative refinement methods should always remove at least the failure trace they initially find if it is not timing consistent, otherwise, as in this example, it can happen that the trace is known to be timing inconsistent but the methodology is not able to remove it! For this reason it is important to derive CESARs that describe as faithfully as possible the state space under analysis.

It remains to be said what is understood by *describe as faithfully as possible*. We need a formal definition in order to be able to produce an adjustment procedure for CESARs. First idea is to require that all traces of the CESAR belong to the state space. For instance in previous example there was no trace in which event a was fired first and then event b. However we can run into troubles in the timing analysis if only this condition is enforced.

For instance in Figure 11.2 (d) we can see a very similar system in which the missing CESAR traces are added to the state space of the system, but the outcome is the same as before. The problem is that timing analysis on a CESAR assumes that concurrent events produce convergent traces (*e.g.* in the example the firing of $s_1 \xrightarrow{a} s_3 \xrightarrow{b}$ should lead to the same state as $s_1 \xrightarrow{b} s_2 \xrightarrow{a}$, which is not the case) and if this is not true, then the timing analysis gives conservative results as there are more ways in which a state is considered that can be reached than there really are. Thus not only traces represented by the CESAR must belong to the system but also they must follow the expected CESAR trace semantics, specifically, the convergence of concurrent paths. In the following section we define the concept of *zone* of a CESAR which allows to formally reason about this questions.

11.2 The Zone

A *zone* of a CESAR CS describes which states and arcs of a system are represented accurately (in terms of timing consistency) by CS. We give its formal definition in Definition 11.1 by establishing a relation between the GRC defined by the words of the timing computation traces of CS and the state space of the system. We use this GRC instead of the GRC of CS, which is defined by $\mathrm{words}(\mathrm{CS})$, simply because as we saw on Chapter 8the relevant set of traces in terms of timing consistency is $\mathrm{tct}(\mathrm{CS})$ and not $\mathrm{traces}(\mathrm{CS})$.

Several mappings appear in the definition that relate states/transistions of the system with configurations/transitions in the GRC. Many of them are required to represent the different degrees of timing information that can be obtained according to the properties of the states. We illustrate this particular point with some examples after the definition.

DEFINITION 11.1 (ZONE OF A **CESAR** IN A **LzTS**)

Given a CESAR CS*, a LzTS* $A = \langle S, \Sigma, T_a, s_0, \mathrm{FR} \rangle$ *and a state* s_z *of* A*, let* $\mathrm{GC} =$

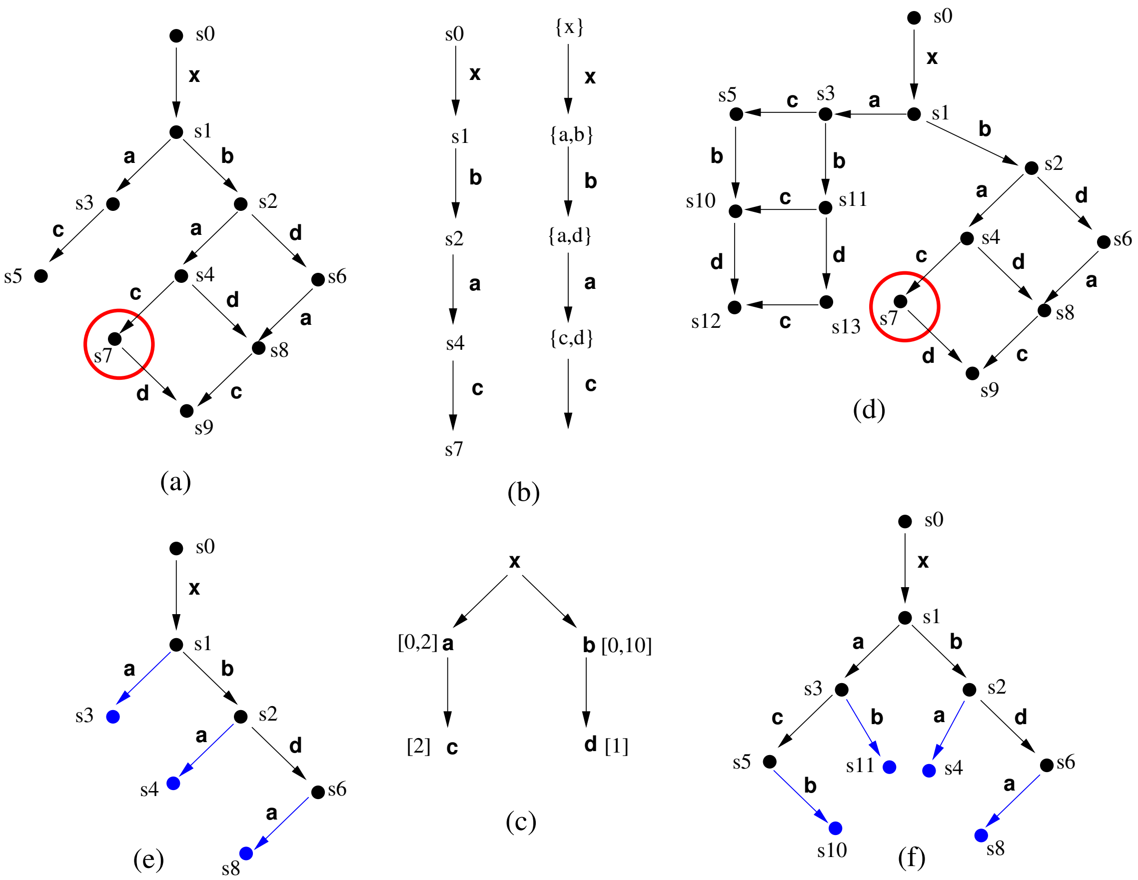


Figure 11.2. Example of failed verification due to inaccurate CESAR: (a) Untimed state space of a TTS. (b) Run to failure and its corresponding trace. (c) CESAR (trigger graph) derived from the trace. No relative timing precedences can be derived from its timing analysis. (d) Untimed state space of a TTS in which all traces covered by the CESAR of subfigure (c) are present, although they do not converge. (e) Zone of the CESAR of subfigure (c) for the system of the subfigure (a). (f) Zone of the same CESAR for the system of the subfigure (d).

$\langle C, \Sigma, T_g, \top\rangle$ be the GRC formed by the words of the set of timing computation traces of CS. *We define the* zone of CS in state s_z *as the subgraph in A formed by all the states and transitions that satisfy the following mappings from the states/transition of* GC *to the states/transitions of A.*

$$\text{states}_{in}(\top) = \text{states}_{out}(\top) = \text{states}_{pa}(\top) = \mathsf{s}_z$$

$$\text{states}_{out}(\mathcal{C}_i) = \bigcup_{\forall \mathcal{C}_j \xrightarrow{\mathsf{e}} \mathcal{C}_i \in T_g} \{\mathsf{s} \mid \mathsf{s}' \xrightarrow{\mathsf{e}} \mathsf{s} \in \text{trans}(\mathcal{C}_j \xrightarrow{\mathsf{e}} \mathcal{C}_i)\}$$

$$\text{states}_{pa}(\mathcal{C}_i) = \bigcap_{\forall \mathcal{C}_j \xrightarrow{\mathsf{e}} \mathcal{C}_i \in T_g} \{\mathsf{s} \mid \mathsf{s}' \xrightarrow{\mathsf{e}} \mathsf{s} \in \text{trans}(\mathcal{C}_j \xrightarrow{\mathsf{e}} \mathcal{C}_i)\}$$

$$\text{states}_{in}(\mathcal{C}_i) = \{\mathsf{s} \mid \text{states}_{pa}(\mathcal{C}_i) \wedge \mathcal{E}(\mathsf{s}) = \mathcal{E}(\mathcal{C}_i)\}$$

$$\text{trans}(\mathcal{C}_j \xrightarrow{\mathsf{e}} \mathcal{C}_i) = (\text{states}_{in}(\mathcal{C}_j) \times \{\mathsf{e}\} \times S) \cap T_a$$

As a zone can be uniquely identified by its starting state i.e. *state* s_z*, we will refer to it as* $\mathrm{zone}(A, \mathrm{CS}, \mathsf{s}_z)$*, which is formally a* **LzTS** *defined as follows:*

$$\langle \bigcup_{\forall \mathcal{C} \in C} \mathrm{states}_{out}(\mathcal{C}), \Sigma, \bigcup_{\forall \mathcal{C}_j \xrightarrow{\mathsf{e}} \mathcal{C}_i \in T_g} \mathrm{trans}(\mathcal{C}_j \xrightarrow{\mathsf{e}} \mathcal{C}_i), \mathsf{s}_z, \mathrm{FR}_z \rangle$$

where FR_z *is* FR *restricted to the transitions that appear in the zone.* ■ 11.1

Although the zone of a CESAR is a subgraph of a LzTS, given the relevance of the mappings onto the GRC (we extensively use them in forthcoming chapters), we provide an alternative definition in which this representation is made explicit.

Definition 11.2 (**GRC** representation of a zone)
Let z *be a zone of a* CESAR CS *in* LzTS A *defined by the mappings* states_{in}*,* states_{pa}*,* states_{out} *and* trans *over the the* GRC G *formed by the words of the set of timing computation traces of* CS. *The* GRC representation of z *is the five-tuple*

$$\langle \mathrm{states}_{in}, \mathrm{states}_{pa}, \mathrm{states}_{out}, \mathrm{trans}, G \rangle.$$

■ 11.2

We show a couple of zone examples in Figure 11.2 (e,f). The states in the mapping states_{in} (drawn in black in the pictures) form the skeleton of the zone as they conform to the CESAR semantics: concurrent paths converge and the expected enablings are respected. Hence these states identify perfectly with the configuration of the GRC to which they are mapped, so any configuration property will also be satisfied by these states. In particular this allows to substitute the configuration of Theorem 8.3 by these states, and makes it possible to apply the timing analysis to any trace obtained by firing some enabled event at that point. The destination states of these traces are precisely the states contained in the mapping states_{out} (drawn in blue). These states do not necessarily satisfy CESAR definition, but the timing analysis results can be applied to them.

The last mapping, states_{pa}, identifies a subset of the states of states_{out} that do not necessarily respect the expected enablings of the CESAR, but in which concurrent paths converge. This is important because the approach used to test if a particular state is reachable through a timing consistent trace of the CESAR depends on whether the state appears in states_{pa} or just in states_{out}.

Let us see an example in Figure 11.3. In (b) there is a CESAR whose zones we compute in two different systems depicted in (a) and (c). Their respective zones are shown in (d) and (f) while the GRC to which they are mapped appears in (e). In one case state s_4 belongs to the states_{pa} mapping (s_4 does not belong to $\mathrm{states}_{in}(\mathcal{C})$ because $\mathcal{E}(\mathsf{s}_4) \neq \mathcal{E}(\mathcal{C})$ as event c is enabled in s_4 but not in configuration $\mathcal{C}_5$), while in the other case it only belongs to states_{out}. If a state belongs to $\mathrm{states}_{pa}(\mathcal{C})$, then it can only be avoided by the timing analysis if this analysis shows that an event in set $\Sigma \setminus \mathcal{C}$ must happen before an event in $\mathcal{C}$ (*i.e.* an event not fired to reach configuration $\mathcal{C}$ has to happen before any of these fired events). This is clearly shown in first example, as although we know that $\mathsf{a} < \mathsf{b}$ from the timing analysis, state s_4 is reachable through a timing consistent trace. In fact any possible outcome of the timing analysis would have yielded the same result as in $\mathcal{C}_5$, the configuration to whom s_4 is mapped, all the events of the CESAR have already fired so there is no possible precedence to avoid it.

On the other hand, to use a similar reasoning in our second example would be wrong. In this case s_4 is avoidable. For instance if $\mathsf{a} < \mathsf{b}$, then there is no timing consistent trace covered by the CESAR from s_0 to s_4. This example illustrates that when concurrent paths do not converge there are more precedences that allow to avoid a given state. In particular the precedences that allow to

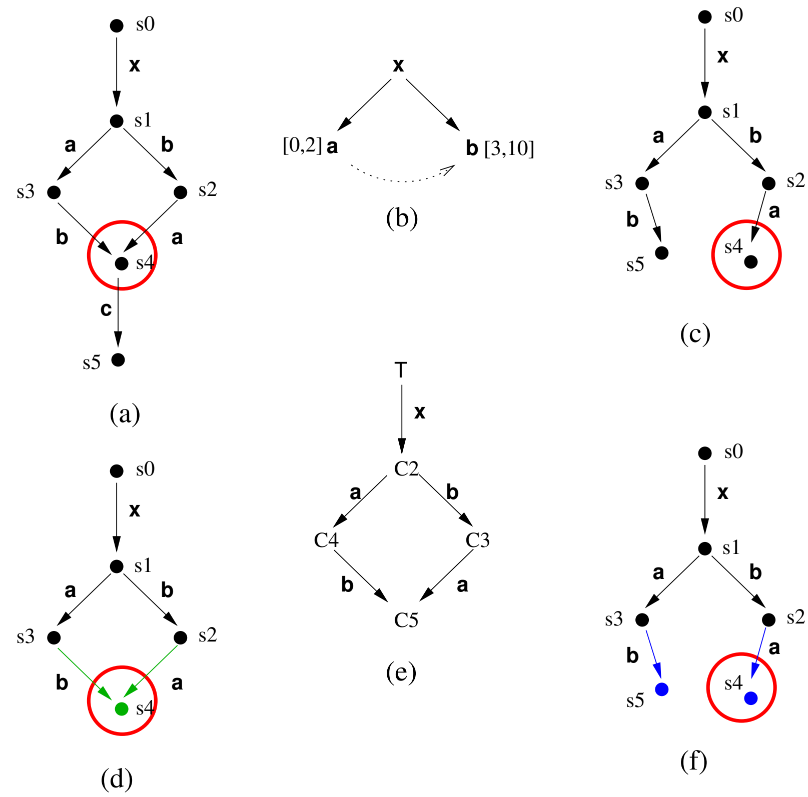


Figure 11.3. Example of the semantics of the different mappings used to define a zone: (a) Untimed state space of a TTS with convergence of concurrent paths. (b) A CESAR after timing analysis (LzCES). (c) Untimed state space of a TTS without convergence of concurrent paths. (d) Zone of the CESAR for the system of the subfigure (a). State s_4 cannot be avoided no matter the outcome of the timing analysis. (e) GRC of the words of the timing computation traces of the CESAR. As there are no disablings or restrictions in the CESAR it is isomorphic with the GRC of the CESAR. (f) Zone for the system of the subfigure (c). Precedence $a < b$ allows to avoid failure state s_4.

avoid an arc $\mathcal{C}_j \xrightarrow{e} \mathcal{C}_i$ are of the form $a < b$ where $a \in \Sigma \setminus \mathcal{C}_i$ and $b \in \mathcal{C}_i$ (these precedences are the same ones that avoid $\mathcal{C}_i$) or where $a \in \Sigma \setminus \mathcal{C}_j$ and $b \in \mathcal{C}_j$ (these precedences are the same ones that avoid $\mathcal{C}_j$). These two options share many of the precedences and the whole expression can be rewritten as: precedence $a < b$ avoids arc $\mathcal{C}_j \xrightarrow{e} \mathcal{C}_i$ iff it avoids $\mathcal{C}_i$ or $b = e$ and $a \in \Sigma \setminus \mathcal{C}_i$ which is easier to test.

If a given state can only be reached by a subset of arcs, then the only precedences that allow to avoid that state can be computed as the intersection of all the precedences that can avoid each one of its incoming arcs. For instance in our example state s_4 can only be reached (in the GRC) through arc $\mathcal{C}_3 \xrightarrow{a} \mathcal{C}_5$ where $\mathcal{C}_3 = \{b\}$ and $\mathcal{C}_5 = \{a, b\}$. As $\Sigma = \{a, b\}$ in this case the only possible precedence to avoid this arc is the one capable of avoiding configuration $\mathcal{C}_3$, that is $a < b$. In fact it is easy to show that when the number of arcs reaching a state in $\mathrm{states}_{out}(\mathcal{C})$ is greater than one, then the only precedences that allow to avoid all arcs are simply the ones that allow to avoid $\mathcal{C}$. We group all these results on the precedences that avoid a particular state or transition of a zone in the following lemma.

LEMMA 11.1 (PRECEDENCES THAT AVOID STATES/TRANSITIONS)

Given a zone of a CESAR, its mappings states_{pa}, states_{out} *and* trans, *and a precedence* $\mathsf{a} < \mathsf{b}$ *between concurrent events of the CESAR.*

- *If state* $\mathsf{s} \in \text{states}_{pa}(\mathcal{C})$, *then it is avoided by precedence* $\mathsf{a} < \mathsf{b}$ *iff* $\mathsf{a} \notin \mathcal{C}$ *and* $\mathsf{b} \in \mathcal{C}$. *In such case we say that* $\mathsf{a} < \mathsf{b}$ skips $\mathcal{C}$.
- *If transition* $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}' \in \text{trans}(\mathcal{C}_i \xrightarrow{\mathsf{e}} \mathcal{C})$, *then it is avoided by a precedence* $\mathsf{a} < \mathsf{b}$ *iff* $\mathsf{a} < \mathsf{b}$ *skips* $\mathcal{C}$ *or* $\mathsf{a} = \mathsf{e} \wedge \mathsf{b} \in \mathcal{C}_i$. *This expression can be simplified to* $\mathsf{a} \notin \mathcal{C}_i \wedge \mathsf{b} \in \mathcal{C}$ *using the fact that* $\mathsf{a} \neq \mathsf{b}$ *because they must be concurrent events of the CESAR. In any case, when any of the previous expressions is satisfied, we say that* $\mathsf{a} < \mathsf{b}$ skips $\mathcal{C}_i \xrightarrow{\mathsf{e}} \mathcal{C}$.
- *If state* $\mathsf{s} \in \text{states}_{out}(\mathcal{C}) \setminus \text{states}_{pa}(\mathcal{C})$ *and*
  - *there is a single path reaching state* s, i.e. *there is only one transition* $\mathsf{s}' \xrightarrow{\mathsf{e}} \mathsf{s}$ *such that belongs to some* $\text{trans}(\mathcal{C}_i \xrightarrow{\mathsf{e}} \mathcal{C})$, *then it is avoided by a precedence* $\mathsf{a} < \mathsf{b}$ *if and only if* $\mathsf{a} < \mathsf{b}$ *skips* $\mathcal{C}_i \xrightarrow{\mathsf{e}} \mathcal{C}$.
  - *there is more than one path reaching* s, *then it is avoided by a precedence* $\mathsf{a} < \mathsf{b}$ *if and only if* $\mathsf{a} < \mathsf{b}$ *skips* $\mathcal{C}$.

■ 11.1

Intuitively a zone is a part of the state space whose traces have their temporal behavior *exactly* represented by the CESAR as shown in Theorem 11.2.

THEOREM 11.2 (EQUIVALENCE BETWEEN TIMING ANALYSIS)

Given a LzTS A, a LzCES $\text{LCS} = \langle \text{CS}, <_{\text{lzy}} \rangle$, *a state* s *of* A, *and a run* ρ *in the zone of* CS *in* s *such that* $\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b \in \rho$, *then* ρ *respects the orderings in* $<_{\text{lzy}}$, *if and only if, for each transition* $\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b$ *of* ρ, *it exists a timing consistent trace of* $\text{tct}(\text{CS})$ *through* $\mathcal{C}_a \xrightarrow{\mathsf{e}} \mathcal{C}_b$ *where* $\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b \in \text{trans}(\mathcal{C}_a \xrightarrow{\mathsf{e}} \mathcal{C}_b)$ *and* trans *is the mapping of transitions used by the zone. Formally,*

$$\begin{aligned} \rho \textit{ satisfies } <_{\text{lzy}} \iff \; & \forall \mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b \in \rho, \exists \theta \in \text{tct}(\text{CS}) : \mathcal{E}(\mathcal{C}_a) \xrightarrow{\mathsf{e}} \mathcal{E}(\mathcal{C}_b) \in \theta \wedge \\ & \wedge\, \mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b \in \text{trans}(\mathcal{C}_a \xrightarrow{\mathsf{e}} \mathcal{C}_b) \wedge \theta \textit{ is timing consistent.} \end{aligned}$$

■ 11.2

Proof: *See Appendix B.11.*

The meaning of the theorem, despite of the formalization, is quite simple: only in those areas of the LzTS where the CESAR really describes accurately the behavior the system, then the timing analysis gives comparable results to the timing-consistency checking of traces. On must be careful with the exact meaning of the term *comparable results*. For instance a run of the zone satisfying all precedences in $<_{\text{lzy}}$ does not necessarily have to be timing consistent. What theorem guarantees is that there is some timing consistent run through each one of the arcs of the run. We provide an example of such phenomenon in Figure 11.4.

The state space of a TTS is shown in Figure 11.4 (a) and the delays associated to its events can be found in (b). In this system, failure trace $\mathsf{s}_0 \xrightarrow{\mathsf{x}} \mathsf{s}_1 \xrightarrow{\mathsf{b}} \mathsf{s}_3 \xrightarrow{\mathsf{a}} \mathsf{s}_4 \xrightarrow{\mathsf{c}} \mathsf{s}_9 \xrightarrow{\mathsf{d}} \mathsf{s}_{10}$ is timing inconsistent. In (b) appears a CESAR whose zone in s_0 coincides perfectly with the state space of the system (*i.e.* the CESAR represents all the behavior of the system). Timing analysis on the CESAR only reveals that event a should happen before c. Consequently the previous failure trace satisfies all the precedences given by the timing analysis (none of its arcs is marked as lazy in the GRC) but, at the same time, we know that is timing inconsistent because event d should happen before c if $\text{trigger}(\mathsf{d}) = \mathsf{a}$. However, as configuration $\mathcal{C}_{10}$, that corresponds to state s_9, is reachable in the

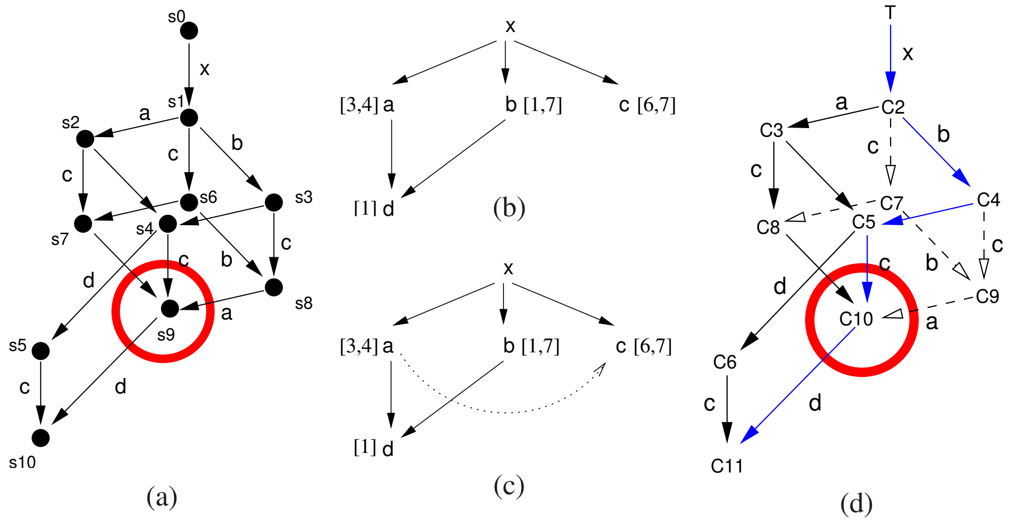


Figure 11.4. (a) Untimed state space of a TTS. (b) CESAR that expresses the causality in all the paths present in the system, with annotated delays for the events. Its zone in state s_0 is precisely all the state space of the system. (c) LzCES obtained after timing analysis, only showing that a must happen before c. (d) GRC of the LzCES. Failure trace $s_0 \xrightarrow{x} s_1 \xrightarrow{b} s_3 \xrightarrow{a} s_4 \xrightarrow{c} s_9 \xrightarrow{d} s_{10}$ is inconsistent, but timing analysis does not falsify it (the corresponding arcs of the trace are depicted in blue).

GRC, we know by Theorem 11.2 that there is another trace reaching this state which is timing inconsistent. The only thing that remains to be done is to find which one is the trace. In §13.4 we present a procedure to do such task when a particular failure is known to be not avoidable.

But before tackling this problem, we must first solve the question of obtaining a CESAR that represents with enough accuracy the relevant parts of the state space of a system. In §11.3 we present an algorithm that, given a trace (or trace fragment), computes a CESAR whose zone is guaranteed to contain the trace. In this way the methodology can take advantage of Theorem 11.2 and safely use timing analysis on CESARs instead of individual trace timing consistency checking.

11.3 Algorithm to compute a CESAR whose zone includes the original trace

In this section we describe the algorithms that allow to obtain a CESAR whose zone contains the original failure trace. This basically consists in incrementally compute the zone of the current proposed CESAR. If at some point of this process a problem is encountered that would make it impossible for the zone to contain the failure trace, then the CESAR is modified to take into account the unexpected behavior. The main algorithm is shown in Algorithm 11.1. From line 1 to 8 the trigger graph CS is built and, from line 9 to 16, a preliminary zone z is constructed that only contains the original failure run ρ. We store the zone information using a GRC, in the same way as in Definition 11.1. However, we use only a single mapping that corresponds to the states_{in} mapping of the definition. Finally, the function `checkNode` is called at each one of the configurations of the zone following the order of the run.

The function `checkNode` is shown in Algorithm 11.2. Its precondition is that, when called, the existing arcs and states in the zone must respect the current CESAR semantics, that is, what has been seen up to this point in the analysis could have been generated by the CESAR. The postcondition is that the zone contains all the runs in which all incoming events to configuration $\mathcal{C}_i$ have been moved "upwards" by exchanging positions with concurrent preceding events. All these

Algorithm 11.1: computeCESARfromTrace

input : $\rho = \mathsf{s}_0 \xrightarrow{\mathsf{e}_0} \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \ldots \mathsf{s}_n \xrightarrow{\mathsf{e}_n} \mathsf{s}_{n+1}$ a timing inconsistent run.
output : CS $= \langle \Sigma, \prec, \triangleright, <_{\mathrm{CS}} \rangle$ a CESAR whose zone contains the the run ρ.

$\prec \leftarrow \emptyset$
$\triangleright \leftarrow \emptyset$
$\Sigma \leftarrow \{\mathsf{e}_0, \ldots, \mathsf{e}_n\}$
forall $i : 0 \leq i \leq n$ **do**
  forall $\mathsf{e} \in \mathcal{E}(\mathsf{s}_{i+1}) \setminus \mathcal{E}(\mathsf{s}_i)$ **do** $\prec \leftarrow \prec \cup \{(\mathsf{e}_i, \mathsf{e})\}$
  forall $\mathsf{d} \in \mathcal{E}(\mathsf{s}_i) \setminus \mathcal{E}(\mathsf{s}_{i+1}) \wedge \mathsf{e}_i \neq \mathsf{d}$ **do** $\triangleright \leftarrow \triangleright \cup \{(\mathsf{e}_i, \mathsf{d})\}$
$<_{\mathrm{CS}} \leftarrow (\prec \cup \triangleright)^*$
$\mathrm{CS} \leftarrow \langle \Sigma, \prec, \triangleright, <_{\mathrm{CS}} \rangle$
$C \leftarrow \emptyset$
$T \leftarrow \emptyset$
forall $i : 0 \leq i \leq n+1$ **do**
  $\mathcal{C}_i \leftarrow \{\mathsf{e}_0, \ldots, \mathsf{e}_i\}$
  $C \leftarrow C \cup \mathcal{C}_i$
  $\mathrm{states}(\mathcal{C}_i) \leftarrow \{\mathsf{s}_i\}$
  if $i > 0$ **then** $T \leftarrow T \cup \{\mathcal{C}_{i-1} \xrightarrow{\mathsf{e}_{i-1}} \mathcal{C}_i\}$
$z \leftarrow \langle \Sigma, C, T, \mathcal{C}_0 \rangle$
forall $i : 0 \leq i \leq n+1$ **do** `checkNode` $(CS, z, \mathcal{C}_i)$
return CS

runs belong to the system *and* are generated by the CESAR (possibly by modifying it). The result that from a trace one can generate all the concurrent ones by exchanging the position of adjacent concurrent events is a known result from Mazurkiewicz's trace theory [Maz88]. As `checkNode` calls in Algorithm 11.1 are done incrementally following the firing order of the run, new events (in this discussion we will consider all events distinct) are introduced one at a time, thus all concurrent traces are generated by swapping this new event with its predecessors (see Figure 11.6 for an example).

The procedure is quite simple: take the last two events fired to reach configuration $\mathcal{C}_i$ present in the GRC that represents the zone and check using the CESAR if both events are completely concurrent. If so, then call function `checkPath` which checks if the concurrent GRC path $\mathcal{C}_k \xrightarrow{\mathsf{e_j}} \mathcal{C}_k \cup \{\mathsf{e}_j\} \xrightarrow{\mathsf{e_k}} \mathcal{C}_i$ also exists in the system *i.e.* if there are states and transitions in the system that could be mapped to this path. This function might call the `checkNode` function, hence the procedure is recursive. As recursive calls are issued on events with lower BFS number (configurations nearer to the initial configuration than the current one) the recursivity is guaranteed to end at some point. Every time function `checkNode` is called, there is a single arc $\mathcal{C}_j \xrightarrow{\mathsf{e_j}} \mathcal{C}_i$ in the GRC. This is verified in Algorithm 11.1 as well as in `checkPath` function that are the only two places where `checkNode` function is used.

Note that in the function the events are checked to see if they are completely concurrent each time before a `checkPath` call is issued instead of computing which pairs are initially concurrent. This is so because a previous call to function `checkPath` might modify the CESAR so that two initially concurrent events are no longer concurrent due to an added causality or precedence arc. Thus worthless work is avoided by skipping pairs of events that have been shown not completely concurrent by previous function calls.

Function `checkPath` is described in Algorithm 11.3. As we will show in Lemma 11.3 there are only two possible scenarios when this function is called: either the first edge and the middle

Algorithm 11.2: checkNode

input : $\rho = \mathsf{s}_0 \xrightarrow{\mathsf{e}_0} \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \ldots \mathsf{s}_n \xrightarrow{\mathsf{e}_n} \mathsf{s}_{n+1}$ a timing inconsistent run.
input/output : $\mathrm{CS} = \langle \Sigma, \prec, \triangleright, <_{\mathrm{CS}} \rangle$ a CESAR.
input/output : $z = \langle \Sigma, C, T, \mathcal{C}_0 \rangle$ GRC used to represent the partially computed zone for the current CESAR.
input : $\mathcal{C}_i$ a configuration in the GRC that we want to check.

forall $\mathcal{C}_j \xrightarrow{\mathsf{e}_j} \mathcal{C}_i \in T$ **do**
  forall $\mathcal{C}_k \xrightarrow{\mathsf{e}_k} \mathcal{C}_j \in T$ **do**
    if $\mathsf{e}_j \not<_{\mathrm{CS}} \mathsf{e}_k \wedge \mathsf{e}_k \not<_{\mathrm{CS}} \mathsf{e}_j$ **then** `checkPath(`ρ,CS,z,$\mathcal{C}_k$,e_j,e_k,$\mathcal{C}_i$`)`

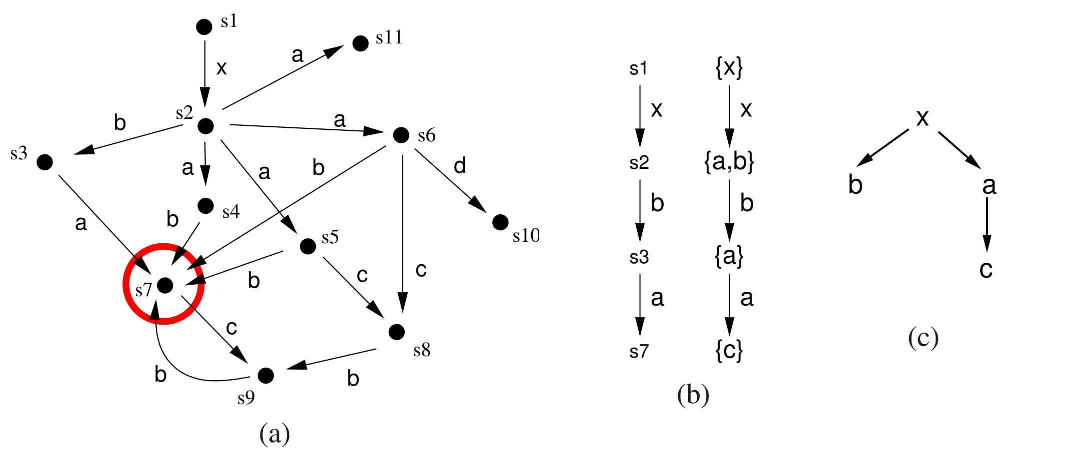


Figure 11.5. Example for `CheckPath` function: (a) Untimed state space of a TTS. (b) Run to failure and its corresponding trace. (c) CESAR (trigger graph) derived from the trace.

configuration of the path already exists in the GRC (hence we do only need to check the final edge of the path) or nothing exists (hence the two edges and the middle configuration must be checked and added to the zone). In the first scenario the function `checkEdge` is the one that does the job, so we will treat this case when explaining this function. The only relevant fact about this function at this moment is that it returns a boolean value indicating whether the checking was a success or not.

In case of failure the path cannot be included in the zone hence pruned in the CESAR. Note, however, that if events e_i and e_j are known to be independent (see §3.3.2) then it is not necessary to do any further checking and the call to function `checkEdge` could be skipped. In case all the elements of the path must be checked and added (this case starts at line 5), the first thing we do is to check if the path is convergent, that is, is it possible to reach the states mapped to target configuration $\mathcal{C}_j$ from the states of source configuration $\mathcal{C}_i$ by firing events $\xrightarrow{\mathsf{e}_i}\xrightarrow{\mathsf{e}_j}$? To decide this question the common states reached by firing e_i forward from $\mathcal{C}_i$ and e_j backwards from $\mathcal{C}_j$ are computed. If this set is empty then there is no convergent path and the path must be removed from the CESAR. This is done by the `prune` function which will be explained later on but simply adds a precedence that removes the conflicting path that also respects the firing order of the original failure run ρ.

If there are convergent states, then we check if these states satisfy the expected enablings according to the current CESAR definition. This is performed by the function `checkEnablings` which returns a list of records in which it is noted for which set of states what kind of violations

Algorithm 11.3: checkPath

input : $\rho = \mathsf{s}_0 \xrightarrow{\mathsf{e}_0} \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \dots \mathsf{s}_n \xrightarrow{\mathsf{e}_n} \mathsf{s}_{n+1}$ a timing inconsistent run.
input/output : $\mathrm{CS} = \langle \Sigma, \prec, \rhd, <_{\mathrm{CS}} \rangle$ a CESAR.
input/output : $z = \langle \Sigma, C, T, \mathcal{C}_0 \rangle$ GRC used to represent the partially computed zone for the current CESAR.
input : $\mathcal{C}_i$ a configuration in the GRC that is the start of the path we want to check.
input : e_i the first event in the path we want to check.
input : e_j the second event in the path we want to check.
input : $\mathcal{C}_j$ a configuration in the GRC that is the end of the path we want to check.

1 $\mathcal{C}_k \leftarrow \mathcal{C}_i \cup \{\mathsf{e}_i\}$
2 **if** $\mathcal{C}_i \xrightarrow{\mathsf{e}_i} \mathcal{C}_k \in T$ **then**
3   **if** $\neg$`checkEdge(`ρ,CS,z,$\mathcal{C}_k$,e_j,$\mathcal{C}_j$`)` **then** `prune(`CS,e_i,e_j`)`
4 **else**
5   convergent $\leftarrow \mathrm{Image}(\mathrm{states}(\mathcal{C}_i), T_{\mathsf{e}_i}) \cap \mathrm{Preimage}(\mathrm{states}(\mathcal{C}_j), T_{\mathsf{e}_j})$
6   **if** convergent $= \emptyset$ **then**
7   `prune(`CS,e_i,e_j`)`
8   **return**
9   violationList $\leftarrow$ `checkEnablings(`CS, $\mathcal{C}_k$, convergent`)`
10   **if** violationList.head().numberOfViolations $= 0$ **then**
11   rightStates $\leftarrow$ violationList.head().states()
12   **else**
13   rightStates $\leftarrow$ `proposeCESARmodifications(`CS,$\mathcal{C}_i$,e_i,e_j,$\mathcal{C}_j$,violationList`)`
14   **if** rightStates $= \emptyset$ **then**
15   `prune(`CS,e_i,e_j`)`
16   **return**
17         $C = C \cup \mathcal{C}_k$
18         $T = T \cup \{\mathcal{C}_i \xrightarrow{\mathsf{e}_i} \mathcal{C}_k, \mathcal{C}_k \xrightarrow{\mathsf{e}_j} \mathcal{C}_j\}$
19   **forall** $\mathsf{a} \in \mathcal{E}(\mathcal{C}_k)$ **do** $\mathrm{validPred}(\mathsf{a}) \leftarrow \mathrm{validPred}(\mathsf{a}) \cap \mathcal{C}_k$
20   `propagate(`z, $\mathcal{C}_k$, rightStates`)`
21   `checkNode(`ρ,CS,z, $\mathcal{C}_k$`)`

of the expected enabling there are. For instance assume that function `checkPath` has been called on the system of Figure 11.5 (a) to check if the path from s_2 to s_7 firing $\xrightarrow{\mathsf{a}}\xrightarrow{\mathsf{b}}$ exists and satsfies current CESAR that is the trigger graph of subfigure (c) obtained from the failure run of subfigure (b). First of all convergent states are $\mathsf{s}_4, \mathsf{s}_5$ and s_6 as they are the only ones that can occupy the intermediate position in sequence $\mathsf{s}_2 \xrightarrow{\mathsf{a}}\xrightarrow{\mathsf{b}} \mathsf{s}_7$. It is expected that events b and c are enabled in these states according to the current CESAR of subfigure (c). However we have that $\mathcal{E}(\mathsf{s}_4) = \{\mathsf{b}\}$, $\mathcal{E}(\mathsf{s}_5) = \{\mathsf{b}, \mathsf{c}\}$ and $\mathcal{E}(\mathsf{s}_6) = \{\mathsf{b}, \mathsf{c}, \mathsf{d}\}$. Function `checkEnablings` will return a list ordered by the number of violations (with the states with less violations first) like this one:

$$\langle \{\mathsf{s}_5\}, \emptyset \rangle, \langle \{\mathsf{s}_4\}, \text{unexpected disabling of } \mathsf{c} \rangle, \langle \{\mathsf{s}_6\}, \text{unexpected enabling of } \mathsf{d} \rangle$$

If there are states that satisfy the expected enablings they are selected as the states that will be mapped to the middle configuration $\mathcal{C}_k$ (line 10). For instance, in the example state s_5 would be selected. Otherwise we can only save the path if the CESAR is modified somehow to fit with the existing enablings. This is done in function `proposeCESARmodifications` that selects which modification will be performed and returns the set of states that satisfies the new enablings. If no modification is possible (remember that CESAR model has some restrictions on the relations it contains) then the path is pruned (lines 14-15). Otherwise the GRC is updated with the new path (lines 17-18) and function `checkNode` is called in the new configuration $\mathcal{C}_k$ to ensure that all CESAR paths are checked.

The function validPred assigns each event to its set of valid causal predecessors. Initially all events have it initialized to Σ, but once a node is added to the zone, any enabled event can only be triggered by events that have fired prior to this point. The purpouse of keeping this information is to prevent from introducing causal relations that violate the information in the already processed configurations (see Algorithm 11.5).

Finally, function `propagate` checks that the mapped states to each configuration only contain the states that really converge with the rest of the configurations. That is, it is possible that some source configuration contains three states but that only one of them really reconverges as expected, hence it is necessary to remove all states that do not satisfy all the conditions.

LEMMA 11.3
When function `checkPath`($\mathrm{CS}, z, \mathcal{C}_i, \mathsf{e}_i, \mathsf{e}_j, \mathcal{C}_j$) is called it can never be the case that the middle configuration ($\mathcal{C}_k$) exists but the first edge of the path ($\mathcal{C}_i \xrightarrow{\mathsf{e_i}} \mathcal{C}_k$) does not yet appear in the zone. ■ 11.3

Proof: *The unique way in which a configuration is added to the GRC representing the zone is through a `checkPath` call, so to generate $\mathcal{C}_k$ without the edge $\mathcal{C}_i \xrightarrow{\mathsf{e_i}} \mathcal{C}_k$ it would have been necessary that `checkPath` has been called by `checkNode`($\mathcal{C}_k$) in the following scenario: $\mathcal{C}_x \xrightarrow{\mathsf{e_i}} \mathcal{C}_y \xrightarrow{\mathsf{e_x}} \mathcal{C}_k$ which would have produced the call `checkPath`($\mathrm{CS}, z, \mathcal{C}_x, \mathsf{e}_x, \mathsf{e}_i, \mathcal{C}_k$) in order to check the concurrent path. However this scenario requires that e_i is fired before e_x, which is a situation that cannot belong to the initial run, as when the function `checkPath`($\mathrm{CS}, z, \mathcal{C}_i, \mathsf{e}_i, \mathsf{e}_j, \mathcal{C}_j$) is called, the event e_i is being "swapped" up to the top, hence e_i is the last event when the current top-level `checkNode` call is issued. As recursive call always go to shorter traces is not possible that e_i appears before some other event unless a previous `checkPath` call has generated the swapping between e_x and e_i. But this swap is only possible in the following scenario: $\mathcal{C}_x \xrightarrow{\mathsf{e_x}} \mathcal{C}_z \xrightarrow{\mathsf{e_i}} \mathcal{C}_k$. As $\mathcal{C}_z \xrightarrow{\mathsf{e_i}} \mathcal{C}_k$ by uniqueness of the configurations we have that $\mathcal{C}_z = \mathcal{C}_i$ so the only possible way that allows $\mathcal{C}_k$ to be already present in the zone is that the arc $\mathcal{C}_i \xrightarrow{\mathsf{e_i}} \mathcal{C}_k$ is already present as well.*

The `checkEdge` function (Algorithm 11.4) guarantees that only the states of $\mathcal{C}_i$ and $\mathcal{C}_j$ that have an arc between them labeled with e_i remain in the system. The procedure is quite simple: if there are no such states, then the arc must be removed from the zone because it does not satisfy the convergence condition. Otherwise we must restrict the mapped states in both configurations to only those who are related to the other configuration through event e_i. If there are states that must be removed, then there can be a chain effect in which other states are removed from other configurations. It can never be the case that such chain effects remove all the states from a configuration because in the current configuration there is at least one state (otherwise would have been pruned and function `propagate` would not have been called) and the rest of the states of the mapped states already satisfy the condition that `checkEdge` enforces. Thus if you remove some state from a configuration it is guaranteed that all incoming/outgoing configurations will have at least one state that still is able to reach/be reached by the states remaining.

In Figure 11.6 we can see the sequence of calls that the `computeCESARfromTrace` algorithm produces in a system that satisfies the specifications of a CESAR in which three events (a,b and c) are completely concurrent. When `checkPath` is called either the full path did not exist or only the first arc and the middle configuration exists, in which case the `checkEdge` function is called.

Function `proposeCESARmodifications` (Algorithm 11.5) is in charge of modifying the CESAR when some enabling violation is found during zone exploration. Loop from line 2 to 8 looks

Algorithm 11.4: checkEdge

input : $\rho = \mathsf{s}_0 \xrightarrow{\mathsf{e}_0} \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \ldots \mathsf{s}_n \xrightarrow{\mathsf{e}_n} \mathsf{s}_{n+1}$ a timing inconsistent run.
input/output : $\mathrm{CS} = \langle \Sigma, \prec, \rhd, <_{\mathrm{CS}} \rangle$ a CESAR.
input/output : $z = \langle \Sigma, C, T, \mathcal{C}_0 \rangle$ GRC used to represent the partially computed zone for the current CESAR.
input : $\mathcal{C}_i$ a configuration in the GRC that is the start of the edge we want to check.
input : e_i the event that labels the edge we want to check.
input : $\mathcal{C}_j$ a configuration in the GRC that is the end of the edge we want to check.

statesFromCi $\leftarrow \mathrm{Image}(\mathrm{states}(\mathcal{C}_i), T_{\mathsf{e}_i})$
if statesFromCi $\cap\, \mathrm{states}(\mathcal{C}_j) = \emptyset$ **then**
    return *false*
else
    $T \leftarrow T \cup \{\mathcal{C}_i \xrightarrow{\mathsf{e}_i} \mathcal{C}_j\}$
    if statesFromCi $\supset \mathrm{states}(\mathcal{C}_j)$ **then**
        `propagate(`z, $\mathcal{C}_i$, $\mathrm{Preimage}(\mathrm{states}(\mathcal{C}_j), T_{\mathsf{e}_i})$`)`
    else
        if statesFromCi $\subset \mathrm{states}(\mathcal{C}_j)$ **then** `propagate(`z,$\mathcal{C}_j$, statesFromCi`)`
    return *true*

Algorithm 11.5: proposeCESARmodifications

input/output : $\mathrm{CS} = \langle \Sigma, \prec, \rhd, <_{\mathrm{CS}} \rangle$ a CESAR.
input : $\mathcal{C}_i$ a configuration in the zone that is the start of the path we want to check.
input : e_i the first event in the path we want to check.
input : e_j the second event in the path we want to check.
input : vList a violation list that contains records in which the violating states as well as the type of violations are kept.

solvableViolations $\leftarrow \emptyset$
while vList *is not empty* **do**
    violation $\leftarrow$ `popFirst(`vList`)`
    prune $\leftarrow$ $false$
    forall d $\in$ `unexpectedDisabledEvents(`violation`)` **do**
        prune $\leftarrow$ prune $\vee$ d $\in \mathcal{E}(\mathcal{C}_i) \vee \mathsf{e}_j \notin \mathrm{validPred}(\mathsf{d})$
    prune $\leftarrow$ prune $\vee$ (`unexpectedEnabledEvents(`violation`)` $\cap\, \Sigma \neq \emptyset$)
    if $\neg$ prune **then** solvableViolations $\leftarrow$ solvableViolations $\cup$ {violation }
if solvableViolations $= \emptyset$ **then return** $\emptyset$
selectedViolation $\leftarrow$ `selectSmallerViolation(`solvableViolations`)`
forall d $\in$ `unexpectedDisabledEvents(`selectedViolation`)` **do** $\prec \leftarrow \prec \cup \{(\mathsf{e}_j, \mathsf{d})\}$
forall e $\in$ `unexpectedEnabledEvents(`selectedViolation`)` **do**
    $\Sigma \leftarrow \Sigma \cup \{\mathsf{e}\}$
    $\prec \leftarrow \prec \cup \{(\mathsf{e}_i, \mathsf{e})\}$
    $\rhd \leftarrow \rhd \cup \{(\mathsf{e}_j, \mathsf{e})\}$
$<_{\mathrm{CS}} \leftarrow (<_{\mathrm{CS}} \cup \prec)^*$
return states(selectedViolation)

to each set of states that has different violation conditions in order to determine if it is possible to arrange the CESAR to include the unexpected behavior by adding new events, causality arcs or disablings. If this is not possible then the only available solution is to prune the path $\mathcal{C}_i \xrightarrow{\mathsf{e_i}} \xrightarrow{\mathsf{e_j}}$ from the GRC. All violations that do not require this pruning (which is considered as the last resort option) are then considered in order to find out which one involves less modifications of the

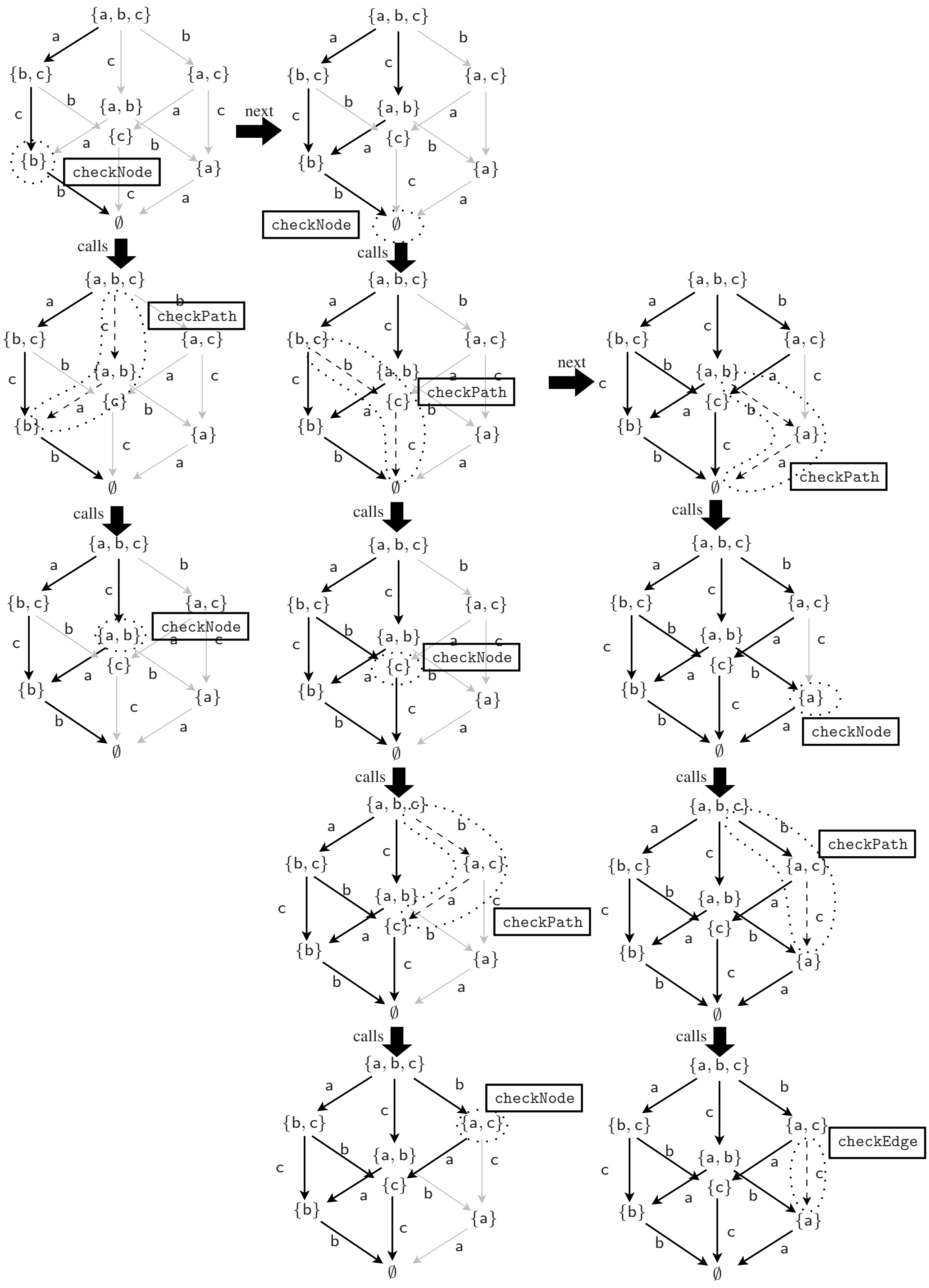


Figure 11.6. Example of execution of the `computeCESARfromTrace` algorithm. Nested calls are shown from top to bottom. Validated zone elements are shown with bold arcs.

CESAR. This election is performed by the auxiliary function `selectSmallerViolation` and then the corresponding fixes are applied to the CESAR (lines 11 to 16).

We will now look carefully to the violation conditions that one can come accross and which type of modification must be introduced to solve the problem. There are two main cases, but they contain several subcases that must be managed differently:

- Unexpected enabling
  - Event was already enabled and was assumed to become disabled, but it was not the case.
  - Event was not enabled but was expected to become enabled later on, however it became enabled sooner than expected.
  - Event was not enabled and was not expected at any point (*i.e.* a new event).
- Unexpected disabling
  - Event was already enabled and was assumed to remain in that state, but it was not the case.
  - Event was not enabled but was expected to become enabled by the firing of e_i, but in fact e_i did not trigger the event.

In the first case, in which an event is already enabled and should become disabled, but the disabling does not happen we are in the situation in which a more complex disabling relation is established between events of the type *event* c *is only disabled when both events* a *and* b *have fired*. Our CESAR model only supports direct disabling relations between events, hence in this case we have no other choice than to prune.

In the second case we have a similar problem: an event becomes enabled sooner than expected which indicates that there is a more complex causality relation between events. Our model supports AND-causality, but if this situation arises it implies the presence of some type of OR-causality which our model does not support. Hence, again, the only possibility is to prune that path. Note that to determine if an event was expected later on an occurrence number has to be assigned to events (following the procedure explained in Definition 5.2) as we are working with distinct events. Then the checking simply reduces to test if the event already belongs to the set of events of the CESAR. This implies that if a particular CESAR contains two events $\langle \mathsf{e}, 1\rangle$ and $\langle \mathsf{e}, 2\rangle$ that correspond to two different repetitions of event e, then depending on the context an unexpected enabling of e can be assigned different occurrence numbers. For instance if ocurrence number 2 is assigned, then we consider that e falls in this case (as $\langle \mathsf{e}, 2\rangle$ already appears the alphabet of events Σ). However should occurrence number 3 have been assigned, then it would be considered a new event and would belong to the next case we will consider.

The third case corresponds to the situation when a new distinct event e is found. This can easily be incorporated into the CESAR, by making $\mathsf{e}_i \prec \mathsf{e}$. However, as this event was not found in $\mathcal{C}_j$ then the only option is that e_j disables e.

These first three cases cover all unexpected enablings. As one can see if the unexpected enabling is produced by some event that belongs already to the alphabet of events (whether if it is actually enabled in $\mathcal{C}_i$ or not, which covers first and second cases) the course of action must be to prune current path (line 7).

Fourth case considers the scenario where an event d is already enabled and has to remain in that state, but becomes disabled by the firing of e_i. If event is enabled in $\mathcal{C}_i$ then in partially computed zone (which contains the original failure trace) is either fired, remains enabled until the end or is disabled by some other event. In all these situations the disabling cannot be modeled by the CESAR because if $\mathsf{e}_i \triangleright \mathsf{d}$ then in all traces of the CESAR it would be disabled, but this is incompatible with the fact that it fires or it remains enabled until the end. In case d is already disabled by some other event, then to add an OR-disabling (that is that two concurrent events disable the same event)

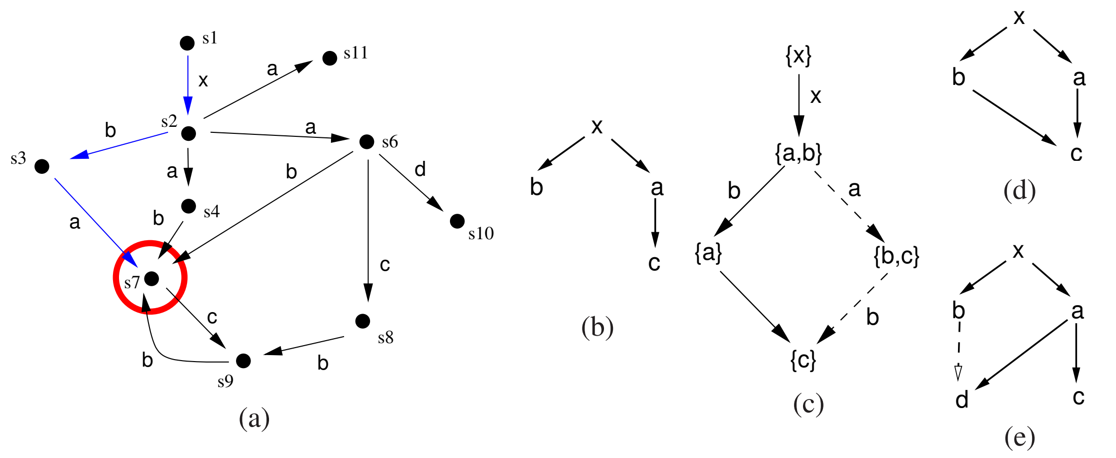


Figure 11.7. Example for `proposeCESARmodifications` function: (a) Untimed state space of a TTS, identical to the TTS of Figure 11.5 (a) except for the fact that state s_5 has been removed. First run to failure found is marked with blue arcs. (b) CESAR initially used (trigger graph) derived from the run. (c) Partially computed zone when `proposeCESARmodification` is called. The dashed path is the one that `checkPath` is currently checking. (d) Modified CESAR that includes state s_4. (e) Modified CESAR that includes state s_6.

violates the CESAR construction conditions of Definition 7.11. Consequently in this situation the only possibility is to prune the path (line 6).

Last case corresponds to the scenario where the event e is expected to become enabled by the firing of e_i, but e_i does not trigger it. Let us name all the elements in the path we are checking as $\mathcal{C}_i \xrightarrow{e_i} \mathcal{C}_k \xrightarrow{e_j} \mathcal{C}_j$. As $e \in \mathcal{E}(\mathcal{C}_j)$ (because e_i was supposed to trigger the event and $e_i \in \mathcal{C}_j$) but $e \notin \mathcal{E}(\mathcal{C}_k)$, then clearly e_j is a trigger of e and a causality arc must be added to the CESAR (line 11). Note that this is only allowed if e_j is a valid causal predecessor of e according to the information previously seen in zone construction (line 6).

We will illustrate how `proposeCESARmodifications` function works in the example of Figure 11.7. Assume that initial trace is $\{x\} \xrightarrow{x} \{a, b\} \xrightarrow{b} \{a\} \xrightarrow{a} \{c\}$ as in Figure 11.5 and that a `CheckNode` has been performed on s_3 and is now currently being performed on s_7. This would activate `CheckPath` on the $\{a, b\} \xrightarrow{a} \{b, c\} \xrightarrow{b} \{c\}$ path. The `CheckEnablings` function would have returned the following violation list:

$$\langle \{s_4\}, \text{unexpected disabling of } c \rangle, \langle \{s_6\}, \text{unexpected enabling of } d \rangle$$

As there is no state in which no violation occurs, then function `proposeCESARmodifications` would have been called. The function would sequentially process the list to see which violations can be solved without pruning (lines 2 to 8). First violation is an unexpected disabling of event c, as $c \notin \{a, b\}$ which does not require a pruning. Second violation is an unexpected enabling of event d which is an event that does not belong to the alphabet of the CESAR, hence pruning can be avoided too. Next step would be to select one of the solvable violations and modify accordingly the CESAR. In terms of arcs introduced in the CESAR, unexpected disablings are more lightweight than unexpected enablings, hence in our case first violation woud have been selected. In subfigure (d) we depict the modified CESAR. Should have been the other modifications selected, the modified CESAR would have been the one shown in subfigure (e).

Note that all the modifications introduced by function `proposeCESARmodifications` satisfy the conditions of Definition 7.10 which guarantee that the resulting CESAR will always be capable of producing the original failure trace. The only remaining element to completely prove that this will actually happen (because Definition 7.10 deals only with CESAs hence the precedences added in case a pruning is required are not considered in the definition) is to check that function `prune` prunes the given path and still respects the original failure trace. In Algorithm 11.6 we show function `prune`.

Algorithm 11.6: prune

input/output : $\mathrm{CS} = \langle \Sigma, \prec, \rhd, <_{\mathrm{CS}} \rangle$ a CESAR.
input : e_i the event that labels the first edge of the path we want to prune.
input : e_j the event that labels the second edge of the path we want to prune.

$<_{\mathrm{CS}} = (<_{\mathrm{CS}} \cup \{(\mathsf{e}_j, \mathsf{e}_i)\})^*$

The algorithm simply adds the precedence that event e_j should happen before e_i. This precedence avoids the path $\xrightarrow{\mathsf{e_i}}\xrightarrow{\mathsf{e_j}}$ for obvious reasons and we will show that it respects the order of the events in the original failure run. Function `prune` is only called in the `CheckPath` procedure and in this last method event e_i is the concurrent event that is being "pushed upwards". This event is the newly introduced event in all the subsequent calls of the `CheckPath` function after the `CheckNode` call of line 17 in the `computeCESARfromTrace` algorithm. As already existing arcs of the zone can only belong to events that are fired earlier in the trace, then to force that any of these events fires before e_j will not violate the firing order of the trace.

Finally, the overall algorithm produces a CESAR that also respects the conditions imposed by timing analysis in Definition 7.11. These conditions are that a disablings cannot be the source or destination of a restriction. The algorithm produces correct CESARs because restrictions are only added between events that have already arcs in the partial zone, which means they fire so they cannot be disabled. Moreover disablings can only be introduced into the CESAR when a new event is found, hence we cannot have neither two conflicting disablings.

Chapter 12

CONVERGENCE

Summary

An interesting question is to determine how “powerful” the approaches based on the enabling compatibility relation can be. In this chapter we discuss the convergence of these methodologies, that is, whether they give a positive or negative answer in a finite number of steps.

We show with examples that there are systems for which they never terminate. We also describe different classes of TTSs for which there is some methodology that terminates, and we provide a hierarchy of these classes. Basically our results show that all incorrect systems are verifiable, and that correct ones are as well if their failure traces satisfy some conditions. In particular that all of them contain a timing inconsistent fragment of a length (after all non-relevant events have been removed) below some finite bound.

Finally we consider the characteristics of a methodology that achieves the best possible degree of theoretical convergence.

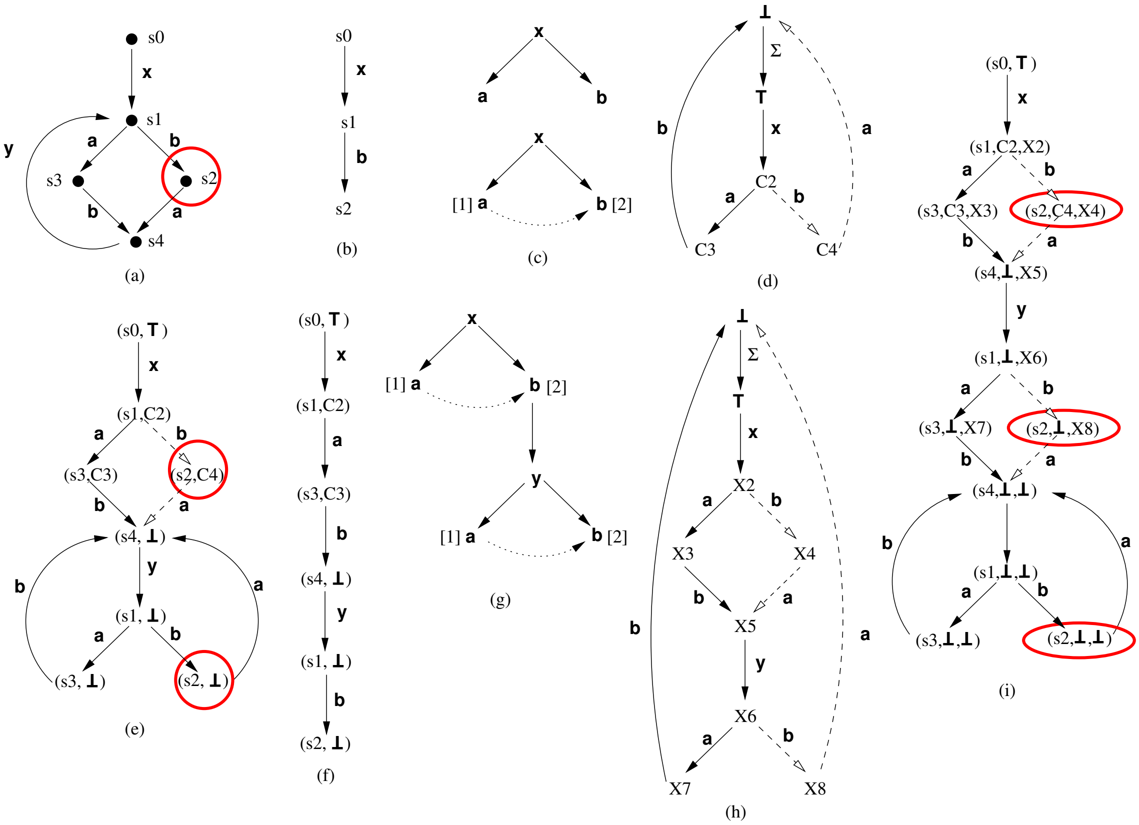


Figure 12.1. Example of divergence of the preliminary verification methodology: (a) TTS that we want to analyze with a failure state in state s_2. Events x and y are unbounded. Delay bounds for remaining events are: $\delta(a) = [1]$ and $\delta(b) = [2]$. (b) Run to failure found in first iteration. (c) Corresponding CESAR and LzCES. (d) Derived ECTRA (simplified depiction). (e) LzTS obtained after composition with the ECTRA. (f) Run to failure state found in second iteration. (g) Corresponding LzCES. (h) Derived ECTRA in the second iteration (simplified depiction). (i) LzTS obtained after composition with the ECTRA of the second iteration.

12.1 Introduction

To introduce the convergence problem we perform several iterations of the preliminary verification methodology on a small example. We will see that the process never stops although the system is clearly correct.

In Figure 12.1 (a) there is a TTS with the following delay bounds: $\delta(a) = [1]$, $\delta(b) = [2]$ and $\delta(x) = \delta(y) = [0, \infty)$. It should be quite clear that state s_2 is not reachable through a valid trace. However the preliminary verification methodology fails to show this simple fact and never terminates. In the first iteration the run to failure state s_2 of (b) is found. Timing consistency of its corresponding trace is checked and shown inconsistent. Then the CESAR is built and timing analyzed giving a LzCES (both appear in (c)). An ECTRA is computed to remove the timing inconsistent traces (subfigure (d)) that is composed with the original system yielding the LzTS of (e). The original trace reaching s_2 is now lazy, but the composition unfolds the system producing a new failure state $(s_2, \perp)$ that is revealed during the reachability analysis at the beginning of the second iteration of the methodology. A new run to failure state is found and the process is repeated again (subfigures (f), (g) and (h)). Again the resulting LzTS after this second composition in subfigure (i) has a replica of the original failure state that is reachable: $(s_2, \perp, \perp)$. This unfolding process goes on infinitely because the entry point (*i.e.* the states mapped to configuration $\top$) to the timing analyzed state space are the states in which event x is enabled. This means that longer traces are not able to “reuse” timing analysis of previous iterations and the unrolling will continue forever.

In [Peñ03] they try to promote the reuse of previous analysis by using a trace different than the full original failure trace. Instead, they use only the minimum suffix of the trace needed to prove timing inconsistency. Moreover they give a description of a subclass of TTSs for which convergence is guaranteed using this approach. In §12.2 we discuss this convergence result.

12.2 Convergence of methods based on ECTRAs and finite trace analysis

There is an inherent difficulty when analyzing the convergence of a family of methods, as each particular method in the family might converge on different kinds of systems. Moreover, the situation can become even more confusing because for a given method the proved convergence result (*i.e.* the class of systems for which the method has been proved to terminate) can be a subclass of the systems for which the method actually converges. In this section we will try to point out the limitations of the family of methods that use composition with ECTRAs to refine the original system. In order to restrict the scope of the discussion we will focus on methods that generate ECTRAs from the analysis of a finite trace (*i.e.* they do not detect loops in the system and extrapolate the current relative timing behavior of the events to infinite unfoldings of the loops) and have a quite definite structure that follows this schema:

- Detect a timing inconsistent failure trace in a finite range of the shortest failure trace.
- Build an ECTRA that removes only timing inconsistent traces including, at least, the original failure trace.
- Compose with the original system.

We call the class of all these methods `ECTRAalg`. More formally a procedure in class `ECTRAalg` is any procedure that satisfies the skeleton of Algorithm 12.1 and fulfills the following conditions:

- All called functions must finish.
- Function `findFailRun` must return a failure run whose length is within a finite bound of the shortest failure run present in the system. If there is no failure run in the system, an empty run is returned. An example of how a verification methodology might not converge if function `findFailRun` does not satisfy these conditions is shown in Appendix C.3.
- Function `isEmpty` returns TRUE if the run passed as a parameter is empty and FALSE otherwise.
- Function `isTimingConsistent` returns TRUE if the trace passed as a parameter is timing consistent and FALSE otherwise.
- Function `buildECTRA` returns an ECTRA that when composed with the system will only remove timing inconsistent traces including, at least, the failure trace passed as a parameter.

All methods following these guidelines will differ only in the policy used to find the failure run (function `findFailRun`) and the one used to select the ECTRA given a particular failure trace (function `buildECTRA`). Note that the schema is simply a specialization of the iterative verification flow shown in Figure 4.7. Strange at it may seem neither the preliminary verification methodology

Algorithm 12.1: ECTRAalg skeleton

input : A a TTS.
output : A timing consistent failure run ρ of A or CORRECT if A does not contain any timing consistent failure trace.

while *true* **do**
  $\rho \leftarrow$ `findFailRun` (A)
  if `isEmpty` (ρ) **then return** CORRECT
  if `isTimingConsistent` (θ_ρ) **then return** ρ
  EC $\leftarrow$ `buildECTRA` (A, θ_ρ)
  $A \leftarrow A \odot$ EC

nor the one presented in [Peñ03] satisfy these requirements for the reasons exposed in Chapter 11. Briefly the problem is that both methods use CESARs to generate the ECTRAs that are not guaranteed to represent accurately the state space of the system and this can lead to the paradoxical situation where the original failure trace is not removed. As both methods are useful in developing the discussion on convergence we will assume in the rest of this section that a phase is added to both methods that guarantees that a useful CESAR is always obtained (an algorithm for this task is given in §11.3).

In the family of methods we are analyzing, one of the less powerful is the preliminary verification methodology. It is not very useful because it always uses the complete failure trace to refine the system. This usually makes the methodology non convergent as seen at the beginning of this chapter, except for very simple systems like systems with only a finite set of traces or that have only loops going to the initial state(s). Most of the examples from previous chapters fall into this class so preliminary methodology works well with them.

12.3 Methods based on splitting at nodal states

In [Peñ03] a different approach is used and only the minimum necessary suffix (needed to prove timing inconsistency) of the failure trace is selected. In this case the convergence of the method is guaranteed for a larger class of systems. In [Peñ03] a first attempt to classify these systems was done in terms of the so-called *nodal states* (see §A.8 for more details). That is states for which the timing consistency of the traces going through these states can be reduced to checking the timing consistency of traces reaching them and the traces leaving them (*i.e.* the problem can be split in two independent parts). We formalize this notion in Definition 12.1.

Definition 12.1 (Nodal state)

Let $A = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$ be a TS. A state $\mathsf{s} \in S$ is called nodal *if any run ρ of A through state s can be decomposed in two fragments $\rho(\rightarrow\mathsf{s})$ and $\rho(\mathsf{s}\rightarrow)$ that correspond to the fragment of ρ reaching s and the fragment of ρ leaving s respectively, such that, given any arbitrary delays to the events in the system, the following statement is satisfied:*

$$\theta_\rho \text{ is timing consistent} \iff \theta_{\rho(\rightarrow\mathsf{s})} \text{ and } \theta_{\rho(\mathsf{s}\rightarrow)} \text{ are timing consistent}$$

■ 12.1

Note that any state without incoming arcs or outgoing arcs is trivially a nodal state. For instance in Figure 12.2 there are four such trivial states: $\mathsf{s}_0, \mathsf{s}_3, \mathsf{s}_5$ and s_8. However there is also one non trivial state: s_4. To show that this state is a nodal state we must check the runs going through it. There

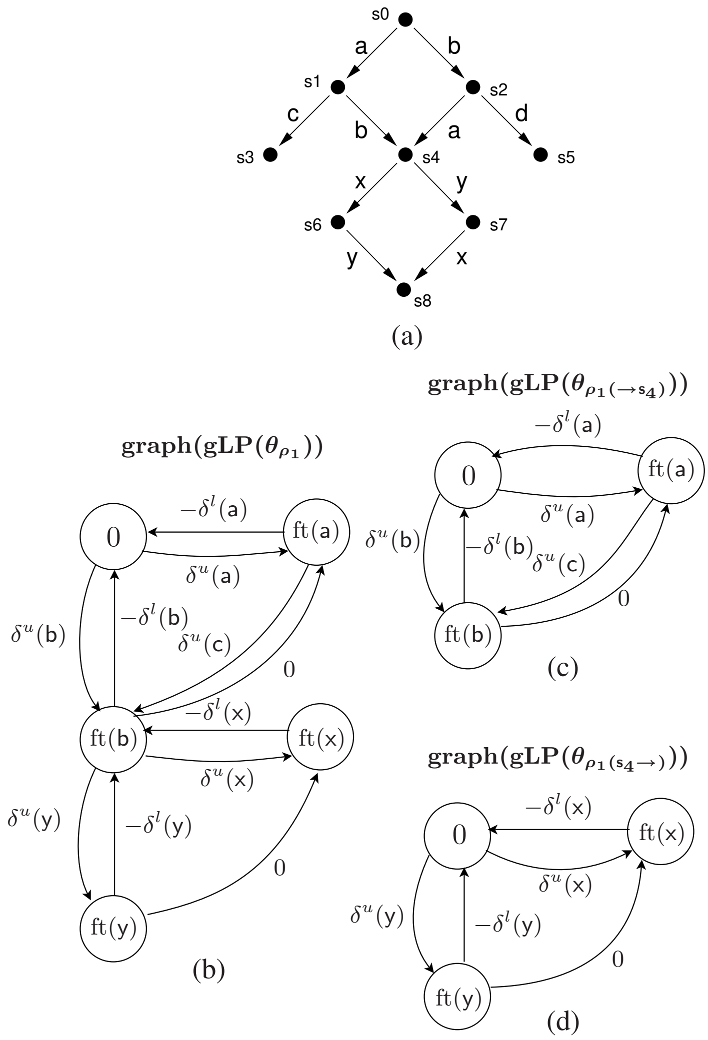


Figure 12.2. Example of a non-trivial nodal state: (a) TS with nodal states in s_0, s_3, s_5, s_8 and s_4. (b) Associated graph of the timing consistency problem of trace $\{a,b\}\xrightarrow{a}\{b,c\}\xrightarrow{b}\{x,y\}\xrightarrow{x}\{y\}\xrightarrow{y}\emptyset$. (c) Associated graph for the part of the trace reaching state s_4. (d) Associated graph for the part of the trace leaving state s_4.

are four runs through state s_4:

$$\rho_1 = s_0\xrightarrow{a}s_1\xrightarrow{b}s_4\xrightarrow{x}s_6\xrightarrow{y}s_8 \qquad \rho_3 = s_0\xrightarrow{b}s_2\xrightarrow{a}s_4\xrightarrow{x}s_6\xrightarrow{y}s_8$$
$$\rho_2 = s_0\xrightarrow{a}s_1\xrightarrow{b}s_4\xrightarrow{y}s_7\xrightarrow{x}s_8 \qquad \rho_4 = s_0\xrightarrow{b}s_2\xrightarrow{a}s_4\xrightarrow{y}s_7\xrightarrow{x}s_8$$

that generate four traces, namely:

$$\theta_{\rho_1} = \{a,b\}\xrightarrow{a}\{b,c\}\xrightarrow{b}\{x,y\}\xrightarrow{x}\{y\}\xrightarrow{y} \quad \theta_{\rho_3} = \{a,b\}\xrightarrow{b}\{a,d\}\xrightarrow{a}\{x,y\}\xrightarrow{x}\{y\}\xrightarrow{y}$$
$$\theta_{\rho_2} = \{a,b\}\xrightarrow{a}\{b,c\}\xrightarrow{b}\{x,y\}\xrightarrow{y}\{x\}\xrightarrow{x} \quad \theta_{\rho_4} = \{a,b\}\xrightarrow{b}\{a,d\}\xrightarrow{a}\{x,y\}\xrightarrow{y}\{x\}\xrightarrow{x}$$

For all those runs ($\forall i : 1 \leq i \leq 4$) it must be true that

$$\theta_{\rho_i} \text{ is timing consistent} \iff \theta_{\rho_i(\to s_4)} \text{ and } \theta_{\rho_i(s_4\to)} \text{ are timing consistent}$$

in order for state s_4 to be nodal. We will show this for one case (run ρ_1) as other cases are very similar. First of all, the condition "θ_{ρ_1} is timing consistent $\Leftrightarrow \theta_{\rho_1(\to s_4)}$ and $\theta_{\rho_1(s_4\to)}$ are timing consistent" is equivalent to say that $\mathrm{gLP}(\theta_{\rho_1})$ and $\mathrm{gLP}(\theta_{\rho_1(\rightarrow s_4)}) \cup \mathrm{gLP}(\theta_{\rho_1(s_4\rightarrow)})$ are mutually timing included. In terms of the graphs associated to the systems of equations this is equivalent to check that, for any representative possible negative cycle in $\mathrm{graph}(\mathrm{gLP}(\theta_{\rho_1}))$ there is a representative possible negative cycle with less weight either in $\mathrm{graph}(\mathrm{gLP}(\theta_{\rho_1(\rightarrow s_4)}))$ or in $\mathrm{graph}(\mathrm{gLP}(\theta_{\rho_1(s_4\rightarrow)}))$ and vice versa.

We have that[1]

$$\mathrm{gLP}(\theta_{\rho_1}) = \{ \quad \begin{array}{ll} \mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{b}) \leq \mathrm{ft}(\mathsf{x}) \leq \mathrm{ft}(\mathsf{y}) & \mathrm{ft}(\mathsf{b}) - \mathrm{ft}(\mathsf{a}) \leq \delta^u(\mathsf{c}) \\ \delta^l(\mathsf{a}) \leq \mathrm{ft}(\mathsf{a}) \leq \delta^u(\mathsf{a}) & \delta^l(\mathsf{x}) \leq \mathrm{ft}(\mathsf{x}) - \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{x}) \\ \delta^l(\mathsf{b}) \leq \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{b}) & \delta^l(\mathsf{y}) \leq \mathrm{ft}(\mathsf{y}) - \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{y}) \quad \} \end{array}$$

and run fragments are $\rho_1(\rightarrow s_4) = s_0 \xrightarrow{\mathsf{a}} s_1 \xrightarrow{\mathsf{b}} s_4$ and $\rho_1(s_4\rightarrow) = s_4 \xrightarrow{\mathsf{x}} s_6 \xrightarrow{\mathsf{y}} s_8$ which create the following systems of equations:

$$\mathrm{gLP}(\theta_{\rho_1(\rightarrow s_4)}) = \{ \quad \begin{array}{l} \mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{b}) \\ \delta^l(\mathsf{a}) \leq \mathrm{ft}(\mathsf{a}) \leq \delta^u(\mathsf{a}) \\ \delta^l(\mathsf{b}) \leq \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{b}) \\ \mathrm{ft}(\mathsf{b}) - \mathrm{ft}(\mathsf{a}) \leq \delta^u(\mathsf{c}) \quad \} \end{array} \qquad \mathrm{gLP}(\theta_{\rho_1(s_4\rightarrow)}) = \{ \quad \begin{array}{l} \mathrm{ft}(\mathsf{x}) \leq \mathrm{ft}(\mathsf{y}) \\ \delta^l(\mathsf{x}) \leq \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{x}) \\ \delta^l(\mathsf{y}) \leq \mathrm{ft}(\mathsf{y}) \leq \delta^u(\mathsf{y}) \quad \} \end{array}$$

The associated graphs can be seen in Figures 12.2 (b,c,d). It is easy to check that subfigure (c) and (d) are subgraphs of (b), and that (b) does not contain any representative possible negative cycle (as defined in Definition 5.13) that contains arcs of both subgraphs, hence all representative possible negative cycles of (b) belong either to (c) or (d) and vice versa.

Nodal states are important because they can be used to split a complex problem in a set of easier subproblems. The question is to obtain a description of nodal states that allows to easily check if a state of a trace is nodal or not. We give such a condition in Lemma 12.1. Note that this condition is weaker than the original one in [Peñ03] (see §A.8) hence more (non-trivial) nodal states might be found.

LEMMA 12.1 (CONDITION FOR NODAL STATE)
Let $A = \langle S, \Sigma, T, s_0 \rangle$ *be a TS. A state* $s \in S$ *is* nodal *(according to Definition 12.1) if* $\forall s' \in \mathrm{Reach}(s_0, T) : s' \xrightarrow{e} s \in T$, $\mathcal{E}(s') \cap \mathcal{E}(s) = \emptyset$. ■ 12.1

Proof: *if* $\forall s' \in \mathrm{Reach}(s_0, T) : s' \xrightarrow{e} s \in T$, $\mathcal{E}(s') \cap \mathcal{E}(s) = \emptyset$*, then any trace reaching* s *will have an enabling in which all previously enabled events are disabled or fired. In terms of the associated graph this creates a node such that there is no arc from previous nodes to the next nodes, thus the graph can be partitioned in two subgraphs as there is no representative possible negative cycle shared between both parts. Consequently timing consistency checking can be performed safely in the subgraphs instead of the whole graph. As this happens for all trace reaching state* s*, the state is nodal.*

As an example, state s_4 in Figure 12.2 satisfies the conditions required by Lemma 12.1.

In [Peñ03] the original trace was split by its nodal states. Then, starting by the last fragment, each fragment was checked to see if it was timing inconsistent. The first timing inconsistent fragment found was then used by inspecting incrementally its suffixes until the minimal suffix to prove timing inconsistency was found. Using this strategy the convergence of the method is guaranteed for all systems that satisfy the following definition.

[1] According to section *Conservative approximations* of §5.3.2, events a and b should have been considered conservative because they are enabled in the initial state, so that $\delta^l(\mathsf{a}) = \delta^l(\mathsf{b}) = 0$. However, in this example we will not consider them as conservative to make the example more illustrative without having to introduce an additional startup event.

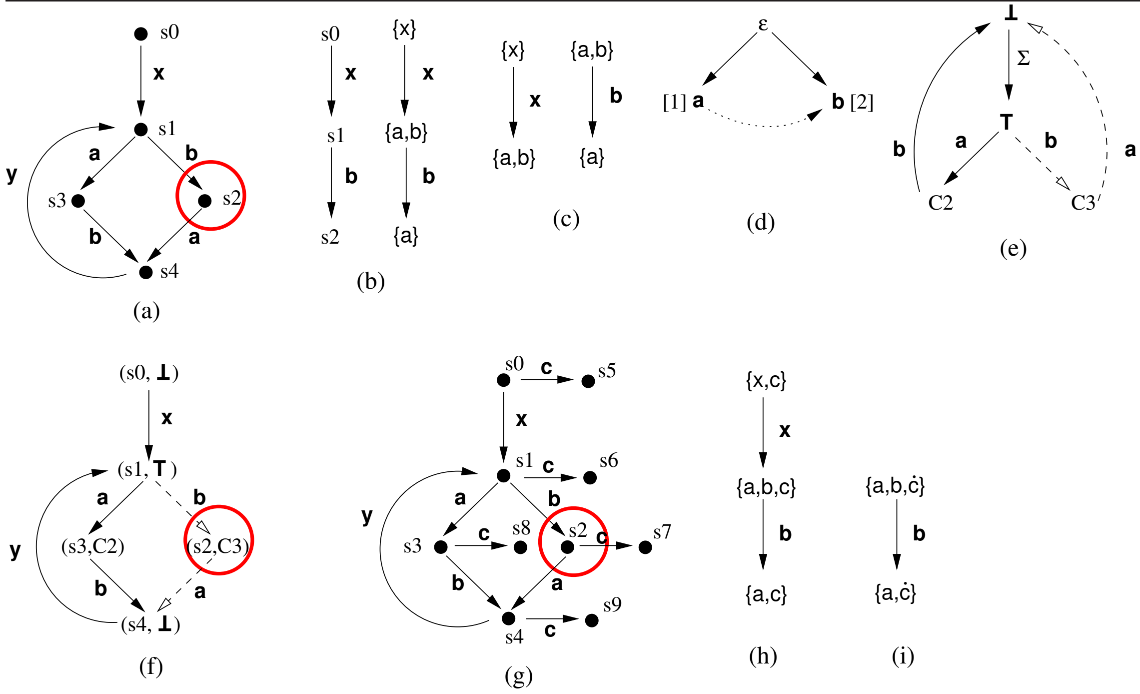


Figure 12.3. Example of verifiable systems with and without non-trivial nodal states: (a) TTS of Figure 12.1 (a). (b) Run to failure state and its corresponding trace. It contains a nodal state. (c) Two fragments obtained after splitting by the nodal point. (d) LzCES obtained from the second trace fragment. (e) Computed ECTRA from LzCES (simplified depiction). (f) LzTS after composition with the ECTRA. Failure state is not reachable through a valid traces, hence the system is proved correct. (g) A similar TTS (events have the same delays as in subfigure (a)) in which an additional concurrent event is added. Although there is no "global" nodal point, the system can be shown correct using the same ECTRA as before, hence convergence criterion can be enlarged if the methodology is adapted. (h) Failure trace. (i) Its minimal fragment showing timing inconsistency. Using it to generate the ECTRA allows to complete the verification task.

DEFINITION 12.2 (STRONGLY SYNCHRONIZED **TS**S)

Let $A = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$ *be a* TS*. We will say that* A *is* strongly synchronized *if there is a nodal state in all cycles in* $\mathrm{Reach}(\mathsf{s}_0, T)$. ■ 12.2

For example, in the TS of Figure 5.2 of page 65, states s_0, s_1 and s_{13} are nodal. All cycles in it contain the fragment $\mathsf{s}_{13} \xrightarrow{\mathsf{y}} \mathsf{s}_0 \xrightarrow{\mathsf{x}} \mathsf{s}_1$, hence all cycles must go through all those nodal states and, consequently, the TS is strongly synchronized. Let name the class of strongly synchronized systems as SSTS. Note that all systems without cycles belong to this class. In general for all systems in SSTS any method that uses nodal points to split the analysis will converge[2]. In this way, the example by which the convergence problem was introduced can be verified without problems as shown in Figure 12.3 (a-f).

There is an immediate improvement of this convergence result as next lemma states.

LEMMA 12.2

Algorithms in `ECTRAalg` *that split the analysis using nodal states will converge on:*

[2] A formal proof of this result with only minor variations can be found in proof of Lemma 12.2

- *All systems which are incorrect.*
- *Correct systems in which only all cycles appearing in traces reaching fail arcs/states also contain a nodal state.*

■ 12.2

Proof: *If system is incorrect, even if the system contains inexistent fails for which an infinite unfolding will be performed, the unfolding always increases the distance of the fail from the initial state. This means that if the method used to return the counterexample always returns the shortest counterexample (or a counterexample in a finite range from the shortest one), then it is guaranteed that the timing consistent failure trace will eventually be found and the methodology will converge. More formally let j be the length of the shortest failure trace in the system and let k be the length of the shortest timing consistent failure trace. Clearly $j \leq k$. Now assume that the search failure trace method gives always a trace whose length is in the interval $[j, j+m]$ for a finite m. As there is a finite number of traces of length $[j, j+m]$ all the traces in the set will eventually either be found by the search method or removed because they are enabling compatible with some timing inconsistent trace. As no trace of greatest length will be found until all traces of length j have been processed any timing consistent trace of length j will eventually be found. If no such trace exists then the interval will move up to the next length j' of the current shortest trace. This means that at some point $k \in [j', j'+m]$ and the trace timing consistent trace of length k will eventually be found. Note that this line of reasoning will work for any method in* `ECTRAalg` *as it does not depend on nodal states.*

*If the system is correct, then take a trace θ to a failure. If θ does not contain any repeated state (*i.e. *has no cycles) then the number of these traces is finite because the underlying* TS *is finite too. As composition must remove at least the original failure trace consequently only a finite number of compositions will be required to remove all these traces. On the other hand if θ contains repeated states, then we have unfolded cycles. As lemma obliges that all cycles of traces reaching a failure state/arc (as θ) contain a nodal state, the analysis can be split in two parts: one using the trace fragment up to the nodal state and one using the trace fragment departing from that state. As this procedure can be done for each one of the cycles present in the trace, the analysis can be reduced to a collection of fragments all of them containing no repeated state, that we already know are finite in number. Thus any algorithm splitting at nodal points will converge when analyzing such systems.*

As there cannot be more failure arcs and fail states than arcs and states it is clear that this class of systems (name them *failure strongly synchronized* and denote by FSSTS its class) is a superset of the former, that is FSSTS $\supset$ SSTS. Note that to determine if a TTS belongs to class FSSTS it is needed to take into account the timing information, while for class SSTS this information is not used. This explains why class SSTS is formed by TSs (which can be considered a particular case of TTS in which all events are unbounded) while FSSTS is formed by TTSs. The use of timing information is precisely what allows that FSSTS $\supset$ SSTS.

The methodology of [Peñ03] verifies systems in FSSTS while the preliminary verification methodology does not. However if the latter is extended to split at nodal points it becomes capable of verifying them.

12.4 Beyond nodal states

Although class FSSTS represents an improvement over class SSTS many systems that one can intuitively analyze do not belong to the former. For instance in Figure 12.3 (g) we can a see a TTS that is equal to the one in subfigure (a) but in which an additional concurrent event c is enabled. Due to the nodal state definition, it is clear that now state s_1 is not nodal, hence for instance the preliminary verification methodology extended to split at nodal points will never converge. In state s_1 there is a failure arc (because arc labeled with event b reaches failure state s_2) and two arcs that can avoid the failure (corresponding to events a and c). It should be clear that using the same ECTRA as in the previous example (subfigure (e)) the fail can be shown inexistent in the timed domain. Now there is a gap between the convergence criterion used up to this point (class FSSTS) and some naive examples.

In order to reason more easily on the subject we formalize the convergence problem. Formally, our algorithms try to determine if $\mathcal{L}(A) \cap \overline{\mathcal{L}(P)} = \emptyset$ or not (see §5.3.1) where A is a TTS $\langle A^-, \delta^l, \delta^u \rangle$ and P is a set of safety properties. To check this, a series of LzTSs A_i that are approximations of A are produced, such that $A_0 = A^-$ and for $i \geq 1$, $A_i = A_{i-1} \odot \mathrm{EC}_i$ where EC_i is the ECTRA used at iteration i. Now the algorithm converges, if and only if, for a finite i either $\exists \theta \in \mathcal{L}(A_i) \cap \overline{\mathcal{L}(P)}$ such that θ is timing consistent, in which case system A is incorrect, or $\mathcal{L}(A_i) \cap \overline{\mathcal{L}(P)} = \emptyset$, in which case the system is correct.

For the cases where A fails, it is guaranteed that there is a finite i that satisfies the above mentioned conditions for the reasons exposed in the proof of Lemma 12.2. For the case where A is correct we have that $\mathcal{L}(A_i) \cap \overline{\mathcal{L}(P)} = \emptyset$. By Theorem 10.1 we know that $\mathcal{L}(A_i) = \mathcal{L}(A_{i-1} \odot \mathrm{EC}_i) = \mathcal{L}(A_{i-1} \setminus \mathsf{EnComp}(\mathcal{L}(A_{i-1}), \Theta_i)$ where Θ_i is the set of lazy traces of EC_i. Expanding this expression until A_0 is reached be obtain $\mathcal{L}(A_i) = \mathcal{L}(A_0) \setminus \bigcup_{0<j\leq i} \mathsf{EnComp}(\mathcal{L}(A_{j-1}), \Theta_j)$ which can also be written as $\mathcal{L}(A_i) = \mathcal{L}(A_0) \setminus \bigcup_{0<j\leq i} \mathsf{EnComp}(\mathcal{L}(A_0), \Theta_j)$ due to the fact that although $\mathsf{EnComp}(\mathcal{L}(A_0), \Theta_j) \supseteq \mathsf{EnComp}(\mathcal{L}(A_{j-1}), \Theta_j)$ the additional traces must be traces that have been removed previously so there is no danger in removing them more than once.

At this point $\mathcal{L}(A_i) \cap \overline{\mathcal{L}(P)} = \emptyset$ converts to $\Big(\mathcal{L}(A_0) \setminus \bigcup_{0<j\leq i} \mathsf{EnComp}(\mathcal{L}(A_0), \Theta_j)\Big) \cap \overline{\mathcal{L}(P)} = \emptyset$ which is the same as $\mathcal{L}(A_0) \cap \overline{\mathcal{L}(P)} \subseteq \bigcup_{0<j\leq i} \mathsf{EnComp}(\mathcal{L}(A_0), \Theta_j)$. As

$$\bigcup_{0<j\leq i} \mathsf{EnComp}(\mathcal{L}(A_0), \Theta_j) = \mathsf{EnComp}(\mathcal{L}(A_0), \bigcup_{0<j\leq i} \Theta_j)$$

this simply states that all failure traces of the system are enabling compatible with some proved timing inconsistent trace of finite set $\bigcup_{0<j\leq i} \Theta_j$.

For systems without Zeno runs it can be guaranteed that set $\bigcup_{0<j\leq i} \Theta_j$ is finite (this is equivalent to say that i is finite, as our definition of the Θ_j sets in equation (10.1) makes them always finite) if all failure traces have a finite timing inconsistent fragment as next lemma states.

LEMMA 12.3

Given a TTS A, there is an algorithm in `ECTRAalg` such that converges analyzing A if all traces in $\mathcal{L}(A_i) \cap \overline{\mathcal{L}(P)}$ have a timing inconsistent fragment of length less or equal to finite fixed bound k. ■ 12.3

Proof: *Take the algorithm in `ECTRAalg` such that, when a failure trace θ is found, constructs an ECTRA EC_i whose set of lazy traces (noted as Θ_i) includes one of the timing inconsistent fragments of θ of length less or equal to k. As fragments of bounded length are finite in number, the number of sets Θ_i will also be, as well as the number of iterations of the method.*

We say that a TTS that is either incorrect or satisfies the conditions of the lemma is a *bounded inconsistent fragments transition system* and we denote the class of these systems as BIFTS. Clearly BIFTS $\supset$ FSSTS as, if any loop of a trace reaching a failure has a nodal state, then the

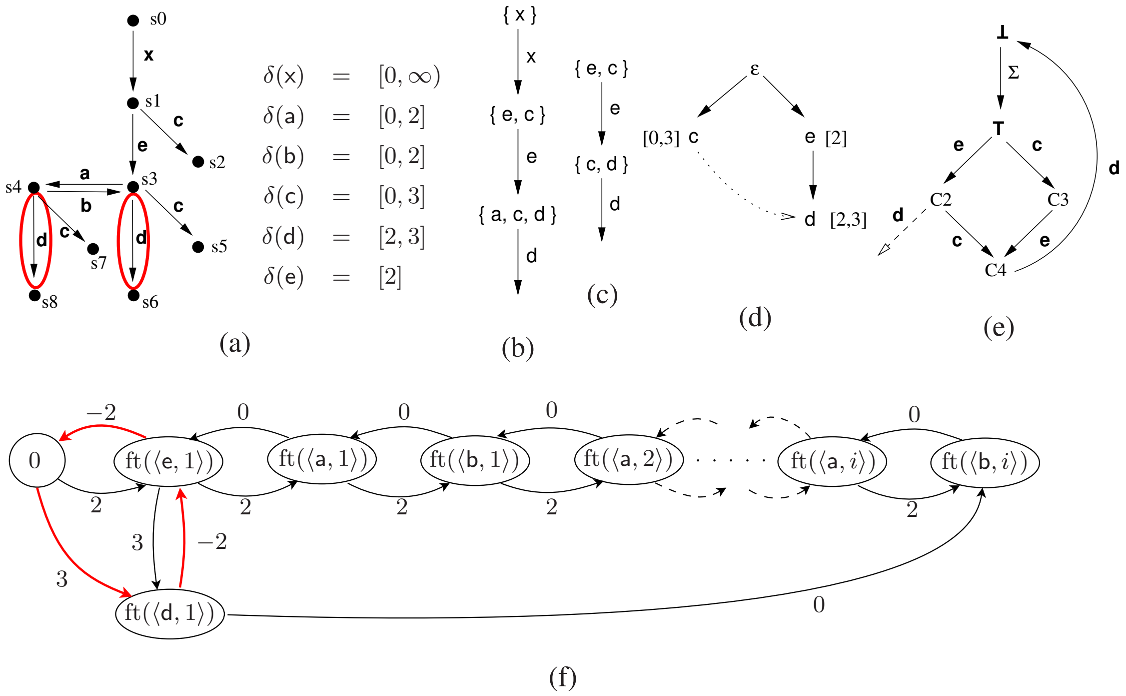


Figure 12.4. Example of verifiable system with an arbitrarily large timing inconsistent trace fragment: (a) TTS that contains Zeno runs that is verifiable by algorithms in ECTRAalg but does not satisfy the convergence conditions of Lemma 12.3 as contains traces with arbitrarily large minimal timing inconsistent fragments. (b) One of the possible failure traces that could have been found at first iteration. (c) Its timing inconsistent reduced version. (d) LzCES obtained after timing analysis of the trigger graph of the reduced trace. (e) Simplified depiction of the ECTRA derived from the LzCES. (f) Graph associated to the timing consistency problem of the trace $\theta = \{\mathsf{c}, \mathsf{e}\} \xrightarrow{\mathsf{e}} (\{\mathsf{a}, \mathsf{c}, \mathsf{d}\} \xrightarrow{\mathsf{a}} \{\mathsf{b}, \mathsf{c}, \mathsf{d}\} \xrightarrow{\mathsf{b}})^i \{\mathsf{a}, \mathsf{c}, \mathsf{d}\} \xrightarrow{\mathsf{d}}$. An arbitrary number of repetitions of the $\xrightarrow{\mathsf{a}}\xrightarrow{\mathsf{b}}$ sequence does not have any effect on the number of negative cycles present in the graph.

analysis can be split in several finite trace fragments (as shown in proof of Lemma 12.2). As system is correct then at least one of these fragments is timing inconsistent, hence there is a timing inconsistent fragment of bounded length for each trace reaching a failure. Note that in system of Figure 12.3 (g) all failure traces have a minimal timing inconsistent fragment of length one, depicted in subfigure (i), and that using this fragment to generate the ECTRA allows to verify the system, hence the system belongs to class BIFTS.

A particular case of Lemma 12.3 is when all timing inconsistent fragments do not have any repeated state, as then the *recurrence diameter* (*i.e.* the maximum possible distance between any state and its repetition in a trace) of the system is the bound of the timing inconsistent fragments. This was the case, for instance of system of Figure 12.3 (g). However there are systems that are verifiable in which repetitions might appear in the timing inconsistent fragments. In some cases even with an arbitrary number of repetitions. An illustrative example of this are some TTSs that contain Zeno runs. Although when TTSs were introduced in Chapter 2 we reduced the scope of our discussion to non-Zeno TTSs, at this particular point these kind of systems are very useful to show the limitations of Lemma 12.3 and to focus on relevant aspects of the discussion.

First thing we will do is to present a TTS that contains Zeno runs that can be verifiable even if the conditions stated in the lemma are not satisfied.

EXAMPLE 12.1 *In Figure 12.4 (a) we can see a* TTS *that contains Zeno runs in which the firing of event* d *is defined as a failure. The system contains traces that have an arbitrarily large minimal timing inconsistent fragment. This can be proved as follows:*

Given any failure trace ending with $\xrightarrow{\mathsf{d}}$*, it is only possible to prove timing inconsistency if a fragment starting at state* s_1 *is used. If a fragment starting at states* s_3 *or* s_4 *is used, then* d *can fire before or after events* a, b *and* c *due to the given delays and the fact that the exact separation time between* c *and* d *is not known as both event are already enabled at states* s_3 *and* s_4.

This implies that runs of the form $\mathsf{s}_0\xrightarrow{\mathsf{x}}\mathsf{s}_1\xrightarrow{\mathsf{e}}(\mathsf{s}_3\xrightarrow{\mathsf{a}}\mathsf{s}_4\xrightarrow{\mathsf{b}})^*\mathsf{s}_3\xrightarrow{\mathsf{d}}$ *(using regular expression notation) have a minimal timing inconsistent fragment* $\mathsf{s}_1\xrightarrow{\mathsf{e}}(\mathsf{s}_3\xrightarrow{\mathsf{a}}\mathsf{s}_4\xrightarrow{\mathsf{b}})^*\mathsf{s}_3\xrightarrow{\mathsf{d}}$ *which can be made infinitely large, hence it is unbounded. However, the* ECTRA *of subfigure (d), which can be obtained from a failure trace (subfigure (b)) by discarding irrelevant events to the timing consistency (obtaining the reduced trace of subfigure (c)), can be used to prove the correctness of the system.*

■ 12.1

As previous example illustrates, this notion of *discarding irrelevant events to the timing consistency* can be used to improve the convergence results, but first of all we need to formalize it.

Given a timing inconsistent trace θ, its inconsistency might be due to several negative simple cycles in the associated graph to the set of inequalities that test its timing consistency. Each one of these negative cycles (in the rest of this section we assume all cycles to be simple except when otherwise stated) identifies a set of sufficient inequalities to prove that no solution is possible. For instance, in our example, all trace fragments of the form

$$\theta = \{\mathsf{c},\mathsf{e}\}\xrightarrow{\mathsf{e}}(\{\mathsf{a},\mathsf{c},\mathsf{d}\}\xrightarrow{\mathsf{a}}\{\mathsf{b},\mathsf{c},\mathsf{d}\}\xrightarrow{\mathsf{b}})^*\{\mathsf{a},\mathsf{c},\mathsf{d}\}\xrightarrow{\mathsf{d}} \tag{12.1}$$

are inconsistent because event c should have fired before event d. This can be seen in the associated graph of Figure 12.4 (f) where there is a negative cycle $0\xrightarrow{3}\mathrm{ft}(\langle\mathsf{d},1\rangle)\xrightarrow{-2}\mathrm{ft}(\langle\mathsf{e},1\rangle)\xrightarrow{-2}0$. This cycle corresponds to cycle $0\xrightarrow{\delta^u(\mathsf{c})}\mathrm{ft}(\langle\mathsf{d},1\rangle)\xrightarrow{-\delta^l(\mathsf{d})}\mathrm{ft}(\langle\mathsf{e},1\rangle)\xrightarrow{-\delta^l(\mathsf{e})}0$ in $\mathrm{graph}(\mathrm{gLP}(\theta))$.

Given a particular negative cycle, it is possible to construct a trace that contains that cycle and to whom original trace θ is enabling compatible. We are interested in obtaining the shortest traces among these ones, that we call *reduced traces*.

DEFINITION 12.3 (TIMING-INCONSISTENT REDUCED VERSION OF A TRACE)

Let θ *be a timing inconsistent finite trace, and* c *one of the negative cycles of its associated graph. The* timing-inconsistent reduced version of θ for c*, denoted as* $\theta[c]$*, is the trace obtained by applying transformations of Lemma 5.8 to reduce as much as possible the number of nodes in its associated graph, while preserving cycle* c.

We say that $\theta[c]$ *is minimal if there is no other negative cycle* c' *in* G *such that* $\theta[c']$ *is shorter than* $\theta[c]$.

■ 12.3

There are only two transformations that can remove nodes from the associated graph: prefix removal and sink event removal. Given a negative cycle, the reduced version of a trace is obtained by removing all fired events until the first fired event in the cycle, and then removing all events that belong to concurrent branches or are causal successors of events involved in the cycle but do not appear in it –by iteratively using the *sink event removal* transformation while possible, sometimes using additional transformations to be able to apply the former transformation. Prefix removal might include the first fired event in the cycle, if the event does not appear in any of the arcs of the cycle. This is because is such case it only works as a "startup" event, a function that can be performed also by special node 0. Moreover, we can use the *promote to conservative events* transformation on all the events in which the lower bound does not appear in the arcs of the cycle.

In our example, using the negative cycle in Figure 12.4 (f) to reduce any trace of the form shown in (12.1), we do obtain the trace in subfigure (c) by repeteadly removing sink events. We start with the last occurrence of event b, however, the transformation requires that removed event triggers no other event, and last a is triggered by it. Thus, we use the *advance disabling* transformation to remove the enabling of this last event a, and then proceed with the removal of b. In this way all the occurrences of a and b can be completely supressed.

The previous procedure to obtain $\theta[c]$ given a negative cycle c can be also summarized by defining which sets of events are preserved. Let Σ_c be the set of events that appears in cycle c, either in the nodes or the arcs, we define $\mathsf{e}_{s'}$ to be the smallest element in Σ_c according to the $<_{\mathsf{fo}}$ relation. That is $\mathsf{e}_{s'}$ is the first event fired among the events in Σ_c. If cycle does not contain $\mathsf{e}_{s'}$ in its arcs, or its node is not the destination of a 0-valued arc, then we define $\mathsf{e}_s = \mathsf{e}_{s'}$, otherwise we define e_s as the event that fires before $\mathsf{e}_{s'}$ in the trace. Now e_s is the last fired event that can be removed by prefix removal transformation. Then for each event e in Σ_c we define the set of preserved events of trace θ as

$$\mathrm{preserveEv}(\mathsf{e}) = \{\mathsf{e}' \mid \mathsf{e}_s <_{\mathsf{fo}} \mathsf{e}' \leq_{\mathsf{tr}} \mathsf{e}\}. \tag{12.2}$$

So, for a cycle c, the complete set of preserved events is $\mathrm{preserveEv}(c) = \bigcup_{\forall \mathsf{e} \in \Sigma_c} \mathrm{preserveEv}(\mathsf{e})$, and the set of events promoted to conservative is $\mathrm{consEv}(c) = \{\mathsf{e} \mid \delta^l(\mathsf{e}) \text{ does not appear in } c\}$.

Minimal timing-inconsistent reduced versions of traces allow to improve the convergence results of Lemma 12.3.

LEMMA 12.4
Given a TTS A, there is an algorithm in `ECTRAalg` such that converges analyzing A if all traces in $\mathcal{L}(A_i) \cap \overline{\mathcal{L}(P)}$ have a timing inconsistent fragment whose minimal timing-inconsistent reduced version is of length less or equal to finite fixed bound k. ■ 12.4

Proof: *Take the algorithm in `ECTRAalg` such that, when a failure trace θ is found, constructs an ECTRA EC_i whose set of lazy traces (noted as Θ_i) includes the timing-inconsistent reduced version θ' of one of the minimal timing inconsistent fragments of θ and θ' has a length less or equal to k. As fragments of bounded length are finite in number, the number of sets Θ_i will also be, as well as the number of iterations of the method.*

As before we say that a TTS that is either incorrect or that satisfies the conditions of this lemma is a *bounded reduced inconsistent fragments transition system* and we denote the class of all such TTSs as BRIFTS. Again, this class encompasses all previous classes as BRIFTS $\supset$ BIFTS.

In order to establish convergence results in terms of negative cycles in traces, we define our last class of TTSs based on this parameter and we show the equivalence of this class with BRIFTS for non-Zeno TTSs.

DEFINITION 12.4 (BOUNDED NEGATIVE CYCLES TRANSITION SYSTEMS)
Class BNCTS is formed by all TTSs that are either incorrect or that all its failure traces have a negative cycle of finite bounded length in the graph associated to its timing consistency problem. ■ 12.4

In previous definitions we have not explicitely formalized which graph checks the timing consistency of a trace. We have simply used the term *the graph associated to trace θ*. This indefinition is not casual. Up to this point, the only graph we have seen associated to a trace θ, is $\mathrm{graph}(\mathrm{LP}(\theta))$. However, any other graph associated to a SPTE $\mathrm{LP}'(\theta)$ such that $\mathrm{sol}(\mathrm{LP}'(\theta)) = \mathrm{sol}(\mathrm{LP}(\theta))$ would work. On the other hand, not all these graphs have the same negative cycles, which affects Definitions 12.3 and 12.4.

Nevertheless, Lemma 12.4 is valid regardless of the graph selected, as long as it correctly tests the timing consistency of θ. However, to prove stronger results (see Theorem 12.6) additional constraints must be imposed. In particular the graphs must contain all the negative cycles of the traces to whom θ is enabling compatible. We discuss this issue in §12.6 but from this point on, we assume the BRIFTS and BNCTS classes are defined using this special type of graph that we call *saturated*.

LEMMA 12.5

If we consider all types of TTSs, including Zeno TTSs, then BRIFTS $\subsetneq$ BNCTS. If we restrict to non-Zeno TTSs, then BIFTS $=$ BRIFTS $=$ BNCTS. ■ 12.5

Proof: *Incorrect systems belong to both classes by definition, so we only have to check correct systems. If we consider all types of TTSs, including Zeno TTSs, then, for correct systems, class BRIFTS requires that all their failure traces have a minimal timing inconsistent reduced version of finite bounded length. Class BNCTS requires that these traces have a negative cycle of finite bounded length. Clearly, any bounded minimal timing inconsistent reduced version of a trace has a bounded negative cycle. Let k be the maximum number of fired events of all the minimal timing inconsistent failure traces, then k is a bound for at least one of the negative cycles of all failure traces. Consequently, BRIFTS $\subseteq$ BNCTS. A similar argument allows to prove that BIFTS $\subseteq$ BNCTS.*

In this general case, moreover, we can prove that BRIFTS $\neq$ BNCTS with a counterexample TTS, that we show in Appendix B.12. Hence BRIFTS $\subsetneq$ BNCTS.

If we restrict to non-Zeno TTSs, we can prove that BNCTS $\subseteq$ BRIFTS with the following argument. In this type of systems, given a finite interval, only a finite number of events can be fired. Given a TTS A in BNCTS, let C be the set of all bounded cycles in A. For each cycle $c \in C$, compute absolute value of the sum of all negative weights in c. Let w be the maximum value among these absolute values. This is a finite number since the number of elements in the cycle is bounded and all negative values come from lower delay bounds, which are also finite. Let k be the maximum number of events that can be fired in w units of time. Then k is a bound for the timing inconsistent reduced version of any timing inconsistent trace of A. Exactly the same argument can be used to prove BNCTS $\subseteq$ BIFTS.

In next section we show an example of a correct system that does not satisfy the conditions of Lemma 12.4 (*i.e.* a system outside class BRIFTS) and is not verifiable by any method in `ECTRAalg`. Moreover, we prove that the violation of the conditions of the lemma on correct systems will always carry the same consequences, hence that class BRIFTS contains only all systems verifiable using the tools developed in this work.

12.5 Divergent systems

In previous section we expanded, step by step, the set of systems for which convergence is guaranteed for some algorithm in `ECTRAalg`. Namely, we proved SSTS $\subset$ FSSTS $\subset$ BIFTS $\subset$ BRIFTS $\subset$ BNCTS. Moreover, if NZTS is the class of non-Zeno TTSs, we showed that BIFTS $\cap$ NZTS $=$ BRIFTS $\cap$ NZTS $=$ BNCTS $\cap$ NZTS (Figure 12.5). However, the convergence criterion cannot be indefinitely enlarged because there are systems for which all relevant

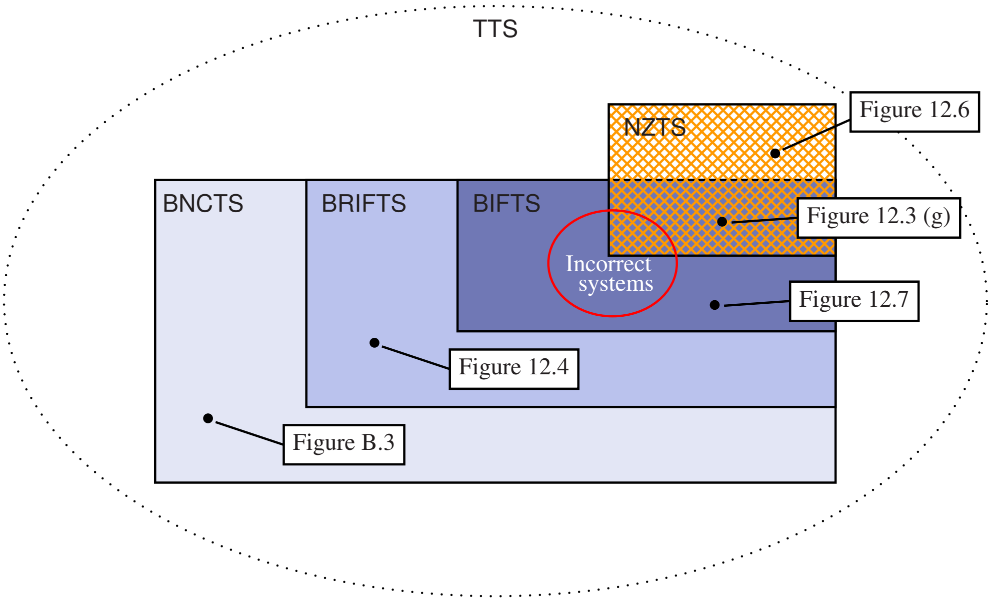


Figure 12.5. Class hierarchy for TTSs. Some examples of TTSs that appear in this work are classified accordingly.

events to a particular failure are inherently not synchronized, hence any approach based on the analysis of finite fragments of traces will inevitably not finish.

For instance, in Figure 12.6 (a), there is a TTS in which a failure arc ($s_3 \xrightarrow{a} s_4$) has been declared. Due to the delays of the events such arc can never be taken (timed state space is shown in subfigure (b)), however, to show that any trace containing it is timing inconsistent, one has to go back up to state s_1, as events a and b are completely decoupled from that point on. The only chance to show that b fires before a is to check the timing from the point in which they were synchronized, otherwise there is not enough information. Consequently, any ECTRA enforcing that b fires before a must have an entry point in state s_1. This means that there will be an infinite unfolding because events c and d create loops to state s_3, which is passed the unique possible entry point of the ECTRA.

Note that this limitation applies to any method using ECTRAs and timing analysis on finite traces. It is possible to build an ECTRA such that when composed with the original system we do obtain the LzTS of Figure 12.6 (b), but it would require to perform a timing analysis in which the loop is taken into account in order to guarantee that all removed traces are timing inconsistent. Hence the timing analysis would be over an infinite set of traces. Alternatively, one could solve the problem by resorting to timed automata alike composition methods instead of using ECTRAs, as these methods are known to be convergent even if only a finite trace is used in timing analysis [AIKY92].

The system of Figure 12.6 (a) does not satisfy the conditions of Lemma 12.4 as given a bound k there is always a trace whose minimal timing inconsistent reduced fragment is larger than k. In particular an arbitrarily large trace of this type can be generated by word $\mathsf{xe(badc)^*a}$ (using regular expression notation) and its shortest timing inconsistent fragment is formed by $\{e,b\} \xrightarrow{e} \{a,b\}(\xrightarrow{b} \{a,d\} \xrightarrow{a} \{c,d\} \xrightarrow{d} \{c,b\} \xrightarrow{c} \{a,b\})^* \xrightarrow{a}$ which coincides with its minimal timing inconsistent reduced version. Because this system belongs to class NZTS, due to the

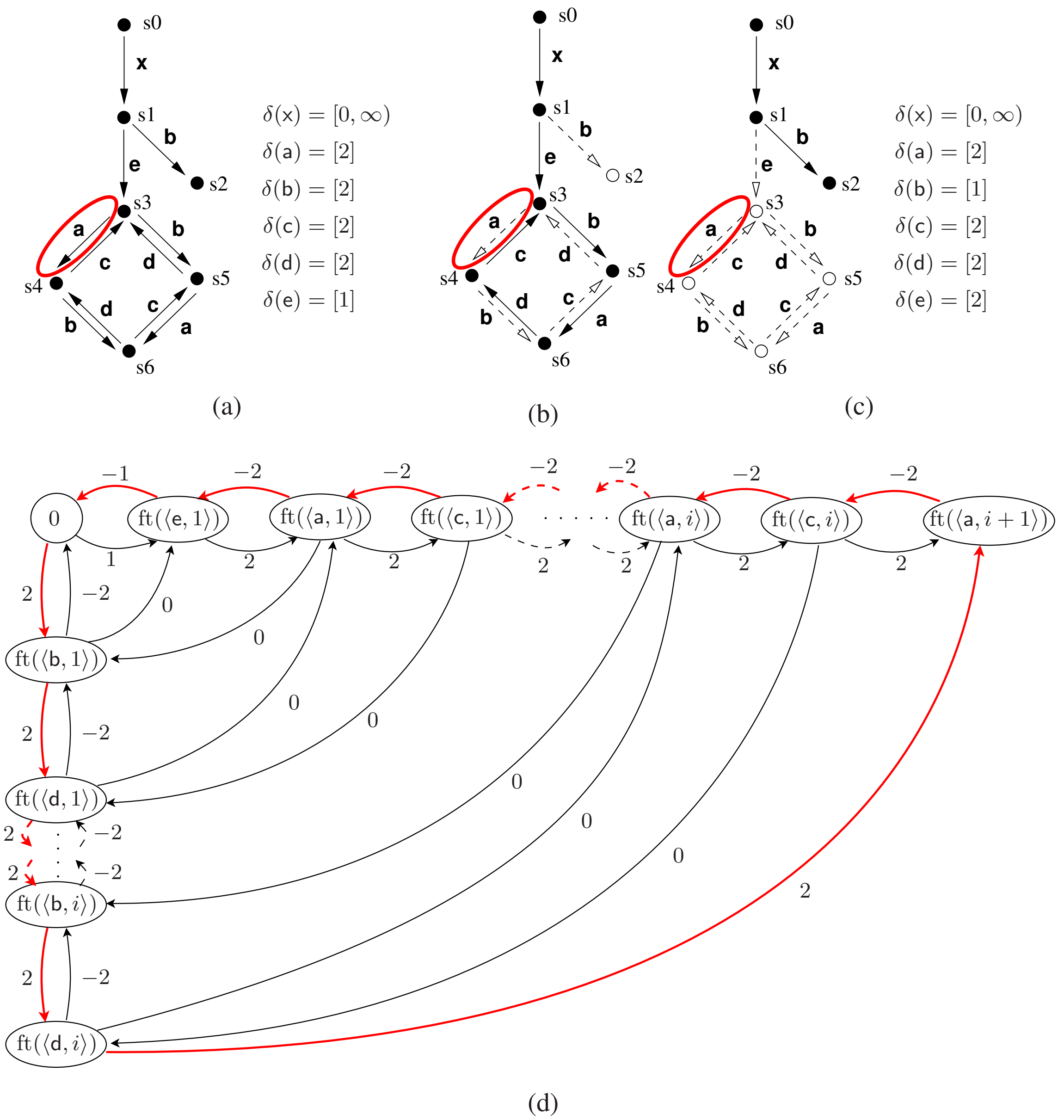


Figure 12.6. Example of divergent system for any algorithm in ECTRAalg class: (a) TTS for which any methodology based on ECTRAs and timing analysis on finite traces will not converge. (b) Timed state space of the TTS. Failure arc is not taken by any valid trace. (c) If the new delays bounds are considered, then the system can be shown correct even by the preliminary verification methodology. (d) Graph associated to the timing consistency problem of a generic trace of the form $\{\mathsf{e},\mathsf{b}\}\xrightarrow{\mathsf{e}}\{\mathsf{a},\mathsf{b}\}(\xrightarrow{\mathsf{b}}\{\mathsf{a},\mathsf{d}\}\xrightarrow{\mathsf{a}}\{\mathsf{c},\mathsf{d}\}\xrightarrow{\mathsf{d}}\{\mathsf{c},\mathsf{b}\}\xrightarrow{\mathsf{c}}\{\mathsf{a},\mathsf{b}\})^i\xrightarrow{\mathsf{a}}$. The only negative cycle of the graph (bold arcs in the picture) can be made arbitrarily large by changing parameter i.

equivalence between class BRIFTS and BNCTS for these type of systems, a minimal negative cycle of arbitrary length is also generated by these family of traces as shown in subfigure (d).

In general, TTSs outside class BRIFTS will diverge for any method in ECTRAalg as next theorem states.

THEOREM 12.6

Given a TTS A that is correct but does not satisfy the conditions of Lemma 12.4 (i.e.

$A \notin$ BRIFTS using the saturated *associated graph to test timing consistency), then there is no algorithm in* `ECTRAalg` *such that converges analyzing A.* ■ 12.6

Proof: *Conditions for convergence appearing in Lemma 12.4 are that all traces in $\mathcal{L}(A_i) \cap \overline{\mathcal{L}(P)}$ must have a timing inconsistent fragment whose timing inconsistent reduced version is of length less or equal to some finite fixed bound k. If these conditions are not fulfilled it means that, given an arbitrary finite bound k, there is some trace $\theta \in \mathcal{L}(A_i) \cap \overline{\mathcal{L}(P)}$ such that its minimal timing inconsistent reduced fragment is longer than k.*

A composition with an ECTRA can only remove traces containing a fragment that is enabling compatible with some of the trace in the set of lazy traces of the ECTRA (we note this set as Θ). We defined the traces in Θ of finite length. If a trace θ has a minimal timing inconsistent reduced fragment larger than the longest trace in Θ then for sure composition will not remove it, otherwise this would imply that the minimal timing inconsistent reduced fragment is enabling compatible with some trace θ' of Θ. But this means that the fragment is not minimal (θ' is also timing inconsistent and is shorter) which is a contradiction since we used the saturated associated graph to derive it. Because the saturated

As trace θ cannot be removed by any composition with an ECTRA with shorter lazy traces, and we know system A has traces of arbitrarily large length of its timing inconsistent reduced fragment, it results that the number of compositions required is infinite and the algorithm will never finish.

By uniting Theorem 12.6, Lemma 12.4 and Lemma 12.5, we obtain a complete specification of the capacity of the theoretical tools developed.

COROLLARY 12.7
Given a TTS A, there is an algorithm in `ECTRAalg` *such that converges analyzing A, if and only if, $A \in$ BRIFTS (this includes the case where A is non-Zeno and $A \in$ BNCTS).*
■ 12.7

12.6 Trace timing consistency revisited

In Theorem 12.6 we require to use the saturated associated graph to a trace θ, that is a graph that contains all the negative cycles of the graphs of the traces to whom θ is enabling compatible. However, the only candidate we have at this point, $\mathrm{graph}(\mathrm{LP}(\theta))$, does not fulfill this conditions and the theorem does not apply in such case as next example illustrates.

For instance, in Figure 12.7 (a) there is a TTS with a failure state, s_5. In subfigure (b) we show the structure of all traces reaching s_5 in the system. Let θ be one of these traces in which sequence $\xrightarrow{\mathsf{a}}\xrightarrow{\mathsf{b}}$ is repeated i times, then $\mathrm{graph}(\mathrm{gLP}(\theta))$ is depicted in subfigure (c). Considering the actual delays of the events there is a single negative cycle in the graph that includes the $\delta^u(\mathsf{e})$ and the $-\delta^l(\mathsf{f})$ and all negative arcs from $\mathrm{ft}(\langle \mathsf{a}, i+1\rangle)$ to $\mathrm{ft}(\langle \mathsf{x}, 1\rangle)$. This negative cycle can be enlarged at will by increasing i, thus if we wrongly apply Theorem 12.6, then the system should not be verifiable by any `ECTRAalg` algorithm. However, it is not difficult to see that the reduced trace of subfigure (d), obtained from θ by prefix removing, yields the LzCES of subfigure (e) that suffices to complete succesfully the verification of the system.

What failed in this case? The proof of the theorem is based in the fact that the shortest negative cycle in the system identifies the minimal set of conditions that allow to prove that the trace is timing inconsistent. While this is true, the problem is that $\mathrm{graph}(\mathrm{LP}(\theta))$ does not contain all the negative cycles it should.

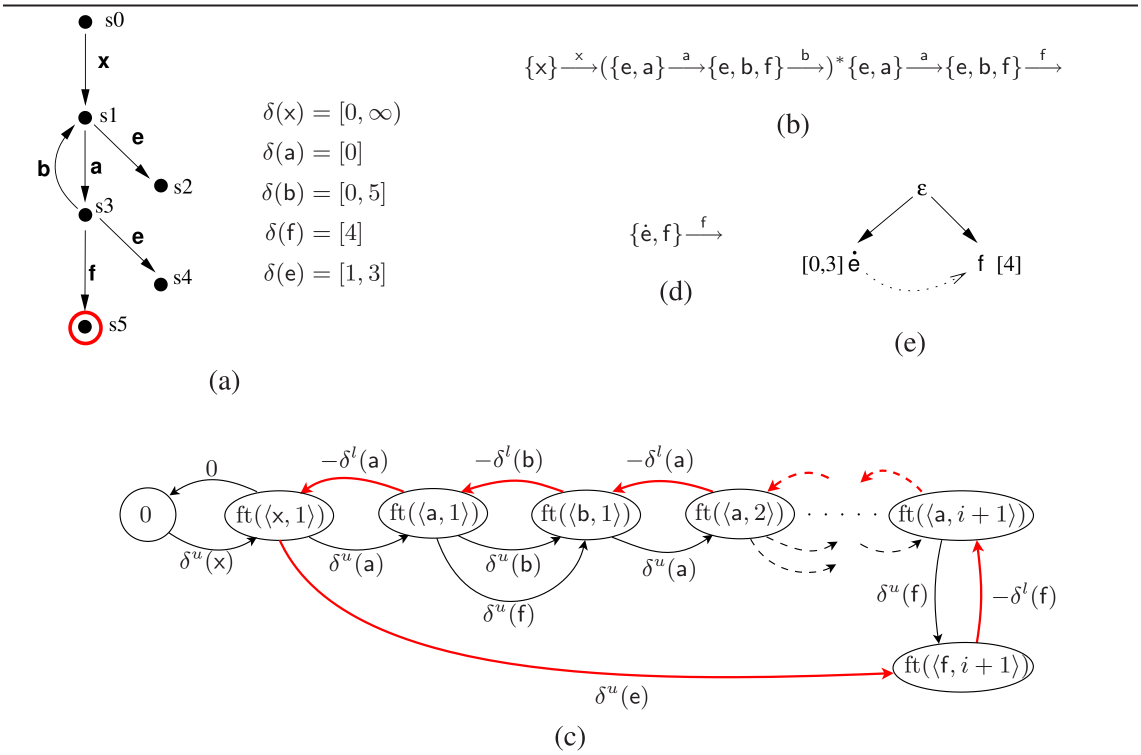


Figure 12.7. Example of verifiable system with an arbitrarily large timing inconsistent reduced trace fragment if $\mathrm{graph}(\mathrm{LP}(\theta))$ is used to check timing consistency: (a) TTS that contains Zeno runs that is verifiable by algorithms in `ECTRAalg` but does not satisfy the convergence conditions of Lemma 12.4 as contains traces with arbitrarily large minimal timing inconsistent reduced fragments. (c) Graph associated to the timing consistency problem of the trace $\theta = \{\mathsf{x}\}\xrightarrow{\mathsf{x}}(\{\mathsf{e},\mathsf{a}\}\xrightarrow{\mathsf{a}}\{\mathsf{e},\mathsf{b},\mathsf{f}\}\xrightarrow{\mathsf{b}})^i\{\mathsf{e},\mathsf{a}\}\xrightarrow{\mathsf{a}}\{\mathsf{e},\mathsf{b},\mathsf{f}\}\xrightarrow{\mathsf{f}}$ using $\mathrm{gLP}(\theta)$. An arbitrary number of repetitions of the $\xrightarrow{\mathsf{a}}\xrightarrow{\mathsf{b}}$ sequence allows to enlarge at will the unique negative cycle present in the graph. (d) Reduced trace sufficient to verify the system. Note that event e is defined as conservative. (e) Its LzCES.

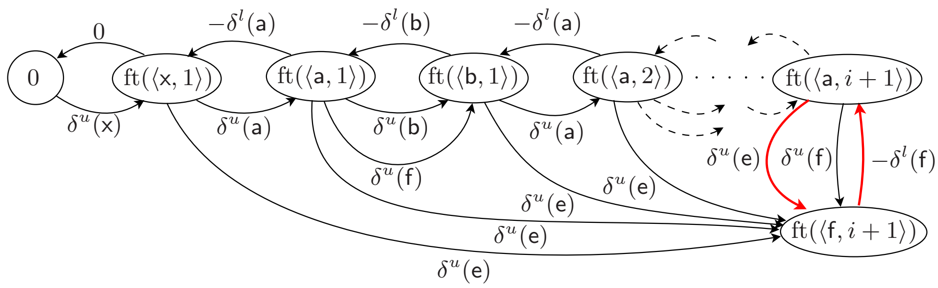
There are several possible formulations for the set of inequalities used to test the timing consistency of a trace that will result in exactly the same DBM after shortest-path closure, hence will have the same solution set. In §5.3.3 we presented just one possibility, $\mathrm{LP}(\theta)$, that has a good balance between the ease of creation and the number of inequalities in the system, because traversing θ once is enough to generate $\mathrm{LP}(\theta)$ without any further processing, and each event generates between one and three inequalities. However other possibilities are available, for instance the approach of [LLPY03] minimizes the number of inequalities, which allows a more efficient storage as a sparse matrix at the price of a more complex creation process. Now we present a third alternative, $\mathrm{LP}^*(\theta)$, that has a useful property: its graph contains all the negative cycles of the graphs of $\mathrm{LP}^*(\theta')$ where $\theta \geq_{\mathsf{ec}} \theta'$.

Definition 12.5 (Saturated timing consistency inequalities system) *Without loss of generality, let θ be a standardized distinct trace $\{\epsilon\}\xrightarrow{\epsilon}E_1\xrightarrow{\mathsf{e}_1}\cdots\xrightarrow{\mathsf{e}_n}E_{n+1}$. The* saturated timing consistency inequalities system *of θ, noted as $\mathrm{LP}^*(\theta)$ is defined by the following inequalities:*

- $\mathrm{ft}(\epsilon) = 0$
- $\forall i, j \quad i < j \Rightarrow \mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{e}_j)$
- $\forall i : 1 \leq i \leq n, \quad \delta^l(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{e}_i) - \mathrm{ft}(\mathrm{trigger}(\mathsf{e}_i))$

$$\{\mathsf{x}\}\xrightarrow{\mathsf{x}}\{\mathsf{e},\langle\mathsf{a},1\rangle\}\xrightarrow{\langle\mathsf{a},1\rangle}\{\mathsf{e},\langle\mathsf{b},1\rangle,\langle\mathsf{f},1\rangle\}\xrightarrow{\langle\mathsf{b},1\rangle}\{\mathsf{e},\langle\mathsf{a},2\rangle\}\ldots\{\mathsf{e},\langle\mathsf{a},i+1\rangle\}\xrightarrow{\langle\mathsf{a},i+1\rangle}\{\mathsf{e},\langle\mathsf{b},i+1\rangle,\langle\mathsf{f},i+1\rangle\}\xrightarrow{\langle\mathsf{f},i+1\rangle}$$

(a)



(b)

$\Sigma = \{\mathsf{e},\langle\mathsf{f},i+1\rangle\}$

(c)

$\emptyset\xrightarrow{\epsilon}\{\mathsf{e}\}\xrightarrow{\epsilon}\{\mathsf{e}\}\xrightarrow{\epsilon}\{\mathsf{e}\}\ldots\{\mathsf{e}\}\xrightarrow{\epsilon}\{\mathsf{e},\langle\mathsf{f},i+1\rangle\}\xrightarrow{\langle\mathsf{f},i+1\rangle}$

(d)

$\{\dot{\mathsf{e}},\mathsf{f}\}\xrightarrow{\mathsf{f}}$

(e)

Figure 12.8. Example of verifiable system with an arbitrarily large timing inconsistent reduced trace fragment if $\mathrm{graph}(\mathrm{LP}(\theta))$ is used to check timing consistency – second figure: (a) Distinct trace of trace $\theta = \{\mathsf{x}\}\xrightarrow{\mathsf{x}}(\{\mathsf{e},\mathsf{a}\}\xrightarrow{\mathsf{a}}\{\mathsf{e},\mathsf{b},\mathsf{f}\}\xrightarrow{\mathsf{b}})^i\{\mathsf{e},\mathsf{a}\}\xrightarrow{\mathsf{a}}\{\mathsf{e},\mathsf{b},\mathsf{f}\}\xrightarrow{\mathsf{f}}$. (b) Graph associated to the timing consistency problem of the trace θ using $\mathrm{gLP}^*(\theta)$ instead of $\mathrm{gLP}(\theta)$. Not all arcs are represented for readability. The shortest negative cycle (marked with bold arcs) contains arcs with weight $-\delta^l(\mathsf{f})$ and $\delta^u(\mathsf{e})$. (c) Set of events relevant to the shortest negative cycle of the graph. Distinct event $\langle\mathsf{a},i+1\rangle$ does not belong to the set because it works as the *startup* event in the cycle. (d) Projection of trace θ using the set of relevant events, first step. Events not relevant are removed from the enablings and are substituted in the arcs by ϵ. (e) Projection of trace θ using the set of relevant events, second step. All $\xrightarrow{\epsilon}$ arcs are removed whenever possible. As the enabling in which event e becomes enabled is removed, the event becomes conservative.

- $\forall i : 1 \le i \le n, \forall \mathsf{e} \in E_i, (\mathsf{e}_j = \mathrm{trigger}(\mathsf{e}) \ \wedge\ \forall k : j \le k < i, \mathrm{ft}(\mathsf{e}_i) - \mathrm{ft}(\mathsf{e}_k) \le \delta^u(\mathsf{e}))$

■ 12.5

Using algebra terminology the negative cycles in $\mathrm{graph}(\mathrm{LP}^*(\theta))$ identify *irreducible inconsistent subsytems* (IIS) of system $\mathrm{LP}^*(\theta)$ because removing some of the arcs of the cycle does not give another cycle (as we are considering just simple cycles) which is equivalent to say that removing the corresponding inequality will produce a feasible subsystem of equations. The fact that $\mathrm{graph}(\mathrm{LP}^*(\theta))$ will contain all the negative cycles of the traces that are enabling compatible with θ will allow to obtain the shortest enabling compatible trace that is able to show timing inconsistency of θ.

LEMMA 12.8
Given a two traces θ and θ' such that $\theta \ge_{\mathsf{ec}} \theta'$ then $\mathrm{graph}(\mathrm{LP}^*(\theta))$ *contains all negative cycles in* $\mathrm{graph}(\mathrm{LP}^*(\theta'))$. ■ 12.8

Proof: *By Lemma 5.8 we know that θ' can be obtained from θ by a sequence of transformations. Any of the allowed transformations can only remove negative cycles from* $\mathrm{graph}(\mathrm{LP}^*(\theta))$*, hence it will contain all the negative cycles of* $\mathrm{graph}(\mathrm{LP}^*(\theta'))$.

To illustrate the validity of this approach we will redo the example of Figure 12.7 using $\mathrm{LP}^*(\theta)$ instead of $\mathrm{LP}(\theta)$. This allows to fullfil the conditions to apply Corollary 12.7. In Figure 12.8 (a) we can see the distinct trace whose graph, $\mathrm{graph}(\mathrm{gLP}(\theta))$, was depicted in Figure 12.7 (c). Subfigure (b), on the other hand shows $\mathrm{graph}(\mathrm{gLP}^*(\theta))$. Because it contains many more arcs and to draw them all would render the figure hard to read we only represent some of them. It contains a small negative cycle whose set of relevant events is shown in subfigure (c). Note that distinct event $\langle \mathsf{a}, i+1\rangle$ does not belong to the set, although it appears in one of the nodes, because it acts as the startup event in the cycle *i.e.* this node would be substituted by node 0 in the timing inconsistent reduced trace. Using this set to project the original trace we obtain its timing-inconsistent reduced version as appears in subfigure (e). Subfigure (d) shows an intermediate step of the projection procedure. Note that subfigure (e) is the same as Figure 12.7 (d), which was the right trace to use in this context to complete the verification task.

To complete the proof of Theorem 12.6, we must show that the minimal timing inconsistent reduced version of a trace is the shortest trace that allows to infere the timing inconsistency of the full trace.

LEMMA 12.9

Let θ be a trace and θ' its minimial timing inconsistent reduced version. There is no other trace θ'' such that $\theta \geq_{\mathsf{ec}} \theta''$, θ'' is timing inconsistent and $|\theta''|<|\theta'|$. ■ 12.9

Proof: *First of all the projection operation using the sets of relevant events gives always an enabling compatible trace. This can be shown because the effects of the projection operator can be reproduced by a sequence of the allowed transformations. Mainly by prefix removal and then supression of the firing of sink events (this might require to tweak some enablings in order to convert events into sink events). All these transformations, in fact, remove nodes from the graph. By definition, θ' is the enabling compatible trace that preserves least nodes among all the enabling compatible traces with θ (because the negative cycle is searched in $\mathrm{graph}(\mathrm{gLP}^*(\theta))$). In the graph of θ', it is not possible to apply a transformation that removes some additional node without destroying the negative cycle, thus rendering θ' timing consistent. Hence, θ' is the shortest trace containing the negative cycle that requires the firing of least events. Consequently, there is no trace θ'' which is timing inconsistent and $|\theta''|<|\theta'|$.*

12.7 Checking BRIFTS/BNCTS membership

As opposed to class SSTS which was defined only taking into consideration the underlying TS hence disregarding timing information, definition of class BRIFTS heavily relies on this kind of information as it is fundamental to determine the timing consistency of trace fragments and to obtain their reduced versions. For instance, in the example of Figure 12.6, by simply changing the delay values of event b and event e, we do obtain another TTS for which even the preliminary verification methodology converges! (see subfigure (c)).

The problem now is to see if checking the membership of a system with respect to class BRIFTS is viable or not. Should it be possible, then we could know before starting the verification if it would be wasted effort or not. Obviously, the cost of this check is not interesting if its magnitude is equal or bigger than the cost of the verification problem itself.

Bad news is that the cost of determining the membership with respect to class BRIFTS is at least the same as the verification problem. The reason is that one of the elements that distinguish between a system in class BRIFTS and the rest is that all incorrect system belong to this class (and such distinction is precisely what the verification problem is all about). Fortunately, there is a way in which a conservative approximation to membership can be tested with low overhead while the

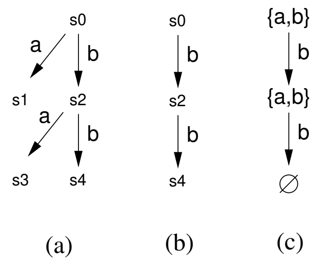


Figure 12.9. Example to illustrate that repeated enablings in trace do not necessarily correspond to repeated states in the run that generated the trace: (a) A TS. (b) One of its runs. (c) Its corresponding trace. Trace contains repeated enablings but run does not contain repeated states.

verification process is going on. The idea is to look at the failure traces found: if the system under analysis does not belong to BRIFTS then we will start seeing always increasing timing inconsistent reduced fragments (*i.e.* minimal negative cycles of increasing length).

For this to happen is a necessary condition that the timing inconsistent reduced fragments contain loops (see §12.8 for a discussion of what is a "loop" in a timing inconsistent reduced version of a trace fragment). Hence if the trace has at least one timing inconsistent reduced fragment that does not contain any repetition of a state, then is safe to continue with the verification flow.

On the other hand, if all timing inconsistent reduced fragments contain repetitions, then it is possible (but not sure) that we found a system outside BRIFTS. One possibility to go ahead is to be sure that the number of loops will be bounded. A loop is bounded if:

- It advances time *i.e.* at least one of the fired events in the loop has minimal delay bound greater than 0, and
- There is a continuously enabled event in all the loop with maximal delay bound different than infinity.

In such situation, the loop can only be unfolded only a finite number of times before the continuously enabled event is forced to fire. Only in the case that a failure trace has loops in all its timing inconsistent reduced fragments and all these loops cannot be shown to be bounded, then is likely that the verification of the system will diverge.

Taking that into account when such situation arises, one can either switch to a more powerful (yet expensive) verification method, try to analyze the timing behavior of loops or just continue with the hope that either a finite number of iterations will suffice or a timing consistent failure will eventually be found. In any case the user can be warned that divergence is possible which is a valuable information to take further decisions on the best verification approach to the problem.

12.8 Unfolding detection

In previous section, loop detection in a trace was a key element in the evaluation of the divergence possibilities. However, there is a technical problem to detect a loop in the timing inconsistent reduced version of a trace: a trace does not have states that can be checked for repetitions, instead has enablings, whose repetitions do not necessarily imply state repetitions (see Figure 12.9). Therefore some kind of run must be assigned to the timing inconsistent reduced version of a trace to be able to work with states. This can be quite easy when the reduction consists simply in selecting a trace fragment. Then we can work with the corresponding fragment of the original run.

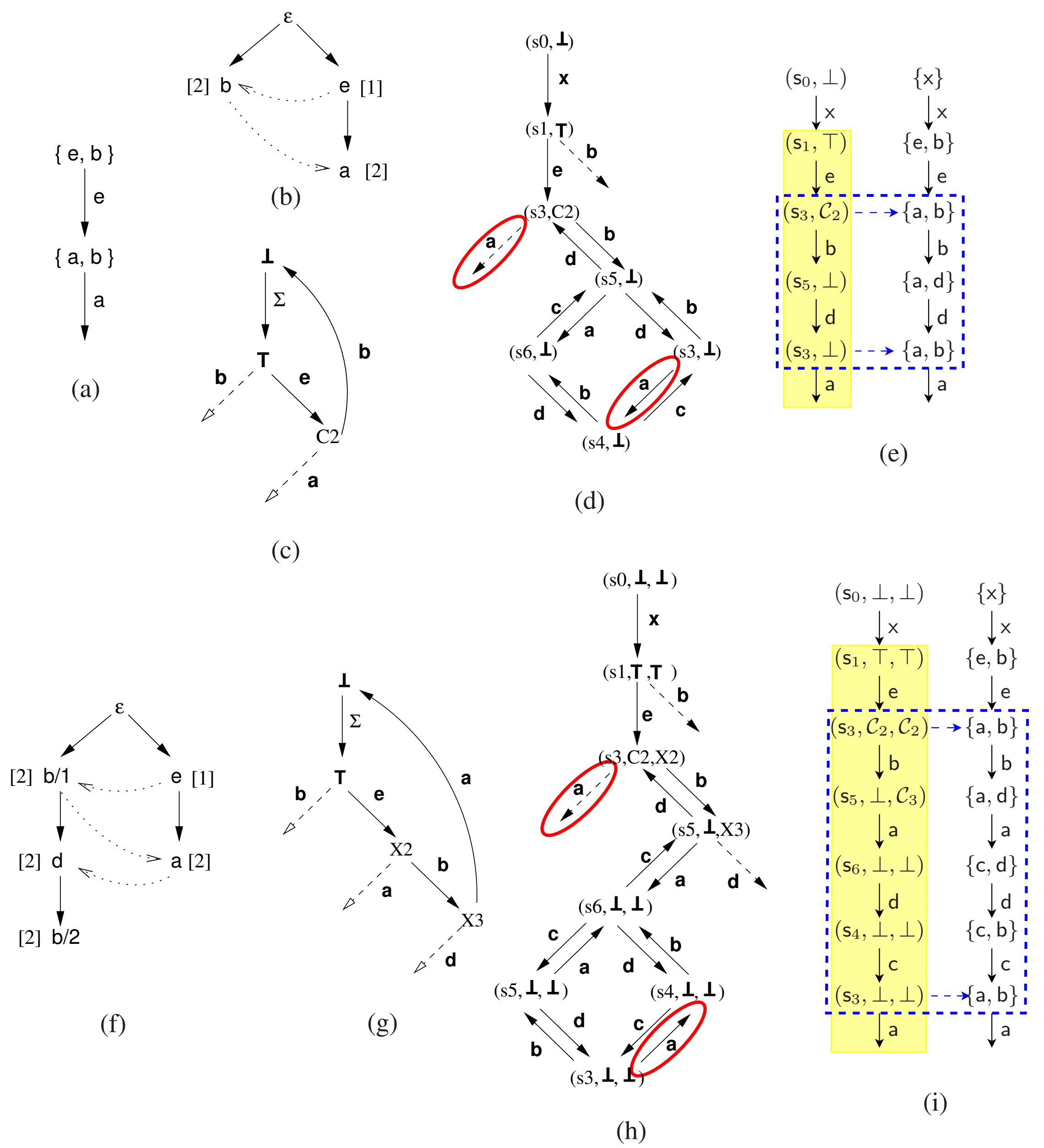


Figure 12.10. Example of unfolding detection and BRIFTS membership checking: (a) Shortest suffix to prove timing inconsistency of first failure trace found in system of Figure 12.6. (b) Derived LzCES after timing analysis. (c) Simplified depiction of its corresponding ECTRA. (d) Resulting classical LzTS after composition. (e) Trace of the next failure run in the system. Shortest suffix to prove inconsistency has been shaded and a dashed box indicates the states/enablings involved in the unfolding detection. (f) Derived LzCES after timing analysis of this second trace. (g) Simplified depiction of its corresponding ECTRA. (h) Resulting classical LzTS after second composition. (i) Trace of the next failure run in the system. There is no event continuously enabled between the two occurrences of state s_3.

For instance, in the example of Figure 12.6, assume that we use the approach of [Peñ03] and a search failure run method that always finds the shortest run to failure. The first run found is $s_0 \xrightarrow{x} s_1 \xrightarrow{e} s_3 \xrightarrow{a} s_4$. The shortest possible suffix to show inconsistency, in this case, is $s_1 \xrightarrow{e} s_3 \xrightarrow{a} s_4$ whose trace is shown in Figure 12.10 (a). After composition (subfigure (d)), next shortest failure run is

$$(\mathsf{s}_0,\bot)\xrightarrow{\mathsf{x}}(\mathsf{s}_1,\top)\xrightarrow{\mathsf{e}}(\mathsf{s}_3,\mathcal{C}_2)\xrightarrow{\mathsf{b}}(\mathsf{s}_5,\bot)\xrightarrow{\mathsf{d}}(\mathsf{s}_3,\bot)\xrightarrow{\mathsf{a}}(\mathsf{s}_4,\bot)$$

in which an unfolding of a loop appears, since we do first encounter state $(\mathsf{s}_3,\mathcal{C}_2)$ and later on we have state $(\mathsf{s}_3,\bot)$. This run together with its trace appear in (e).

Note that, in all the enablings between the enablings of $(\mathsf{s}_3,\mathcal{C}_2)$ and $(\mathsf{s}_3,\bot)$ (both included), event a is continuously enabled. As all involved events have finite upper delay bound and lower delay bounds greater than 0, then the number of unfoldings is known to be bounded for the reasons exposed in §12.7. After composition with the ECTRA of (g) we obtain the system of (h) in which one of the shortest failure runs is

$$(\mathsf{s}_0,\bot,\bot)\xrightarrow{\mathsf{x}}(\mathsf{s}_1,\top,\top)\xrightarrow{\mathsf{e}}(\mathsf{s}_3,\mathcal{C}_2,\mathcal{C}_2)\xrightarrow{\mathsf{b}}(\mathsf{s}_5,\bot,\mathcal{C}_3)\xrightarrow{\mathsf{a}}(\mathsf{s}_6,\bot,\bot)\xrightarrow{\mathsf{d}}(\mathsf{s}_4,\bot,\bot)\xrightarrow{\mathsf{c}}(\mathsf{s}_3,\bot,\bot)\xrightarrow{\mathsf{a}}$$

shown in (i) together with its trace. Again, we find an unfolding between state $(\mathsf{s}_3,\mathcal{C}_2,\mathcal{C}_2)$ and state $(\mathsf{s}_3,\bot,\bot)$, but in this case there is no event which is continuously enabled between these two points. As trace can only be falsified using at least this fragment –its trace is the minimal timing inconsistent reduced trace–, a warning will be issued that verification process might diverge, according to what has been discussed in previous section.

On the other hand, when reduced versions are not simply fragments of the trace but also remove irrelevant concurrent events, we do no longer have a one-to-one relation between the original run and the trace.

For instance assume that, in system of Figure 12.4, the search failure run method returns run $\mathsf{s}_0\xrightarrow{\mathsf{x}}\mathsf{s}_1\xrightarrow{\mathsf{e}}\mathsf{s}_3\xrightarrow{\mathsf{a}}\mathsf{s}_4\xrightarrow{\mathsf{b}}\mathsf{s}_3\xrightarrow{\mathsf{d}}$. The minimal timing inconsistent reduced version of its trace is $\{\mathsf{c},\mathsf{e}\}\xrightarrow{\mathsf{e}}\{\mathsf{c},\mathsf{d}\}\xrightarrow{\mathsf{d}}$. Original run contains a loop, but we already know that using the previous reduced trace the verification task can be successfully completed. Therefore we must develop a method to discover that, in fact, the loop of the run does not affect the reduced trace.

To test if a reduced trace contains an unfolding of a loop we use a similar approach to the projection of states introduced in §10.6.3. The main idea is that there is a one-to-one correspondence between the original run and its trace, as well as an enabling compatible mapping between the trace and its reduced version. Composing both relations we obtain a mapping from the states of the original run to the reduced trace. At this point, we consider that there is a loop in the reduced trace when a repeated state appears in different enablings of the trace. A repeated state can be a direct repetition, like in this example, or a state that has unfolded due to previous composition, like in the previous example, where states $(\mathsf{s}_3,\bot)$ would be considered a repetition of state $(\mathsf{s}_3,\mathcal{C}_2)$. In our last example state s_1 maps to enabling $\{\mathsf{c},\mathsf{e}\}$ and states s_3 and s_4 map to enabling $\{\mathsf{c},\mathsf{d}\}$, hence there is no loop as expected.

12.9 A sketch of a more convergent methodology

With all the previous discussion of §12.2 and §12.8 in mind, we propose the following verification methodology in order to maximize its convergence:

- Search all[3] simple negative cycles in the graph associated to the timing consistency checking (*i.e.* $\mathrm{graph}(\mathrm{LP}^*(\theta))$) of the failure trace found. Compute the relevant set of events and the corresponding timing inconsistent reduced version according to these sets.
- Divide reduced versions in two sets: one group, Θ_1, that contains all traces in which there is no unfolding, and Θ_2 that holds the rest of traces.

[3] Because some cycles involve a proper subset of relevant events with respect to other negative cycles, actually not all simple negative cycles are considered, only the ones whose relevant set of events is not a proper superset of a relevant set of some other cycle.

- If $\Theta_1 \neq \emptyset$, use some heuristic to select among these traces the one to use in composition.
- Otherwise, split set Θ_2 in two sets: Θ_3 are the traces that have a guaranteed number of unfoldings according to the conditions seen in §12.7, and Θ_4 are the rest.
- If $\Theta_3 \neq \emptyset$, use some heuristic to select among these traces the one to use in composition.
- Otherwise, issue a warning to the user and then use some bounding strategy or change methodology.

The methodology of [Peñ03], as already mentioned, tried to solve the convergence problem by using the smallest trace suffix that showed timing inconsistency. This is only an heuristic which tries to obtain a small timing inconsistent fragment and, with some postprocessing –not documented in [Peñ03]–, compute an approximation to its timing inconsistent reduced version. However, this approach converges in much less situations than the proposal made in this section.

For instance, in the example of Figure 12.6 (a) if delays of event x is changed to $\delta(\mathsf{x}) = [5, \infty)$ and event b is made enabled also in state s_0 by adding transition $\mathsf{s}_0 \xrightarrow{\mathsf{b}} \mathsf{s}_7$ (and assuming initially enabled events are not conservative), then the system is trivially correct but using the shortest suffix strategy will produce an infinite unfolding.

Another valid strategy to attain the theoretical convergence is to use always the shortest timing inconsistent reduced traces that correspond to the simple negative cycles. Although this allows to use fragments that might produce unfoldings, the length of these fragments will always increase. This guarantees that, eventually, all good candidates are used in composition.

The problem of the methodologies proposed in this section is that several compositions can be performed that are very local to disprove the current failure trace, and that some other trace, with only minor variations in the firing order of the events, is found in subsequent steps. To avoid this phenomenon we will, instead of using the proposed methodology, use a technique much more aware of the surrounding state space in order to make the compositions more effective. Only when unfoldings are detected we would use the approach outlined in this section to guarantee that we attain the theoretical convergence.

Chapter 13

NEW METHODOLOGY

Summary

With all the theory presented until this point we propose a new methodology for the verification of timed systems. Although the basic concept of iterative refinement is still used as in [Peñ03], many more processing steps are introduced. The additional effort in each iteration is used to make the algorithm “wiser”. Ideally this would allow to reduce the number of iterations needed to decide if the system is correct or not. Although the time needed at each iteration will grow, the reduction in the number of iterations could yield a smaller total verification time depending on how these two parameters are actually modified. In Chapter 15 we will see that it is indeed the case that this new approach performs better than a more straightforward and “blindfold” implementation.

The design of the methodology is difficult because we must find an equilibrium between several parameters. In particular our objective is to obtain a methodology that:

- Converges as much as it can, up to the limitations exposed in Chapter 12.
- Uses CESARs instead of single trace analysis to speed up verification.
- Reduce the extra work in each iteration. Since the nature of iterative verification is “lazy” in the sense that only when fails are detected an iteration is started, it is sensible to follow the same philosophy in each iteration. That is to try to do only the work that is required.

One of the major problems is that, to ensure convergence, the methodologies we have proposed (§12.9) work at the trace level. While this is useful to reveal the theoretical limitations of the approach, it contradicts the objective of using CESARs for efficiency. Hence we must find a way to preserve convergence and work with CESARs instead of single traces. Moreover, to accomplish our third objective, we must try that the actions taken to achieve this goal do not introduce too much overhead in the verification iteration.

13.1 Overview

In Figure 13.1 we show the general outline of the new methodology as a UML 2.0 activity diagram.

The first part of the verification flow is equal to the general iterative refinement flow. Verification starts by abstracting time from the system description (TTS box on the left upper corner). In the resulting LzTS we search for a trace to a failure. If no trace is found, then we can assure the system is correct. Otherwise, we break the trace into several fragments using the nodal points as bounds and we check their timing-consistency. If they are all consistent, then the set of timing inconsistent fragments is empty and we already have a counterexample by rebuilding the trace. It might seem silly to split first the trace in fragments and then have to rebuild it. However, this happens only if the original found trace is timing consistent. One of the advantages of the methodology is its capability to detect "nearby" traces that are timing consistent. This involves the substitution of timing inconsistent trace fragments by timing consistent ones in later steps of the flow.

Let us go back to the trace fragments. For each one we compute a CESAR that is representative enough (see Chapter 11), so that the fragment is contained in the zone of the CESAR and the timing analysis is guaranteed to be accurate. Precisely with the information obtained from this analysis the LzCES is constructed.

Next step is to compute the zone (see Definition 11.1) of the CESAR. Although the zone is partially computed as a subproduct of the CESAR adjustment, it has to be completed once we have the right CESAR. Moreover, this allows to mark all the failures in the zone. With the timing information of the LzCES we can check if these failures can be avoided or not (see Lemma 11.1). If some failure cannot be avoided, then by Theorem 11.2, we know there is a timing consistent trace reaching it. We explain the procedure to obtain such trace in §13.4.

On the other hand, if all failures can be avoided there might be several combinations of precedences between events that we can use to prove this. Given that possible solutions can behave differently in terms of convergence and the state space pruned, we explore all of them using a *branch and bound* algorithm. We select the smaller LzCES that behaves better in terms of convergence (§13.3). If at some point the minimal LzCES has "bad" convergence expectations, then we switch to the trace level verification of §12.9 to guarantee maximal convergence[1].

After all fragments have been processed, either we have minimal LzCES or we were able to generate timing consistent fragments for all originally timing inconsistent ones. In the latter case, we rebuild a counterexample and we finish. In the other case, we generate the ECTRA from the minimal LzCES as explained in §8.5.

Finally, once the ECTRA is available, we compose it with the original system to incorporate the timing information into the model. The exact procedure is explained in §13.5 but mainly deals with indeterminacy issues analyzed in Chapter 10.

In the following sections we explain the parts of the methodology that have not been previously covered.

13.2 Failures? Which failures?

Our algorithm computes the zone of each trace fragment. We know there is a failure state/arc in the zone of the last fragment, but besides this one it is perfectly possible that no other failure is present in the zone of any of the fragments. In such cases, the meaning of "Are failures avoided in the zone?" is not completely clear.

[1]This alternative flow is not shown in the activity diagram for readability

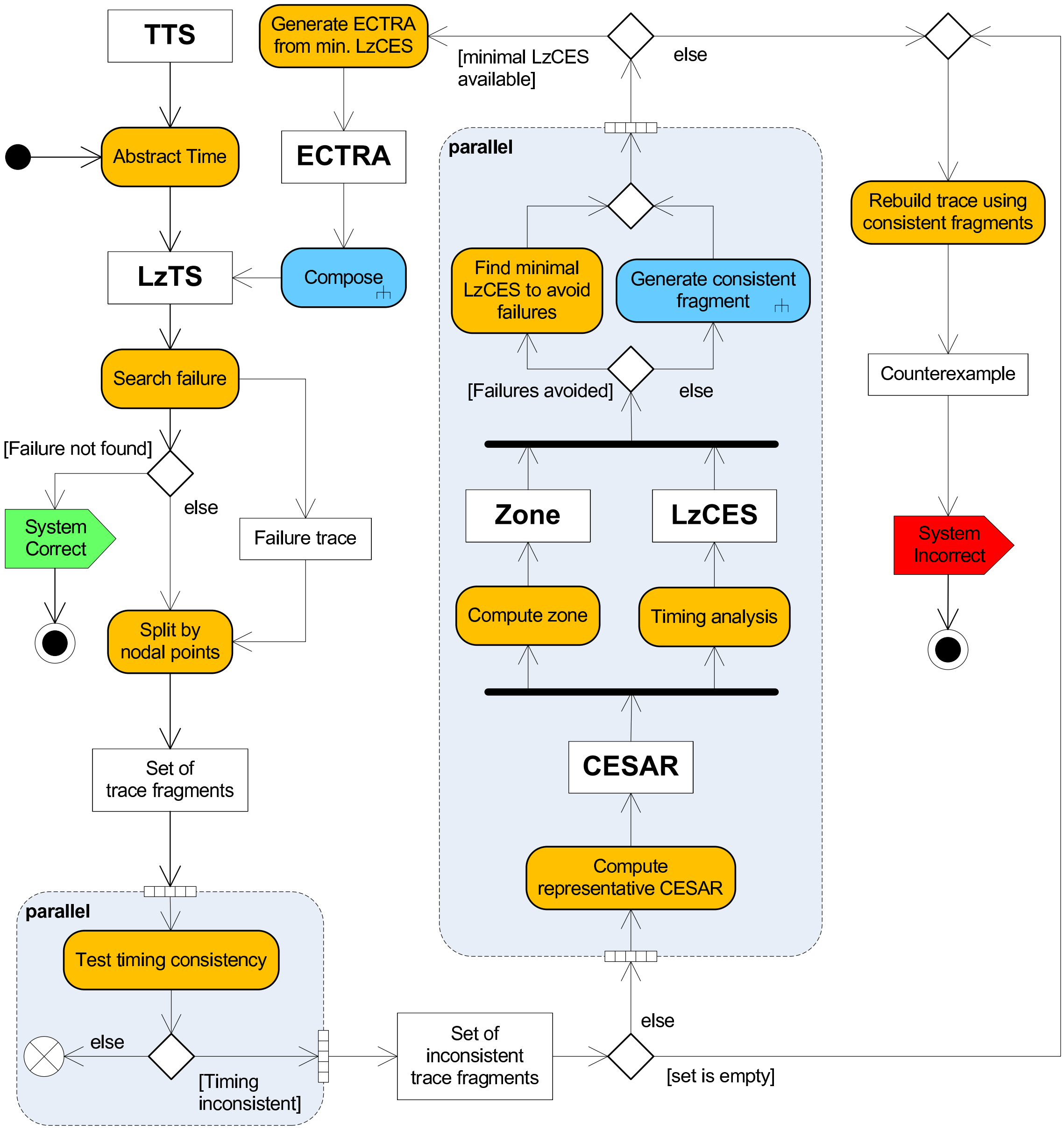


Figure 13.1. General schema of the verification flow as a UML 2.0 activity diagram.

To store the zone information, we use a GRC in which we map the states/transitions of the zone. For zones of fragments that are not the last one, the last configuration of its GRC is *always* considered a configuration containing a failure, irrespective of the states mapped on it. Of course, any other configuration/arc that contains a mapped failure state/transition, is also considered, but the last configuration is the linking point with the next fragment. If it can be avoided, then the rest of the trace is avoidable. Furthermore, if it cannot be avoided then we can find a timing consistent trace that goes from the start of the zone to its end, hence can replace perfectly the current timing inconsistent fragment.

In the case in which a fragment different than the last cannot avoid a failure and the failure is not this *dummy* failure, then the fragments that appear after it can be ignored, as we have found a shorter trace to a failure situation.

13.3 The *Find minimal LzCES to avoid failures* action

The objective of this phase is to find the most suitable LzCES to generate the ECTRA used in composition. *Most suitable* in this context means that, after composition, all failure traces of the zone are pruned, and that it guarantees the convergence of the methodology as much as possible.

To find a LzCES of these characteristics we use several observations. First, not all relative timing precedences found in timing analysis are equally important to avoid the failures of the zone. Second, the LzCES might be simplified depending on which precedences we want to be preserved. Finally, the convergence when ECTRAs from LzCES are used –instead of being generated by minimal timing inconsistent reduced traces– is still an open question.

In view of these observations we proceed as follows. We compute the minimal sets of precedences that suffice to avoid all failures. There can be many of these sets, and each one might produce a different minimal LzCES. So, for each one of the sets, we compute its minimal LzCES (§13.3.1). To select the best candidate in terms of convergence, we classify them using a similar criterion to the one in §12.9. That is, if there are repetitions in the zone, does this LzCES unfold loops? We prefer LzCES that do not unfold. Among the ones that unfold, we prefer the ones that have bounded unfoldings. Finally if, for all timing inconsistent fragments, only unbounded unfoldings are available, then we switch to one of the single trace methodologies of §12.9 that guarantee maximal convergence at the expense of more iterations.

A sketch of our implementation can be found in Algorithm 13.1. As each precedence avoids a particular set of failures, and all failures must be avoided, the problem of computing the minimal sets of precedences is essentially the *set covering problem*, which is known to be NP-complete. A number of standard algorithms to solve it are reviewed in [CFT98].

In our algorithm, before line 15, we compute the set of failures that each precedence in $<_{\mathrm{lzy}}$ avoids. A given configuration is avoided by a precedence if the precedence is not useless (*i.e.* it has some pruning capability also when the effect of disabled events is considered) and the partial order satisfied by all the traces reaching it is no longer an order when the precedence is added (*i.e.* a cycle appears). A precedence $\mathsf{a} < \mathsf{b}$ violates an order $<'$ if, and only if, $(b, a) \in <'$. In lines 7 and 11 we do test this.

Function order computes the partial order respected by all traces reaching a particular configuration or arc. For the case of a configuration $\mathcal{C}$, it is formed by all $\mathsf{a} <_{\mathrm{CS}} \mathsf{b}$ such that $\mathsf{a} \in \mathcal{C}$ and also by $\mathsf{a} < \mathsf{b}$ where $\mathsf{a} \in \mathcal{C}$, and $\mathsf{b} \notin \mathcal{C} \cup \mathcal{D}(\mathcal{C})$, where $\mathcal{D}(\mathcal{C})$ is the set of disabled events by the firing of the events in $\mathcal{C}$. That is, traces must respect the restriction order of CS for all the fired events needed to reach the configuration, and these fired events must happen before any other event to be able to get to $\mathcal{C}$. For the case of an arc $\mathcal{C}_i \xrightarrow{\mathsf{e}}$ the order is the same but also adds that event e fired before any other of the enabled events in $\mathcal{C}_i$. That is incorporating precedences $\mathsf{e} < \mathsf{c}$ where $\mathsf{c} \in \mathcal{E}(\mathcal{C}) \setminus \{\mathsf{e}\}$. Formally,

$$\mathrm{order}(\mathcal{C}) = (\prec \cup \{(\mathsf{a}, \mathsf{b}) \in <_{\mathrm{CS}} \mid \mathsf{a} \in \mathcal{C}\} \cup \{(\mathsf{a}, \mathsf{b}) \mid \mathsf{a} \in \mathcal{C} \wedge \mathsf{b} \notin \mathcal{C} \cup \mathcal{D}(\mathcal{C})\})^*$$
$$\mathrm{order}(\mathcal{C}_i \xrightarrow{\mathsf{e}}) = (\mathrm{order}(\mathcal{C}_i) \cup \{(\mathsf{e}, \mathsf{c}) \mid \mathsf{c} \in \mathcal{E}(\mathcal{C}) \setminus \{\mathsf{e}\}\})^*$$

We explained in §8.4.2 that a precedence $\mathsf{a} < \mathsf{b}$ starts pruning paths when b is enabled and event a has not fired yet. If before this point any of the two events is disabled, then the precedence becomes useless. Precisely function `useless` tests this. It returns true whenever the set of events

Algorithm 13.1: findMinimalLzCES

input : LCS $= \langle \mathrm{CS}, <_{\mathrm{lzy}} \rangle$ a LzCES.
input : $z = \langle \Sigma, C, T, \mathcal{C}_0 \rangle$, GRC used to represent the zone of CS.
input : failure a vector of configurations/arcs of the zone that contain failures and must be avoided.
output : minLzCES minimal LzCES that avoids all failures in the zone and considers convergence factors.

```
useful ← ∅
forall (a, b) ∈ <lzy do
    avoidedBy (a <lzy b) ← ∅
    forall f ∈ failure do
        if type (f) = configuration then
            C ← f.configuration
            if ¬useless(a < b, order(C)) ∧ (b, a) ∈ order(C) then
                avoidedBy (a <lzy b) ← avoidedBy (a <lzy b) ∪ {f}
        else
            (C_i --e--> C_j) ← f.arc
            if ¬useless(a < b, order(C_i --e-->)) ∧ (b, a) ∈ order(C_i --e-->) then
                avoidedBy (a <lzy b) ← avoidedBy (a <lzy b) ∪ {f}
    if avoidedBy (a <lzy b) ≠ ∅ then
        useful ← useful ∪ {(a, b)}
minLzCES ← generateAllMinCoverSets(LCS,failure,useful,avoidedBy)
return minLzCES
```

fired before the enabling of b includes a disabler of a or b. Formally, useless($a < b$, order) $= \Box$ if, and only if, $\exists d \in \mathrm{fbe}(b, \mathrm{order}) : d \triangleright a \vee d \triangleright b$.

After the set of failures that each precedence avoids, we call the `generateAllMinCoverSets` function (Algorithm 13.2) which solves the set covering problem using a tailored branch and bound implementation. Only the precedences that avoid some failure are passed to this algorithm, together with the set of failures to avoid, and the computed set of failures avoided by each precedence. To increase efficiency this function uses several optimizations (many of them standard):

- First, it orders the precedences using the number of failures avoided, and starts searching solutions using the precedences that are more useful *i.e.* avoid more failures. (line 1)
- Solutions are explored in a DFS manner. When no current bound (different than the original one) exists, then the algorithm adds precedences until a full solution is found. Then it compute its minimal LzCES (line 11, see §13.3.1) and sets it as the current bound.
- Once a bound is available, for each partial solution, we compute its minimal LzCES (line 18) and we classify this LzCES. If current bound is of a better class, then it is discarded. If it is of the same class, then we compare their sizes (line 19). If bigger, then it is also discarded.
- Given a partial solution, only precedences that increase the coverage of the solution are used to expand it. (lines 15 and 24).

The algorithm uses some auxiliary functions that we describe now:

- `append` appends an element to the end of a sequence.
- `prepend` adds an element to the beginning of a sequence.
- `popFirst` removes the head element of a sequence and returns it.

Algorithm 13.2: generateAllMinCoverSets

input : LCS $= \langle \mathrm{CS}, <_{\mathrm{lzy}} \rangle$ a LzCES.
input : failure a vector of configurations/arcs of the zone that contain failures and must be avoided.
input : useful set of precedences that avoid some of the failure in failure.
input : avoidedBy function that relates each precedence in useful with the failures it avoids.
output : bestL minimal LzCES that avoids all failures in the zone and considers convergence factors.

```
1  precedence ← orderByNumFailures(useful)        ▷ precedence is a vector of |precedence| elements
2  bestL ← LCS
3  partial ← ∅                                    ▷ partial is a sequence of partial solutions to explore
4  forall i such that 0 ≤ i <|precedence| do
5      append(partial, {i})                       ▷ Start with partial solutions of one element
6  while partial ≠ ∅ do
7      sol ← popFirst(partial)                    ▷ Remove first element in sequence and put it in sol
8      avoids ← ⋃_{∀i∈sol} avoidedBy(precedence[i])
9      if bestL = LCS then
10         if avoids ⊇ failure then
11             L ← minLzCEStoPreserve(LCS, {precedence[i] | ∀i ∈ sol}, failure)
12             bestL ← L
13         else
14             forall j : max_{∀i∈sol}(i) < j < |precedence| do
15                 if avoidedBy(precedence[j]) ⊈ avoids then
16                     prepend(partial, sol ∪ {j})
17     else
18         L ← minLzCEStoPreserve(LCS, {precedence[i] | ∀i ∈ sol}, failure)
19         if class(L) < class(bestL) or (class(L) = class(bestL) and size(L) < size(bestL)) then
20             if avoids ⊇ failure then
21                 bestL ← L
22             else
23                 forall j : max_{∀i∈sol}(i) < j < |precedence| do
24                     if avoidedBy(precedence[j]) ⊈ avoids then
25                         prepend(partial, sol ∪ {j})
26 return bestL
```

- `minLzCEStoPreserve` is a function that computes the minimal LzCES that preserves a given set of relative timing relations. See §13.3.1.

- `class` returns the class number of a LzCES according to a convergence criterion. These classes and how they can be computed are explained in §13.3.2.

- `size` returns the size of the alphabet of a LzCES.

Note that in line 2 we initialize the current best solution found to the initial LzCES LCS. However, this is only done the first time the algorithm is called for a given trace. For the rest of the timing inconsistent fragments, there is already a bound available from previous iterations, hence at this point the best among the current complete LzCES LCS and the previous best solution is taken. The implementation is a little bit more complex if fragments are tested in parallel, but modifications are straightforward.

13.3.1 Computing the minimal LzCES

Given a LzCES LCS and a set of relative timing precedences P that must be preserved, we can compute a LzCES L such that L preserves the precedences in P and the size of its alphabet is minimal. Moreover, L has to prune all the failure traces of the zone. This implies that all failure traces must go through the entry point of L.

Definition 13.1 (minimal **LzCES** to contain a set of precedences)
Given a LzCES $L = \langle \mathrm{CS}, <_{\mathrm{lzy}}, \mathrm{context}\rangle$ *with* $\mathrm{CS} = \langle \Sigma, \prec, \triangleright, <_{\mathrm{CS}}\rangle$ *and a subset P of the relative timing precedences in* $<_{\mathrm{lzy}}$ *(i.e.* $P \subseteq <_{\mathrm{lzy}}$*), let* $L' = \langle \mathrm{CS}', <'_{\mathrm{lzy}}, \mathrm{context}'\rangle$ *be a LzCES obtained after the timing analysis of CESAR* $\mathrm{CS}' = \langle \Sigma', \prec', \triangleright', <'_{\mathrm{CS}}\rangle$*. We say that* L' contains P *if* $P \subseteq <'_{\mathrm{lzy}}$ *and all failure traces in the zone of* CS *are enabling compatible with some trace of* CS'.
If there is no other LzCES $L'' = \langle \mathrm{CS}'', <''_{\mathrm{lzy}}\rangle$ *that contains P with* $\mathrm{CS}'' = \langle \Sigma'', \prec'', \triangleright'', <''_{\mathrm{CS}}\rangle$ *and* $|\Sigma''|<|\Sigma'|$*, then we say that L' is the* minimal LzCES to contain P. ■ 13.1

The basic idea to compute the minimal LzCES that contains P is to simplify the underlying CESAR of the LzCES and then redo the timing analysis on this simplified CESAR to obtain a smaller LzCES that contains P. Later on we will see the conditions under which this smaller LzCES is known to be the minimial one. By now we consider the simplifications of the CESAR that guarantee that all failure traces go through the entry point and that the timing relations in P are preserved.

The simplification comprises two phases. First, events not related to the precedences in P are removed from the CESAR. After that we find the latest nodal point in the zone of the new CESAR such that all failures are reachable from it. All the events fired before that nodal point can be safely discarded. To ease the formalisation we introduce function inf that, given a set and an ordering relation, returns the least elements of the set according to the relation. Formally, $\inf(S, <) = \{x \in S \mid \nexists y \in S : y < x\}$. Conversely, function sup returns the greatest elements, that is $\sup(S, <) = \{x \in S \mid \nexists y \in S : x < y\}$.

Lemma 13.1
Given a LzCES $L = \langle \mathrm{CS}, <_{\mathrm{lzy}}, \mathrm{context}\rangle$ *with* $\mathrm{CS} = \langle \Sigma, \prec, \triangleright, <_{\mathrm{CS}}\rangle$*, a subset P of the relative timing precedences in* $<_{\mathrm{lzy}}$ *and a context selection function* $\mathrm{context}_s$ *that, for each precedence p in P, returns one of the timing zones in* $\mathrm{context}(p)$*, then the LzCES* $L' = \langle \mathrm{CS}', <'_{\mathrm{lzy}}, \mathrm{context}'\rangle$*, obtained after the timing analysis of CESAR* $\mathrm{CS}' = \langle \Sigma', \prec', \triangleright', <'_{\mathrm{CS}}\rangle$*, contains P if:*

- $\Sigma' = (\downarrow\Sigma_p)_{\prec} \setminus (\downarrow\Sigma_c)_{\prec}$ *where* $\Sigma_p = \{\mathsf{e} \in \Sigma \mid \mathsf{a} <_{\mathrm{lzy}} \mathsf{b} \in P \wedge \big((\mathsf{a} = \mathsf{e} \vee \mathsf{b} = \mathsf{e}) \vee (\mathsf{c} < \mathsf{d} \in \mathrm{context}_s(\mathsf{a} <_{\mathrm{lzy}} \mathsf{b}) \wedge (\mathsf{c} = \mathsf{e} \vee \mathsf{d} = \mathsf{e}))\big)\}$ *and* $\Sigma_c \subseteq \Sigma$ *such that*
  (i) $\forall \mathsf{e}, \mathsf{e}' \in \Sigma_c, \quad \mathsf{e} \not<_{\mathrm{CS}} \mathsf{e}'$.
  (ii) $\forall \mathsf{e} \in \Sigma_c, \forall \mathsf{e}' \in \sup((\downarrow\Sigma_p)_{\prec} \setminus (\Sigma_c\uparrow)_{<_{\mathrm{CS}}}, <_{\mathrm{CS}}), \quad \mathsf{e}' <_{\mathrm{CS},\mu} \mathsf{e}$ *. Where* $<_{\mathrm{CS},\mu}$ *is the covering relation of* $<_{\mathrm{CS}}$.
  (iii) $\forall \mathsf{e} \in \Sigma_c, \forall \mathsf{e}' \in \inf(\Sigma', \prec), \quad \mathsf{e} \prec_\mu \mathsf{e}'$.

  We call an event set Σ_c *satisfying these conditions a* cutset of CS with respect to Σ_p.
- *Relations* $\prec'$, $\triangleright'$ *and* $<'_{\mathrm{CS}}$ *are equal to their* $\prec$, $\triangleright$ *and* $<_{\mathrm{CS}}$ *counterparts, but restricted to the events in* Σ'.

■ 13.1

Proof: *Let θ be any of the traces of* CS*, we will show that we can find a trace θ' such that θ' is a trace of* CS' *and* $\theta \geq_{\mathsf{ec}} \theta'$*. Let θ' be the trace obtained by removing all events not*

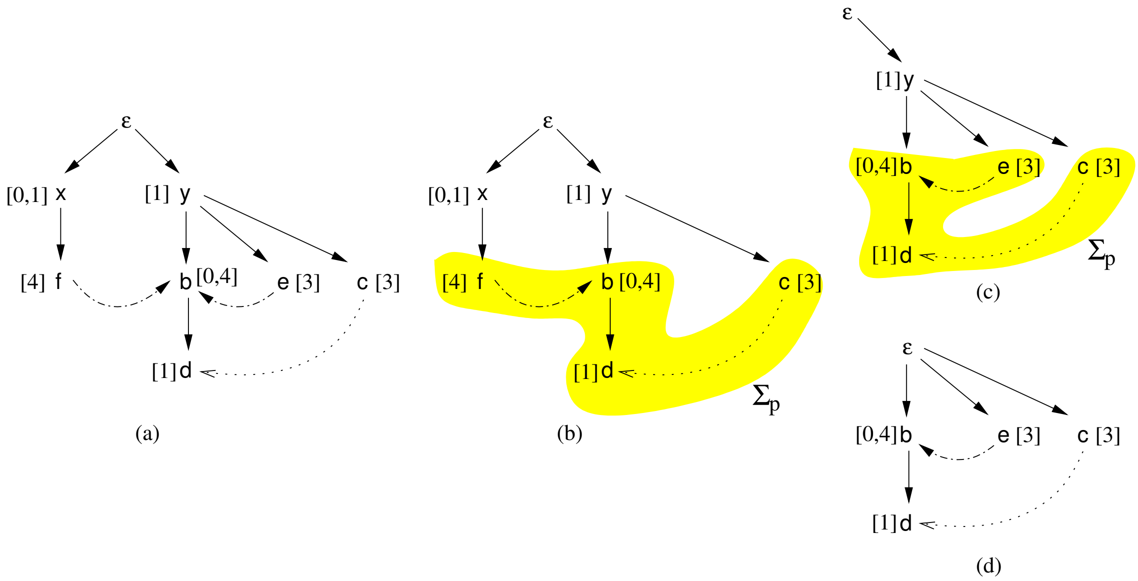


Figure 13.2. Example of different LzCES produced by different context selection. (a) Original LzCES. (b) Resulting LzCES of the application of Lemma 13.1 using $\mathrm{context}_s(\mathsf{c} <_{\mathrm{lzy}} \mathsf{d}) = \{\mathsf{f} < \mathsf{b}\}$. (c) LzCES obtained by the same procedure but using $\mathrm{context}_s(\mathsf{c} <_{\mathrm{lzy}} \mathsf{d}) = \{\mathsf{e} < \mathsf{b}\}$ and cutset $\Sigma_c = \{\epsilon\}$. (d) LzCES using the same context but with cutset $\Sigma_c = \{\mathsf{y}\}$.

in Σ' from θ. This trace belongs to CS' since it respects the relations in CS'. Moreover, we can obtain also θ' from θ by using enabling compatible preserving transformations of Lemma 5.8, hence $\theta \geq_{\mathsf{ec}} \theta'$. Basically, the transformation process consists in removing sink events until only events in $(\downarrow\Sigma_p)_\prec$ remain, and then using the prefix removal transformation to remove all the events in $(\downarrow \Sigma_c)_\prec$. Because of (ii) and (iii), we know that any event in $(\downarrow\Sigma_c)_\prec$ fires before any event in Σ', thus it is possible to remove the prefix in all traces.

The timing relations in P are preserved because, by splitting at a nodal point, we guarantee that all timing relations between the preserved events still hold, since they are synchronized from that point on. Note that this only applies to timing relations whose context contains at least one timing zone formed, exclusively, by preserved events. Otherwise the lineal equations that caused the timing relation are no longer present.

If, for all precedence $\mathsf{a} <_{\mathrm{lzy}} \mathsf{b}$, $\mathrm{context}(\mathsf{a} < \mathsf{b})$ contains a timing zone formed by events in Σ', then the obtention of L' does not require to perform a timing analysis on CS'. All the timing relations between the preserved events Σ' still hold because the LzCES is cut at a nodal point and all necessary lineal equations are also preserved, thus $<'_{\mathrm{lzy}}$ equals $<_{\mathrm{lzy}} \cap (\Sigma' \times \Sigma')$. By construction all the precedences in P are in this case, for this reason $P \subseteq <'_{\mathrm{lzy}}$. However, for all precedences in which no timing zone of its context is completely preserved, we need a timing analysis on CS' to determine if the precedences still hold in L'.

Several possible LzCES containing P can be obtained using this lemma. There are two reasons for this: first, the context selection of the precedences in P can modify the resulting LzCES; and second, for a given LzCES there can be many possible cutsets. Obviously, in the latter case we are interested in the cutset that produces a smaller LzCES. That is, that for any possible cutset Σ'_c, $\Sigma_c \subseteq (\Sigma'_c\uparrow)_\prec$.

EXAMPLE 13.1 *In Figure 13.2 (a) we can see a LzCES. Timing analysis reveals that* $\mathsf{c} <_{\text{lzy}} \mathsf{d}$. *The context of this precedence is* $\text{context}(\mathsf{c} <_{\text{lzy}} \mathsf{d}) = \{\{\mathsf{f} < \mathsf{b}\}, \{\mathsf{e} < \mathsf{b}\}\}$ *because in either of these two timing zones it is verified that* $\mathsf{c} <_{\text{lzy}} \mathsf{d}$ *(but not in timing zone* $\emptyset$*). If we use Lemma 13.1 with a context selection function* $\text{context}_s(\mathsf{c} <_{\text{lzy}} \mathsf{d}) = \{\mathsf{f} < \mathsf{b}\}$ *then we obtain LzCES in (b). On the other hand, by using* $\text{context}_s(\mathsf{c} <_{\text{lzy}} \mathsf{d}) = \{\mathsf{e} < \mathsf{b}\}$ *we obtain the smaller LzCESs shown in (c,d). For instance, in this latter cases,* $\Sigma_p = \{\mathsf{c}, \mathsf{d}, \mathsf{b}, \mathsf{e}\}$ *and the difference between the two is caused by a different cutset selection. For (c), the cutset* Σ_c *is formed by the trivial set* $\{\epsilon\}$*, while in (d)* $\Sigma_c = \{\mathsf{y}\}$*. As* $\{\mathsf{y}\} \subseteq (\{\epsilon\}\uparrow)_{\prec}$ *we prefer this latter option, as we know it provides a smaller LzCES.* ■ 13.1

However, the use of the lemma does not guarantee to find the minimal LzCES. There are also other possibilities of LzCES that ensure that all failure traces go through its entry point, *but* do not guarantee that the timing relations in P are preserved. These LzCES are generated similarly to the previous ones, but relaxing the conditions of (iii) to simply guarantee that prefix removal is possible for all traces. In this case a timing analysis has to be performed to check if all the precedences in P still hold, as it is no longer guaranteed. If they do not hold, then it is not a valid minimal LzCES.

Let C_f be the set of all configurations, in the GRC that represents the zone, that either contain a failure state or are the source of a failure arc. Since we do not have to guarantee that all traces in LCS go through the entry point of L, only the ones leading to failure configurations, we can refine the precedence relation of the underlying CESAR of LCS by adding the common partial order of the orders of the traces going through each $\mathcal{C}$ in C_f. That is, if $<_{\mathcal{C}}$ is the partial order respected by all traces crossing $\mathcal{C}$, then we can add $\bigcap_{\forall \mathcal{C} \in C_f} <_{\mathcal{C}}$ to the precedence relation. This can lead to closer cutsets, thus to smaller LzCES.

LEMMA 13.2

Given a LzCES $L = \langle \text{CS}, <_{\text{lzy}}, \text{context}\rangle$ *with* $\text{CS} = \langle \Sigma, \prec, \triangleright, <_{\text{CS}}\rangle$*, a subset* P *of the relative timing precedences in* $<_{\text{lzy}}$ *and a context selection function* context_s *and a partial order* $<_f$ *satisfied by all traces of the zone of* CS *reaching a failure (*i.e. $<_f \supseteq <_{\text{CS}}$*), then the LzCES* $L' = \langle \text{CS}', <'_{\text{lzy}}, \text{context}'\rangle$*, obtained after the timing analysis of CESAR* $\text{CS}' = \langle \Sigma', \prec', \triangleright', <'_{\text{CS}}\rangle$*, contains* P *if:*

- $P \subseteq <'_{\text{lzy}}$.
- $\Sigma' = (\downarrow \Sigma_p)_{\prec} \setminus (\downarrow \Sigma_c)_{<_f}$ *where* $\Sigma_p = \{\mathsf{e} \in \Sigma \mid \mathsf{a} <_{\text{lzy}} \mathsf{b} \in P \wedge ((\mathsf{a} = \mathsf{e} \vee \mathsf{b} = \mathsf{e}) \vee (\mathsf{c} < \mathsf{d} \in \text{context}_s(\mathsf{a} <_{\text{lzy}} \mathsf{b}) \wedge (\mathsf{c} = \mathsf{e} \vee \mathsf{d} = \mathsf{e})))\}$ *and* $\Sigma_c \subseteq \Sigma$ *such that*
  - *(i)* $\forall \mathsf{e}, \mathsf{e}' \in \Sigma_c, \quad \mathsf{e} \not<_f \mathsf{e}'$. i.e. *All events in* Σ_c *are concurrently fired in* $<_f$.
  - *(ii)* $\forall \mathsf{e} \in \Sigma_c, \forall \mathsf{e}' \in \sup((\downarrow \Sigma_p)_{\prec} \setminus (\Sigma_c\uparrow)_{<_f}, <_f), \quad \mathsf{e}' <_f \mathsf{e}$. i.e. *All relevant events that are not fired after the events in* Σ_c*, must fire before them.*
  - *(iii)* $\forall \mathsf{e} \in \Sigma_c, \forall \mathsf{e}' \in \inf(\Sigma', <_f), \quad \mathsf{e} <_f \mathsf{e}'$. i.e. *All preserved events in* Σ' *must fire after the events in* Σ_c.
- *Relations* $\prec'$, $\triangleright'$ *and* $<'_{\text{CS}}$ *are equal to their* $\prec$, $\triangleright$ *and* $<_{\text{CS}}$ *counterparts, but restricted to the events in* Σ'.
- *All events in* Σ' *that do not become enabled because of other events in* Σ' *or events in* Σ_c*, must be promoted to conservative events. Formally, any* e' *in* $\inf(\Sigma', \prec)$ *is promoted to conservative event if exists* $\mathsf{e} \in \Sigma_c$ *such that* $\mathsf{e} \not\prec_{\mu} \mathsf{e}'$.

■ 13.2

Proof: *The proof is identical to the proof of Lemma 13.1, but easier, since it is not required that the entry point is now a synchronization (nodal) point. Equation in (iii), although it is*

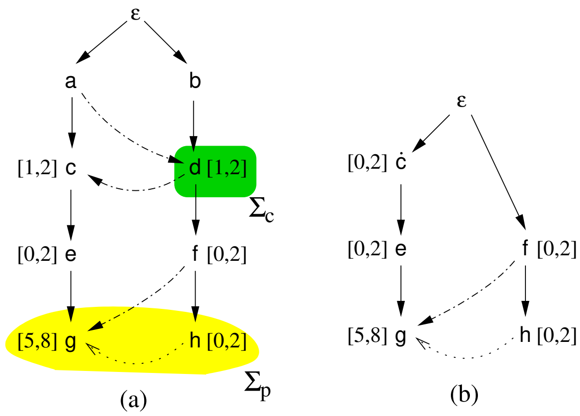


Figure 13.3. (a) LzCES obtained from a mutual exclusion protocol, in which we want to compute the minimal LzCES that preserves the precedence $\mathsf{h} <_{\text{lzy}} \mathsf{g}$. (b) The minimal LzCES obtained using cutset $\{\mathsf{d}\}$. Event c is now conservative because not all its causal predecessors were preserved or in the cutset.

a relaxed version of the one in Lemma 13.1, still guarantees –together with (ii)– that any event in $(\downarrow \Sigma_c)_{\prec}$ fires before any event in Σ' for any of the failure traces, so prefix removal is possible.

Precisely because Lemma 13.2 is, in some aspects, a relaxed version of Lemma 13.1, we cannot guarantee that precedences in P are preserved in L', thus we must incorporate the checking explicitely ($P \subseteq <'_{\text{lzy}}$). This implies that a timing analysis is required for each candidate LzCES.

The partial order $<_f$, as explained before, is computed by intersecting the partial orders respected by all traces reaching a failure configuration or arc. These latter orders can be obtained from $<_{\text{CS}}$ and the partial order described in Definition 13.3. For instance, given a failure configuration $\mathcal{C}$, let $<'_{\mathcal{C}}$ be the order obtained by using Definition 13.3, then $<_{\mathcal{C}} = (<_{\text{CS}} \cup <'_{\mathcal{C}})^*$.

EXAMPLE 13.2 *In Figure 13.3 (a) we can see a LzCES coming from a mutual exclusion protocol example (although event names have been changed for readability). Events* a *and* b *are unbounded. In this case the precedence we want to preserve is* $\mathsf{h} <_{\text{lzy}} \mathsf{g}$. *The context of this precedence is* $\{\mathsf{a} < \mathsf{d}, \mathsf{d} < \mathsf{c}, \mathsf{f} < \mathsf{g}\}$*, however, the precedence does not require all these restrictions to be satisfied. Doing a timing analysis considering the subsets of the context, we find out that restriction* $\mathsf{f} < \mathsf{g}$ *is enough to force* $\mathsf{h} <_{\text{lzy}} \mathsf{g}$.

Using $\Sigma_p = \{\mathsf{f}, \mathsf{g}, \mathsf{h}\}$*, the first possible cutset, according to Lemma 13.2, is* $\{\mathsf{d}\}$*. The resulting LzCES is shown in subfigure (b). Note that event* c *becomes conservative because one of its causal predecessors (event* a*) is not preserved. Timing analysis shows that* $\mathsf{h} <_{\text{lzy}} \mathsf{g}$ *holds, so this is the minimal LzCES.* ■ 13.2

Previous example introduces a new element to consider: the dependencies between restrictions allow to avoid computing an exponential number of timing analysis (§8.4.1), however this also can hide which are the minimal set of restrictions that produce a particular precedence. Should we have these minimal sets, we would simply use them as Σ_p and we would find candidate cutsets. Nevertheless both problems are related and can be solved more efficiently using a combined approach.

The idea is to compute the closest cutset to the events in Σ_p, with Σ_p only containing the events in the precedences that we want to preserve. Once the cutset is found, we do a first simplification:

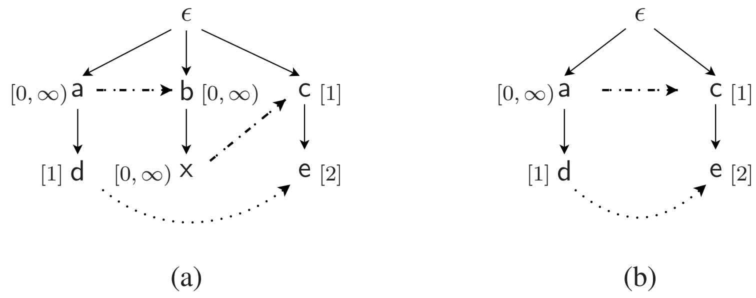


Figure 13.4. Assume we want to preserve $\mathsf{d} <_{\mathrm{lzy}} \mathsf{e}$ in the LzCES of subfigure (a). Its context is $\mathrm{context}(\mathsf{d} <_{\mathrm{lzy}} \mathsf{e}) = \{\{\mathsf{a} < \mathsf{b}, \mathsf{x} < \mathsf{c}\}\}$, thus the first impression is that the LzCES cannot be simplified. However, a more careful analysis reveals that they key restriction is $\mathsf{a} < \mathsf{c}$, which is obtained by transitivity, since this restrictions allows to supress events b and x and obtain a smaller LzCES preserving $\mathsf{d} <_{\mathrm{lzy}} \mathsf{e}$ (in subfigure (b)).

- we remove everything "above" the cutset (included), and
- we iteratively remove all sink events that are not the origin or destination of a restriction or disabling arc until no further modification is possible.

This simplified CESAR is timing analyzed, if the relevant precedences do not hold, then we find a previous cutset and the process starts again.

Eventually for some cutset the required precedences will hold. Then we perform a second simplification in which we will determine which restrictions are really needed and produce the smallest LzCES. If this information is available the simplification process is simple, it just consists in removing any event that is not a causal predecessor of the events in Σ_p or the essential restrictions (obviously including Σ_p and the events in the restrictions). Now we focus on how we can determine which restrictions are needed. For this task we use the context information provided by the timing analysis. This might give several options for each precedence, so each one has to be tested (in conjunction with the other possibilities of the rest of precedences, if they have more than one option) to see which combination produces the smallest LzCES. The number of combinations is potentially very high, because not only the direct restrictions appearing in the LzCES have to be considered and also the restrictions that are implicit by transitivity must be taken into account. We illustrate this in Figure 13.4. Fortunately in most of the cases the LzCES are quite small after previous simplification phases, thus even an exhaustive search provides reasonable running times.

13.3.2 Classifying LzCES based on convergence criteria

Given a zone represented with a GRC, we can detect pairs of configurations $(\mathcal{C}_i, \mathcal{C}_j)$ such that $\mathcal{C}_i \subset \mathcal{C}_j$ and in which a state is repeated. This allows to reveal when the LzCES is already working with a loop unfolding or will produce one. As in §12.8, we consider a state to be repeated when it is equal to another state after abstracting all the newly introduced state variables by compositions (that is preserving only the original state variables).

To find the pairs of conflicting configurations, first of all we do only consider the configurations visited by some valid trace (according to the $<_{\mathrm{lzy}}$ in the LzCES). We explore these configurations in a BFS manner and we accumulate the abstracted states from incoming configurations into the one we are processing. If at some point the states of the current configuration intersect with any of the accumulated sets in the incoming configurations, then some repeated states have been found (and we have to backtrack until we find the configurations that contain these states in their mapped set of states –not the accumulated one).

Once the pairs of conflicting configurations are available, if the LzCES contains any of the events that differentiate any of these pairs (*i.e.* given a pair $(\mathcal{C}_i, \mathcal{C}_j)$, any event in $\mathcal{C}_j \setminus \mathcal{C}_i$), then the LzCES will preserve the unfolding.

To implement a bounding strategy we classify LzCES in three categories, based on their behavior with respect to unfoldings.

Class 1 If they do not contain any unfolding.

Class 2 If they contain unfoldings, but they are all bounded. That is, there is a distinct event enabled in all the configurations sharing a state that has a non-infinite upper bound delay, and at least one of the events that are different in the configurations has a lower bound greater than 0. Formally, for each pair $(\mathcal{C}_i, \mathcal{C}_j)$ in which the unfolding is preserved, an event in $\mathcal{E}(\mathcal{C}_i) \cap \mathcal{E}(\mathcal{C}_j)$ has a finite upper bound delay and an event in $\mathcal{C}_j \setminus \mathcal{C}_i$ has a lower bound greater than 0.

Class 3 If they contain an unbounded unfolding.

13.4 The *Generate consistent fragment* activity

If the relative timing information contained in the LzCES shows that some of the failures in the zone cannot be avoided, then, by Theorem 11.2, we know that these unavoided failures are reachable through a timing consistent trace. In this section we deal with the problem of actually finding one of these traces.

A trivial solution is to check the timing consistency of all valid traces of the LzCES. However, there can be many traces and a more efficient approach would be desirable. To reduce the number of problems to solve, we use the observation that each LP problem can test the timing consistency of more than one trace, as long as all the traces have the same triggers.

Figure 13.5 shows the corresponding activity diagram. First, an unavoided failure of the zone is selected and we choose the trigger for all the events that have more than one direct causal predecessor (trigger sets). This selection takes into account the $<_{\mathrm{lzy}}$ to avoid proposing trigger combinations that we already know are impossible[2]. For each trigger selection, an LP problem is built that tests the timing consistency of all the traces that reach the failure and satisfy the given trigger selection. If the problem is unfeasible, we discard the set. Otherwise, we can build a trace fragment from the solutions and we use this timing consistent fragment to replace the corresponding original timing inconsistent fragment.

Although our implementation is iterative (one set is tested after the other), we marked the expansion region as *parallel* because there are no dependencies between the different iterations and it could be implemented this way.

13.4.1 Timing-consistency of sets of traces through a failure

In §5.3.3 we showed how the timing consistency problem for a single trace could be transformed into a LP problem. However, a LP problem is capable of verifying more than one trace at a time. Each LP problem is formed by two sets of inequations (Lemma 5.2), one that enforces each firing event in the trace to have smaller firing time than its successor (sequence order equations),

[2] In fact the best option is to consider the order $<_{\mathrm{TZ}}= \{(\mathsf{a}, \mathsf{b}) \mid \mathrm{mst}_{\mathrm{twd}(\mathrm{CS}_{\mathrm{TZ}})}(\mathsf{b}, \mathsf{a}) < 0\}$, where TZ is the timing zone of the selected failure, instead of $<_{\mathrm{lzy}}$. The reason is that, by Theorem 11.2, we know that the former order is satisfied by all timing consistent traces reaching an enabling of the timing zone. Note that $<_{\mathrm{TZ}}$ might include orderings not present in $<_{\mathrm{lzy}}$, in particular between events fired before entering the timing zone.

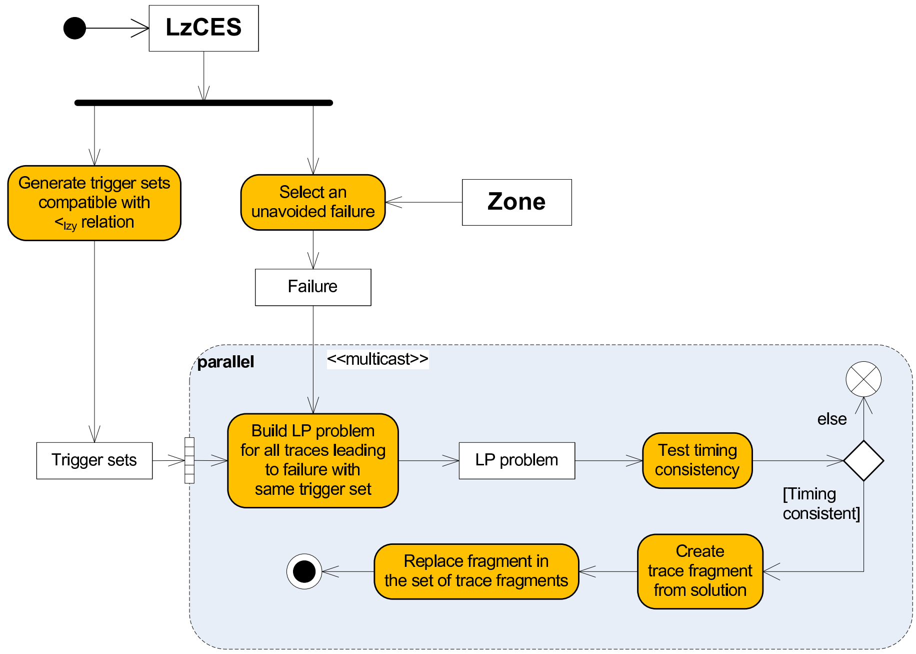


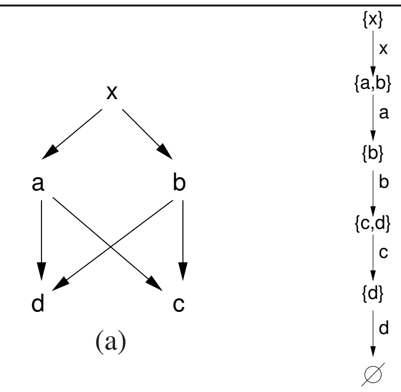
Figure 13.5. Activity diagram of the *Generate consistent fragment* activity.

and the other that deals, broadly speaking, with causal relationships between events (causal order equations). However, by removing some of the restrictions in the first set, we can check if a set of traces is timing consistent, as long as we respect the causal relationships between events described by the second set.

EXAMPLE 13.3 *In Figure 13.6 (a) we can see a* CESAR*. Assume this* CESAR *describes* faithfully *a* LzTS, i.e. *its zone contains all the state space of the* LzTS*. In subfigure (b) a possible trace of this* CESAR *is shown, and the corresponding LP problem appears in subfigure (c). In this LP problem we can replace the first sequence of inequalities by just* $\mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{b})$ *and then our new LP problem, in subfigure (d), will check the timing consistency of the set of traces depicted in subfigure (e). This means that if a solution to that problem is possible, either one of the two traces is possible, while if the problem is unsolvable, then both traces are timing inconsistent.* ■ 13.3

In the example we cannot remove the condition $\mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{b})$ because, otherwise, we would allow a to fire after b, but then events c and d would not be triggered by b. Thus the equations $\delta^l(\mathsf{c}) \leq \mathrm{ft}(\mathsf{c}) - \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{c})$ and $\delta^l(\mathsf{d}) \leq \mathrm{ft}(\mathsf{d}) - \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{d})$ would not describe the real causal relationship between events, and the LP problem will not adjust to the CESAR we started from. As firing times depend on the trigger events, we must guarantee that only one causal predecessor for each event is chosen as the triggering event. Besides that, no other order is enforced, thus allows to represent the timing consistency of more than one trace.

In this section we will create an LP problem to test the timing consistency of all the traces with the same triggers that go trough a given failure. To do so, we use the causal order equations of Lemma 5.2 to whom we add several orderings that events must respect. One of the orderings ensures that all events have the same triggers, and another that only the traces going trhough the failure are examined.



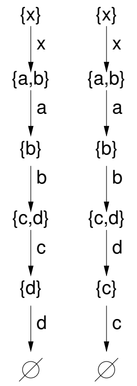
(a) (b)

$$\begin{gathered}
\mathrm{ft}(\mathsf{x}) \leq \mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{b}) \leq \mathrm{ft}(\mathsf{c}) \leq \mathrm{ft}(\mathsf{d}) \\
0 \leq \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{x}) \\
\delta^l(\mathsf{a}) \leq \mathrm{ft}(\mathsf{a}) - \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{a}) \\
\delta^l(\mathsf{b}) \leq \mathrm{ft}(\mathsf{b}) - \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{b}) \\
\delta^l(\mathsf{c}) \leq \mathrm{ft}(\mathsf{c}) - \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{c}) \\
\delta^l(\mathsf{d}) \leq \mathrm{ft}(\mathsf{d}) - \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{d})
\end{gathered}$$

(c)

$$\begin{gathered}
\mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{b}) \\
0 \leq \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{x}) \\
\delta^l(\mathsf{a}) \leq \mathrm{ft}(\mathsf{a}) - \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{a}) \\
\delta^l(\mathsf{b}) \leq \mathrm{ft}(\mathsf{b}) - \mathrm{ft}(\mathsf{x}) \leq \delta^u(\mathsf{b}) \\
\delta^l(\mathsf{c}) \leq \mathrm{ft}(\mathsf{c}) - \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{c}) \\
\delta^l(\mathsf{d}) \leq \mathrm{ft}(\mathsf{d}) - \mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{d})
\end{gathered}$$

(d)



(e)

Figure 13.6. Example of testing the timing consistency of a set of traces with a single LP problem: (a) CESAR that represents a set of traces of a system. (b) One of its traces. (c) LP problem associated to the trace. (d) Another LP problem in which the first row of inequalities has been substituted by $\mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{b})$. Now it tests the timing consistency of the two traces in (e).

DEFINITION 13.2 (ORDER IMPOSED BY TRIGGER SELECTION)

Given a LzCES $\mathrm{LCS} = \langle \mathrm{CS}, <_{\mathrm{lzy}} \rangle$ *with* $\mathrm{CS} = \langle \Sigma, \prec, \triangleright, <_{\mathrm{CS}} \rangle$, *and a function* $\mathrm{trigger}_{\mathrm{CS}}$ *that select the triggers of the events that can be triggered by more than one event, we define the relation* $<_{\mathrm{trigger}_{\mathrm{CS}}}$ *as the order that forces all the events in* Σ *to have the trigger imposed by* $\mathrm{trigger}_{\mathrm{CS}}$.

Let us use the term $\mathrm{dpr}(\mathsf{e})$ *to identify the set of events that are direct causal predecessors of* e, i.e. $\{\mathsf{e}_j \mid \mathsf{e}_j \prec_\mu \mathsf{e}\}$. *We introduce a new term,* $\mathrm{dpr}_{\mathrm{conc}}(\mathsf{e})$, *that corresponds to the subset of* $\mathrm{dpr}(\mathsf{e})$ *such that, for all events* e' *in* $\mathrm{dpr}_{\mathrm{conc}}(\mathsf{e})$, *event* e' *is not forced by* $<_{\mathrm{lzy}}$ *to fire before any other event in the set. Formally,* $\mathrm{dpr}_{\mathrm{conc}}(\mathsf{e}) = \{\mathsf{e}' \in \mathrm{dpr}(\mathsf{e}) \mid \nexists \mathsf{e}'' \in \mathrm{dpr}(\mathsf{e}) : \mathsf{e}' <_{\mathrm{lzy}} \mathsf{e}''\}$. *We call this set the set of concurrent direct causal predecessors of* e.

Let Σ_m *be the set of all the events with more than one concurrent direct causal predecessor,* i.e. $\Sigma_m = \{\mathsf{e} \in \Sigma \mid \; |\mathrm{dpr}_{\mathrm{conc}}(\mathsf{e})| > 1\}$. *Formally,* $\mathrm{trigger}_{\mathrm{CS}} : \Sigma_m \rightarrow \Sigma$ *is a function that, for each one of the events in* Σ_m, *selects one of their concurrent direct causal predecessors as the actual trigger. This function must satisfy that events with the same set of concurrent direct causal predecessors select the same trigger, that is* $\forall \mathsf{e}, \mathsf{e}' \in \Sigma_m, (\mathrm{dpr}_{\mathrm{conc}}(\mathsf{e}) = \mathrm{dpr}_{\mathrm{conc}}(\mathsf{e}')) \Rightarrow (\mathrm{trigger}_{\mathrm{CS}}(\mathsf{e}) = \mathrm{trigger}_{\mathrm{CS}}(\mathsf{e}'))$.

To force that all the events in Σ *have a single possible trigger specified by* $\mathrm{trigger}_{\mathrm{CS}}$*, for each event with multiple concurrent direct causal predecessor, the selected trigger event must fire later than any other of its causal predecessors. We use the term* $<_{\mathrm{trigger}_{\mathrm{CS}}}$ *to refer to this order, that we formally define as follows:*

$$<_{\mathrm{trigger}_{\mathrm{CS}}} = \{(\mathsf{e}_i, \mathrm{trigger}_{\mathrm{CS}}(\mathsf{e})) \mid \mathsf{e} \in \Sigma_m \wedge \mathsf{e}_i \in \mathrm{dpr}_{\mathrm{conc}}(\mathsf{e}) \wedge \mathsf{e}_i \neq \mathrm{trigger}_{\mathrm{CS}}(\mathsf{e})\}$$

■ 13.2

We also need to be able to check only the traces that go through a failure. Because there can be different types of failures in the zone, this involves several cases:

Definition 13.3 (Order of traces going through a failure)
Given a ***GRC*** *representation of a zone* $\langle \mathrm{states}_{in}, \mathrm{states}_{pa}, \mathrm{states}_{out}, \mathrm{trans}, G\rangle$ *with* $G = \langle C, \Sigma, T_G, \top\rangle$*, and a configuration* $\mathcal{C} \in C$ *that contains a failure state* s_f *or a transition* $\mathcal{C}_i \xrightarrow{\mathsf{e}_f} \mathcal{C}_j \in T_G$ *that contains a failure arc* $\mathsf{s}_1 \xrightarrow{\mathsf{e_f}} \mathsf{s}_2$*, all the traces going through the failure satisfy order* $<_{\mathrm{f}}$ *defined as follows:*

(i) If the failure is a failure arc, then

$$<_{\mathrm{f}} = \{(\mathsf{e}, \mathsf{e}_f) \mid \exists \mathcal{C}_k \in C : \mathcal{C}_k \xrightarrow{\mathsf{e}} \mathcal{C}_i \in T_G\} \cup \{(\mathsf{e}_f, \mathsf{e}) \mid \exists \mathcal{C}_m \in C : \mathcal{C}_j \xrightarrow{\mathsf{e}} \mathcal{C}_m \in T_G\}$$

(ii) If failure is a failure state, then

(a) if $\mathsf{s}_f \in \mathrm{states}_{out}(\mathcal{C})$*, then*

$$<_{\mathrm{f}} = \{(\mathsf{e}, \mathsf{e}') \mid \exists \mathcal{C}_k, \mathcal{C}_m \in C : \mathcal{C}_k \xrightarrow{\mathsf{e}} \mathcal{C}, \mathcal{C} \xrightarrow{\mathsf{e}'} \mathcal{C}_m \in T_G\}$$

(b) however, if $\mathsf{s}_f \in \mathrm{states}_{out}(\mathcal{C}) \setminus \mathrm{states}_{pa}(\mathcal{C})$*, then there are less valid paths to reach* s_f *and additional orderings must be imposed. So, in addition to the inequalities in (a), let* ArcsToFailure *be all the arcs of the* ***GRC*** *that reach* s_f*. Formally,* $\mathrm{ArcsToFailure} = \{\mathcal{C}_k \xrightarrow{\mathsf{e}} \mathcal{C} \in T_G \mid \exists \mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}_f \in \mathrm{trans}(\mathcal{C}_k \xrightarrow{\mathsf{e}} \mathcal{C})\}$*.Then the arcs reaching the failure must fire later than the ones not reaching it:*

$$<_{\mathrm{f}} = <_{\mathrm{f}} \cup \{(\mathsf{e}', \mathsf{e}) \mid \exists \mathcal{C}_k \xrightarrow{\mathsf{e}} \mathcal{C} \in \mathrm{ArcsToFailure} \wedge \exists \mathcal{C}_l \xrightarrow{\mathsf{e}'} \mathcal{C} \in T_G \setminus \mathrm{ArcsToFailure}\}$$

■ 13.3

Definition 13.4 (Consistency of traces through a failure)
Given:

- *a* ***LzCES*** $\mathrm{LCS} = \langle \mathrm{CS}, <_{\mathrm{lzy}}\rangle$ *with* $\mathrm{CS} = \langle \Sigma, \prec, \triangleright, <_{\mathrm{CS}}\rangle$*,*
- *a trigger function* $\mathrm{trigger}_{\mathrm{CS}}$ *as defined in Definition 13.2,*
- *a* ***GRC*** *representation of a zone* $\langle \mathrm{states}_{in}, \mathrm{states}_{pa}, \mathrm{states}_{out}, \mathrm{trans}, G\rangle$ *with* $G = \langle C, \Sigma, T_G, \top\rangle$*, and*
- *a failure of the zone,*

let $<_{\mathrm{trigger}_{\mathrm{CS}}}$ *and* $<_{\mathrm{f}}$ *be the orders described in Definition 13.2 and Definition 13.3. Then, the LP problem that tests the timing consistency of all the traces* θ *of the zone that:*

- *go through the given failure and,*
- *for all events* e *where there is more than one possible trigger, they share the same triggers so that* $\mathrm{trigger}(\theta, \mathsf{e}) = \mathrm{trigger}_{\mathrm{CS}}(\mathsf{e})$*,*

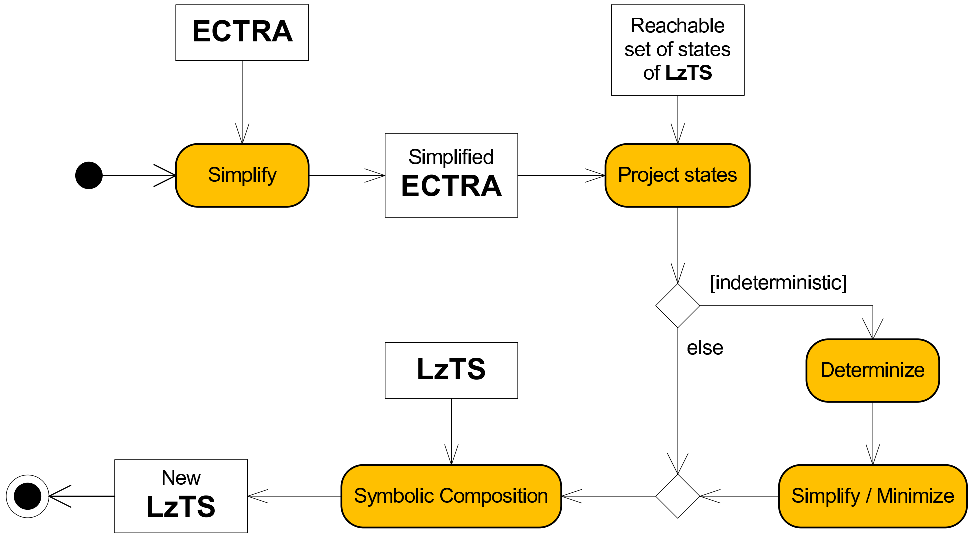


Figure 13.7. The *Compose* activity.

is composed by the following set of inequalities:

(i) For each event e *in the alphabet* Σ*, add the corresponding inequality from Table 5.1.*

(ii) $\forall(\mathsf{e}_i, \mathsf{e}) \in (<_{\mathrm{lzy}} \cup <_{\mathrm{trigger}_{\mathrm{CS}}} \cup <_{\mathrm{f}}) : \mathsf{e}_i \not\prec \mathsf{e}, \quad \mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{e})$

■ 13.4

If this LP problem is solvable, a trace can be generated from the firing times obtained. Obviously, for any two events e and e' such that $\mathrm{ft}(\mathsf{e}) < \mathrm{ft}(\mathsf{e}')$ we know that e happens before e' in the trace. However, many events can have the same firing time which does not provide an immediate information of the firing order of the events. In such case one must be careful to respect all the orderings imposed by the $(<_{\mathrm{lzy}} \cup <_{\mathrm{trigger}_{\mathrm{CS}}} \cup <_{\mathrm{f}})^*$ relation.

13.5 The *Compose* activity

Once an ECTRA has been selected for composition, several steps have to be performed before the actual composition can take place. The activity diagram of Figure 13.7 shows these steps.

First of all we simplify the ECTRA. This is a lightweight alternative to minimization, in which

- all lazy arcs, and
- all arcs that go to a configuration from which we cannot reach (unless by reenter arcs) a lazy arc,

are redirected to $\perp$ configuration. After this, all configurations no longer reachable from $\top$ can be removed.

Next step is to perform a projection of the composed states, as explained in §10.6.3. This allows to remove most of the false indeterminacy without the cost of an actual composition. Then we check if some indeterminacy remains. In such case the determinization procedure of §10.4 is invoked. Depending on the size of the resulting ECTRA we can simplify again or minimize (§10.5) it.

Finally, the ECTRA obtained after all this steps is composed with the LzTS as explained in §6.6. In our case, this composition is symbolical (in the sense that the automata involved are represented symbolically). To keep the number of new variables introduced low, some specific details have to be considered. We cover them, together with some other optimization techniques in §14.1.

The composition yields a new LzTS in which the timing information contained in the ECTRA has been incorporated. This new LzTS is the starting point of a new refinement iteration as shown in Figure 13.1.

Chapter 14

PRACTICAL CONSIDERATIONS AND OPTIMIZATIONS

Summary

Good ideas might fail due to a careless implementation. In this chapter we focus our attention in how the symbolic composition of an ECTRA and a LzTS, a crucial step in our methodology, can be performed. As the results in Chapter 15 show, the way in which it is implemented can have a significant impact in the overall performance.

The other big subject we discuss here are optimizations of the methodology introduced in Chapter 13. Because of its complexity there are plenty of opportunities to incorporate optimizations, but we restrict our discussion to the addition of a prephase that with simple reachability analysis techniques can discharge the methodology from the most trivial timing inconsistent traces.

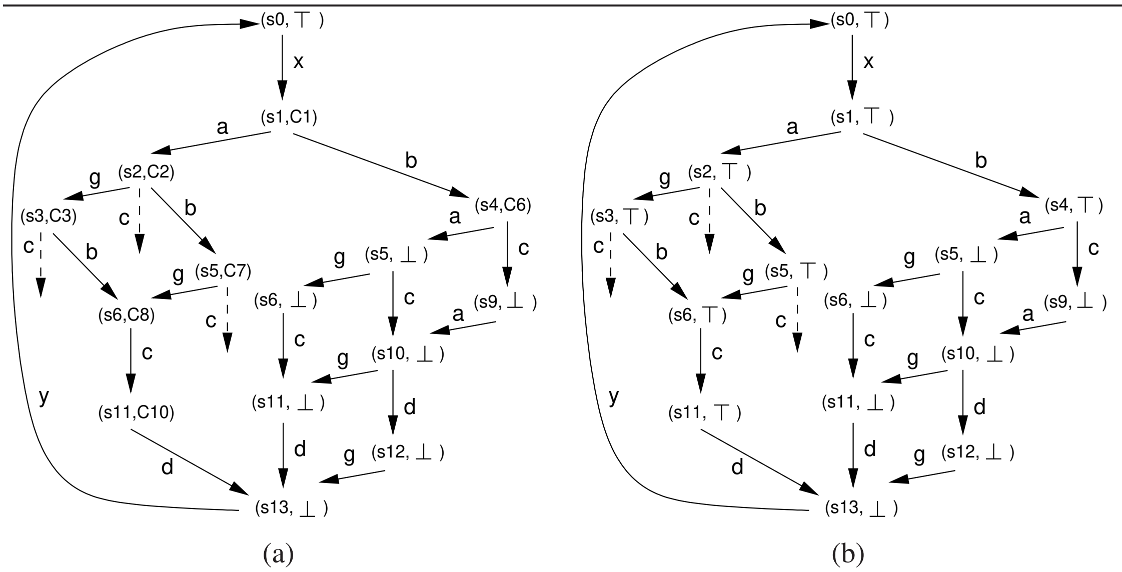


Figure 14.1. Example of possible encodings for symbolic composition: (a) Composed LzTS of Figure 5.3 –page 66. It requires at least three new Boolean variables to be able to encode configurations $\{\perp, \top, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_6, \mathcal{C}_7, \mathcal{C}_8\}$. (b) The same composed LzTS but using an alternative encoding in which a single Boolean variable is used.

14.1 System composition in a symbolic setting

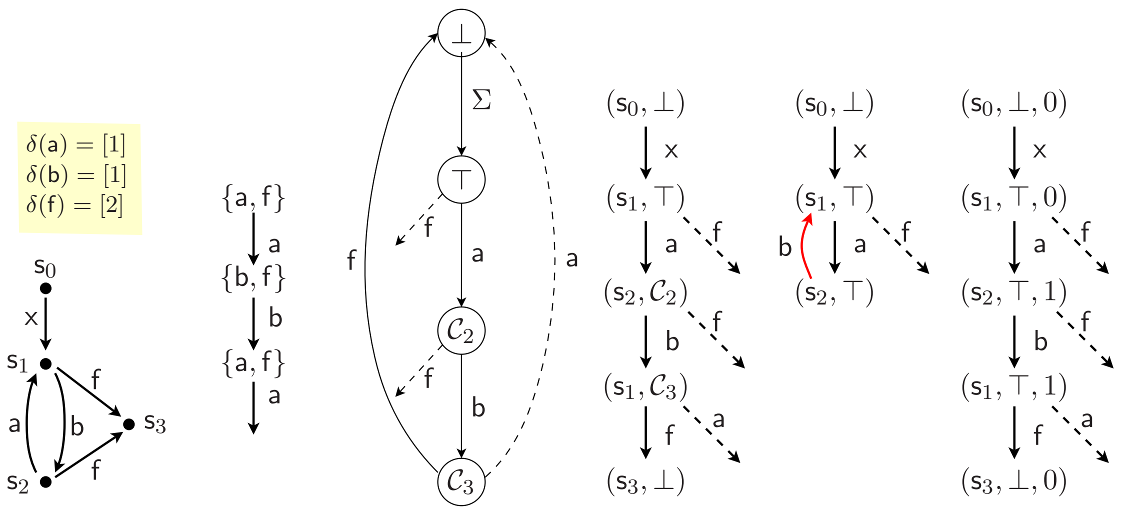
In §6.6 we presented the composition of a LzTS and an ECTRA. In this composition, the state space of the LzTS and the state space of the ECTRA (*i.e.* the configurations) are merged. However, if this approach is directly translated to a symbolic setting, it requires to introduce a number of new Boolean variables to encode all the configurations. This could be a limiting factor of the methodology, since the number of variables a BDD package can efficiently handle is on the order of the hundreds.

On the other hand it is possible to take advantage of the structure of the transition relations of the LzTS to reduce the number of variables that have to be introduced. In fact, given that the state space of the ECTRA is an abstraction of a part of the state space of the LzTS, in many cases only a single Boolean variable is enough to implement the composition. This new variable will differentiate the timed states (the states that are affected by the composition) from the rest.

EXAMPLE 14.1 *In Figure 14.1 we show the standard encoding we have used in the previous parts of this work (a), and the encoding that is implemented in practice. Note that three Boolean variables are required in the first case to encode all the relevant configurations of the ECTRA. In fact this implies to add six new Boolean variables as also next-state variables must be introduced. However, only one Boolean variable (two considering also the next-state variables) is needed using the encoding of (b).* ■ 14.1

Note that this "reduced" encoding is only possible if no transition of the LzTS appears in different arcs of the ECTRA. In such case, typically arising when loops are unfolded, we need to add more variables to distinguish between these transitions.

EXAMPLE 14.2 *In Figure 14.2 (a) we can see a TTS, and in (b) a fragment of a timing inconsistent trace of the system. If we build an ECTRA to remove it, and we do a multiple trace analysis (§6.5.2) so that other timing inconsistent are also removed, we obtain the ECTRA of (c). Composing it with the original system we obtain LzTS of (d), given that we use an encoding for*



(a) (b) (c) (d) (e) (f)

Figure 14.2. (a) A TTS. (b) Fragment of a timing inconsistent trace. (c) Simplified depiction of an ECTRA that removes the previous timing inconsistent trace and some others (multiple trace analysis, §6.5.2). (d) Resulting LzTS after composition using distinguished configurations. (e) Resulting LzTS after composition using a single Boolean variable. Clearly, there is a problem since it is no longer possible to distinguish $(\mathsf{s}_1, \top)$ from $(\mathsf{s}_1, \mathcal{C}_3)$, and both states have to be different as the lazyness of events a and f is exchanged. (f) The problem can be solved by introducing a new Boolean variable that is set by one of the events (in this case a).

each configuration. However, if we try to simply use a single Boolean variable we obtain (e), in which we have a problem since states $(\mathsf{s}_1, \top)$ *and* $(\mathsf{s}_1, \mathcal{C}_3)$ *of (d) are now represented with only one state in (e). Nevertheless both states must be distinguished because the lazyness of events* a *and* f *is different in each one of the states. Thus, the only solution is to incorporate some additional variable.* ■ 14.2

An algorithm to determine if additional variables are needed –and to assign them in a way such that resulting TRs have suitable representations– is given in Algorithm 14.1. The key idea is to use the firing of events to set a variable. This way the other events can ignore the variable in many of the transitionts of their TRs.

EXAMPLE 14.2 (CONT.) *If we project the reachable set of states on the ECTRA of Figure 14.2 (c) we obtain the following mapping:*

$$
\begin{aligned}
\mathrm{states}(\bot) &= \{\mathsf{s}_0, \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3\} \\
\mathrm{states}(\top) &= \{\mathsf{s}_1\} \\
\mathrm{states}(\mathcal{C}_2) &= \{\mathsf{s}_2\} \\
\mathrm{states}(\mathcal{C}_3) &= \{\mathsf{s}_1\}
\end{aligned}
$$

Algorithm 14.1 will detect that configurations $\top$ *and* $\mathcal{C}_3$ *share states. Next, it checks for shared transitions starting at* s_1*. In this case, for instance,* $\mathsf{s}_1 \xrightarrow{\mathsf{f}} \mathsf{s}_3$ *forms part of the arc* $\top \xrightarrow{f} F$ *and of* $\mathcal{C}_3 \xrightarrow{f} \bot$ *(we do not show the full labels of the ECTRA arcs for readability), hence the algorithm puts the pair* $(\top, \mathcal{C}_3)$ *in the conflict list.*

Algorithm 14.1: Making configurations unique

input : $E = \langle C, T, T_E, \bot, \mathrm{FR} \rangle$ an ECTRA where all lazy arcs go to special configuration F.
input : states a function mapping the reachability set of a LzTS onto the configurations of E.
output : Σ' a set of events that have to be assigned an additional Boolean variable to make all configurations unique.

$C_{sel} \leftarrow C \setminus \{\bot, F\}$
Conflicts $\leftarrow \emptyset$
forall $\mathcal{C}_i \in C \setminus \{\bot, F\}$ **do**
  $C_{sel} \leftarrow C_{sel} \setminus \{\mathcal{C}_i\}$
  forall $\mathcal{C}_j \in C_{sel}$ **do**
    Shared $\leftarrow \mathrm{states}(\mathcal{C}_i) \cap \mathrm{states}(\mathcal{C}_j)$
    if Shared $\neq \emptyset$ **then**
      forall $\mathcal{C}_i \xrightarrow{\langle \mathsf{S}_{s1}, \Sigma_1, \mathsf{S}_{d1} \rangle} \mathcal{C}_k \in T_E$ **do**
        forall $\mathcal{C}_j \xrightarrow{\langle \mathsf{S}_{s2}, \Sigma_2, \mathsf{S}_{d2} \rangle} \mathcal{C}_m \in T_E$ **do**
          if Shared $\cap\, \mathsf{S}_{s1} \cap \mathsf{S}_{s2} \neq \emptyset$ *and* $\Sigma_1 \cap \Sigma_2 \neq \emptyset$ *and* $\mathsf{S}_{d1} \cap \mathsf{S}_{d2} \neq \emptyset$ *and* $\mathcal{C}_k \neq \mathcal{C}_m$ **then**
            Conflicts $\leftarrow$ Conflicts $\cup\, (\mathcal{C}_i, \mathcal{C}_j)$

if $|$ Conflicts $| = 0$ **then return** $\emptyset$
$\Sigma' \leftarrow \emptyset$
Candidates $\leftarrow \Sigma$ ▷ Candidates is assigned the full set of events
forall $(\mathcal{C}_i, \mathcal{C}_j) \in$ Conflicts **do**
  LocalCand $\leftarrow (\mathcal{C}_j \setminus \mathcal{C}_i) \cup (\mathcal{C}_i \setminus \mathcal{C}_j)$ ▷ Only events that distinguish the configurations are relevant
  if $\Sigma' \cap$ LocalCand $= \emptyset$ **then**
    NewCand $\leftarrow$ Candidates $\cap$ LocalCand ▷ Events able to solve the conflicts seen so far
    if NewCand $= \emptyset$ **then**
      $\Sigma' \leftarrow \Sigma' \cup \{$one of the elements of Candidates $\}$
      Candidates $\leftarrow$ LocalCand
    else
      Candidates $\leftarrow$ NewCand

if Candidates $\neq \emptyset$ **then** $\Sigma' \leftarrow \Sigma' \cup \{$one of the elements of Candidates $\}$
return Σ'

Finally, the difference between both configurations is $\{\mathsf{a}, \mathsf{b}\}$, *hence any of these two events can be used to set a new Boolean variable as shown in Figure 14.2 (f).* ■ 14.2

Note that the use of events to disambiguate configurations is only safe when the ECTRA has some specific structure. For instance Algorithm 14.1 assumes that all ECTRA arcs either remain in the same configuration or “advance” (go to a configuration with a greater BFS number) with the only exceptions of the arcs going to configurations $\top$ and $\bot$. This is the case for all the ECTRAs obtained by the construction procedures of §6.2 and §8.5. However, it is not guaranteed if the ECTRAs are determinized and minimized. In this latter case we use a more standard approach, which is to assign a new code to all conflicting configurations. The number of new variables required is then $\log_2$ of the number of codes required. We call this latter option *configuration disambiguation* as opposed to *event disambiguation*.

We develop the symbolic composition using event disambiguation because tends to produce smaller TRs and most of the ECTRAs satisfy the required conditions. However, once the equations have been presented, we give also their corresponding versions using configuration disambiguation.

14.1.1 Computing the symbolic composition using event disambiguation

To perform the composition in a symbolic setting means, in practice, to modify the characteristic functions of the TRs of the LzTS. In the following we detail how these modifications have to be performed and how they can be optimized to reduce the representation requirements of the TRs. As we use classical LzTSs in our implementation, and this is a question more related to implementation, we develop this section using classical LzTSs.

Given a Boolean encoded classical LzTS $A = \langle \mathcal{V}, \mathcal{V}', \Sigma, \mathrm{TR}, \mathcal{X}_{\mathsf{S}_{in}}, \mathrm{EF}\rangle$ and an ECTRA $E = \langle C, T, T_E, \perp, \mathrm{ER}_E\rangle$ in which the states of the reachability set are mapped to its configurations using the function states computed by Algorithm 10.2, then we define its symbolic composition $A \odot E$ as a Boolean encoded classical LzTS $A_c = \langle \mathcal{V}_c, \mathcal{V}_c', \Sigma, \mathrm{TR}_c, \mathcal{X}_{\mathsf{S}_{in},c}, \mathrm{EF}\rangle$ where,

- $\mathcal{V}_c = \mathcal{V} \cup \mathcal{U} \cup \{v_t\}$ with $v_t \notin \mathcal{V} \cup \mathcal{U}$ and $\mathcal{U} \cap \mathcal{V} = \emptyset$, where
  - $\mathcal{U}$ is the set of new Boolean variables assigned to the disambiguating events Σ' computed by Algorithm 14.1, so that $\mathcal{U} = \{u_{\langle \mathsf{e}, i\rangle} \mid \forall \langle \mathsf{e}, i\rangle \in \Sigma'\}$. Note that the events are distinct events.
  - Boolean variable v_t is a new variable used to indicate if the states are affected by the ECTRA or not.
- $\mathcal{V}_c' = \mathcal{V}' \cup \mathcal{U}' \cup \{v_t'\}$ with $v_t' \notin \mathcal{V}' \cup \mathcal{U}'$ and $\mathcal{U}' \cap \mathcal{V}' = \emptyset$, where $\mathcal{U}'$ and v_t' are the next-state versions of the previously introduced variables.
- $\mathcal{X}_{\mathsf{S}_{in},c} = \left(\mathcal{X}_{\mathsf{S}_{in}} \cdot v_t \cdot \mathcal{X}_{enter} \;+\; \mathcal{X}_{\mathsf{S}_{in}} \cdot \overline{v_t} \cdot \overline{\mathcal{X}_{enter}}\right) \cdot \prod_{\forall u \in \mathcal{U}} \overline{u}$, where

$$\mathcal{X}_{enter} = \begin{cases} \mathcal{X}_{\mathrm{SE}(\top)} & \text{if } \mathrm{LB}(\top) = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

- For all event e in Σ

$$\mathrm{TR}_c(\mathsf{e}) = \sum_{\forall \mathcal{C}_i \xrightarrow{\langle \mathsf{s}_a, \Sigma_d, \mathsf{s}_b\rangle} \mathcal{C}_j \in T_E : \exists k : \langle \mathsf{e}, k\rangle \in \Sigma_d} \mathcal{X}_{\mathsf{S}_a} \cdot \mathrm{TR}(\mathsf{e}) \cdot \mathcal{X}'_{\mathsf{S}_b} \cdot \mathrm{cond}_{\mathrm{extra}} \cdot \mathrm{cond}_{\mathrm{timed}} \tag{14.1}$$

  where,

  - $\mathrm{cond}_{\mathrm{extra}}$ is the formula that handles the additional variables used for disambiguation, defined as

$$\mathrm{cond}_{\mathrm{extra}} = \begin{cases} \overline{u_{\langle \mathsf{e}, k\rangle}} \cdot u'_{\langle \mathsf{e}, k\rangle} \cdot \prod_{\forall u \in \mathcal{U} \setminus \{u_{\langle \mathsf{e}, k\rangle}\}} (u = u') & \text{if } \mathcal{C}_j \notin \{\top, \perp\} \wedge \langle \mathsf{e}, k\rangle \in \Sigma' \\ \prod_{\forall u \in \mathcal{U}} (u = u') & \text{if } \mathcal{C}_j \notin \{\top, \perp\} \wedge \langle \mathsf{e}, k\rangle \in \Sigma' \\ \prod_{\forall u \in \mathcal{U}} \overline{u'} & \text{if } \mathcal{C}_j \in \{\top, \perp\} \end{cases}$$

    if $\Sigma' \neq \emptyset$, or simply 1 if $\Sigma' = \emptyset$. Note that most of the above formula is used to indicate that the variables keep their value. When using partial TRs this part is represented implicitely, so the modification of the TR due to these variables is really small.
  - $\mathrm{cond}_{\mathrm{timed}}$ is the formula that sets v_t to its correct value, defined as

$$\mathrm{cond}_{\mathrm{timed}} = \begin{cases} \overline{v_t} \cdot \overline{v_t'} & \text{if } \mathcal{C}_i = \perp \wedge \mathcal{C}_j = \perp \\ \overline{v_t} \cdot v_t' & \text{if } \mathcal{C}_i = \perp \wedge \mathcal{C}_j \neq \perp \\ v_t \cdot \overline{v_t'} & \text{if } \mathcal{C}_i \neq \perp \wedge \mathcal{C}_j = \perp \\ v_t \cdot v_t' & \text{if } \mathcal{C}_i \neq \perp \wedge \mathcal{C}_j \neq \perp \end{cases}$$

In the previous definition we assumed that the arcs of the ECTRA were labeled using distinct events. This is possible if the ECTRA satfies the conditions that allow to use event disambiguation.

14.1.2 Computing the symbolic composition using configuration disambiguation

Basically, the construction is the same with some minor modifications. For instance, let C' be the set of conflicting configurations and $\mathcal{U}$ the set of new Boolean variables such that $|\mathcal{U}| = \lceil \log_2 |C'| \rceil$. Let $\zeta(\mathcal{C})$ be the formula of the encoding assigned to $\mathcal{C}$ using the present-state variables, and $\zeta'(\mathcal{C})$ using the next-state ones. All the configurations in C' have different codes, and for the rest of configurations, one of the codes is assigned to all of them. Note that $\zeta(\perp)$ identifies this code, as this configuration can never be a conflicting one.

Now initial state is $\mathcal{X}_{\mathsf{S}_{in},c} = \mathcal{X}_{\mathsf{S}_{in}} \cdot v_t \cdot \mathcal{X}_{enter} \cdot \zeta(\top) \;+\; \mathcal{X}_{\mathsf{S}_{in}} \cdot \overline{v_t} \cdot \overline{\mathcal{X}_{enter}} \cdot \zeta(\perp)$ and TR_c is computed as follows:

$$\mathrm{TR}_c(\mathsf{e}) = \sum_{\forall \mathcal{C}_i \xrightarrow{\langle \mathsf{S}_a, \Sigma_d, \mathsf{S}_b \rangle} \mathcal{C}_j \in T_E : \mathsf{e} \in \Sigma_d} \mathcal{X}_{\mathsf{S}_a} \cdot \zeta(\mathcal{C}_i) \cdot \mathrm{TR}(\mathsf{e}) \cdot \mathcal{X}'_{\mathsf{S}_b} \cdot \zeta'(\mathcal{C}_j) \cdot \mathrm{cond}_{\mathrm{timed}}. \tag{14.2}$$

Note that, contrary to what happens in §14.1.1, here the events of the ECTRA cannot be distinct, which involves some minor changes in the formulas.

14.1.3 Optimization of the TRs

Equations (14.1) and (14.2) allow to compute the right TRs to achieve the desired composition effect. However, the resulting expressions are not minimial *i.e.* smaller TRs exist that would yield the same effect. We illustrate this with the following example.

EXAMPLE 14.3 *In Figure 14.3 (a) we show a LzTS and (b) contains its encoding. Subfigure (c) shows the equations of the TRs (in full form to avoid the complexity of performing the computation using the partial TRs) together with their enabling functions, while the ECTRA we want to compose with the system appears in (d). Take, for instance, the computation of* $\mathrm{TR}(\mathsf{x})$. *Although there more arcs that have to be considered, for simplicity, assume that only the arc* $\top \xrightarrow{\langle \mathsf{x} \wedge \bar{\mathsf{a}} \wedge \bar{\mathsf{b}} \wedge \bar{\mathsf{c}}, \{\mathsf{x}\}, \mathsf{a} \wedge \mathsf{b} \wedge \mathsf{c} \rangle} \mathcal{C}_2$ *is relevant. The contribution of this arc to* $\mathrm{TR}(\mathsf{c})$, *according to Equation (14.1), is*

$$\mathcal{X}_{\mathrm{ER}(\mathsf{x}) \cap \overline{\mathrm{ER}(\mathsf{a})} \cap \overline{\mathrm{ER}(\mathsf{b})} \cap \overline{\mathrm{ER}(\mathsf{c})}} \cdot \mathrm{TR}(\mathsf{x}) \cdot \mathcal{X}'_{\mathrm{ER}(\mathsf{a}) \cap \mathrm{ER}(\mathsf{b}) \cap \mathrm{ER}(\mathsf{c})} \cdot 1 \cdot v_t \cdot v'_t$$

which is equivalent to

$$\mathrm{EF}(\mathsf{x}) \cdot \overline{\mathrm{EF}(\mathsf{a})} \cdot \overline{\mathrm{EF}(\mathsf{b})} \cdot \overline{\mathrm{EF}(\mathsf{c})} \cdot \mathrm{TR}(\mathsf{x}) \cdot \mathrm{EF}'(\mathsf{a}) \cdot \mathrm{EF}'(\mathsf{b}) \cdot \mathrm{EF}'(\mathsf{c}) \cdot v_t \cdot v'_t.$$

After all these operations are performed, we finally obtain the expression $\overline{v_1} \cdot \overline{v_2} \cdot \overline{v_3} \cdot \overline{v_4} \cdot v_t \cdot v'_1 \cdot \overline{v'_2} \cdot \overline{v'_3} \cdot \overline{v'_4} \cdot v'_t$. *If we convert this TR into a partial TR (using Algorithm 2.1) we obtain* $\langle \overline{v_1} \cdot \overline{v_2} \cdot \overline{v_3} \cdot \overline{v_4} \cdot v_t \cdot v'_1, \{v_1\} \rangle$.

This pTR perfectly describes the transition $(0000, 1) \xrightarrow{\mathsf{x}} (1000, 1)$ *in subfigure (f). However, a smaller pTR could have been used with the same effect. For instance,* $\langle \overline{v_1} \cdot v'_1, \{v_1\} \rangle$ *is enough. The reason is that there is only one state with* $v_1 = 0$ *in our reachability set, so any modification to all the states that satisfy* $v_1 = 0$, *in fact, only affects state* $(0000, 1)$. ■ 14.3

To obtain smaller partial TRs, the available information of the reachable states can be used. The main idea is that, given a particular set of states, to identify a subset of these states can be achieved by simpler means. For instance, consider the set of sections of this document, if we want to identify

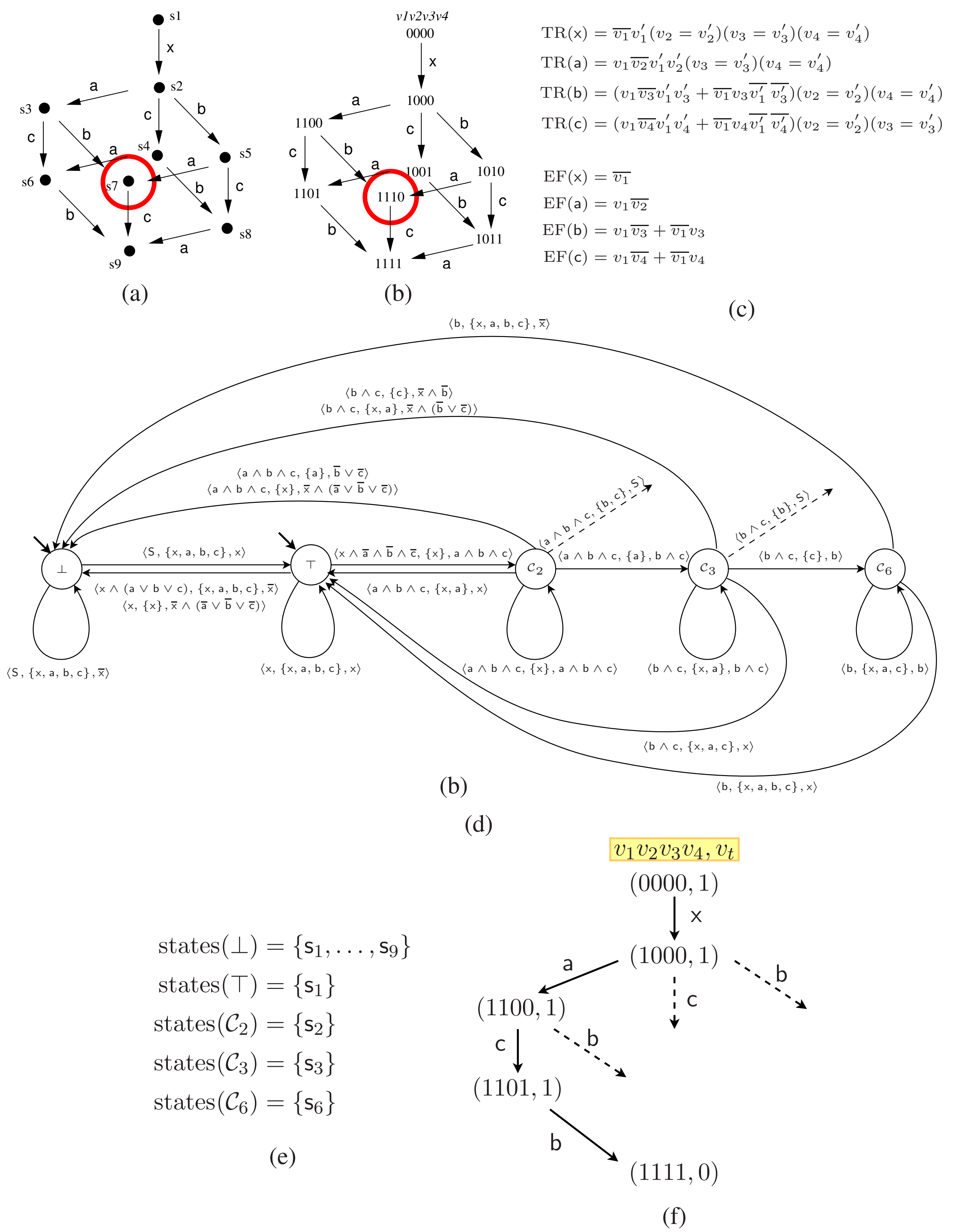


Figure 14.3. (a) State space of a LzTS. (b) Encoding of its state space. (c) Transition relations of the systemm (not using partial TRs for clarity) together with their enabling functions. (d) ECTRA we want to compose with the LzTS. (e) Mapping of the states resulting from the projection of the reachability set onto the ECTRA. (f) Resulting LzTS after symbolic composition.

a particular section we must give both the chapter and the section number. However if our universe restricts to a particular chapter, using only the section number is enough.

Similarly, when sets are identified using Boolean characteristic functions and we want to distinguish the states of g out from the states of f, then instead of g we can use any function h such that $f \cdot g = f \cdot h$. There exist several Boolean operators that compute such function h that try to obtain a function with a small representation (a problem known as *don't care minimization*). Some examples are the *constrain* [CM90], *restrict* [CM90], and *leaf-identifying* compaction [HBBM97] operators. Although the problem of finding a minimal h is NP-complete [SW96], which makes exact approaches [OCVSV98] usually infeasible, the previous heuristics can give good results in practice.

We use the term $\mathrm{DCmin}(g, f)$ to refer to the result of a *don't care minimization* operator, given that several choices are available. We can introduce this operator in Equations (14.1) and (14.2) to reduce the size of the resulting TR.

First of all we need to introduce some auxiliary notation. Let

$$\mathrm{states}_{ev}(E) = \sum_{\forall \mathcal{C} \in C : \mathcal{C} \neq \perp} \mathrm{states}(\mathcal{C}) \cdot \mathrm{extra}_{\mathrm{vars}}(\mathcal{C})$$

where

$$\mathrm{extra}_{\mathrm{vars}}(\mathcal{C}) = \begin{cases} \prod_{\forall \langle \mathsf{e}, i\rangle \in \Sigma'} \overline{u_{\langle \mathsf{e}, i\rangle}} & \text{if } \mathcal{C} = \perp \\ 1 & \text{if } \mathcal{C} \neq \perp \text{ and } \Sigma' = \emptyset \\ \prod_{\forall \langle \mathsf{e}, i\rangle \in \mathcal{C} \cap \Sigma'} u_{\langle \mathsf{e}, i\rangle} \cdot \prod_{\forall \langle \mathsf{e}, i\rangle \in \overline{\mathcal{C}} \cap \Sigma'} \overline{u_{\langle \mathsf{e}, i\rangle}} & \text{if } \mathcal{C} \neq \perp \text{ and } \Sigma' \neq \emptyset \end{cases}$$

and let $\mathcal{X}_{\mathrm{RS}}$ be the characteristic function of the reachability set of LzTS A.

Then we rewrite Equation (14.1) as

$$\mathrm{TR}_c(\mathsf{e}) = \sum_{\forall \mathcal{C}_i \xrightarrow{\langle \mathsf{s}_a, \Sigma_d, \mathsf{s}_b \rangle} \mathcal{C}_j \in T_E : \exists k : \langle \mathsf{e}, k\rangle \in \Sigma_d} \mathrm{From}_{ev} \cdot \mathrm{TR}(\mathsf{e}) \cdot \mathrm{To}'_{ev} \cdot \mathrm{cond}_{\mathrm{extra}} \cdot \mathrm{cond}_{\mathrm{timed}} \tag{14.3}$$

where

$$\mathrm{From}_{ev} = \begin{cases} \mathrm{DCmin}\left(\mathcal{X}_{\mathrm{states}(\mathcal{C}_i)} \cdot \mathrm{extra}_{\mathrm{vars}}(\mathcal{C}_i) \cdot \mathcal{X}_{\mathsf{s}_a}, \mathcal{X}_{\mathrm{states}_{ev}(E)} \cdot \mathrm{EF}(\mathsf{e})\right) & \text{if } \mathcal{C}_i \neq \perp \\ \mathrm{DCmin}\left(\mathcal{X}_{\mathsf{s}_a}, \mathcal{X}_{\mathrm{RS}} \cdot \mathrm{EF}(\mathsf{e})\right) & \text{if } \mathcal{C}_i = \perp \end{cases}$$

and

$$\mathrm{To}_{ev} = \begin{cases} \mathrm{DCmin}\left(\mathcal{X}_{\mathrm{states}(\mathcal{C}_j)} \cdot \mathrm{extra}_{\mathrm{vars}}(\mathcal{C}_j) \cdot \mathcal{X}_{\mathsf{s}_b}, \mathcal{X}_{\mathrm{states}_{ev}(E)}\right) & \text{if } \mathcal{C}_j \neq \perp \\ \mathrm{DCmin}\left(\mathcal{X}_{\mathsf{s}_b}, \mathcal{X}_{\mathrm{RS}}\right) & \text{if } \mathcal{C}_j = \perp \end{cases}$$

Changes to Equation (14.2) are similar. First we define

$$\mathrm{states}_{cf}(E) = \sum_{\forall \mathcal{C} \in C : \mathcal{C} \neq \perp} \mathrm{states}(\mathcal{C}) \cdot \zeta(\mathcal{C})$$

and then

$$\mathrm{TR}_c(\mathsf{e}) = \sum_{\forall \mathcal{C}_i \xrightarrow{\langle \mathsf{s}_a, \Sigma_d, \mathsf{s}_b \rangle} \mathcal{C}_j \in T_E : \mathsf{e} \in \Sigma_d} \mathrm{From}_{cf} \cdot \mathrm{TR}(\mathsf{e}) \cdot \mathrm{To}'_{cf} \cdot \mathrm{cond}_{\mathrm{timed}} \tag{14.4}$$

where

$$\mathrm{From}_{cf} = \begin{cases} \mathrm{DCmin}\left(\mathcal{X}_{\mathrm{states}(\mathcal{C}_i)} \cdot \mathcal{X}_{\mathsf{s}_a} \cdot \zeta(\mathcal{C}_i), \mathcal{X}_{\mathrm{states}_{cf}(E)} \cdot \mathrm{EF}(\mathsf{e})\right) & \text{if } \mathcal{C}_i \neq \perp \\ \mathrm{DCmin}\left(\mathcal{X}_{\mathsf{s}_a}, \mathcal{X}_{\mathrm{RS}} \cdot \mathrm{EF}(\mathsf{e})\right) & \text{if } \mathcal{C}_i = \perp \end{cases}$$

and

$$\mathrm{To}_{cf} = \begin{cases} \mathrm{DCmin}\left(\mathcal{X}_{\mathrm{states}(\mathcal{C}_j)} \cdot \mathcal{X}_{\mathsf{S}_b} \cdot \zeta(\mathcal{C}_j), \mathcal{X}_{\mathrm{states}_{cf}(E)}\right) & \text{if } \mathcal{C}_j \neq \bot \\ \mathrm{DCmin}\left(\mathcal{X}_{\mathsf{S}_b} \cdot \zeta(\bot), \mathcal{X}_{\mathrm{RS}}\right) & \text{if } \mathcal{C}_j = \bot \end{cases}$$

14.2 Static Analysis

We use the term *static analysis* to refer to the techniques applied before the iterative flow begins. Their objective is to minimize as much as possible the state space of the system by performing the simplest timing analysis. The concept of static analysis is not new as, for instance, the tool of [Peñ03] already included similar (but not equal) optional preprocessing options (see Appendix A.9).

The methodology we have developed has many more steps than previous iterative refinement methods. This increment is needed for convergence and efficiency reasons. However in many systems, at least the first iterations, the LzCES used in the composition is usually minimal and represents the precedence of a single event over another. For instance a typical LzCES of this type will show that if event r and t have the delay bounds $\delta(\mathsf{r}) = [0, 2]$ and $\delta(\mathsf{t}) = [3, \infty)$, then, if both events are enabled at the same time, r must happen first.

Although the methodology tries to stop the processing as soon as simple compositions are found, it is still inefficient to do all the processing and finally obtain a result that could have been easily generated by simple inspection of the event delay bounds. One possibility could be to analyze the delay bounds and detect pairs of events suitable for creating a minimal LzCES. However this would introduce many new variables in the system (at least one per composition) which is a high price for only a potential benefit.

The computational effort spent in static analysis should be kept low with respect to the normal operation of the methodology, precisely because it is not possible to know in advance if the work done would be helpful or not. It could be that pruning achieved only affects parts of the system in which no failures exist. Additional effort is only worth it if potential benefits are high and the cost is low, specially if the benefits are not guaranteed.

We present several possibilities of static analysis that differ in the amount of effort they require. They all have in common that they do not modify the transition relations nor introduce new variables in the model. We explain them in the following sections ordered by their cost (from the cheapest to the most expensive).

14.2.1 The responsible rabbit and the turtle

If two runners start the race at the same time, the faster will win. We can even relax the initial conditions and still have the same outcome: if the faster runner starts sooner or, at most, at the same time as the slower one, he/she wins. If this is the situation in all the races between the two, every time we have a contest between them we could always declare the winner immediately without even starting the race.

Exactly the same reasoning applies to events: let r be an event faster than event t, if r is always enabled before (or at the same time as) t, then we know that r will always fire before t. This preference of r over t can be very easily implemented by modifying the firing function of t as follows: $\mathrm{FR}(\mathsf{t}) := \mathrm{FR}(\mathsf{t}) \cap \overline{\mathrm{FR}(\mathsf{r})}$. This is different than composing with a LzCES that expresses the same relative ordering. On one hand it does not modify the TRs of the events nor introduces any new variable in the system. On the other hand, we can only use this simplification if *in all cases* c is enabled before or at the same time as t, while composition automatically affects only the states that satisfy this condition even if the system also contains states in which r is triggered after t. In Figure 14.4 (d,e,f) we give an example of this difference.

Now we have to answer two questions:

- What makes an event faster than another?
- How do we check that r is always enabled before (or at the same time as) t?

The first question is easy: we consider an event r faster than another event t if $\delta^u(\mathsf{r}) < \delta^l(\mathsf{t})$. This ensures that if both are enabled simultaneously, r is forced to happen first.

We solve the second question by testing the opposite condition: if there exists some transition in our system that triggers r while t remains enabled, then exists at least one race in which the rabbit was not responsible enough and starts later than the turtle. Consequently we cannot perform our simplification. Formally, for a LzTS $\langle \Sigma, S, T, \mathsf{s}_0, \mathrm{FR} \rangle$ and for two events $\mathsf{r}, \mathsf{t} \in \Sigma$ such that $\delta^u(\mathsf{r}) < \delta^l(\mathsf{t})$, r happens always before t if:

$$\nexists \mathsf{e} \in \Sigma : \mathsf{e} \neq \mathsf{t} \wedge \mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}' \in T \wedge \mathsf{s} \in \overline{\mathrm{ER}(\mathsf{r})} \cap \mathrm{ER}(\mathsf{t}) \wedge \mathsf{s}' \in \mathrm{ER}(\mathsf{r}) \cap \mathrm{ER}(\mathsf{t}) \qquad (14.5)$$

We give an implementation of this type of static analysis in Algorithm 14.2. To reduce the overhead introduced we order the different checks by cost, so that we can easily discard pairs of events with cheap computations and the most complex checks are performed only when required.

Algorithm 14.2: SimpleStaticAnalysis

input/output : $A = \langle \Sigma, S, T, \mathsf{s}_0, \mathrm{FR} \rangle$ a LzTS.
input : RS the reachability set of A.

1 **forall** $\mathsf{r}, \mathsf{t} \in \Sigma$ **do**
2   **if** $\delta^u(\mathsf{r}) < \delta^l(\mathsf{t})$ **then**
3   **if** $\mathrm{RS} \cap \mathrm{FR}(\mathsf{t}) \cap \mathrm{FR}(\mathsf{r}) \neq \emptyset$ **then**
4   Source $\leftarrow \mathrm{RS} \cap \overline{\mathrm{ER}(\mathsf{r})} \cap \mathrm{ER}(\mathsf{t})$ ▷ Proceed only if there are states in which both events can fire
5   Dest $\leftarrow \mathrm{ER}(\mathsf{r}) \cap \mathrm{ER}(\mathsf{t})$
6   delayt $\leftarrow true$
7   **forall** $\mathsf{e} \in \Sigma \setminus \{\mathsf{r}, \mathsf{t}\}$ **do**
8   **if** $\mathrm{Image}(\mathsf{Source}, T_{\mathsf{e}}) \cap \mathsf{Dest} \neq \emptyset$ **then**
9   delayt $\leftarrow false$ ▷ In some situation r gets enabled after t is already enabled
10   **break** ▷ so we cannot guarantee it will always happen before t and we abort
11   **if** delayt **then** $\mathrm{FR}(\mathsf{t}) \leftarrow \mathrm{FR}(\mathsf{t}) \cap \overline{\mathrm{FR}(\mathsf{r})}$

In line 6 we also excluded r from the events checked, although in Equation (14.5) it is not. This is a straightforward optimization since *Source* states belong to $\overline{\mathrm{ER}(\mathsf{r})}$ hence $\mathrm{Image}(\mathsf{Source}, T_{\mathsf{r}}) = \emptyset$. In terms of complexity, besides the cost of the BDD operations, the algorithm is cubical in the number of events in Σ.

EXAMPLE 14.4 *In Figure 14.4 (a) we show the untimed state space of a* TTS *together with the delay bounds of its events. Algorithm 14.2 reveals that in all states in which events* c *and* d *are enabled, all previous states either none of them was enabled or only* c *was. Since* $\delta^u(\mathsf{c}) = 2$ *and* $\delta^l(\mathsf{d}) = 3$ *it can modify the firing function of* d *–subfigure (b).*

Note that, with the reachable set of states of (a), although $\delta^u(\mathsf{f}) = 5$ *and* $\delta^l(\mathsf{e}) = 7$*, it is not possible to give preference to* f *over* e *because of state* s_6*. However, after the modification of* d*, the reachable set changes and a subsequent call to Algorithm 14.2 would yield the* LzTS *of (c).*

Subfigure (d) shows another TTS *obtained from (a) by adding two arcs and a state. In this context, Algorithm 14.2 cannot prune any transition. Nevertheless, by using a simple* LzCES

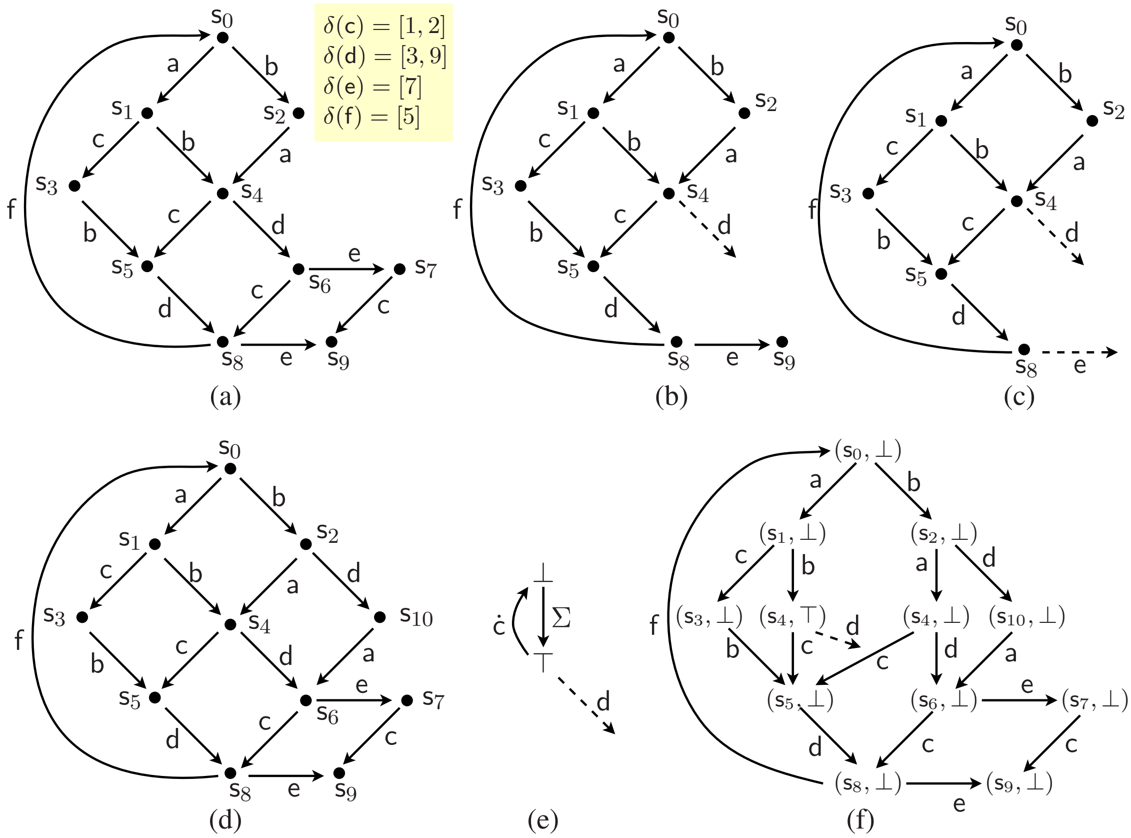


Figure 14.4. (a) A TTS. (b) Resulting LzTS after Algorithm 14.2 has been used. Note that the pruning opens new optimization oportunities since state s_6, in which event e was enabled, is no longer reachable. (c) Resulting LzTS after Algorithm 14.2 is used again. (d) A very similar TTS for which the algorithm is not able to produce any refinement because state s_4 is reachable from state s_2 in which event d is enabled but c is not. (e) Simplified depiction of an ECTRA in which event c fires before d. (f) Resulting LzTS obtained by composing (d) with (e). This produces some refinement but also causes state splitting.

(e) that expresses the precedence of c *over* d *some refinement can be achieved, although the composition also causes some state splitting (f), in this case of state* s_4. *Note that event* c *was defined as conservative in the LzCES, otherwise the composition would have been completely useless, because in* $st1$ c *is already enabled.* ■ 14.4

As previous example shows, Algorithm 14.2 and reachability analysis can be repeteadly invoked until a fixpoint is reached, *i.e.* no event is modified.

14.2.2 The sometimes lazy rabbit and the turtle

In our previous approach, if in a single occasion the rabbit arrives later than the turtle to the start of the race, then no modification is done, even if there are lots of other situations where the rabbit arrives on time or earlier than the turtle. An obvious improvement is to be more selective by identifying the states that satisfy the conditions, and then only modify the firing functions for these states. However it is computationally more expensive to find out these states as one can see by inspecting Algorithm 14.3.

Algorithm 14.3: CarefulStaticAnalysis

input/output : $A = \langle \Sigma, S, T, \mathsf{s}_0, \mathrm{FR} \rangle$ a LzTS.
input : RS the reachability set of A.

forall $\mathsf{r}, \mathsf{t} \in \Sigma$ **do**
  if $\delta^u(\mathsf{r}) < \delta^l(\mathsf{t})$ **then**
    if $\mathrm{RS} \cap \mathrm{FR}(\mathsf{t}) \cap \mathrm{FR}(\mathsf{r}) \neq \emptyset$ **then**
      $D_{i+1} \leftarrow \mathrm{RS} \cap \overline{\mathrm{ER}(\mathsf{r})} \cap \mathrm{ER}(\mathsf{t})$ ▷ Initialize fixpoint
      repeat
        $D_i \leftarrow D_{i+1}$ ▷ Temporal copy to be able to detect when fixpoint is reached
        forall $\mathsf{e} \in \Sigma \setminus \{\mathsf{t}\}$ **do**
          $D_{i+1} \leftarrow D_{i+1} \cup (\mathrm{ER}(\mathsf{r}) \cap \mathrm{ER}(\mathsf{t}) \cap \mathrm{Image}(D_{i+1}, T_{\mathsf{e}}))$
      until $D_i = D_{i+1}$
      $P \leftarrow \mathrm{RS} \cap \mathrm{ER}(\mathsf{r}) \cap \mathrm{ER}(\mathsf{t}) \setminus D_i$ ▷ P contains all the states in which r has to fire before t
      if $P \neq \emptyset$ **then** $\mathrm{FR}(\mathsf{t}) \leftarrow \mathrm{FR}(\mathsf{t}) \setminus P$ ▷ Render t unfireable in these states

The main idea in this case is that the states in which r will always win t are those in which both events are enabled *and* each one of its predecessor states s satisfies that either:

- none of the events is enabled in s, or
- only event r is enabled, or
- both events are enabled *and* the fired event is t or s recursively satisfies these conditions.

Since the definition is recursive, clearly to compute it requires a fixpoint. However, instead of directly computing the "safe" states, the algorithm computes the violating states. It starts from the states in which t is enabled, but r is not. Then computes all the states in $\mathrm{ER}(\mathsf{r}) \cap \mathrm{ER}(\mathsf{t})$ reachable from the former set. These states do not satisfy the conditions, hence all their successor states neither will do.

The complexity of the algorithm is $|\Sigma|^3 \cdot |S|$, since the number of iterations of the fixpoint can be, in the worst case, equal to the number of states in the system. In practice, however, usually only a few number of them are required.

In terms of efficiency there a few interesting implementation details not shown in the algorithm for clarity. First, it is possible to finish earlier the fixpoint when we detect that $D_{i+1} = \mathrm{ER}(\mathsf{r}) \cap \mathrm{ER}(\mathsf{t})$, because in such case no modification of t will happen. This can save some of the iterations of the innermost loop.

Another important question must be considered when the implementation uses classical LzTS s and partial TRs. In this case, the modification is made on $\mathrm{TR}(\mathsf{t})$ and, clearly, it is crucial for efficiency that the support of the TR is as small as possible. However, usually the representation of P includes most of the variables of the system, which are introduced in the TR when using P to compute the new TR. An alternative way of modifying the TR which alleviates this problem is as follows: $\mathrm{TR}(\mathsf{t}) \leftarrow \mathrm{TR}(\mathsf{t}) \cap (\overline{\mathrm{ER}(\mathsf{r})} \cup \mathrm{DCmin}(D_i, \mathrm{RS} \cap \mathrm{ER}(\mathsf{t})))$. In this case the use of $\overline{\mathrm{ER}(\mathsf{r})}$, which usually involves few variables, and the use of *Don't care* minimization, which removes many of the unnecessary variables present in D_i, produces smaller TR sizes.

EXAMPLE 14.5 *In Figure 14.5 (a) we show the untimed state space of a TTS together with the delay bounds of its events. Algorithm 14.2 produces no pruning because there exists state s_9 that can be reached from state s_7 in which* d *is already enabled while* c *is not. However, Algorithm 14.3 is capable of pruning the arcs of the states in which the conditions are fulfilled* (s_4). ■ 14.5

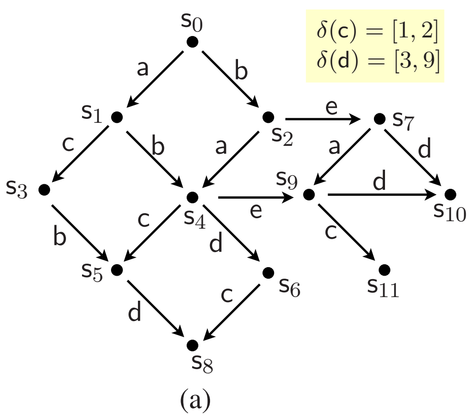


Figure 14.5. (a) A TTS. (b) Resulting LzTS after Algorithm 14.3 has been used. Only the states in which event c is enabled before or at the same time as d are affected by the pruning.

Chapter 15

EXPERIMENTAL RESULTS

Summary

One of the main objectives of this work is to improve the timed verification methodology using relative timing introduced in [Peñ03]. In Part II and Part III we have seen, step by step, the different elements that have to be considered, which come together to create the methodology exposed in Chapter 13. Many of the differences with [Peñ03] have been introduced to improve the correctness of the method, and some others to improve its performance.

Basically, the changes for correctness involve additional checkings, which consume time. Nevertheless this additional work might reduce the number of total iterations required to finish the verification task, which can reduce the total time spent. In present chapter we experimentally verify this claim by comparing with TRANSYT, the tool that implements the methodology of [Peñ03], in several sets of benchmarks.

There have been very few performance comparisons between relative timing methods and other state-of-the-art verification tools. Thus the significance of these methods could not be fully appreciated. We also shed some light on this issue by comparing with several tools that use different approaches. However, most of them compute the exact timed state, which makes the comparison not completely fair, since we compute an approximation to it. For this reason we also compare with other iterative refinement approaches, based on some of the previous work in this field.

The results of our experiments are promising since in many cases we obtain better results than any other tool or method. However, it is not always like this in all our benchmarks, hence we analyze the bottlenecks of our methodology and we describe the type of systems for which it is best suited.

Circuit	$\mathcal{V}$	Σ	S	S_f	R_f	C
sbuf-read-ctl	26	18	74	16	20	Y
rcv-setup	46	13	78	34	49	N
alloc-outbound	32	21	82	20	24	Y
ebergen	22	17	83	22	26	N
mp-forward-pkt	26	18	186	70	90	Y
dff	22	13	225	164	225	N
half	18	14	227	133	173	N
chu133	24	17	288	204	282	N
converta	28	23	408	244	309	N
nowick	26	20	510	292	362	Y
chu150	24	18	520	339	424	N
sbuf-send-ctl	32	24	1592	1081	2194	N
rpdft	24	19	2612	1841	3111	N
sbuf-send-pkt2	38	25	4544	4044	7024	Y
pe-rcv-ifc	44	27	5268	4058	6588	N
vme (big)	36	28	10568	9693	18435	N
sbuf-ram-write	42	30	14016	12362	20488	N
wrdatab	40	30	15000	10978	15820	Y
ram-read-sbuf	44	33	19328	17488	39088	Y
mr1	44	31	21076	11574	12689	Y
mr0	56	39	727304	642302	1.55e6	Y
tsend-bm (big)	54	39	830560	780780	1.99e6	N
trimos-send	54	45	2.10e6	1.80e6	4.36e6	N
mmu	58	44	5.60e6	5.20e6	1.59e7	N

Table 15.1. Speed-independent circuits.

15.1 The set of benchmarks

To analyze the performance of our methodology and compare it with other alternatives, we use three different sets of benchmarks: speed-independent circuits implemented with two input gates, mutual exclusion protocols and complex timed circuits.

15.1.1 Speed-independent circuits implemented with two input gates

The first set is formed by speed-independent circuits under the assumption that complex gates are available. That is circuits that work correctly assuming unbounded delays for the (complex) gates. However, if we implement them using only two input elemental gates, their correct operation is no longer guaranteed for this pessimistic unbounded delay model. But considering that gates have a bounded delay it is possible that the implementation is still correct. More details can be found on [Peñ03].

We summarize some relevant characteristics of these systems in Table 15.1. Column $\mathcal{V}$ contains the number of Boolean variables of the system (considering only the present state variables, *i.e.* the variables needed to encode the states, not the transition relations), Σ the number of events, and S the number of states reachable disregarding any timing information. Columns S_f and R_f indicate the number of failure states and failure arcs present in the reachable state space, respectively. The source states of failure arcs were counted also as failure states in these figures. Finally, column C specifies if the system is correct or not.

In all these circuits the number of events with a bounded delay equals Σ, thus all the events present in the system require a clock. With the exception of circuit `chu150`, the maximum delay

Fischer (**F**), software implementation described in [AIKY92]

#proc.	4	5	6	7	8	10	12	14	16	32
$\mathcal{V}$	30	36	42	48	56	68	80	92	106	204
Σ	28	35	42	49	56	70	84	98	112	224
Σ_{clk}	20	25	30	35	40	50	60	70	80	160
S	7633	63803	516921	4.10e6	3.21e7	1.90e9	1.09e11	6.10e12	3.37e14	2.13e28
S_f	910	11315	122532	1.20e6	1.14e7	8.78e8	6.08e10	3.92e12	2.40e14	2.04e28

smallFischer (**sF**), automata implementation described in [BN03]

#proc.	4	5	6	7	8	10	12	14	16	32
$\mathcal{V}$	22	26	30	34	40	48	56	64	74	140
Σ	20	25	30	35	40	50	60	70	80	160
Σ_{clk}	12	15	18	21	24	30	36	42	48	96
S	752	3552	16320	73600	327424	6.29e6	1.17e8	2.15e9	3.87e10	3.14e20
S_f	244	1560	8794	45906	227560	5.05e6	1.03e8	1.98e9	3.68e10	3.13e20

Alur-Taubenfeld (**AT**), software implementation described in [AIKY92]

#processes	3	4	5	6
$\mathcal{V}$	38	52	54	62
Σ	39	52	65	78
Σ_{clk}	33	44	55	66
S	15754	382564	8.11e6	1.57e8
S_f	426	19500	625690	1.64e7

Table 15.2. Mutual exclusion protocols.

constant is 111.1 for all of them. For `chu150` it is 4.4. In all cases it is sufficient to multiply by 10 to make all the delays have integer bounds.

This set of benchmarks has also been used in [BJMY02] and [Peñ03], with the exception of the `wrdatab` and `pe-rcv-ifc` circuits. Some of the circuits have also been verified in [KBS02, NMY03, KYM05]. Despite its popularity as a benchmark, this set of circuits has the drawback that the decomposition of complex gates has been done independently in each paper. Consequently there are differences in the number of gates and the correctness of the resulting circuits in some cases, so direct comparisons are sometimes not possible.

15.1.2 Mutual exclusion protocols

The second set of benchmarks is composed of well-known mutual exclusion protocols. They are one of the most popular benchmarks in the related literature, specially Fischer's protocol [Lam87]. For this particular example, we used two implementations, one consisting in seven states per process, like the one used in [AIKY92], and a simplified one with only four states, appearing in [BN03]. In this set we also include a more complex mutual exclusion protocol presented in [AT92] that was also analyzed in [AIKY92]. We refer to this latter protocol as the Alur-Taubenfeld protocol.

The advantage of protocols (and one of the reasons for their popularity as benchmarks) is that they are often scalable. By increasing the number of processes involved one can observe the assymptotic behavior of different verification methods. In Table 15.2 we give some model information for several sizes of the problems. Column Σ_{clk} indicates the subset of events that has a bounded delay (*i.e.* this is equivalent to the number of clocks needed in the system). With respect to Table 15.1 we have removed column R_f since the property checked is that no more than one

Circuit	$\mathcal{V}$	Σ	max δ	S	S_f	R_f	C
gals	84	15	666.7	1232	136	8	N
andor	28	25	500	2048	1990	2494	Y
ipcmos-1	26	21	22.5	2516	1331	1947	Y
ipcmos-2	42	35	22.5	372368	254108	461488	Y
pcc	48	30	111.1	1.06e6	107712	107712	Y
stari-1	20	20	330	576	376	284	Y
stari-2	32	32	330	12336	9560	6608	Y
stari-3	44	44	330	255744	220948	164332	Y

Table 15.3. Complex timed circuits.

process can be in the critical section at the same time, thus only state failures are defined in these benchmarks.

For the implementation of Fischer's protocol in [AIKY92] we have used the same delay values as them. That is, read/write actions have a delay of $[0, 2]$ while the delay action takes a value in the interval $[5, 8]$ to complete. The same values are used in the Alur-Taubenfeld protocol. For the implementation of Fischer's protocol in [BN03] we have also used the same parameter values. In this case, setting the variable has a delay $[0, 1]$ and the wait and critical sections take at least 2 units of time to execute (they are upper unbounded).

15.1.3 Complex timed circuits

The third set of benchmarks comprises timed circuits described in the literature, whose correctness depends crucially on the delay of its components. In particular we verify the following systems:

GALS A Globally-Asynchronous Locally-Synchronous design in [MVF00].

AndOr A two-level unfooted delayed-reset domino gate that computes the combinatorial Boolean function $f(a, b, c) = (a + b) \cdot c$ [BMH99].

IPCMOS A pulse-based controller for asynchronous pipelines in [SRC$^+$00].

PCC Pausible Clock Controller for heterogeneous systems in [YD99].

STARI A self-timed pipeline in [Gre93].

For the `stari` and `ipcmos` benchmarks, the number in the circuit name indicates how many replicated cells form the pipeline. Apart from the `gals` circuits, all circuits were specified with a set of delays that allows a correct operation.

This set of benchmarks, with the exception of the `andor` circuit, was used in [PP03] to evaluate the performance of a hybrid reachability analysis combining simulation and guided-traversal without considering the timing issues (*i.e.* untimed verification). The `stari` circuit is quite popular and has been also the subject of a timed verification in [BMT99, YR99, ZMM03, KOY$^+$01].

15.2 The compared methods

We compare our methodology with implementations of:

(i) standard verification algorithms,

(ii) adaptations to timed transition systems of some of the iterative verification methods.

Versions of representative methods of each one of these families are available in our tool OOTRANSYT, this allows a higher degree of profiling than using other tools. Because we have selected benchmarks that have been previously studied, we also compare to published results. If the involved tools and models were available, we have repeated the experiments to remove the distortion caused by different underlying hardware, so in many cases we compare with state-of-the-art applications.

We avoided automatic translations between input formalisms, therefore not all benchmarks have been studied in all tools. The reason for this limitation is that the translation process is error prone or can be done in such a way that hampers their performance. Moreover most of the tools do not provide enough information (*e.g.* the number of reachable states) to check if the translations are correct. Consequently, we have choosed to use only the benchmarks provided by the creators of the tools. As an exception, if some of the benchmarks could be easily adapted from these ones, then we have manually specified them.

15.3 Experimental set-up

All the experiments have been conducted on a PC with an Intel Pentium IV processor with a clock frequency of 3 GHz. The system has 1 Gbyte of physical memory and runs Linux 2.6. In our experiments we have limited the amount of memory allowed to be used by our tool to 256 Mbytes and the maximum execution time to 1 hour per benchmark.

We use the expressions *memout* or *mo* in the tables to indicate that the memory limit has been reached. Similarly the terms *timeout* or *to* are used for the time limit.

15.4 Results

Since the main objective of this work is to improve the results of the proposed methodology of [Peñ03], we first compare the results of the tools TRANSYT (an implementation of that methodology) and OOTRANSYT (which implements the methodology described in Chapter 13). Afterwards we will generalize this comparison to the rest of the methods.

15.4.1 TRANSYT versus OOTRANSYT

TRANSYT has a big number of parameters which can be used to tune or modify the behavior of the tool. In all the experiments we used the scripts of [Peñ03] when they were available. The standard options were to disable static analysis, to use chained traversal to find failures and to reorder variables every nine compositions. We have used the same options for the benchmarks for which no script was available.

For OOTRANSYT we disabled dynamic reordering. Only in the few cases in which the performance was severely affected by a bad variable ordering, then it was enabled. In particular this is only the case of the `mr0` example when static analysis is not used.

In Table 15.4 we present the results for the circuits summarized in Table 15.1. We compare three methodologies, the one implemented in the tool TRANSYT and our methodology without using the static analiysis prephase (that appears under the name of *OwoSA* in the table), or using it (column OOTRANSYT). The static analysis used is the one presented in §14.2.2 which is more exact but also more expensive.

For each one of these methods we provide the number of compositions performed (column *TC*) and the total verification time in seconds. When static analysis is used, we add three columns named $\mathrm{SA}_{\mathrm{it}}$, SA_p and $\mathrm{SA}_{\mathrm{time}}$. They indicate the number of iterations required by the static analysis, how many precedences were detected in that phase and the time required to complete it, respectively.

	TRANSYT		OwoSA		OOTRANSYT					
Benchmark	TC	time	TC	time	TC	SA_{it}	SA_p	SA_{time}	time	Correct
sbuf-read-ctl	4	0.4	3	0.4	**0**	1	6	0.1	**0.1**	Y
rcv-setup	3	0.2	2	0.2	**0**	2	12	0.1	**0.1**	N
alloc-outbound	5	0.9	3	0.4	**0**	1	9	0.1	**0.1**	Y
ebergen	1	0.1	**0**	**0.1**	**0**	2	8	0.1	0.3	N
mp-forward-pkt	16	6.8	4	0.7	**0**	1	12	0.1	**0.1**	Y
dff	6	**0.4**	4	**0.4**	**2**	3	34	0.1	**0.4**	N
half	4	0.3	**0**	**0.1**	**0**	3	25	0.1	0.2	N
chu133	15	6.4	3	0.6	**1**	2	21	0.1	**0.5**	N
converta	*	*	3	0.6	**2**	3	16	0.2	**0.5**	N
nowick	11	9.0	3	0.5	**0**	1	17	0.1	**0.1**	Y
chu150	10	5.3	2	0.6	**0**	2	18	0.1	**0.3**	N
sbuf-send-ctl	9	5.1	5	1.2	**0**	2	30	0.1	**0.6**	N
rpdft	7	1.0	1	**0.2**	**0**	4	61	0.3	0.9	N
sbuf-send-pkt2	18	45.4	15	7.4	**4**	3	59	0.3	**1.4**	Y
pe-rcv-ifc	*	*	8	3.0	**3**	4	198	1.1	**2.4**	N
vme (big)	39	1342.0	4	**1.6**	**2**	5	226	1.4	2.2	N
sbuf-ram-write	25	361.4	4	3.2	**3**	3	117	0.6	**2.2**	N
wrdatab	>45	*timeout*	14	**100.4**	**6**	3	210	1.8	161.4	Y
ram-read-sbuf	40	2057.7	10	24.0	**6**	3	84	0.5	**4.5**	Y
mr1	23	399.9	14	83.1	**0**	2	239	1.3	**1.4**	Y
mr0	51	3495.5	23	165.0†	**5**	5	721	10.3	**28.1**	Y
tsend-bm (big)	24	130.3	4	**2.0**	**0**	5	1241	23.5	25.7	N
trimos-send	**0**	0.5	**0**	**0.2**	**0**	3	229	13.5	13.8	N
mmu	6	25.5	**1**	**1.1**	**1**	6	855	28.6	29.1	N

† in this particular case we enabled variable reordering every nine iterations.

Table 15.4. Comparative results for TRANSYT and OOTRANSYT tools on speed-independent circuits.

The number of precedences detected has been computed considering all the iterations, so the same precedence is counted many times if new states in which the precedence is valid are discovered in different iterations.

The first conclusion is that the new methodology is efficient in reducing the number of composition required with respect to TRANSYT. There are two reasons for this: it uses a more representative CESAR, which tends to produce composition that remove more traces to failures, and, in case the system is not correct, it is capable in many cases to find a timing consistent failure trace without having to compose. Although each iteration requires more computations, running time clearly benefits in all examples, with the only exception of the `mr0` benchmark. In this latter case differences in the composition method and the variable order negatively affect OOTRANSYT without static analysis, and it is not possible to verify the circuit within the time limits without variable reordering. However, if variable reordering is allowed, then running time improves by an order of magnitude with respect to TRANSYT (using the same setting for reordering in both tools).

There are two benchmarks, namely `converta` and `pe-rcv-ifc` which could not be verified using TRANSYT. The first one because the outcome was not correct (it says the system is correct while the rest of the methodologies give a timing consistent counterexample) and the second because there is an assert violation at iteration 22.

If static analysis is incorporated, the number of compositions is reduced even more. In many cases the static analysis alone is capable of verifying the system (either because there are no

F		#proc.	4	5	6	7	8	10	12	14	16	32
	TRANSYT	TC	20	37	>63							
		time (s)	8.7	97.9	*timeout*							
	OwoSA	TC	12	20	30	42	>45					
		time (s)	3.1	17.5	144.8	2496.8	*timeout*					
	OOTRANSYT	TC	0	0	0	0	0	0	0	0	0	0
		SA_{it}	1	1	1	1	1	1	1	1	1	1
		SA_p	20	35	54	77	104	170	252	350	464	1952
		time (s)	0.2	0.5	0.7	1.4	2.9	7.7	18.8	43.0	88.7	2933.1
sF		#proc.	4	5	6	7	8	10	12	14	16	32
	TRANSYT	TC	13	28	38	62	95	>55				
		time (s)	1.3	10.0	30.9	138.7	1372.7	*				
	OwoSA	TC	12	20	30	42	56	90	>44			
		time (s)	1.1	2.8	7.1	21.1	72.3	1250.5	*memout*			
	OOTRANSYT	TC	0	0	0	0	0	0	0	0	0	0
		SA_{it}	1	1	1	1	1	1	1	1	1	1
		SA_p	22	38	58	82	110	178	262	362	478	1982
		time (s)	0.1	0.2	0.3	0.4	0.8	1.7	3.6	5.2	10.2	312.8
AT		#proc.	3	4	5	6						
	TRANSYT	TC	43	>139								
		time (s)	48.2	*timeout*								
	OwoSA	TC	18	36	>24							
		time (s)	18.2	493.5	*timeout*							
	OOTRANSYT	TC	6	12	20	>18						
		SA_{it}	2	3	3	3						
		SA_p	69	200	335	504						
		SA_{time}	1.1	21.4	81.6	543.4						
		time (s)	4.6	60.7	2533.2	*timeout*						

Table 15.5. Comparative results for TRANSYT and OOTRANSYT tools on mutual exclusion protocols.

remaining failures or because the first failure trace found is timing consistent). In most cases running time also greatly benefits from it, specially on some difficult examples like `ram-read-sbuf`, `mr1` and `mr0`. However it is not always better than the other two methods because the overhead introduced by the prephase is sometimes not compensated. For instance in the `trimos-send` bechmark, the first failure trace found is timing consistent, hence the static analysis in this case is useless. On the other hand this method is able to finish all the set of benchmarks and is the one provinding more consistent results.

For the protocols analyzed in Table 15.5 the results are similar but the tendency is much more clear: TRANSYT is the least efficient method while OOTRANSYT with static analysis provides the best results. For both versions of Fischer's protocol, the number of compositions is always less in OOTRANSYT. It must be noted that the tool performs the minimum number of compositions when not using the static analysis prephase. Since each process must set a variable before the delay phase of the other processes has finished, in order to make the protocol work correctly, the minimal number of composition is $n \cdot (n-1)$, where n is the number of processes.

If static analysis is used, it is able to remove all failure states without having to perform any composition. Moreover, only a single iteration is required. We do not give the time for the static analysis phase, since it is almost equal to the complete verification time. Although it is a very efficient approach for this benchmark, its running time is at least cubical with respect to the number of processes. There are two main reasons for this:

- The variable order was not fixed to one of the known orders which have quadratic representation, instead in the first reachability analysis we allowed some automatic reordering and then dynamic reordering was deactivated.
- The cost of the static analysis is inherently cubical at least, because it compares each event with each other event and for each pair performs a fixpoint (which requires to consider again all the events at least once). As the number of events is a multiple of the number of processes, the cost is also cubical in the number of processes. Moreover the time needed for the fixpoint does not only depend on the number of events, since more processes mean more Boolean variables which make each image computation more expensive.

The Alur-Taubenfeld protocol is correct under a much more complex set of conditions. To work correctly, three of the instructions of each process must happen before the delay of each one of the rest of the processes has expired. This implies that the minimum number of compositions is $3n \cdot (n-1)$, which is, again, achieved by OOTRANSYT. Static analysis is able to prove two of the three required precedences, thus the number of compositions in this case reduces to $n \cdot (n-1)$.

For this particular benchmark, using the standard configuration of OOTRANSYT we could only verify 5 processes in the one hour limit. The main limitation in this case is the composition cost. Our tool implements the approach explained in §14.1, but other alternatives are possible. For instance, TRANSYT uses a composition method that does not depend on the mapped states in each configuration but is based exclusively on the enabling functions of the events.

This approach can be very efficient, since states sets tipically contain most of the variables in the system while enabling functions usually contain few of them. The problem is that lots of disambiguations between configurations are required and is difficult to perform them using only the enabling functions and their negations. This might involve to use additional disambiguation variables. Moreover one must be extremely careful in this process to avoid altering the capacity to detect all the enabling compatible traces.

For all these reasons OOTRANSYT uses the composition method of §14.1.1. Nevertheless in some situations the alternative is better. For instance, we have implemented a composition method similar in concept to the one in TRANSYT. Although its performance is often worse than our current implementation, for the Alur-Taubenfeld protocol provides the following verification times (with static analysis):

#processes	3	4	5	6
TC	6	12	20	30
time (s)	3.8	38.6	274.9	3263.8

Taking into account that in [AIKY92] it took more than one hour to verify three processes using also an iterative refinement approach, the three orders of magnitude of improvement are still significant, even when considering the mitigating factor of technology improvement. Now in one hour we were able to verify a system of double of the original size.

In Table 15.6 we show the results for the complex timed circuits. The `ipcmos-1` circuit shows an interesting effect: the use of static analysis produces more compositions than not using it. Although in general, as we have seen in previous results, the reduction of the state space yields less iterations, there are cases in which this would not happen. The reason is that the CESARs found in the complete state space can be different than the ones found in the pruned state space, and they might be more effective in removing failures.

A similar effect appears in the `andor` circuit. TRANSYT finishes after 23 composition, while OOTRANSYT without static analysis needs 26 iterations. Although the CESAR generation routine of OOTRANSYT is much more complex than the one used in TRANSYT, final performance heavily

	TRANSYT		OwoSA		OOTRANSYT					
Benchmark	TC	time	TC	time	TC	SA_{it}	SA_p	SA_{time}	time	Correct
gals	*	*	**2**	**1.9**	**2**	4	31	0.5	4.2	N
andor	23	6.7	26	13.6	**10**	3	99	0.4	**3.4**	Y
ipcmos-1	30	28.8	**14**	**27.5**†	16	3	101	0.7	54.9†	Y
ipcmos-2	*	*	≥ 48	*timeout*†	≥ 48	4	547	10.5	*timeout*†	Y
pcc	≥ 31	*timeout*	6	**29.3**	**5**	5	691	70.2	87.8	Y
stari-1	*	*	13	**2.0**	**10**	2	21	0.1	**2.0**	Y
stari-2	*	*	29	51.0	**27**	2	41	0.6	**46.3**	Y
stari-3	*	*	≥ 51	*timeout*	≥ 50	2	61	1.2	*timeout*	Y

† in this particular cases we enabled variable reordering every nine iterations.

Table 15.6. Comparative results for TRANSYT and OOTRANSYT tools on complex timed circuits.

depends on the failure trace found. We always use the shortest failure trace, however in many cases working with larger traces allows to remove more failures. This circuit is a perfect example: TRANSYT uses a failure trace search mechanism that does not guarantee to return the shortest trace, thus in some cases several failure elements are processed in a single iteration because the trace captures more than one problem. We preferred to use the shortest failure trace because it provides more stable results (other methods might give different traces between different runs of the application), however we believe our approach still outperforms [Peñ03]. For instance, if a failure trace search mechanism similar to the one in TRANSYT is used in OOTRANSYT, the required iterations reduce to 21.

15.4.2 Comparison with published results and other tools

Besides the comparison with TRANSYT, we also compare with other available alternatives, specially those developped for timed automata, which has been the formalism that has received more attention by the research community. Such a comparison was not performed in [Peñ03], thus it was unknown if its performance was better or worse than state-of-the-art methods.

Speed-independent circuits

For the decomposed speed-independent circuits one can have a partial comparison with the published results in [BJMY02, KBS02, NMY03, KYM05, CC05]. However, results should be interpreted with caution because the decompositions used vary from work to work and not all the circuits have been used in all the papers.

Bozga et al. used the standard reachability analysis for timed automata (see §4.4.1) implemented in the tool OpenKronos, and six of the largest benchmarks could not be fully verified, although in two of them an error could be detected before running out of memory. So finally only four circuits could not be determined to be correct or incorrect, namely, `mmu`, `mr0`, `ram_read_sbuf` and `trimos_send`. OpenKronos took about 10 minutes to exhaust the memory resources of their computer (a SUN UltraSparc 10 with 2Gb of memory), which makes a favorable comparison for TRANSYT for these hardest examples. However, they compute the exact timed state space in which other properties different than safety ones could be checked, which is something that TRANSYT cannot do (or at least it would require a modification of the model checked). Additionally, to store the states explicitely guarantees that the algorithm will not terminate in any benchmark with a large enough number of states. This makes the comparison not very fair.

Nelson et al. used the results from [BJMY02] and [PCKP00] to compare their approach. However they seem not to be aware that the decomposition used in the different papers are not the same, thus that the results are not fully comparable. They also include the comparison with explicit state verifier ATACS running on a 900 Mhz 256Mb Pentium IV, which verifies almost all benchmarks in less than one second. The exceptions are the `tsend-bm` and the `trimos-send` circuits, which take 5.3 and 10.7 seconds to complete. Their paper present a conservative algorithm which halfs the time needed by ATACS to verify the circuits, but in 6 of the circuits it is not able to determine if they are correct or not.

The rest of the papers we have cited do not exactly solve the verification problem. In some cases they solve harder problems and with their computed solutions it is usually trivial to check if the actual circuit works or not.

Kim et al. verified 10 of the circuits in the set [KBS02], being `ram-read-sbuf` the largest of them. Running times on a 450MHz 1Gb Ultra SPARC60 are over 100 seconds for this circuit and also for `converta`, but they claim their implementation is naive. However, they also solve a different verification problem. Instead of verifying the circuit using some given delay bounds, they compute which precedences between events guarantee the correctness of the circuit (and they try to minimize the number of these precedences). Although it is an interesting approach, the problem of verifying that a real circuit satisfies this set of precedences is not trivial.

A similar poblem was solved by Kitai et al. in which the value of the delay bound of some events was computed so that the circuit worked correctly. However in this case the solution they provide can be used to easily solve the verification problem (by simply checking that the actual delay values in the circuit are inside the safe computed bound). They used only 6 of the circuits in the set and they spent between 1 and 3 seconds to solve each one.

Clarisó et al. used abstact interpretation to compute which relations between the delays of a system should be satisfied to correctly operate. They verify 8 of the circuits, most of them in less than two seconds (with the exception of `converta` that requires 20 seconds). The verification problem is trivially solved from their solution.

Mutual exclusion protocols

Mutual exclusion protocols allow us to compare with other tools and approaches. The Alur-Taubenfeld's protocol was studied in [AIKY92], whose results we have already discussed in §15.4.1.

For Fischer's protocol we can also compare against [AIKY92]. Using an iterative refinement approach they managed to check the correctness of three processes in five iterations and three minutes. With half of that time we verify 16 processes. However they used an explicit state representation and it would have been very difficult to verify more than 7 processes (which has more than one million reachable states after timing variables have been abstracted).

More recent data is available for the small version of the protocol. In [BN03] Beyer compares the performance of three tools: RABBIT 2.1 (using discretization and BDDs), RED 3.1 (using CDDs) and UPPAAL 3.2.4 (with explicit representation and DBMs). In less than one hour, UPPAAL could only verify up to 6 processes, while RED achieved 10 processes. By using the best variable order possible, RABBIT verifies 64 processes (and 128 in less than two hours). In comparison our best result is 32. This is much better than RED or UPPAAL, but one has to consider that they solve a harder problem, since they obtain the exact timed state space.

Our results are worse than RABBIT because our technique is, at least, cubical in the number of processes, while for this particular example, if discretizing and using BDDs with the right variable order, it is $\Theta(c^3 n^2 \log_2 c)$ where c is the value of the largest delay constant and n are the number of processes. However, in that paper they used the smallest possible delay constant (*i.e.* $c = 2$)

which clearly helps in achieving these results. For big values of c or a different variable order the number of verified processes quickly diminishes.

For instance, we repeated the experiments with the Fischer's protocol with RABBIT 2.1 on our machine. Using the default value for c we achieved a result than is about the half of the published running time in [BN03], presumably because of better hardware/OS. However, by using $c = 20$ instead of 2, only up to 16 processes could be verified in one hour. We summarize these experiments in the following table:

#processes	4	5	6	7	8	10	12	14	16	32	64
time for $c = 2$	0.1	0.1	0.1	0.1	0.2	0.3	0.5	0.7	1.1	21.1	205.4
time for $c = 20$	1.6	4.6	13.2	56.7	154.8	536.5	938.8	1878.4	3262.1	*to*	

In contrast, UPPAAL, RED and our technique do not depend on the size of the delay constants (if this does not alter the timed state space) and can be more fairly compared (with the already commented exception of the difficulty of the task). This makes a favorable comparison because we verify 32 processes in less time that RED needs to complete the task for 12 processes, according to the data in [BN03]. To compare in equal conditions we have repeated these experiments on our machine with RED 5.0. The following table contains the results for the three protocols:

F	#processes	4	5	6	7	8	10	12	14	16
	time	22.8	1343.8	*to*						
sF	#processes	4	5	6	7	8	10	12	14	16
	time for $c = 2$ or $c = 20$	0.4	1.2	3.5	9.8	23.0	122.4	543.4	2601.8	*to*
AT	#processes	3	4	5	6					
	time	16.5	723.8	*to*						

As expected the performance is better than in previously published results, but still our approach verifies 32 processes of the small version of the Fischer's protocol in less time than RED needs to verify 12 processes. Comparison with the complete Fischer's protocol is even more dramatic.

One of the main reasons for these figures is that RED is negatively affected by the variable order. The tool uses *location-first* order, so that clocks are checked after regular state variables in the data structure. For this ordering, the representation size is $\Omega(2^n)$, however using an interleaved order like the one in RABBIT, it is $\Theta(n^3)$. Using this other variable order the results would have been much better using CDDs, and probably better than using BDDs if $c \backsim n$ for instance, as our experiments with RABBIT have suggested. Even with a non-suitable variable ordering RED outperforms RABBIT up to 10 processes if $c = 20$ is used.

To compare the performance of TRANSYT and OOTRANSYT with an explicit state verifier in our computer, we have used UPPAAL 4.0.6 to verify the same protocols as in RED. The results are summarized in the following table:

F	#processes	4	5	6	7	8	9	10	11	12
	time	0.8	43.1	*mo*						
sF	#processes	4	5	6	7	8	9	10	11	12
	time for $c = 2$ or $c = 20$	0.1	0.1	0.2	0.4	1.9	7.6	31.3	131.6	*mo*
AT	#processes	3	4	5	6					
	time	0.3	33.1	*mo*						

When compared to the results in [BN03], the current version of UPPAAL verifies up to 11 processes of the Fischer's protocol, almost the double than in previous results. The difference, besides of the underlying hardware performance and that we have used a newer version of the tool, is also caused by the fact that in [BN03] the approximation mechanism of UPPAAL was disabled, while we kept the default option in our experiments, which is to use conservative approximations. Moreover the running times are not affected by the delay constant, so it is quite easy to find a value large enough so that the BDD approach has worse results than the explicit approach.

However, even when the conditions are more favorable to BDDs, *i.e.* $c = 2$, the explicit approach can be tuned to have a better performance. For instance, using the *symmetry reduction* option of UPPAAL the verification time is 51 seconds for 64 processes and about half an hour for 128 processes. However no counterexample can be generated using this option, so we choosed not to activate it to keep similar options accross all the tools.

Complex timed circuits

For the complex timed circuits, the STARI pipeline has been studied in [BMT99, YR99, ZMM03, KOY$^+$01].

Bozga et al. used the tool KRONOS following a discretization schema and using BDDs to represent the timed state space. They managed to verify a set of properties different than the ones we use, for a pipeline of 18 stages in about 3 hours. In one hour they were able to verify up to 12 stages. This particular benchmark is very well suited for the use of BDDs (using explicit states they only achieved 8 stages) and, for instance, our own implementation of the discretization approach using BDDs (see §15.4.3) computes the timed state space of 12 stages in less than 40 seconds.

However, even better results can be achieved using other techniques. For instance, Zheng et al. use abstractions in the ATACS tool. Without abstractions the tool only verifies up to 12 stages in about 17 minutes. The enabling of abstractions allows to verify up to 15 stages in 27 minutes. These results are better than KRONOS but worse than our BDD implementation.

More impressive results can be achieved by using partial order reductions. For instance, Yoneda et al. manage to verify 11 stages in less than one second, and Zheng et al. achieve the verification of 18 stages in less than 3 seconds.

Compared to all this methods the results obtained by OOTRANSYT are not competitive. However, elements of the abovementioned approaches could be incorporated into the tool (*e.g.* abstraction), so in some aspects they can be viewed as orthogonal techniques.

Discussion

Unfortunately, all the tools we have used in our tests have very little profiling capacities, and many of the ones appearing in the related papers are not publicly available. This prevents to repeat the experiments on a common hardware (if the tool is not available) or to obtain some interesting data, like, for instance, if the final approximation of the state space obtained by TRANSYT or OOTRANSYT is close or not to the timed state space. Moreover none of the circuits we use as benchmarks is directly available in the input formalisms of the available tools, and we wanted to avoid a translation which is an error-prone process.

For this reason, and to have results executed on the same hardware for all our benchmarks, we will compare with a method that uses an implicit storage of the states (as explicit approaches would be inherently limited in some benchmarks). With this intention we have implemented a BDD approach based on the discretization of time, in a similar vein to [BLN03]. This will also allow to implement some of the other iterative refinement methods on top of this implementation, so that their performance can be compared (see §15.4.5).

15.4.3 Timed state space generation using BDDs

We will briefly explain how we can generate the timed state space of a timed transistion system using BDDs. First important thing is that the timed state space can be generated using discrete-time semantics [HMP92b], thus we start by assuming that our delays are, in fact, rational. Then they are scaled to integers by multiplying by a suitable factor. Now a clock variable is assigned to each event that has a bounded delay. The number of Boolean variables used to encode each clock must allow to represent the value of the maximum delay of the event or, if it is upper unbounded, the value of the minimum delay.

The basic idea is to modify the TRs of the events so that each event can only fire if its clock has a value inside its delay bound. However, contrary to timed automaton, the reset of clocks is not explicit in TTSs and corresponds to the situation in which an event becomes enabled. So each event must take care of resetting the clocks of the events it triggers. Causality analysis (see Chapter 3) can be used to determine which clocks can potentially be reset by a given event.

Let $\mathrm{eftr}(\mathsf{e})$ be the set of events that are effectively triggered (see §3.3.1) by an event e. With this information we can compute the new TR of the event e as follows[1]:

$$\mathrm{TR}(\mathsf{e}) := \mathrm{TR}(\mathsf{e})\cdot(\delta^l(\mathsf{e}) \le \mathrm{clk}(\mathsf{e}) \le \delta^u(\mathsf{e}))\cdot \prod_{\forall \mathsf{e}_s \in \mathrm{eftr}(\mathsf{e})} [\alpha\cdot(\mathrm{clk}'(\mathsf{e}_s) = 0) + \overline{\alpha}\cdot(\mathrm{clk}'(\mathsf{e}_s) = \mathrm{clk}(\mathsf{e}_s))]$$

where, $\alpha = \begin{cases} \overline{\mathrm{EF}(\mathsf{e}_s)}\cdot \mathrm{EF}'(\mathsf{e}_s) & \text{if } \mathsf{e}_s \neq \mathsf{e} \\ \mathrm{EF}'(\mathsf{e}) & \text{otherwise} \end{cases}.$

A *tick* event is added so that it increases all the active clocks by one unit as long as none of them has reached its upper bound. Formally, $\mathrm{TR}(\mathsf{tick}) := \prod_{\forall \mathsf{e}\in\Sigma_b}(\mathrm{clkcond}(\mathsf{e}) + \overline{\mathrm{EF}(\mathsf{e})})$ where Σ_b is the set of all events with bounded delay in the system, and $\mathrm{clkcond}(\mathsf{e})$ is a function defined as follows:

$$\mathrm{clkcond}(\mathsf{e}) = \begin{cases} [(\mathrm{clk}(\mathsf{e}) \le \delta^l(\mathsf{e}) - 1)\cdot(\mathrm{clk}'(\mathsf{e}) = \mathrm{clk}(\mathsf{e}) + 1) + \\ \quad +(\mathrm{clk}(\mathsf{e}) = \delta^l(\mathsf{e}))\cdot(\mathrm{clk}'(\mathsf{e}) = \mathrm{clk}(\mathsf{e}))]\cdot \mathrm{EF}(\mathsf{e}) & \text{if } \delta^u(\mathsf{e}) = \infty \text{ and } \delta^l(\mathsf{e}) > 0 \\ (\mathrm{clk}(\mathsf{e}) \le \delta^u(\mathsf{e}) - 1)\cdot(\mathrm{clk}'(\mathsf{e}) = \mathrm{clk}(\mathsf{e}) + 1)\cdot \mathrm{EF}(\mathsf{e}) & \text{if } 0 < \delta^u(\mathsf{e}) < \infty \\ 1 & \text{if } \delta^u(\mathsf{e}) = \delta^l(\mathsf{e}) = 0 \end{cases}$$

The underlying idea is also simple. All active clocks have to advance (*i.e.* be increased in one unit) at the same time. A clock is active only in the enabling region of its corresponding event, if it is not active (*i.e.* in $\overline{\mathrm{EF}(\mathsf{e})}$) then all clock valuations are permited, which makes BDDs simpler. Clock increment is allowed to happen until some active clock reaches its upper bound. If its upper bound is 0, then no tick event can be fired. Finally if some event is only lower bounded, then the counting for that clock is stopped when this limit has been reached, but still the rest of clocks are allowed to be incremented, contrary to what happens when an upper bound is reached.

Grouping clocks

Previous translation schema is straightforward, but has the inconvenience that it requires a clock for every bounded event. Typically there can be many of such events, and it is known that the difficulty of the verification problem is related also to the number of clocks in the system. Besides that, many of the clocks are not used at the same time [BMT99] therefore we could reuse the same clock for a group of events. The only condition we have to check is that none of the events in the group is concurrently enabled with some of the other events in the same group. That is, if $g \subseteq \Sigma_b$ is the set of events sharing a clock, then $\forall \mathsf{e}_1, \mathsf{e}_2 \in g : \mathsf{e}_1 \neq \mathsf{e}_2, \mathrm{EF}(\mathsf{e}_1)\cdot\mathrm{EF}(\mathsf{e}_2) = 0$.

To group the events we create a graph with a node for each event in Σ_b, then we connect two nodes if they are never enabled concurrently. We do check this in the untimed reachability space we

[1] In this expression we mix both Boolean and natural expressions to simplify the notation. Obviously all natural expressions are Boolean encoded in the actual TR.

obtain before the system is modified. Hence two nodes are connected if they are never effectively concurrent, using the terminology introduced in Chapter 3. Once the graph is complete, we search for cliques in it. The optimal solution depends on which data structure you use. If it is sensitive to constant values, like BDDs, the objective should be to minimize the number of Boolean variables introduced. If the data structure is not affected by constant values, like CDDs, then the objective should be to minimize the number of cliques.

In our case we do use an heuristic to solve the problem. We start by treating the nodes with less degree (*i.e.* number of connections) and we add direct neighbourgs to a group as long as it forms a clique. Neighbourghs are scanned also in an order that depends on the value of the maximum constant they need. The idea is that events with similar maximum constants are assigned together, because the number of bits of a clock depends on this value. Thus we want to avoid groups in which a clock of 10 bits is needed just because the clock of one event in the group reaches value 999 while the rest of the events only use the clock up to value 3.

Once event groups have been formed we can modify the system in a similar way as before. For instance the modifications of the TRs of the events are almost equal. The only difference is that now the clock of each event is the clock assigned to the group it belongs to. Similarly, the clocks reset are now the clocks of the groups to which the triggered events belong to.

The tick event involves slightly more changes, since now the clock of a group is inactive only when none of the events of the group are enabled. Let G be the set of event groups sharing a clock. Then $\mathrm{TR}(\mathsf{tick}) := \prod_{\forall g \in G}(\sum_{\forall \mathsf{e} \in g} \mathrm{clkcond}(\mathsf{e}) + \prod_{\forall \mathsf{e} \in g} \overline{\mathrm{EF}(\mathsf{e})})$. As before, the clocks associated to each event appearing in $\mathrm{clkcond}(\mathsf{e})$ are changed to the clocks of the group the event belongs to.

Finally, as noted in [BN03], the order in which clock variables are added into the BDD structure has a deep impact on performance. In our case we chosed a location-first order, that is, we put all clock variables at the bottom of the BDDs. This order is suitable when the number of untimed states is small, this is the case of most of the circuits we verified. However, this order is known to produce exponential BDDs in many benchmarks, for instance Fischer's protocol.

Reachability analysis

Our implementation uses the same strategy as RABBIT. Each iteration of the tool consists in two fixpoints, in the first one, regular events are fired, while in the second only the tick event is used. Because the size of the tick event can become huge, we also used a conjunctive partitioned representation for it. If a monolithic tick event could fit into memory we use it, otherwise we aggregate groups until a threshold is surpassed, the moment at which we start another partition. The results we provide have a default threshold of 10000 BDD nodes per partition.

15.4.4 Timed state space generation versus iterative refinement

In Table 15.7 we give a summary of the results of generating the complete state space using BDDs. Failures are not checked while the state space is generated, only at the end, and counterexamples are *not* searched although the system is proved incorrect. Column *Iter* is the number of iterations needed to complete the analysis. The search method is the one explained in previous section, that is, in each iteration two fixpoints are computed, one involving only regular events, and the other using only the special *tick* event. Column S_u indicates the number of reachable states once all clock variables have been abstracted away.

We compare the performance of the same algorithm but using two approaches to deal with clock variables. In the first approach, one clock is used for each bounded event. The second option is to group the clocks of events that are never enabled at the same time. For these two approaches

			1 clk/event		grouped clocks		
Benchmark	Iter.	S_u	$\mathcal{V}_{\text{clk}}$	time	$\mathcal{V}_{\text{clk}}$	time	Correct
sbuf-read-ctl	21	35	123	2.1	40	0.8	Y
rcv-setup	14	42	98	2.9	45	1.0	N
alloc-outbound	21	47	146	2.9	39	1.0	Y
ebergen	20	44	112	1.8	40	1.1	N
mp-forward-pkt	15	63	123	2.1	52	1.2	Y
dff	12	115	88	2.4	52	2.0	N
half	12	34	99	4.0	58	3.5	N
chu133	18	38	117	2.5	69	1.5	N
converta	31	63	148	8.1	52	1.6	N
nowick	22	46	135	1.4	69	0.8	Y
chu150	20	59	102	1.0	48	0.6	N
sbuf-send-ctl	26	79	159	11.4	82	2.6	N
rpdft	53	67	134	2.8	59	1.7	N
sbuf-send-pkt2	18	72	170	4.6	98	2.4	Y
pe-rcv-ifc	56	101	182	7.0	92	4.9	N
vme (big)	26	88	182	4.5	99	3.4	N
sbuf-ram-write	35	137	198	27.5	109	19.6	N
wrdatab	46	229	200	492.7	98	122.8	Y
ram-read-sbuf	30	115	217	71.9	110	15.9	Y
mr1	>28	>72	206	*memout*	92	*timeout*	Y
mr0	48	226	259	212.0	152	49.3	Y
tsend-bm (big)	>100	>1161	259	*timeout*	145	*timeout*	N
trimos-send	>27	>807	285	*memout*	141	*memout*	N
mmu	33	210	284	194.4	164	117.0	N

Table 15.7. Results for the complete timed state space generation using BDDs for speed-independent circuits.

we provide the number of Boolean variables to encode (in present state form) the clocks (shown in column $\mathcal{V}_{\text{clk}}$) and the elapsed time to obtain the complete timed state space.

Since using grouped clocks performs consistently better (although it does not provide an improvement of orders of magnitude) we use only this option in the rest of this chapter.

The performance of the grouped clock approach on the mutual exclusion protocols and the complex timed circuits can be seen in Table 15.8 and in Table 15.9 respectively.

In Tables 15.7, 15.8 and 15.8 we have assumed that we want to generate the complete timed state space, however this is not the usual practice. The standard algorithm stops when a failure is encountered. Also it is desirable to return a counterexample trace. If we modify the previous algorithm to incorporate this features we obtain the results of Table 15.10. The verification times for correct circuits are also provided because, to enable counterexample generation, we have to keep track of the state space generated at several intermediate points during the exploration. Typically the size of these intermediate BDDs is larger than the original and final ones, and to keep them makes the overall process slower (sometimes sensibly slower).

Column *Iter* shows the number of iteration at which the failure was found, and S_u the number of untimed states reached at that time. Precisely the time to detect the failure is listed in column *gen.time* while θ_c indicates the length of the found counterexample (without counting the tick events). Finally, *total time* summarizes the total verification time, including *gen.time*.

Since all protocols and all complex timed circuits are correct, with the only exception of the `gals` circuit, we do not provide the results for these benchmarks in a separate table for the case in

F	#processes	4	5	6	7	8	10	12
	Iterations	6	6	6	6			
	S_u	1877	10058	52743	≥ 273388			
	$\mathcal{V}_{\text{clk}}$	16	20	24	28			
	time (s)	1.8	19.3	248.1	*memout*			
sF	#processes	4	5	6	7	8	10	12
	Iterations	3	3	3	3	3	3	>2
	S_u	220	727	2378	7737	25080	260998	≥ 2129860
	$\mathcal{V}_{\text{clk}}$	8	10	12	14	16	20	24
	time (s)	0.2	0.4	1.2	4.7	21.5	395.0	*memout*
AT	#processes	3	4	5	6			
	Iterations	6	7	7	>3			
	S_u	3910	46732	547026	≥ 5416550			
	$\mathcal{V}_{\text{clk}}$	12	16	20	24			
	time (s)	2.3	54.7	1946.0	timeout			

Table 15.8. Generation of the timed state space of mutual exclusion protocols.

Benchmark	Iter.	S_u	$\mathcal{V}_{\text{clk}}$	time	Correct
gals	≥ 3	≥ 12	50	*memout*	N
andor	11	124	59	1.0	Y
ipcmos-1	16	99	39	0.6	Y
ipcmos-2	28	1075	64	25.7	Y
pcc	≥ 5	≥ 972	113	*timeout*	Y
stari-1	13	39	35	0.4	Y
stari-2	39	110	40	0.7	Y
stari-3	81	284	52	1.3	Y

Table 15.9. Results for the complete timed state space generation using BDDs for complex timed circuits.

which timed state space generation stopping at first failure found is used. Instead this information can be found in the summary tables Table 15.11 and Table 15.12, under column *TLGe*. In the case of the only incorrect bechmark, *i.e.* the `gals` circuit, this strategy also exhausts the memory limit before finding any failure state.

In Table 15.11 we summarize, for easy comparison, some of the results we have presented. Columns indicate the running times of four methods, namely, timed state space generation (complete or with early stop) and OOTRANSYT with and without static analysis. The name *TLGe* is used to refer to the state space generation using grouped clocks, stopping at first failure found and providing a counterexample, while *TLGc* is used for the complete state space generation without counterexample generation. Note that, to be fair, this latter column should only be compared to other methods for correct systems, since the rest of methods provide counterexamples.

For speed-independent circuits, state space generation (in any of its variants) sometimes wins one of the two methods in OOTRANSYT, but is never the absolute winner on any benchmark. Differences are not very big with the only exception of the `mr1` benchmark. Note that the generation of counterexamples in some cases accounts for a large portion of the time spent by *TLGe*, however, if the table is redone without counting this time, it still never wins (although some of the differences are sensibly reduced).

On the other hand, for complex timed circuits, OOTRANSYT only wins in two of the benchmarks, namely `gals` and `pcc`. In the case of the `gals` example, the main reason for the bad results of the BDD approaches is that the largest constant is 6667 (after all delays have been converted to

Benchmark	Iter.	S_u	gen.time	θ_c	total time
sbuf-read-ctl	21	35	1.0	-	1.0
rcv-setup	2	13	0.4	3	0.4
alloc-outbound	21	47	1.4	-	1.4
ebergen	5	17	0.7	6	1.4
mp-forward-pkt	15	63	1.5	-	1.5
dff	5	16	0.6	7	1.7
half	2	2	0.4	2	0.5
chu133	15	33	3.1	19	16.4
converta	13	26	0.7	16	2.8
nowick	22	46	1.3	-	1.3
chu150	19	59	0.9	25	2.0
sbuf-send-ctl	9	24	2.5	15	71.6
rpdft	2	5	0.6	3	0.7
sbuf-send-pkt2	18	72	7.6	-	7.6
pe-rcv-ifc	18	72	4.3	33	27.9
vme (big)	16	43	2.7	19	9.0
sbuf-ram-write	21	83	32.7	33	46.8
wrdatab	46	229	227.4	-	227.4
ram-read-sbuf	30	115	87.7	-	87.7
mr1	>28	>72	*timeout*	-	*timeout*
mr0	48	226	129.7	-	129.7
tsend-bm (big)	4	26	8.6	9	9.5
trimos-send	4	32	7.9	7	8.6
mmu	4	55	14.0	8	20.6

Table 15.10. Results for the timed state space generation using BDDs and grouped clocks. Analysis was stopped when failure is encountered, and a full counterexample is generated.

integers), which makes the BDDs too big to be handled. For the `pcc` benchmark, the cause is the big number of clock variables that have to be introduced and the complexity of the generated state space.

The difference in performance for the pipeline examples (*i.e.* `stari` and `ipcmos`) is huge. This is caused by the fact that the number of composition needed to show their correctness increases dramatically with each new cell added, thus our approach is limited to a very small number of cells. As shown in [Peñ03] this drawback is not an insurmountable obstacle, since the combination with other techniques may be powerful enough to prove their correctness. For instance in that case they used an assume guarantee argument to show that IPCMOS pipelines of arbitrary length were correct, which requires several verification steps, one of which is to verify `ipcmos-1`. However this is an orthogonal approach that could have been used with any of the methods we are comparing.

We show the results for protocols in Table 15.12. In this case we incorporate three additional rows with the running times of the tools UPPAAL, RED and RABBIT. The small Fischer's protocol is available in all tools and we have manually specified the other two protocols starting from this model.

A brief account of the results would be that static analysis wins on the (full) Fischer's protocol, and the larger instances of the Alur-Taubenfeld's protocol –if the alternative composition method is used. On the other hand RABBIT wins on the small Fischer protocol.

Note, however, that we can divide the methods between the ones that are affected by the constant values and the ones that are robust to changes in the delay bounds (obviously this only if these changes do not affect the resulting timed state space). The methods/tools affected by delay values

Benchmark	TLGc	TLGe	OwoSA	OOTRANSYT	Correct
sbuf-read-ctl	0.8	1.0	0.4	**0.1**	Y
rcv-setup	1.0	0.4	0.2	**0.1**	N
alloc-outbound	1.0	1.4	0.4	**0.1**	Y
ebergen	1.1	1.4	**0.1**	0.3	N
mp-forward-pkt	1.2	1.5	0.7	**0.1**	Y
dff	2.0	1.7	**0.4**	**0.4**	N
half	3.5	0.5	**0.1**	0.2	N
chu133	1.5	16.4	0.6	**0.5**	N
converta	1.6	2.8	0.6	**0.5**	N
nowick	0.8	1.3	0.5	**0.1**	Y
chu150	0.6	2.0	0.6	**0.3**	N
sbuf-send-ctl	2.6	71.6	1.2	**0.6**	N
rpdft	1.7	0.7	**0.2**	0.9	N
sbuf-send-pkt2	2.4	7.6	7.4	**1.4**	Y
pe-rcv-ifc	4.9	27.9	3.0	**2.4**	N
vme (big)	3.4	9.0	**1.6**	2.2	N
sbuf-ram-write	19.6	46.8	3.2	**2.2**	N
wrdatab	122.8	227.4	**100.4**	161.4	Y
ram-read-sbuf	15.9	87.7	24.0	**4.5**	Y
mr1	*timeout*	*timeout*	83.1	**1.4**	Y
mr0	49.3	129.7	164.4	**28.1**	Y
tsend-bm (big)	*timeout*	9.5	**2.0**	25.7	N
trimos-send	*memout*	8.6	**0.2**	13.8	N
mmu	117.0	20.6	**1.1**	29.1	N
gals	*memout*	*memout*	**1.9**	4.2	N
andor	**1.0**	1.3	13.6	3.4	Y
ipcmos-1	**0.6**	0.8	27.5	54.9	Y
ipcmos-2	**25.7**	33.6	*timeout*	*timeout*	Y
pcc	*timeout*	*timeout*	**29.3**	87.8	Y
stari-1	**0.4**	**0.4**	2.0	2.0	Y
stari-2	**0.7**	1.0	51.0	46.3	Y
stari-3	**1.3**	2.1	*timeout*	*timeout*	Y

Table 15.11. Comparing timed state space generation using BDDs and OOTRANSYT on speed-independent circuits and complex timed circuits.

are TLGc, TLGe and RABBIT, while RED, UPPAAL and OOTRANSYT are robust. This means that the scenario for the small Fischer's protocol can be changed completely if we use a different constant value. To be able to compare with published results we have used the smallest possible value (*i.e.* 2) which favours BDD approaches, however, repeating the experiment with a value of 20 heavily penalizes these approaches. In this case OOTRANSYT would also be the winner, since other methods only verify up to 16 processes, while our tool verifies a system of double this size in about five minutes.

15.4.5 Comparing against other iterative refinement strategies

The exact computation of the timed state space is usually difficult as we have seen in previous sections. For this reason iterative refinement methods have emerged as an interesting alternative. However, it would also be interesting to compare between the performance of different iterative refinement strategies. Note that this type of comparison has never been done in the literature. Typically all works implement their technique and then their compare with a non-iterative approach.

F	#processes	4	5	6	7	8	10	12	14	16	32
	TLGc	1.8	19.3	248.1	*mo*						
	TLGe	2.0	24.1	301.7	*mo*						
	UPPAAL	0.8	43.1	*mo*							
	RED	22.8	1343.8	*to*							
	RABBIT	1.6	4.6	13.2	32.8	86.7	209.9	481.7	857.3	1378.7	*to*
	OwoSA	3.1	17.5	144.8	2496.8	*to*					
	OOTRANSYT	**0.2**	**0.5**	**0.7**	**1.4**	**2.9**	**7.7**	**18.8**	**43.0**	**88.7**	**2933.1**
sF	#processes	4	5	6	7	8	10	12	14	16	32
	TLGc	0.2	0.4	1.2	4.7	21.5	395.0	*mo*			
	TLGe	0.2	0.4	1.2	5.0	23.4	*mo*				
	UPPAAL	**0.1**	**0.1**	0.2	0.4	1.9	31.3	*mo*			
	RED	0.4	1.2	3.5	9.8	23.0	122.4	543.4	2601.8	*to*	
	RABBIT	**0.1**	**0.1**	**0.1**	**0.1**	**0.2**	**0.3**	**0.5**	**0.7**	**1.1**	**21.1**
	OwoSA	1.1	2.8	7.1	21.1	72.3	1250.5	*mo*			
	OOTRANSYT	**0.1**	0.2	0.3	0.4	0.8	1.7	3.6	5.2	10.2	312.8
AT	#processes	3	4	5	6						
	TLGc	2.3	54.6	*mo*							
	TLGe	2.4	54.6	*mo*							
	UPPAAL	**0.3**	33.1	*mo*							
	RED	16.5	723.8	*to*							
	RABBIT	2.3	**22.6**	403.0	*to*						
	OwoSA	18.2	493.5	*to*							
	OOTRANSYT	4.6	60.7	2533.2	*to*						
	OOTRANSYT †	3.8	38.6	**274.9**	**3268.3**						

† using the alternative composition method.

Table 15.12. Comparing OOTRANSYT with timed state space generation using BDDs and other tools, on mutual exclusion protocols.

In this section we implement some variants of the methods reviewed in §4.5. In particular we use the modification of delays proposed [AIKY92] but with a symbolic representation of the states.

Two versions of the algorithm are studied. The first one simply scales all clocks: when a failure trace is detected we test if it is timing consistent using the original delays. If it is, we have a counterexample, if it is not, then we compute the biggest scale factor that allows to show that this trace is timing inconsistent and all the delays are scaled accordingly. This has the advantage that less iterations are required, however it might more easily produce unfeasible systems. We call this strategy *global delay iterative refinement*.

On the other hand, the second version uses the original idea of [AIKY92], for each failure trace, first study which events are involved in the timing inconsistency and, only for these ones, refine their delay bounds so that the same trace cannot be generated again. We refer to this strategy as *local delay iterative refinement*.

Global delay iterative refinement

The results when the global delay strategy is used are shown on Table 15.13. Column *Iter* indicates the number of iterations required, while *Pr* is the percentage of precision needed to either prove or disprove the correctness of the circuit. A precision of 0% is equivalent to assume unbounded delay bounds for all events, while a 100% precision is to use the exact delay bounds. Given a certain precision value p, the delay bounds for each event are computed as $\delta(\mathsf{e}) := [\lfloor \delta^l(\mathsf{e}) \cdot p \rfloor, \lceil \delta^u(\mathsf{e}) \cdot p \rceil]$.

Column *max* δ contains the maximum value for a delay bound (initially all delay bounds are converted to integers). As before, $\mathcal{V}_{\text{clk}}$ is the number Boolean variables needed by the clocks, *time*

Benchmark	Iter.	Pr.(%)	max δ	$\mathcal{V}_{\text{clk}}$	time	θ_c	Correct	TLGe	OwoSA	OT
sbuf-read-ctl	1	0.2	3	7	0.3	-	Y	1.0	0.4	**0.1**
rcv-setup	1	0.2	3	8	0.3	4	N	0.4	0.2	**0.1**
alloc-outbound	1	0.2	3	7	0.4	-	Y	1.4	0.4	**0.1**
ebergen	0	-	-	0	**0.1**	5	N	1.4	**0.1**	0.3
mp-forward-pkt	1	0.2	3	9	0.4	-	Y	1.5	0.7	**0.1**
dff	2	14	159	31	0.7	7	N	1.7	**0.4**	**0.4**
half	0	-	-	0	**0.1**	1	N	0.5	**0.1**	0.2
chu133	3	14	159	42	2.2	19	N	16.4	0.6	**0.5**
converta	1	0.2	3	9	**0.5**	15	N	2.8	0.6	**0.5**
nowick	1	0.2	3	12	0.4	-	Y	1.3	0.5	**0.1**
chu150	2	44	20	34	1.4	25	N	2.0	0.6	**0.3**
sbuf-send-ctl	2	28	317	58	3.1	15	N	71.6	1.2	**0.6**
rpdft	2	14	159	32	0.7	3	N	0.7	**0.2**	0.9
sbuf-send-pkt2	2	7	80	41	1.6	-	Y	7.6	7.4	**1.4**
pe-rcv-ifc	4	100	1111	92	35.3	33	N	27.9	3.0	**2.4**
vme (big)	1	14	159	57	2.7	19	N	9.0	**1.6**	2.2
sbuf-ram-write	2	14	159	67	6.0	33	N	46.8	3.2	**2.2**
wrdatab	2	57	633	85	**72.1**	-	Y	227.4	100.4	161.4
ram-read-sbuf	4	14	159	65	8.6	-	Y	87.7	24.0	**4.5**
mr1	2	6	62	41	3.4	-	Y	*to*	83.1	**1.4**
mr0	3	59	659	135	62.7	-	Y	129.7	164.4	**28.1**
tsend-bm (big)	2	14	159	87	9.1	9	N	9.5	**2.0**	25.7
trimos-send	1	14	159	78	3.9	7	N	8.6	**0.2**	13.8
mmu	2	7	80	71	9.8	8	N	20.6	**1.1**	29.1
gals	3	2	159	30	11.7	15	N	*mo*	**1.9**	4.2
andor	4	26	129	44	2.1	-	Y	**1.3**	13.6	3.4
ipcmos-1	3	21	48	38	1.8	-	Y	**0.8**	27.5	54.9
ipcmos-2	3	21	48	61	59.7	-	Y	**33.6**	*to*	*to*
pcc	≥ 3	≥ 5.3	≥ 60	≥ 52	*memout*	-	Y	*to*	**29.3**	87.8
stari-1	2	0.02	8	13	0.6	-	Y	**0.4**	2.0	2.0
stari-2	2	0.05	16	16	1.3	-	Y	**1.0**	51.0	46.3
stari-3	2	0.05	16	20	3.3	-	Y	**2.1**	*to*	*to*

Table 15.13. Global delay iterative refinement on speed-independent circuits and complex timed circuits.

is the total time spent in all iterations and θ_c is the length of the counterexample found, if any. To easily compare with previous approaches we have included three additional columns at the end of the table, for space constraints we have labeled the column OOTRANSYT as *OT*.

Global delay iterative refinement is a simple strategy: we start with precision $\frac{1}{d}$ where d is the largest lower delay bound in the system. A lower precision has no chance to remove any trace from the timed state space, since all lower delays would be 0. If the system is correct, then we finish. If a failure trace is found, we check if it is timing consistent (using the original delays). If it is, we already have a counterexample. If it is not, then we start multiplying precision by 2, until the trace is timing inconsistent using the modified delays. With this new precision we recompute the state space and we search for failure traces as before.

Surprisingly, the increase in precision does not necessarily imply a smaller timed state space (although most of the times it would have this effect). The reason is that, individually, each delay bound is overapproximated by a quantity that depends on the modulus of the quantum used and the actual value of the bound. The quantum is the inverse of the precision, and given an interval $[l, u]$, the value of the distortion introduced by scaling using the quantum is $(l \bmod \frac{1}{p}) + ((\frac{1}{p} - (u \bmod \frac{1}{p})) \bmod \frac{1}{p})$. So using a precision p it is only guaranteed that the distortion is in the following interval $[0, 2(\frac{1}{p} - 1)]$, but it can perfectly be that in some case the distortion is 0 with a precision $p' < p$ and that is bigger than 0 with p.

As an example consider a system in which there are two events a and b with delays $\delta(\mathsf{a}) = [0, 2]$ and $\delta(\mathsf{b}) = [4, \infty)$. Assume they both become enabled at the same time in some state. Clearly, a will fire before b. The same conclusion is achieved using a precision of $\frac{1}{2}$, which transforms the delays into $\delta(\mathsf{a}) = [0, 1]$ and $\delta(\mathsf{b}) = [2, \infty)$ and introduces no distortion. However, if we use a greater presicion of $\frac{2}{3}$ the delays become $\delta(\mathsf{a}) = [0, 2]$ and $\delta(\mathsf{b}) = [2, \infty)$, thus allows b to fire at the same time as a.

The results of Table 15.13 show that the global delay approach usually requires very few iterations to produce an approximation good enough. For most of the benchmarks the required precision is quite low, except for the `pe-rcv-ifc` circuit. This allows to save a considerable amount of Boolean variables to represent the clocks. For instance, in the `mmu` example, TLGe needs 164 variables while global delay refinement only needs 71.

The strategy is more successful than TLGe for the decomposed speed-independent circuits, with the only exception of `pe-rcv-ifc` circuit, in which full precision is required. The most relevant result is that circuit `mr1` can now be verified with a small time while TLGe exhausts the time limit.

If compared to OOTRANSYT, it ties with it in a couple of benchmarks in which the untimed analysis already reveals a timing consistent failure trace. It wins on the `wrdatab` example, which is one of the difficult examples for OOTRANSYT for this set of benchmarks (an analysis of the bottlenecks for this circuit can be found in §15.5).

For the complex timed circuits the scenario is completely different, since it contains benchmarks that are very suitable for the timed state space generation as well as benchmarks for which this task is very difficult. If we look to this difficult examples (`gals` and `pcc`), the global delay refinement can now complete one of them. In this case the blowup problem was caused by a large value of a delay bound, but since a precision of 2% is enough to find a counterexample, now the analysis finishes in a reasonable time (although OOTRANSYT is five times faster). The `pcc` could not be completed even with a small precision showing that the problem is related to the resulting complexity of the state space once clocks are added into the system.

The rest of the benchmarks of this set are better verified by the TLGe method. Although many of them can be verified using a small precision (or very small, like in the `stari` suite), the overhead of creating the different instances of the problem and generating the state space several times does not compensate for the cost of considering the complete clocks.

For mutual exclusion protocols the global delay refinement approach only provides some benefit for the (big) Fischer's protocol. The small version of the protocol already has the smallest possible delay constants, thus precision required is 100%. On the other hand, the big version only needs a 40% precision and two iterations to finish with substantial gains in the running time. However the unoptimal order of the clock variables still does not allow to verify more than six processes. The results are shown in the next table:

F	#processes	4	5	6	7
global delay	Iter.	2	2	2	≥ 2
	Pr.(%)	40	40	40	≥ 40
	max δ	4	4	4	4
	$\mathcal{V}_{\text{clk}}$	24	30	36	42
	time	0.9	5.3	52.3	*mo*
TLGe	time	2.0	24.1	301.7	*mo*
OOTRANSYT	time	**0.2**	**0.5**	**0.7**	**1.4**

Benchmark	Iter.	Pr.(%)	max δ	Σ_{clk}	$\mathcal{V}_{\mathrm{clk}}$	time	θ_c	Correct	TLGe	OwoSA	OT
sbuf-read-ctl	3	0.2	2	5	4	0.8	-	Y	1.0	0.4	**0.1**
rcv-setup	1	0.2	3	2	3	0.3	4	N	0.4	0.2	**0.1**
alloc-outbound	5	0.2	2	8	7	1.6	-	Y	1.4	0.4	**0.1**
ebergen	0	-	-	0	0	**0.1**	5	N	1.4	**0.1**	0.3
mp-forward-pkt	4	0.2	2	6	6	1.2	-	Y	1.5	0.7	**0.1**
dff	5	28	256	7	26	1.8	8	N	1.7	**0.4**	**0.4**
half	0	-	-	0	0	**0.1**	1	N	0.5	**0.1**	0.2
chu133	8	28	256	10	33	9.2	20	N	16.4	0.6	**0.5**
converta	1	0.2	2	2	3	**0.3**	15	N	2.8	0.6	0.5
nowick	2	0.2	2	4	4	0.6	-	Y	1.3	0.5	**0.1**
chu150	1	44	16	2	9	**0.3**	25	N	2.0	0.6	**0.3**
sbuf-send-ctl	5	57	512	10	39	3.1	15	N	71.6	1.2	**0.6**
rpdft	3	28	256	4	17	0.8	3	N	0.7	**0.2**	0.9
sbuf-send-pkt2	11	28	256	14	55	16.2	-	Y	7.6	7.4	**1.4**
pe-rcv-ifc	10	100	900	15	60	78.6	33	N	27.9	3.0	**2.4**
vme (big)	8	15	133	14	39	6.5	19	N	9.0	**1.6**	2.2
sbuf-ram-write	10	30	330	18	47	23.8	33	N	46.8	3.2	**2.2**
wrdatab	≥ 14	100	≥ 900	≥ 17	≥ 76	*mo*	-	Y	227.4	**100.4**	161.4
ram-read-sbuf	16	28	256	24	61	50.8	-	Y	87.7	24.0	**4.5**
mr1	≥ 12	100	≥ 900	≥ 18	≥ 76	*mo*	-	Y	*to*	83.1	**1.4**
mr0	≥ 22	100	≥ 900	≥ 28	≥ 124	*to*	-	Y	129.7	164.4	**28.1**
tsend-bm (big)	6	100	900	9	40	11.5	8	N	9.5	**2.0**	25.7
trimos-send	1	7	3	3	4	0.8	7	N	8.6	**0.2**	13.8
mmu	2	28	256	5	20	3.7	8	N	20.6	**1.1**	29.1
gals	4	3	16	7	16	**1.6**	15	N	*mo*	1.9	4.2
andor	24	51	256	24	52	20.5	-	Y	**1.3**	13.6	3.4
ipcmos-1	18	42	64	17	41	12.2	-	Y	**0.8**	27.5	54.9
ipcmos-2	≥ 27	100	≥ 151	≥ 30	≥ 80	*mo*	-	Y	**33.6**	*to*	*to*
pcc	≥ 12	≥ 16	≥ 144	≥ 16	≥ 50	*mo*	-	Y	*to*	**29.3**	87.8
stari-1	12	100	330	19	32	5.4	-	Y	**0.4**	2.0	2.0
stari-2	14	100	330	30	40	29.7	-	Y	**1.0**	51.0	46.3
stari-3	18	100	330	43	49	119.3	-	Y	**2.1**	*to*	*to*

Table 15.14. Local delay iterative refinement on speed-independent circuits and complex timed circuits.

For the Alur-Taubenfeld's protocol the results are worse than TLGe because a 100% precision is required to verify it.

Local delay iterative refinement

Instead of considering all the clocks, another possible strategy is to only bound the events that are related to failures. The *local delay iterative refinement* starts by finding the shortest negative cycle in the timing consistency graph associated to a failure trace. This negative cycle reveals which events are essential to prove the timing inconsistency of the trace. Then we compute the precision required for this particular set of events to still have a negative cycle, and the involved events are added to the set of bounded events. All bounded events are then scaled using the least precision that allows to prove that all negative cycles found are still negative.

Table 15.14 shows the results obtained by the method on speed-independent circuits and complex timed circuits. When compared to the global delay approach, local delay requires more iterations to finish. The reason is that it is possible that the smallest negative cycle is associated to a trivial bad ordering of events fired, thus that a trace in which some firing order is exchanged exists in the system. It is difficult for this method to distinguish between an essential inconsistency that will allow to remove many traces or a very local inconsistency that can be easily solved by exchanging the order of the fired events. This is a common problem to approaches considering single traces (like TRANSYT) or in which the smallest negative cycle is used (as in [BSV93]).

Moreover, in some benchmarks the precision required is also greater than when the global delay is used. The reason in this case is also related to the use of the smallest negative cycle. In this situation, usually, the amount of relaxation that one can introduce is much less than if a more negative (and longer) cycle is considered. Since global delay takes into account all the events in the trace, it is possible that with less precision the trace is shown timing inconsistent than if we only restrict to the smallest negative cycle. This is the case, for instance, of the `tsend-bm` benchmark. On the other hand, to consider the minimum set of events for each individual failure trace often results in having a small set of bounded events and this might compensate for the previous problem. As an example, the `tsend-bm` benchmark requires 100% precision with local delay, when only a 14% was needed using the global delay. However, the amount of clock variables needed is 40 compared to the 87 of the global delay. This reduction is achieved because only 9 of the 39 events are actually necessary to find the first timing inconsistent failure trace.

Another important difference with respect to the global delay approach is that local delay uses the information of the negative cycles to determine, for each event, which of the bounds have to be considered. It can perfectly be that only the lower bound of an event is needed, so it is defined as upper unbounded even if in the original delay interval it has a finite bound. This explains why in Table 15.14 there are benchmarks with a higher precision than in Table 15.13 that have smaller maximum constants.

Compared with the approach of [AIKY92], this methodology is similar to one of the proposals presented as future work in that paper (a work never continued as far as we know). In fact the method they actually implemented spends more time determining which events are worth considering. Instead of just adding the new relevant events (based on the last failure trace) to the ones previously found, all the set of failure traces encountered is considered. This makes the optimization problem harder but has a positive sideefect: it is less likely that in the set of relevant events there are events that contradict a single trace because of trivial bad event ordering, although this effect increases as long as the set of failure traces becomes bigger. We believe a similar benefit can be achieved by introducing some partial order reasoning when the negative cycles are considered, but is an alternative we have not explored yet.

For mutual exclusion protocols, as before, the local delay refinement approach only provides some benefit for the (big) Fischer's protocol. The results are shown in the next table:

F	#processes	4	5	6	7	8
global delay	Iter.	4	6	7	9	11
	Pr.(%)	40	40	40	40	40
	max δ	2	2	2	2	2
	Σ_{clk}	8	10	12	14	16
	$\mathcal{V}_{\text{clk}}$	16	20	24	28	32
	time	1.0	3.0	9.1	47.5	334.1
TLGe	time	2.0	24.1	301.7	*mo*	
OOTRANSYT	time	**0.2**	**0.5**	**0.7**	**1.4**	**2.9**

Notably, this approach verifies up to eight processes thus it is more successful than the global delay approach. Although more iterations are needed, the fact that only the lower bound of the delay event is used, allows to reduce the maximum constant from three to two. This, combined with the fact that only two events per process are needed, reduces considerably the amount of clock variables that must be introduced, thus allows a faster verification and reduces to some extent the BDD blowup due to unoptimal variable ordering.

Benchmark	Total	SA (%)	Search (%)	Adjust (%)	Best (%)	Compose (%)
wrdatab (SA)	161.4	1.6 (1%)	2.1 (1%)	129.9 (80%)	22.4 (14%)	3.0 (2%)
mr0 (R)	165.0	-	22.5 (14%)	38.4 (23%)	7.7 (5%)	64.6 (39%)
fischer7	2496.8	-	114.2 (5%)	8.6 (0%)	0.3 (0%)	2350.8 (94%)
fischer8 (L)	948.1	-	399.7 (42%)	23.1 (2%)	0.5 (0%)	490.1 (52%)
fischer32 (SA)	2933.1	2922.1 (100%)	8.9 (0%)	-	-	-

Table 15.15. Time spent in each phase for some examples with significant verification time.

15.5 Analysis of the proposed methodology

Wihtout a doubt, it is important to know why something works. But to improve it, one has to focus on the reasons why it does not work. In this section we analyze the bottlenecks of our approach.

We obtain some interesting insight by dividing the time spent to verify the larger systems between the several phases of the methodology. Note that the aggregation of all the times of the phases is slightly smaller than the total time because of some intermediate steps that we did not consider as they are not relevant.

In Table 15.15 we can see the detail of the verification of the benchmarks with larger running times. The names of the benchmarks sometimes contain indications of the parameters used in the analysis. For instance *(SA)* indicates that static analysis was used, *(R)* marks when reordering was activated and *(L)* denotes the fact that the alternative composition method was used.

The meaning of the columns is as follows: *Total* is the total time, *SA* is the time used by static analysis, *Search* is the time spent in searching a failure trace, *Adjust* is the time needed to find A CESAR whose zone contains the failure trace and perform the timing analysis on it, *Best* is the time required to find the smallest CESAR that avoids all failures in the zone and, finally, *Compose* is the time spent in the composition of the ECTRA and the original system. In each column we also provide the percentage of time that represents that phase over the total time.

For the `wrdatab` example with static analysis, most of the time is spent adjusting the CESAR. The reason is that deep failure traces (about 90 fired events) are found in the latest iterations. This is not necessarily a problem because it can be that restrictions have to be added very soon in the exploration process that limits the concurrency (hence the size) of the zone. This restrictions can come from the shape of the system or from previous compositions. In this particular case the first long trace found is the main responsible of the total time needed to adjust the CESARs: restrictions do not prune the zone early and the resulting structure has lots of configurations and arcs to be checked. On the other hand, the rest of failure traces, which are larger, take much less time to adjust. The reason is that the precedences introduced in the latter composition allow to prune the zone much earlier afterwards.

For the `mr0` example the time is distributed basically between the adjust phase, the composition phase and reordering (it does not appear on the table but it represents most of the 21% of the total time). If reordering is not used total time increases to 10 minutes. Most of the increase goes to the search and composition phases. The reason is that BDD sizes of the TRs grow fast and then any phase that requires intensive manipulation of them is severly penalized. This highlights the fact that the order in which the timed variables are introduced should be further studied (both TRANSYT and OOTRANSYT simply add them at the bottom of the order).

Although a good position for the added variables could be computed, in some cases variable reordering would be still required. The reason is that composition tends to create correlations between variables that, for instance, never appeared together in the same TR before. At the same time the shape of the reachable set of states changes. Consequently, it is likely that reordering between preexistant variables helps reducing the representation size. However the benefits of reordering are also less probable as more compositions have been previously performed.

Tipically at the beginning of the verification process most the representation size is due to the reachability set. But as more iterations are done, the weight changes from the reachability set to the TRs. As TRs start incorporating more and more variables they become of the same complexity as the reachability set, with the difference that there is only one reachable set but there can be hundreds of TRs. This makes it more difficult for the reordering algorithm to obtain good results in later iterations: if to find a suitable order for a single Boolean function is difficult, the same problem when many big functions have to be minimized is much more complex (and the space saving can be much less).

For the Fischer's protocol with 7 processes (without static analysis) the bottleneck is clearly the cost of the composition. This should come as no sourpise because all the failure traces in this case are short and the zone computation is very easy. Moreover to find a good CESAR is also simple, so most of the time the algorithm is performing compositions. It is interesting to note that, like in the Alur-Taubenfeld protocols, if an alternative composition scheme is used, better results can be achieved. Repeating the experiments using this other composition method we obtain the following results for OOTRANSYT without static analysis:

#processes	4	5	6	7	8	10
time (default composition)	3.1	17.5	144.8	2496.8	*to*	
time (alternative composition)	2.3	7.5	31.1	131.2	948.1	*to*

This shows the importance of the exact composition implementation, a subject that deserves a deeper study. Note that it is not the case that the alternative composition method always outperforms the default one (although for our protocol benchmarks it is the case because they have very small ECTRAs in which disambiguation problems seldom appear). For instance for the circuit benchmarks it is usually the case that it the same performance and in many cases is notably slower than the default one.

If we analyze in detail the usage of time in the Fischer's protocol with 8 processes using the alternative composition, we find that now time is almost equally distributed between the search and the composition phases. In this case about 90% of the time spent in the search phase is spent during the last iteration, in which the state space is completely generated. Because by default this search is done using BFS exploration, which is not very suitable for concurrent systems, but is very convenient to find the shortest counterexample trace, while there are shallow errors the failure search is fast, but when they become deeper and this is combined with big TRs, then this part becomes slow. This benchmark is a perfect example, since until last iteration all failures are of length 10, the search time increases slowly (because of the bigger TRs), but in the last iteration, in which the radius of the system, of length 38, has to be traversed, then it takes an enormous amount of time in comparison with previous iterations.

When using static analysis on the Fischer's protocol, results are much better because the static analysis is able to remove all failures, thus in fact all the time is spent in that phase as the `fischer32` example shows.

15.6 Conclusion

Methodologies based on iterative refinement techniques are inherently complex, thus there are plenty of places in which bad decisions can lead to poor performance. Conversely, there are lots of opportunities to improve it. Although our implementation cannot be considered as tuned for performance, the results are promising, since it compares favorably in many cases to other approaches, including other iterative refinement methodologies, as well as state-of-the-art verification tools. On the other hand it is also true that other alternatives are more successful in some specific benchmarks.

This should come at no surprise if the set of experiments considers a wide enough set of benchmarks and a wide range of techniques. Given the difficulty of the problem that we want to solve, it is clear that, in general, there is no best verification technique. We believe there are only techniques more suited to some type of systems than others.

Under this perspective it is interesting to outline for which kind of systems our methodology works well. Clearly the performance is related to the number of iterations, which is related to the number of *reasons* why a particular benchmark is correct. For systems in which there are few of these *reasons*, our technique should be interesting. To increase the number of systems that fall inside this cathegory, we have also introduced some static analysis techniques that allow, in some cases, to discharge some of the proofs and change them for a series of reachability analysis.

Chapter 16

CONCLUSIONS

Summary

This chapter presents the conclusions of this work. In first place, its contributions in several aspects of the formal verification of timed concurrent systems are summarized. Next, we overview a number of open issues for future developments paying special attention to the most relevant bottlenecks we have found in our work. Chapter concludes with some considerations on the role of our proposals in the formal verification scene.

16.1 Introduction

Technology based societies rely on the correct operation of their technology. Formal verification is the discipline of proving that a system works as intended by using mathematical reasoning, usually with the assistance of computers. The aim of this work is to increase the syze of the systems that can be verified and the speed at which this task is completed.

16.2 Contributions

One of the most straightforward verification techniques is to generate the state space of the system and check that no bad behavior is present. Given that the number of states in non-trivial systems is often very large, symbolic representations are normally used.

The generation of the state space for timed systems is a complex problem due to the necessity to keep track of the time elapsed since an event has become enabled. A successful approach to cope with this problem is to use iterative refinements: first timing information is abstracted, and is only incorporated gradually into the system as long as it is required to disprove the failures found during the state space generation.

Although results have supported the validity of iterative refinement techniques, there are few works in this area. One of them uses the relative order between events, rather than an explicit clock, to add timing information into the model. We have extensively studied this approach.

First of all we have described the fundamental relation by which a timing inconsistent trace can be used to prove than another trace is also timing inconsistent. This relationship is too much general to be used in practice, but is useful to derive from it other restricted types of relations that can have an easy symbolical implementation. This is the case of the enabling compatible relation between traces. Although this concept is not new, we have reformulated it to be more general than its preexisting versions. Moreover we have shown that all enabling compatible traces with a given trace θ can be obtained from θ by applying a small number of simple transformations.

The core of the method is to remove all the enabling compatible traces with a timing inconsistent failure trace. To do so we rely on an automaton that recognizes them (ECTRA) that is composed with the original system. This approach differs from previous works, but allows to use all the automata theory to our advantage. Nevertheless it also helps to point out some problems that were previously unnoticed. For instance this is the case of indeterminacy, which we have discussed in Chapter 10.

If indeterminacy is not considered, then it might be the case that not all enabling compatible traces are removed. Given the nature of our automata, which usually have an enormous input alphabet, it is not possible to use the standard determinization algorithms, thus we provide a tailored one. However to use it is expensive and useless in many cases, so we also give some algorithms to detect if the determinization is really needed.

We have also considered the question of the convergence of algorithms based on the composition with this kind of automata (Chapter 12). Our conclusion is that if algorithms are not "loop aware" then the algorithms are less powerful than regular timed state space generation. We have described several classes of systems for which there is an algorithm that (theoretically) terminates and we have established a hierarchy between these classes.

ECTRAs can be generated from a single trace, but this strategy would yield a very slow verification process. Fortunately they can be used to work with several concurrent traces at the same time, although it requires an efficient way to make a timing analysis on this set of traces. For this reason an structure that concisely represents them is introduced (CESAR). This event structure is different from previously used ones to avoid many problems arising in several phases of the method, that are discussed in Chapter 7. Basically it has an asymmetric disabling relation plus a relation that can be used to restrict the set of considered traces. This allows to represent arbitrary sets without affecting the set of enabled events in each trace (a siedeffect of using the causality relation with this intention).

CESARs allow a fast analysis of which of the traces it represents are timing inconsistent. However the analysis is not as simple as previously thought, as shown in Chapter 8. The reason is that each restriction can only have an effect once they appear in the traces (*i.e.* timing consistency does not depend on future events), thus timing analysis has to be performed in several timing zones, depending on which restrictions have been seen or not. Moreover the coherence of timing analysis imposes some conditions on the definition of event structure that were previously unrealized.

The complexities of using CESARs do not end here. In Chapter 11 we show that if a CESAR does not accurately represent parts of the state space, then it can be the case that a timing inconsistent failure trace cannot be removed from the system. Hence we provide an algorithm to adjust a CESAR to the real state space by explicit exploration.

Taking into account all these elements we have proposed a verification methodology, that we have implented in the tool OOTRANSYT. This approach works well if the number of compositions required is not high. Sometimes many of the compositions are trivial precedences between events and it is not worth the effort to go through all the steps, thus we have also introduced some optimizations that allow to easily remove these precedences by performing a particular type of reachability analysis.

To check the validity of our approach, we have compared OOTRANSYT with the tool TRANSYT, that implements the methodology more closely related to ours. Results show that performance improves in many aspects: running time is smaller as well as the number of compositions required.

We have also compared our technique with other state-of-the-art verification tools, with published results on the same benchmarks and with our implementation of a timed state space generation using BDDs. Results here are not clear, since the best methodology heavily depends on the benchmark. Not only this but also the parameters used have a big impact. For instance by changing the parameters of the Fischer's protocol, one of the more popular and easier benchmarks, one can make any of the studied tools to win. We believe that the relevant aspect here is that performance in many cases is comparable to state-of-the-art tools, a fact that was not previously known for this type of technique.

Finally we have also compared to other iterative refinement techniques, since tipically state-of-the-art tools compute the exact timed state space (although some of them use conservative approximations). In this case the results are generally favorable to our tool with the exception of benchmarks for which the number of compositions is very high.

16.3 Future work

For untimed state space generation, clearly the problem are that BDDs are not the best option for concurrent systems. Althouhg partial TRs solve part of the problem of using BDDs, still most of the structure has to be traversed to produce even a very local modification affecting few variables. When the total number of variables is big, this overhead becomes very important. An intersting possibility would be to use our scheduling algorithms with a data structure that allows a more direct acces to the relevant nodes of the structure, like MDDs.

Regarding the verification of timed systems using our methodology, there are several aspects that can be improved. First of all the algorithm to adjust the CESARs does not make use of any partial order technique to simplify the checking. Secondly the method of composing an ECTRA and a LzTS should be studied more in depth, as results have shown that different composition methods might have a big impact on running times. Another possibility is to avoid completely the creation of a monolithic LzTS when composing, instead one could keep the ECTRAs with the mapping of states. Although this option would alleviate the problem of the system size grow the complexity of finding a failure trace in this scenario would also be greater, thus it is likely that a mixed strategy (*i.e.* use standard composition until a threshold is reached and then switch to this latter form of composition) would be the best approach.

In some cases many similar failure traces have to be processed because of the inability of CESARs to express disjunctive causality. Using a more general model would reduce the number of iterations in these cases, however the algorithms to deal with timing analysis and to adjust the event structures would be more complex.

In other examples most of the time is spent in reachability analysis or searching for a counterexample. Another possibility would be to use SAT solvers to perform this part of the analysis (bounded model checking), since they are very well suited for this task, specially when failures are shallow. However, currently the translations of timed concurrent systems to Boolean formulas are not as efficient as they synchronous counterparts.

16.4 Conclusion

Looking at the results achieved, our conclusion is that our methodology is interesting for systems in which the number of "reasons" why the system is correct is not very high. From our experience it would be a surprise that a verification requiring more than a hundred of compositions finishes. For this kind of systems other approaches are more suited.

Clearly it has two important drawbacks. The first one is that is less convergent than exact timed state space generation. The second one is its complexity: while the explanation of the standard algorithms to compute the timed state space requires few pages and few weeks to be implemented, the size of this work speaks for itself. Nevertheless benefits are sometimes important, but still it is possible that they can be matched by standard algorithms by introducing some optimizations or by combining with other techniques. Note that some techniques are orthogonal to our methodology as well as standard algorithms so both approaches could benefit from them, however not all techniques enjoy this property.

To conclude and revisit our discussion on lazyness that starts this work, relative timing approaches offer good opportunities, but are not appropriate for lazy people.

Appendix A

Detailed comparison with [Peñ03]

I have not failed. I've just found 10,000 ways that won't work!

—Thomas Alva Edison

A.1 Model differences

Both works use the same models, with only a minor variation on the definition of LzTS (see Definitions 2.7 and 2.8). Both versions have their own advantages, but we preferred to develop our theory with a model that is able to keep decoupled the untimed and timed behaviors of the system.

A.2 Trace timing consistency checking differences

In [Peñ03] the case where an event remains enabled until the end of the trace was not considered, hence the related equations were not added into the corresponding LP problem. However this case must be taken into account when dealing with fragments of traces.

Even if just considering complete traces (that is, either finite traces ending in a sink state or infinite traces) one must handle this cases: traces ending in a sink state cannot have any remaining event to fire, obviously, but infinite traces yes *e.g.* a system with a loop in which an event e is always enabled in any of the states of the loop. In this example failing to require that the upper delay bound of e is ∞ will end up in the possibility that the trace is considered timing consistent while clearly it is not if the upper bound is finite.

A.3 Enabling compatible mapping differences

Definition A.1 (Enabling-compatible trace mapping [Peñ03])

Let $\theta = \cdots \longrightarrow E_0 \xrightarrow{\mathsf{e}_0} E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots \xrightarrow{\mathsf{e}_n} E_{n+1} \longrightarrow \cdots$ *be a trace over the alphabet of events* Σ *and let* $\theta' = E'_1 \xrightarrow{\mathsf{e}'_1} E'_2 \xrightarrow{\mathsf{e}'_2} \cdots \xrightarrow{\mathsf{e}'_m} E'_{m+1}$ *be a trace over the alphabet* $\Sigma' \subseteq \Sigma$. *Let* $\theta_t = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots \xrightarrow{\mathsf{e}_n} E_{n+1}$ *be a fragment of* θ.

An enabling-compatible *mapping of* θ_t *onto* θ' *is a function* $\mathsf{mp} : \{E_1, \ldots, E_{n+1}\} \mapsto \{E'_1, \ldots, E'_{m+1}\}$ *such that:*

a) $\mathsf{mp}(E_1) = E'_1$ (initialization)

b) $\forall\, 1 \leq i \leq n,\ \mathsf{mp}(E_i) = E_i \cap \Sigma'$ (projection)

c) $\forall\, 1 \leq i \leq n,\ (\mathsf{mp}(E_i) = \mathsf{mp}(E_{i+1}) \ \wedge\ \mathsf{e}_i \notin \Sigma') \ \vee$
$(\mathsf{mp}(E_i) = E'_j \ \wedge\ \mathsf{mp}(E_{i+1}) = E'_{j+1} \ \wedge\ \mathsf{e}_i = \mathsf{e}'_j)$ (firing)

■ A.1

The mapping of θ onto θ' is a function that preserves the enabledness of the events in Σ'. Initially, the events enabled in E'_1 must also be enabled in E_1 (initialization condition). Next, the events of

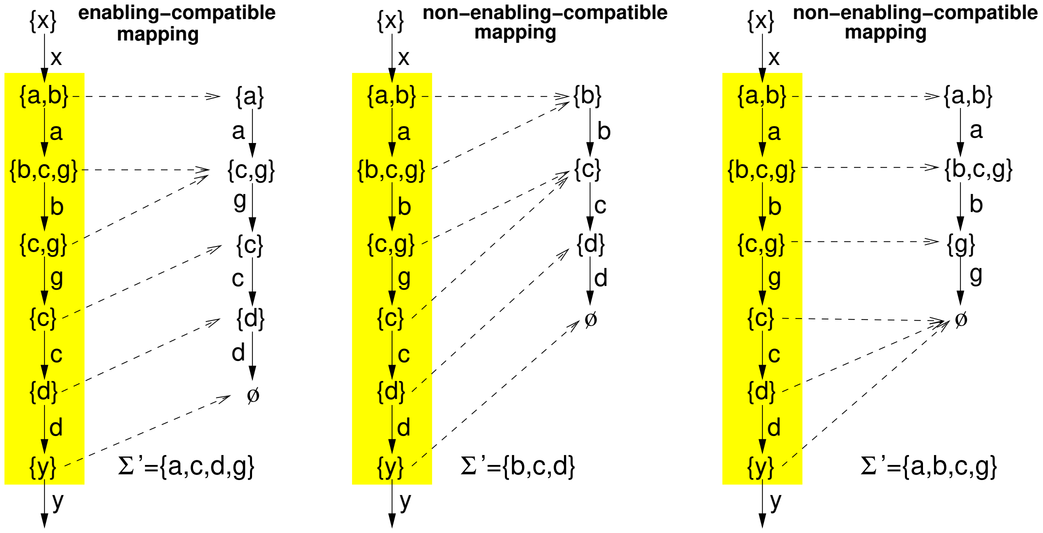


Figure A.1. Enabling-compatible (left) and non-enabling-compatible mappings (center and right).

Σ' enabled along θ and θ' must be the same (projection condition). Moreover, θ may fire events that are not relevant to θ' (when $\mathsf{mp}(E_i) = \mathsf{mp}(E_{i+1})$ in the firing condition). The second part of of the firing condition captures implicitly the disabling of events produced by the firing of e_i in θ and the firing of e'_j in θ', that is if an event is disabled in one trace it must be disabled also in the other trace. Since the firing time of an event only depends on its enabling time and its delay (see §2.2), this notion will allow us to apply the timing analysis of θ' to θ in the fragment θ_t.

Figure A.1 shows three examples of trace mapping of the shadowed fragment. The mapping at the left, with $\Sigma' = \{\mathsf{a},\mathsf{c},\mathsf{d},\mathsf{g}\}$, is enabling-compatible. The mapping at the center, with $\Sigma' = \{\mathsf{b},\mathsf{c},\mathsf{d}\}$, is not enabling-compatible since it violates the projection condition when taking $E_i = \{\mathsf{b},\mathsf{c},\mathsf{g}\}$ and $\mathsf{mp}(E_i) = \{\mathsf{b}\}$. Clearly, a enables c in θ, whereas c is enabled by b in θ'. The mapping at the right, with $\Sigma' = \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{g}\}$, is not enabling-compatible according to Definition A.1, since it violates the projection condition when taking $E_i = \{\mathsf{c},\mathsf{g}\}$ and $\mathsf{mp}(E_i) = \{\mathsf{g}\}$. The firing of b in θ' disables c whereas this does not happen in θ. It must be noted that this particular trace is considered enabling compatible by Definition 5.16. In Figure A.2 we show the associated graphs. Second graph contains possible negative cycles $\delta^u(\mathsf{a}) + \delta^u(\mathsf{c}) + 0 - \delta^l($b and $\delta^u(\mathsf{a}) + \delta^u(\mathsf{g}) + 0 - \delta^l($b which also appear in the first graph. Thus trace θ is timing included in θ'. Definition 5.16 subsumes Definition A.1 so pruning capabilities of derived composition methods will also respect this hierarchy.

Please note that Definition A.1 only works under the assumption that all events are distinct (*i.e.* no event repetition is allowed). We illustrate this in Figure A.3.

Also the definition does not consider conservative events. This may cause undesired results as in Figure A.4 where traces θ_6 and θ_7 satisfy Definition A.1. However it is the case that θ_7 can be timing inconsistent while θ_6 is timing consistent (for instance with $\delta^u(\mathsf{x}) = 3$, $\delta^u(\mathsf{b}) = 1$ and $\delta^l(\mathsf{a}) = 2$), so obviously there is a problem in the definition.

The following theorem was the main theoretical result of the verification approach presented in [Peñ03]. The theorem relies on the notion of enabling-compatibility between traces. Since the time at which events fire or are disabled only depends on the enabling instant plus certain delay, the timing analysis on a trace is also applicable to all the traces that share the same enabling, disabling and firing order. That is the timing analysis applies to the set of enabling-compatible traces.

THEOREM A.1 (ENABLING-COMPATIBILITY AND TIMING CONSISTENCY)
Let θ, θ' and θ_t be traces with the same conditions as in Definition A.1. Let mp *be an enabling-compatible mapping from θ_t onto θ'. Let δ^l and δ^u be two functions that*

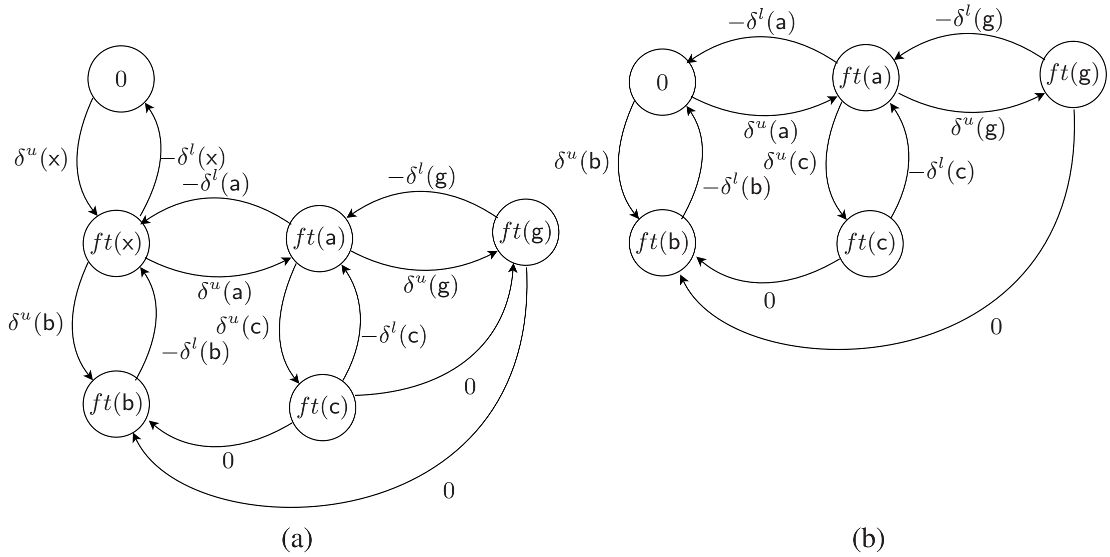


Figure A.2. (a) The associated graph to the generic timing consistency system of the fifth trace appearing in Figure A.1. (b) Associated graph to the generic timing consistency system of the last trace of the same figure. Although enabling compatibility Definition A.1 states that both traces are not enabling compatible, using Definition 5.16 they are. Comparison of associated graphs allows to determine that first trace is timing included in the second one.

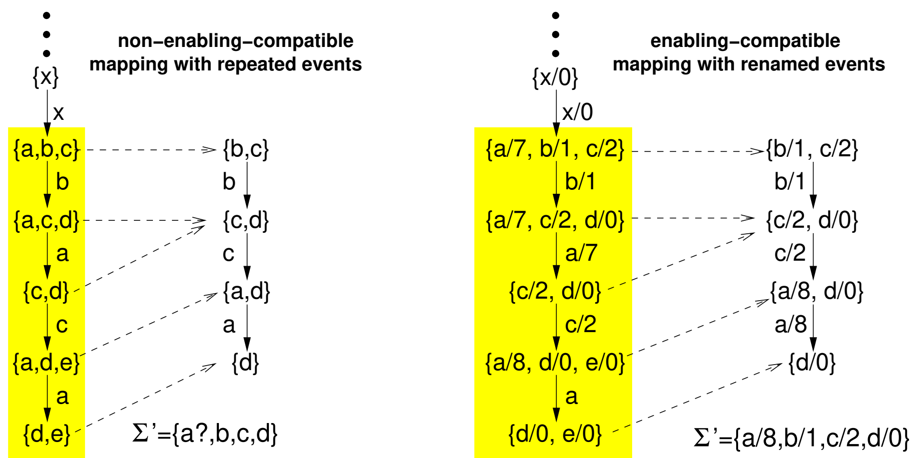


Figure A.3. (left) Non-enabling compatible mapping according to Definition A.1 because of the repeated events *i.e.* event a cannot belong to Σ' and, at the same time, must belong to it. Using Definition 5.16 the mapping is enabling compatible. (right) Renaming allows to create an enabling compatible mapping according to Definition A.1.

assign arbitrary min/max delays to the events in Σ' *and* 0 *and* ∞ *delays to the events in* $\Sigma \setminus \Sigma'$*, respectively. Then,* θ *is timing consistent* $\iff$ θ' *is timing consistent.* ■ A.1

The theorem in itself is correct, as well as the proof that we omitted here. The problem is that there is a missing step: we want to show that the timing analysis of a given trace can be reduced to the timing analysis of those events that are causally related (events in Σ') and, therefore, the events that are concurrent with all the events of Σ' can be abstracted out. However the theorem assigns a delay $[0, \infty)$ to the events not in Σ', then the theorem states something about the timing analysis of a trace that is not the one that we actually have. Obviously, when all the events can have arbitrary

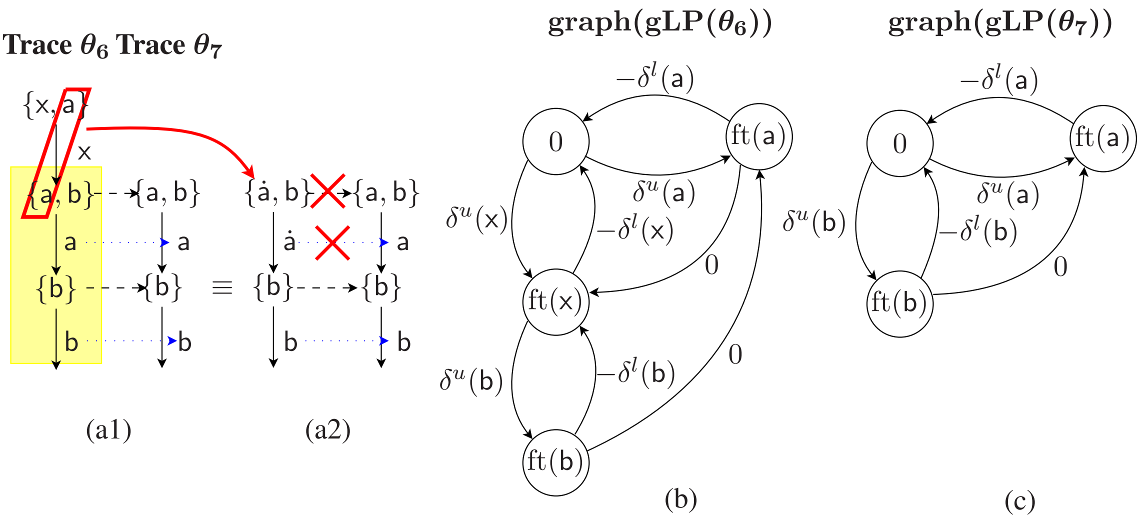


Figure A.4. Example showing two traces that should not be enabling compatible in order to guarantee structural timing inclusion.

delays (as in a real trace) it is no longer true that timing consistency in the mapped trace implies the timing consistency of the original trace, as is shown in Figure A.5, but the implication is still true in the other sense, which is enough for our purposes.

To complete the proof using this approach we should add a proposition in which we state that the timing consistency of a trace implies the timing consistency of the same trace in which some of its events are assigned unbounded delays. Then we have the full chain as timing consistency of the trace in which some of its events are assigned unbounded delays implies, by Theorem A.1, the timing consistency of the abstracted trace. By reversing implications we can then safely prune all enabling compatible traces with a timing inconsistent one as we are sure that they will be also timing inconsistent.

Alternatively we could have used directly this other version of the theorem.

THEOREM A.2 (ENABLING-COMPATIBILITY AND TIMING CONSISTENCY (ALT.)) *Let θ, θ' and θ_t be traces with the same conditions as in Definition 5.16. Let mp be an enabling-compatible mapping from θ_t onto θ', and δ^l and δ^u be two functions that assign arbitrary min/max delays to the events in Σ. Then,*

$$\theta \text{ is timing consistent} \implies \theta' \text{ is timing consistent.}$$

■ A.2

Proof:

Let $\tau_1 \leq \cdots \leq \tau_n$ be the time stamps assigned to $E_1, \ldots, E_{n+1}$ that make θ timing consistent. The same time stamps can be assigned to θ' as follows. Let j be the smallest index such that $\mathsf{mp}(E_j) = E'_i$. Then we assign the time stamp t_j to E_i. Under this assignment we have that for any $\mathsf{e}_k \in \Sigma'$, the time stamp assigned to $\mathsf{FirstEnabled}(\theta, E_j, \mathsf{e}_k)$ is the same as the one assigned to $\mathsf{FirstEnabled}(\theta', \mathsf{mp}(E_j), \mathsf{e}_k)$. This is ensured by Definition A.1, that enforces the set of enabled events in Σ' to be the same in E_j and $\mathsf{mp}(E_j)$. Then, since the disabling of an event $\mathsf{e} \in \Sigma \cap \Sigma'$ must be due to the firing of another event $\mathsf{e}_l \in \Sigma \cap \Sigma'$ and such events are enabled and fire at the same time in both sides, the disabling of e must also occur at the same time stamp in θ and θ'. Now, by Definition 2.6 of timing consistency, it immediately follows that the assignment of time stamps also makes θ' timing consistent.

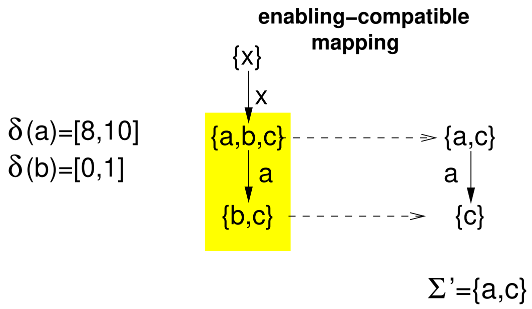


Figure A.5. Example of trace showing that timing consistency in the mapped trace does not imply the timing consistency of the original trace. If $\delta(\mathsf{a}) = [8, 10]$, $\delta(\mathsf{b}) = [0, 1]$ and the rest of events have the interval $[0, \infty]$ as delay, the trace in the left is timing inconsistent as b should fire before a. However the trace on the right is perfectly time feasible as a can fire before c.

We could also have proven this theorem using Lemma 5.2, by showing that the system of equations induced by θ gives a solution subspace of the system induced by θ'. This is easily shown because all the inequations of θ' are also present in the system of θ as $\Sigma' \subseteq \Sigma$ and the mapping enforces that both traces have the same triggers and disabler for all the event in Σ'. Then by Lemma 5.2, the inequation system for θ will contain all the inequation for θ' plus all those afecting events not in Σ'. Thus the firing times that are solution to this last system are also solutions to the system of θ'.

A.4 Event structures

The CESA formalism we have presented in our work (Definition 7.7) corresponds to the CES formalism of [Peñ03]. However a number of changes have been introduced in the definition for several reasons. We give the original definition of [Peñ03] and then we will justify the changes introduced.

Definition A.2 (Causal event structure)

A causal event structure (CES)*,* $CS = \langle \Sigma, \prec, \rhd \rangle$*, is a finite set* Σ *of* events*, a* precedence relation $\prec \subset \Sigma \times \Sigma$ *(irreflexive, antisymmetric and transitive) called the* causality relation*, and a* conflict relation $\rhd \subset \Sigma \times \Sigma$ *(irreflexive and antisymmetric).* $\rhd$ *is inherited via* $\prec$ *in the sense of:* $\forall\, \mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3 \in \Sigma \mid \mathsf{e}_1 \rhd \mathsf{e}_2 \;\wedge\; \mathsf{e}_1 \prec \mathsf{e}_3 \;\Rightarrow\; \mathsf{e}_3 \rhd \mathsf{e}_2$*. Moreover* $\prec$ *and* $\rhd$ *satisfy the following two properties:*

- $\prec \cap \rhd = \emptyset$ *and*
- $\mathsf{e}_1 \rhd \mathsf{e}_2 \;\Rightarrow\; \nexists \mathsf{e}_3 \in \Sigma \;:\; \mathsf{e}_2 \prec \mathsf{e}_3$.

That is, the causality and the disabling relations are disjoint, and disabled events cannot be causal predecessors of other events in the CES .

■ A.2

First of all in CESA definition we require $\prec^{-1} \cap \rhd = \emptyset$. This is not guaranteed by Definition A.2, hence we could build a CES in which an event disables one of its own predecessors. Secondly, CESAs also enforce that $\mathsf{e}_1 \rhd \mathsf{e}_2 \;\Rightarrow\; \nexists \mathsf{e}_3 \in \Sigma \;:\; \mathsf{e}_2 \rhd \mathsf{e}_3$. Again this does not appear in the previous definition, thus a disabled event is allowed to disable events! Finally, also the checking that a disabled event cannot be disabled by two concurrent events is missing, which causes the timing analysis to fail in such situations. However this last omission had no further impact on [Peñ03] because timing analysis on event structures is already faulty (see §A.4.1).

Definition A.3 (Causal event structure generated by a trace)

Let $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots \xrightarrow{\mathsf{e}_{n-1}} E_n \xrightarrow{\mathsf{e}_n} E_{n+1}$ *be a finite trace with* $D(\theta)$ *as the set of*

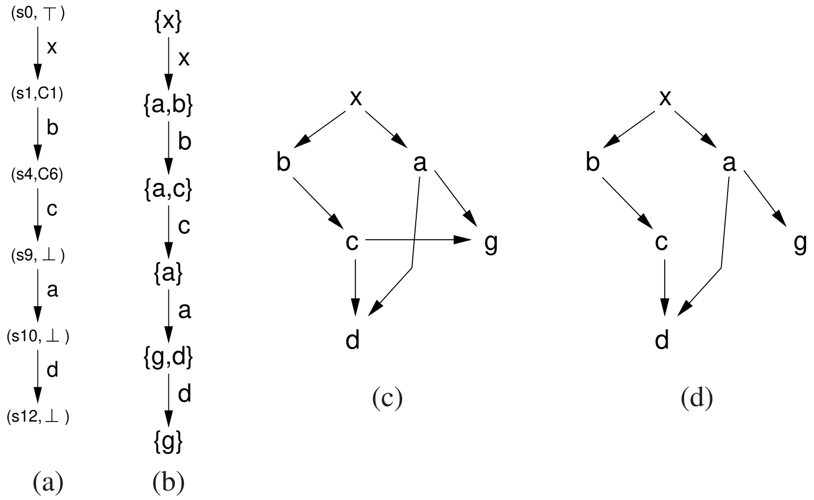


Figure A.6. Building a CES from a trace: (a) A run leading to a failure state extracted from the LzTS of Figure 5.3 (e). (b) Trace extracted from previous run. (c) CES obtained from the trace without considering structural independence. (d) CES when structural independence is considered.

disabled events along it. The causal event structure $\mathrm{CS}_\theta = \langle \Sigma, \prec \rangle$ *generated from* θ *is defined as follows:*

- $\Sigma = \bigcup_{i=1}^{n} E_i$. i.e. *Not only the firing events are included, but all those enabled along the trace, including the disabled ones.*
- *We define the relation* $\prec$ *as follows:*

$$\mathsf{e}_i \prec \mathsf{e}_j \quad \Leftrightarrow \quad i < j \ \wedge \ \{\nexists E_k \in \theta \ : \ \{\mathsf{e}_i, \mathsf{e}_j\} \subseteq E_k\} \ \wedge \ \mathsf{e}_i \notin D(\theta) \ \wedge$$
$$\wedge \ \mathsf{e}_i \textit{ structurally triggers } \mathsf{e}_j$$

The condition $\mathsf{e}_i \notin D(\theta)$ *emphasizes the fact that effectively disabled events can not be causal predecessors of other event since they do not fire along the trace. While last condition enforces that only events that can have a causal relation are taken into account.*

■ A.3

In [PCKP00] the condition “e_i structurally triggers e_j” did not appear in the definition, but in [Peñ03] the structural triggering was used in the implementation. Although it never appeared in the theory, previous definition has been updated to reflect actual practice.

Requiring that e_i structurally triggers e_j we only alleviate this problem and we do not solve it completely. To get an exact result we should know if one event *locally* triggers the other (in the scope of the trace analyzed) and we know that structural triggering does not imply local triggering (see Chapter 3).

However introducing such a false contraint does not invalidate the results obtained by this method (even if removing the *structural triggering* requirement from the definition), it will only take longer to prune the inexistant states in the timed domain by using this overly contrained CES.

In Figure A.6 (a) we show the same run presented in Figure 5.4 (a) (only up to the point where the failure is produced). In (b) we depict the trace derived from this run. If we build a CES from this trace using Definition A.3 without considering the restriction that an event must *structurally trigger* another in order to consider them related by the $\prec$ relation, then we end up with a CES like the one shown in (c). The reason is that c appears before g and they are never enabled at the same time. In this case the equations of the events needed to check structural triggering are

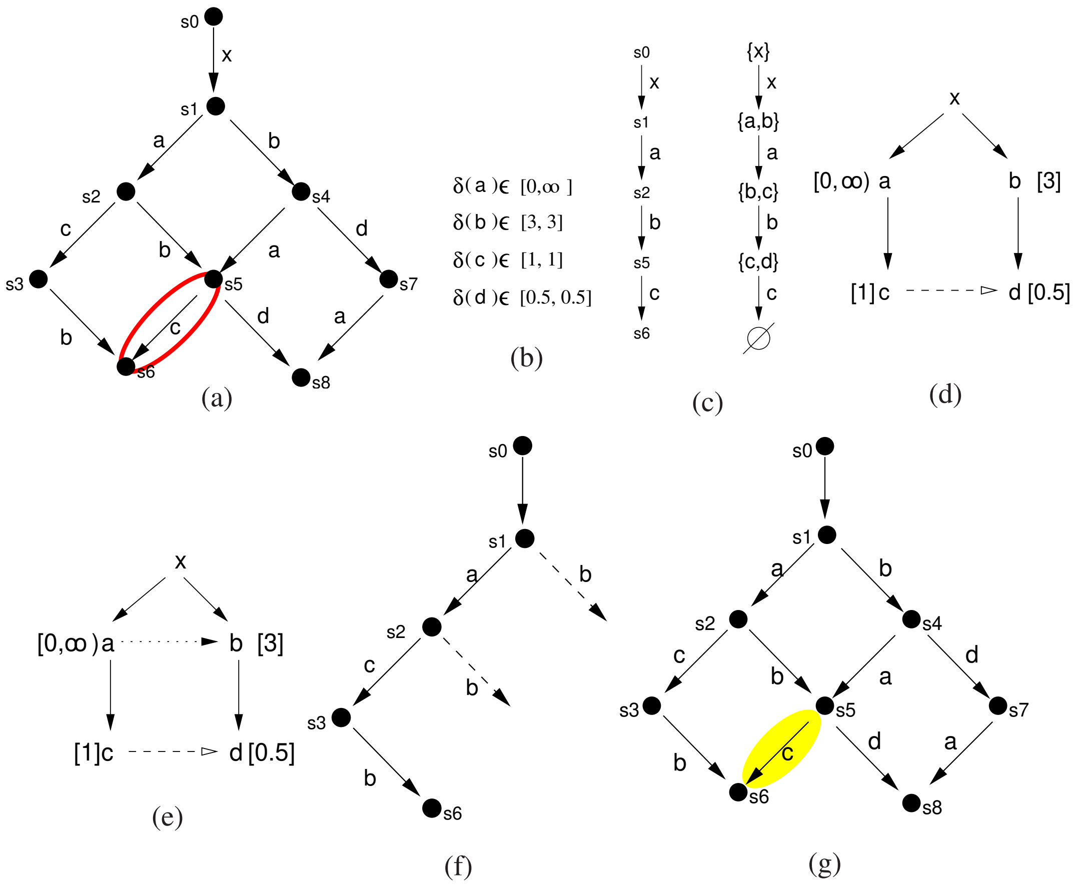


Figure A.7. Example of timing analysis problem: (a,b) TTS and delay intervals. Circled arcs $s_5 \xrightarrow{c} s_6$ is designed to be a failure arc. (c) A possible run to this failure state and its corresponding trace. They are timing inconsistent. (d) Event structure derived from the previous trace in which disabling of d by c is present. (e) Event structure annotated with timing arcs. (f) Classical LzTS obtained after composition. Many timing consistent traces are removed accidentally from the system!! (g) Real timed state space of the system (shaded arcs are unfireable). A correct refinement should never produce a smaller state space.

not provided, but assuming they describe exactly the reachable state space, then by inspection of Figure 5.2 (a) we can see that never that c is fired, g *becomes* enabled. If such considerations are taken into account, then we finish with the CES of subfigure (d).

The following sequences of events are words of CES in subfigure (d): xagcbd, xbcadg, x, xab,...

Consider ω = xbcad, then ω_1 = x is the first prefix of ω, ω_2 = xb is the second prefix of ω, etc. The events enabled by ω_2 are $\{a, c\}$. Finally, according to Definition 7.5, ω generates the trace of Figure A.6 (b).

A.4.1 Differences on the timing analysis of event structures

In [PCKP00] the algorithm of [McM92] is used to obtain the maximum separation times of events, which allow to determine the relative orderings of the events in the timed domain. Due to the simplified model of event structure used in [PCKP00] their approach is correct. However in [Peñ03] disablings (hence restrictions) were introduced into the CES model and the only change in the theory was to use the algorithm of [McM92] for max constraints and linear constraints. This might lead to wrong results as shown in Figure A.7.

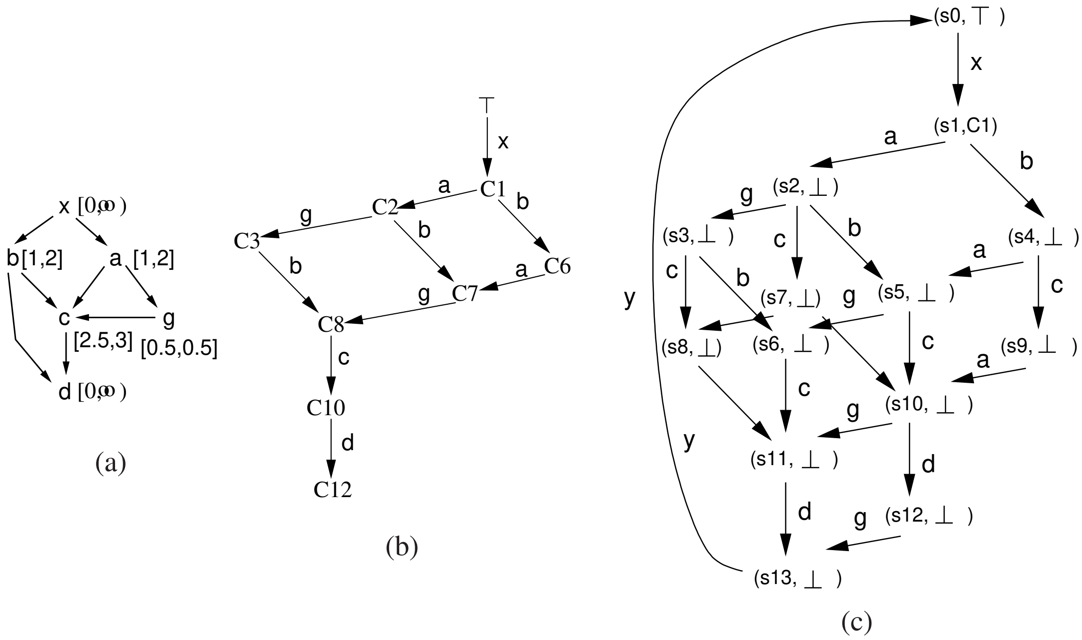


Figure A.8. Example of failed verification because of a faulty LzCES definition: (a) The event structure annotated with timing arcs. (b) State space of the event structure. (c) Classical LzTS obtained after composition. The refinement is completely useless.

Timing-consistency of traces does not depend on a particular ordering of future events, thus we cannot apply to all regions of the state space the precedences found by a timing analysis on a CES that considers some kind of restriction.

Moreover in [PCKP00, Peñ03] these constraints were added directly to the CES as extra causal relations as next definition shows:

DEFINITION A.4 (LAZY **CES** GENENERATED BY A TRACE [PCKP00])
Let $\theta = E_1 \xrightarrow{\mathsf{e}_1} \cdots E_n \xrightarrow{\mathsf{e}_n} E_{n+1}$ *be a trace of a* TTS $A = \langle A^-, \delta^l, \delta^u \rangle$ *, and let* $\mathrm{CS}_\theta = \langle \Sigma, \prec \rangle$*. The tuple* $\mathrm{LCS} = \langle \Sigma, \prec' \rangle$ *is called a* lazy causal event structure (LzCES)*, where* $\prec' = \prec \cup T$ *, and* $T \subseteq \Sigma \times \Sigma$ *is a set of* lazy relations *such that* $T = \{(\mathsf{e}_i, \mathsf{e}_j) \in \Sigma \times \Sigma \mid \mathsf{e}_i \not\prec \mathsf{e}_j \;\wedge\; \mathrm{mst}_{\mathrm{twd(CS)}}(\mathsf{e}_j, \mathsf{e}_i) < 0\}$.
■ A.4

However, this approach is wrong because then the enabling compatibility between traces does not work any more. We show this with an example: in Figure 5.3 (c) we can see a CES annotated with lazy arcs. Using Definition A.4 one must consider lazy arcs as causal arcs (dotted arcs convert to solid arcs), as shown in Figure A.8 (a). As a result, the state space of the event structure is not the one depicted in Figure 5.3 (d), instead it is equal to Figure A.8 (b) (just the unshadowed states from previous figure).

Is there any difference between using one of the two GRCs? Yes, with all the information of subfigure (d) we know that in $C2$, events b, c and g are enabled but event c cannot be fired due to the delays of the events. On the other hand, if we remove from the picture all the shadowed configurations (and their incoming arcs), as in Figure A.8 (b), in state $C2$ *only* events b and g are enabled. This detail makes the refinement to fail, as now state s_2 (in which events b, c and g are enabled) is no longer enabling compatible with state $C2$, thus instead of composed state $(\mathsf{s}_2, C2)$ we have $(\mathsf{s}_2, \perp)$ and refinement does not remove any timing inconsistent trace as shown in Figure A.8 (c).

Moreover in Figure 8.2 (a), on page 124, we can see an example of lazy relation that affects already fired events (*i.e.* $\mathsf{a} <_{\mathrm{lzy}} \mathsf{b}$ is only true when $\mathsf{c} < \mathsf{d}$), thus is useless. This kind of selection is not performed in Definition A.4 which results in an incorrect pruning of the state space. Finally, in [Peñ03], it was never considered the effect of the disablings in the lazy relations. Again, if the question is disregarded, there is the danger that the composition prunes timing consistent traces.

A.5 The enabling compatible product

In [Peñ03] the refinement of the model was not done through the composition with an ECTRA with the original system. Instead, a complex product operation between the original system and the LzCES obtained after timing analysis on a CESA was defined. The operation was called *enabling compatible product*. This strategy had the advantage that most of the theoretical work was developed using CESAs and GRCs, which are much more friendly than the ECTIAs and ECTRAs. However complexity is displaced to the composition operation, which has an equivalent complexity to the ECTIA construction presented in this work. The difference is that the ECTIA formalism enables to use standard automaton theory to solve some problems like indeterminacy that could not be solved with the enabling compatible product.

In [Peñ03] the enabling compatible product produced a classical LzTS instead of the LzTS model we have used, and was defined as follows:

Given the system $A = \langle S, \Sigma_A, T_A, \mathsf{s}_0, \mathsf{EnR}_A \rangle$ and the state space of the LzCES containing the relative timing constraints $G = \langle C \cup \bot, \Sigma_G, T_G, \top, \mathsf{EnR}_G \rangle$, with $\Sigma_G \subseteq \Sigma_A$, the enabling-compatible product of A and G is a new LzTS $\langle S', \Sigma_A, T', \mathsf{s}'_0, \mathsf{EnR}' \rangle$ where:

- $S' \subseteq S \times (C \cup \bot)$,
- $\mathsf{s}'_0 = (\mathsf{s}_0, \top)$ if $\mathcal{E}(\top) \subseteq \mathcal{E}(\mathsf{s}_0)$, and $\mathsf{s}'_0 = (\mathsf{s}_0, \bot)$ otherwise, and
- $\forall \mathsf{e} \in \Sigma_A,\ \mathsf{EnR}'(\mathsf{e}) = \{(\mathsf{s}, \mathcal{C}) \in S' \mid \mathsf{s} \in \mathsf{EnR}_A(\mathsf{e})\}$.

Remark: The alphabet Σ_G is not properly a subset of Σ_A. In fact Σ_G might contain several *instances* of any event in Σ_A. That is, provided an event $\mathsf{e} \in \Sigma_A$, a set $\{\mathsf{e}/1, \mathsf{e}/2, \ldots\}$ of instances of e might be present in Σ_G. This is the case for example when the corresponding LzCES is generated from a trace of A where several occurrences of event e appear along the trace. Thus we define $\Sigma_G \downarrow \Sigma_A$ as the *projection* of the event occurrences in Σ_G over the actual events in Σ_A. For example if $\Sigma_A = \{\mathsf{a},\ \mathsf{b},\ \mathsf{c}\}$ and $\Sigma_G = \{\mathsf{a}/1,\ \mathsf{a}/2,\ \mathsf{c}/1\}$, $\Sigma_G \downarrow \Sigma_A = \{\mathsf{a},\ \mathsf{c}\}$.

The transition relation T' is defined by the rules below. The rules are implied by the conditions on the enabling-compatibility of traces. The fact that $(\mathsf{s}, \mathcal{C}) \in S'$ denotes that s and $\mathcal{C}$ have been reached by prefixes that are enabling-compatible, and that $\mathsf{mp}(\mathcal{E}(\mathsf{s})) = \mathcal{E}(\mathcal{C})$. Given a state of the product $(\mathsf{s}, \mathcal{C})$ with $\mathcal{C} \neq \bot$, we will say that the state is in the timed domain, indicating that the timing analysis performed on CS_θ can be applied to s.

The rules that define the enabling-compatible product are as follows:

Transitions entering the timed domain

Transition	Conditions
$(\mathsf{s}_1, \bot) \xrightarrow{\mathsf{e}} (\mathsf{s}_2, \top)$	$\mathsf{enter} \equiv \mathsf{s}_1 \xrightarrow{\mathsf{e}} \mathsf{s}_2 \in T_A \wedge \mathcal{E}(\top) \downarrow \Sigma_A \subseteq \mathcal{E}(\mathsf{s}_2) \cap \Sigma_G \downarrow \Sigma_A$

These transitions are fired when the events enabled in $\top$ are also enabled in s_2. Thus, timing analysis can start being applied from $(\mathsf{s}_2, \top)$.

Staying inside the timed domain

Transition	Conditions
$(\mathsf{s}_1, \mathcal{C}_1) \xrightarrow{\mathsf{e}} (\mathsf{s}_2, \mathcal{C}_1)$	$\mathsf{inside1} \equiv \mathsf{s}_1 \xrightarrow{\mathsf{e}} \mathsf{s}_2 \in T_A \wedge \mathcal{E}(\mathsf{s}_1) \cap \Sigma_G \downarrow \Sigma_A = \mathcal{E}(\mathsf{s}_2) \cap \Sigma_G \downarrow \Sigma_A$
$(\mathsf{s}_1, \mathcal{C}_1) \xrightarrow{\mathsf{e}} (\mathsf{s}_2, \mathcal{C}_2)$	$\mathsf{inside2} \equiv \mathsf{s}_1 \xrightarrow{\mathsf{e}} \mathsf{s}_2 \in T_A \wedge \mathcal{C}_1 \xrightarrow{\mathsf{e}} \mathcal{C}_2 \in T_G \wedge$ $\mathcal{E}(\mathsf{s}_2) \cap \Sigma_G \downarrow \Sigma_A = \mathcal{E}(\mathcal{C}_2) \downarrow \Sigma_A$

Inside1 corresponds to the condition in which e does not synchronize with G. Here the enablings of configuration $\mathcal{C}_1$ must be preserved, *i.e.* the firing of e cannot disable or enable events in Σ_G.

For inside2, both A and G make a synchronized move which might affect the events from Σ_G in exactly the same way: if $\mathsf{a} \in \Sigma_G$ becomes enabled in A due to this move, it should also become enabled in G, and vice versa.

Exiting or staying outside the timed domain

Transition	Conditions
$(\mathsf{s}_1, \mathcal{C}_1) \xrightarrow{\mathsf{e}} (\mathsf{s}_2, \perp)$	$\mathsf{exit} \equiv \mathsf{s}_1 \xrightarrow{\mathsf{e}} \mathsf{s}_2 \in T_A \wedge \neg(\mathsf{enter} \vee \mathsf{inside1} \vee \mathsf{inside2})$

A.5.1 Comments on transitions entering the timed domain

Given definition does not deal with the conservativeness of the events. It simply assumes that all events enabled at the begining in the GRC are conservative, otherwise the product is wrong as it is not checked that events that are not conservative are not enabled in s_1. Conservativeness of events is critical because the timing analysis might give different results depending on whether some event is considered to be conservative or not.

Although theoretically it is possible to assume that all events are initially conservative in the selected trace suffix (*i.e.* it is always possible to take a longer suffix in which the starting events are conservative) it is a source of inefficiency and without further simplification techniques can make the verification approach not to converge. Imagine a trace in which the firing of each event enables another one (all events with lower delay greater than 0). The only suffix of this trace that has all initial events as conservative is the full trace (initially enabled events of the system are always considered as conservative). If the trace was obtained by unfolding some cycle in the system, then composition will only affect the unfolding at present level, but another trace with similar characteristics but with one more unfolding will be found in next iteration. Hence verification procedure will never terminate.

A.5.2 Comments on transitions staying inside the timed domain

Definition of transitions not synchronizing (*i.e.* equivalent to the *keeping the mapping* transitions in the ECTIA) is overly restrictive as not a single event appearing in Σ_G can become enabled. This is forced in the definition because it is not checked when advancing the mapping (*i.e.* inside2 condition) that newly enabled events are previously disabled. If G contains many events this can cause to lose most of the enabling comptatible benefits as most of the concurrent events will exit the mapping.

There is also a problem with self-enabled events as they can satisfy both conditions inside1 and inside2 so product is indeterministic in this case.

Finally reenter conditions are not specified, so product fails to prune all the traces it should. Actually reenter arcs are forced to exit the timed domain in the definition.

A.5.3 Comments on transitions exiting or staying outside the timed domain

Here there is a formalisation problem. The rule as it is says that everything else that is not considered by previous rules must exit (or remain outside). However this causes that all the arcs that should be pruned because there is not the corresponding arc in G but the event is enabled (*i.e.* a lazy arc due to timing analysis) instead of becoming lazy they exit the timed domain! This renders the composition completely useless.

A.6 The enabling compatible product implementation

Surprisingly the implementation of the ideas of §A.5 solves some of the abovementioned problems, but also introduces new flaws. For instance it considers the reenter and lazy arcs, but for this latter type of arcs the description of the implementation is not correct because of a bad LzCES definition (see §A.4).

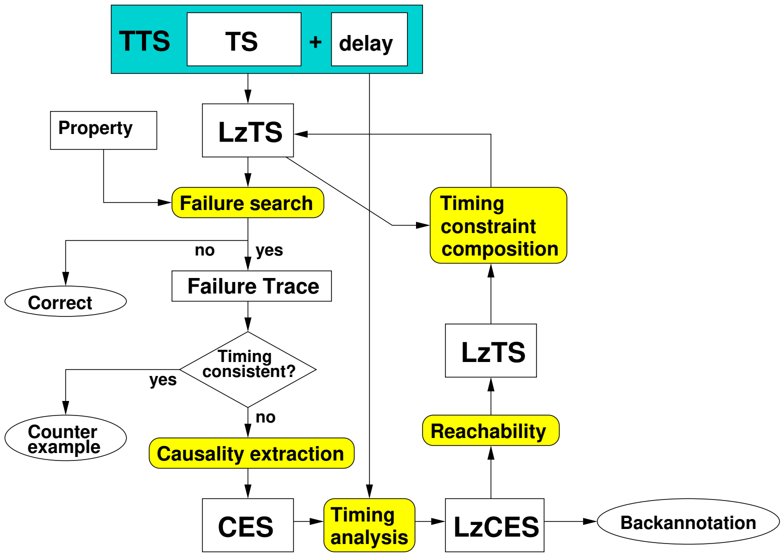


Figure A.9. Flow of the verification methodology of [Peñ03]

One of the key elements in their implementation is the use of an encoding function for each configuration. Namely,

$$\gamma(\mathcal{C}) = \prod_{\mathsf{e}/i \,\in\, \Sigma_G} \begin{cases} \mathrm{EF}(\mathsf{e}) & \text{if } \mathsf{e} \in \Sigma_A \;\; \wedge \;\; \mathsf{e}/i \in \mathcal{E}(\mathcal{C}) \\ \overline{\mathrm{EF}(\mathsf{e})} & \text{if } \mathsf{e} \in \Sigma_A \;\; \wedge \;\; \mathsf{e}/i \notin \mathcal{E}(\mathcal{C}) \end{cases}$$

This definition contains a flaw since Σ_G is the alphabet of distinct events. Therefore if two different instances of the same event are present, they will never be enabled at the same time in a configuration. This causes that the enabling function of the event and its negation appear in the product, which becomes 0. Besides this small formalization flaw, the effectiveness of the implementation is limited by the enabling compatibility definition (see A.3). We believe that many of these problems are related to the fact that distinct events simplify enabling compatibility definition but the implementation has to be done considering repetitions of the events.

On the other hand, from our experimental results we have seen that, in some cases, this type of enconding works better than the approach of §14.1. We have implemented a similar approach, but our enabling compatibility definition is more general and many disambiguation problems arise, which are sometimes difficult to circumvent without losing pruning capabilities. A more restrictive encoding diminishes the probability of disambiguation between configurations, at the price of less pruning capabilities. For many simple systems, however, the extra capabilities of our enabling compatibility definition are not required or have little effect, thus this encoding works well in these cases, specially if the number of system variables is big. Anyway, it is also true that in some other cases our composition method works better, so this topic deserves further study.

A.7 Verification methodology

The verification flow of [Peñ03] is depicted in Figure A.9. In this section we briefly describe it and compare it with the methodology exposed in Chapter 13.

A.7.1 Iterative refinement

Figure A.10 shows the verification algorithm, where A is the TTS that models the system under verification, and P is a safety property.

function *timed_verification* ($A = \langle S_A, \Sigma_A, T_A, \mathsf{s}_{0A}, \delta^l, \delta^u \rangle$, P)
    $A' = \langle S_A, \Sigma_A, T_A, \mathsf{s}_{0A}, \mathrm{FR} \rangle$;
    repeat
        θ := *untimed_verification*(A', P);
        if (empty θ) **return**(SUCCESS);
        LCS := *build_event_structure*(A', θ, δ^l, δ^u);
        if (empty LCS) **return**(FAIL, θ);
        A'' := *compose*(A', LCS);
        $A' := A''$;
    end repeat
end function

Figure A.10. Main algorithm of the relative timing-based verification approach.

function *build_event_structure* ($A' = \langle S, \Sigma, T, \mathsf{s}_0, \mathrm{FR} \rangle$, θ, δ^l, δ^u)
    θ'' := *shortest_suffix*(θ);
    repeat
        θ'' := *add_predecessor*(θ'', θ);
        CS := *build_event_structure*(A', θ'');
        if (*timing_consistent*(CS, δ^l, δ^u))
            L := *compute_lazy_arcs*(CS, δ^l, δ^u);
            LCS := *add_lazy_arcs*(CS, L);
            return (LCS);
        end if
    while ($\theta'' \neq \theta$);
    return (empty CS);
end function

Figure A.11. Algorithm for the derivation of a LzCES from a trace.

First, a LzTS A' is obtained corresponding to the underlying TS of A. The enabling information in A' coincides with that of firing, since no refinement with the timing information has been carried out yet.

The function *untimed_verification* checks whether a trace violating the property P is present in A'. If such a trace exists, a finite prefix, θ, demonstrating the wrong behavior is returned. This prefix is checked for timing-inconsistency by building and analyzing the corresponding causal event structure (see function *build_event_structure* in Figure A.11). If no CES can disprove the feasibility of the trace θ the verification returns θ as an example of violation of P. Otherwise the system is refined through the composition with the LzCES LCS. LCS contains a set of relative timing constraints that apply over a set of enabling-compatible traces, including θ.

The *timed_verification* algorithm does not depend on any particular implementation of the *untimed_verification* function. However the implementation of [Peñ03] uses symbolic model checking techniques [BCM$^+$92]. Basically, theye explore A' looking for failure states where P is violated. Then, a backward traversal is performed to generate a trace, leading from the initial state to the failure one, reproducing the discrepancy with P.

In this approach trace fragments are not used as a way to simplify the analysis. However the main differences with our technique are in the construction of the event structure.

A.7.2 Off-line timing analysis of failures

The function *build_event_structure* (see Figure A.11) builds the shortest suffix θ'' of the trace θ generated by the function *untimed_verification*. θ'' is built such that the timing analysis shows a

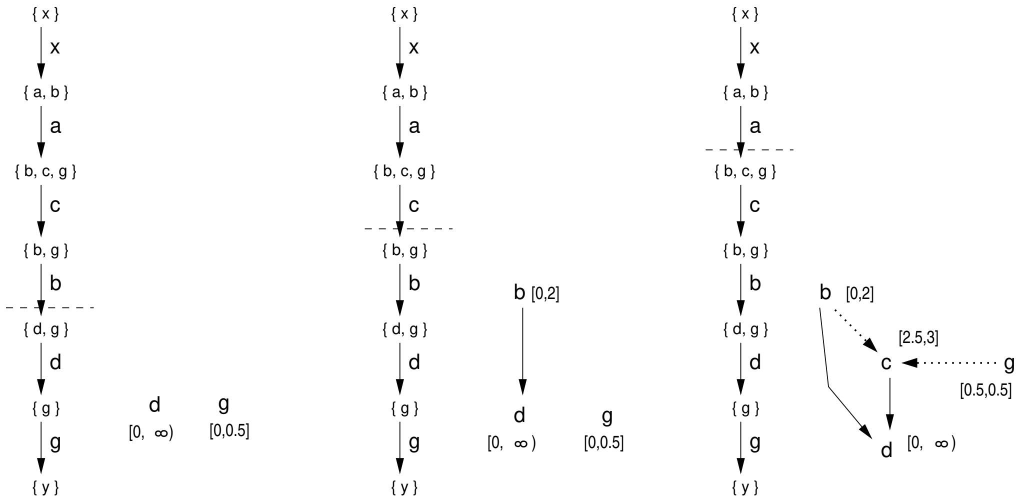


Figure A.12. Generation of the sufficient shortest suffix of a trace. Three steps are needed to obtain a LzCES that proves the timing-inconsistency of the trace.

timing-inconsistency with the delays δ^l and δ^u imposed by A. A causal event structure CS is constructed by using the causal relations of the events in θ'' (see Chapter 7).

Function *timing_consistent* performs timing analysis over CS. It implements the algorithm described in [MD92] for timing analysis over an acyclic graph of events with min/max and linear constraints (see §8.3).

If the timing analysis shows that the trace is not timing consistent, function *compute_lazy_arcs* extracts a set of relative timing constraints from CS, *i.e.* a set of additional orderings between the events of θ'' imposed by the delay bounds. These new constraints are added to the initial CS in the form of lazy arcs by the function *add_lazy_arcs*. The resulting lazy event structure LCS models only those orderings of the events of θ'' which are timing consistent with the delays imposed by A.

EXAMPLE A.1 *Recall the example developed in §5.2. Consider the* TS *of Figure 5.2 (a) and the delay bounds specified in Figure 5.2 (b). Recall also that, in this example, the property being verified states that event* g *must always fire before event* d*. Thus, the following trace* $\{x\}\xrightarrow{x}\{a,b\}\xrightarrow{a}\{b,c,g\}\xrightarrow{c}\{b,g\}\xrightarrow{b}\{d,g\}\xrightarrow{d}\{g\}\xrightarrow{g}\{y\}$ *illustrates a violation of the property.*

The shortest possible suffix is given by the trace $\{d,g\}\xrightarrow{d}\{g\}\xrightarrow{g}\{y\}$*, from which the simple event structure of Figure A.12 (a) is derived. Clearly, the timing analysis cannot be exact since* g *was already enabled in the pre-history of the trace. Thus, the lower delay bound of* g *is conservatively set to* 0*. The timing analysis can conclude nothing about the occurrence order of events* d *and* g*, since both can fire concurrently. The algorithm continues by moving one step backwards along the trace and repeats the same process again, building the corresponding* CES *incrementally. Figure A.12 depicts the three attempts needed to find the shortest sufficient suffix of the original failure trace.*

According to the causality relations extracted from the suffix of Figure A.12 (c), timing analysis concludes that events b *and* g *occur before event* c *(and consequently before* d*). This relative timing constraints are depicted by the dotted arcs in the corresponding lazy event structure. The derived timing relations demonstrate the infeasibility of the given failure trace in the timed domain.*

■ A.1

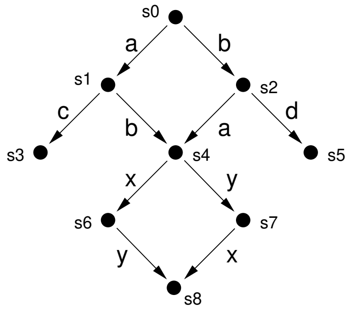


Figure A.13. TS with nodal states in s_0, s_3, s_5, s_8 and s_4. State s_4 does not satisfy Definition A.5.

As we have seen in Chapter 12, the use of the shorter suffix does not guarantee the maximum effectiveness of the later refinement of the state space of the system, with the timing constraints in the LzCES.

In [Peñ03] a number of optimizations, like removing from the LzCES timing-unrelated concurrent events, were implemented (although not documented). However this still does not guarantee maximum convergence. Furhtermore, this approach is not conscious of which precedences are really useful in prunning the state space, although it implements some heuristics to decide which precedences have to be applied first (also not documented).

A.8 Convergence

DEFINITION A.5 (NODAL STATE [PEÑ03])

Let $A = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$ *be a TS. A state* $\mathsf{s} \in S$ *is called* nodal *if* $\forall \mathsf{s}' \in S : \mathsf{s}' \xrightarrow{\mathsf{e}} \mathsf{s} \in T$, $|\mathcal{E}(\mathsf{s}')| = 1$. ■ A.5

Definition A.5 requires that all direct predecessors of a nodal state are synchronized in that state, *i.e.* at the moment when a system arrives to a nodal state all concurrent activities have been finished.

Nodal states are natural points from which the timing analysis is convenient to start. Any event enabled somewhere in a path to a nodal state must fire before reaching this state and, hence, timing analysis from a nodal state does not depend on the prehistory of the process behavior. However Definition A.5 only represents a subset of the states where this independency is present. For instance in Figure A.13 we can see a TS where state s_4 is clearly nodal: relative ordering of events x and y does not depend on anything previous to s_4 for any trace traversing that state. Definition A.5 will not consider s_4 as nodal because $\mathcal{E}(s_1) = \{b, c\}$.

The problem with Definition A.5 is that instead of defining the really important fact (*i.e.* the independency of timing analysis) it tries to give a topological description of when this happens.

In [Peñ03] the definition of strongly synchronized TS is given as:

a TS in which every trace goes through at least one nodal state is called *strongly synchronized*.

They claim that for strongly synchronized systems the method is guaranteed to converge. However it is not difficult to check that there is an error in the definition which makes this claim false: imagine a TS $A = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$ where there is a cyclic behaviour (thus creating an unfolding problem) for which the methodology is not guaranteed to converge, we can convert this system into a *strongly synchronized* by simply adding the prefix $\mathsf{s}_a \xrightarrow{\mathsf{dummy}_1} \mathsf{s}_b \xrightarrow{\mathsf{dummy}_2} \mathsf{s}_0$ to it, with $\mathsf{dummy}_1, \mathsf{dummy}_2 \notin \Sigma$ and $\mathsf{s}_a, \mathsf{s}_b \notin S$. That is, we create a new TS $A' = \langle S \cup \{\mathsf{s}_a, \mathsf{s}_b\}, \Sigma \cup \{\mathsf{dummy}_1, \mathsf{dummy}_2\}, T \cup \{\mathsf{s}_a \xrightarrow{\mathsf{dummy}_1} \mathsf{s}_b, \mathsf{s}_b \xrightarrow{\mathsf{dummy}_2} \mathsf{s}_0\}, \mathsf{s}_a \rangle$. Clearly s_b is a nodal point of A', and any trace in A' will go trough it, thus A' should be strongly synchronized, guaranteing the convergence. However it is also clear that the complexity of the timing analysis should be the same as in A (that is not guaranteed to converge)!

The problem of the former definition is that, in order to satisfy the idea that any cyclic behaviour can be reduced to a single analysis from a nodal point, we must guarantee that there is such nodal point in each possible cycle as it appears in Definition 12.2.

A.9 Static analysis

In [Peñ03] several static analysis options are available.

zero delay pruning When an event has delay bound $[0, 0]$ they consider it has precedence over any other event with positive lower bound. However, this can be a wrong assumption as Figure A.14 shows. Possibly the problem is that they used the delay bound $[0, 0]$ as a mechanism to express *urgent* actions *i.e.* actions that have precedence over the rest, except other urgent actions. Nevertheless urgent actions should be specified by other means, precisely to avoid this kind of problems.

nodal point pruning Nodal states of the system are computed and, for each nodal state, each pair of events a and b fireable in it are tested to see if $\delta^u(\mathsf{a}) < \delta^l(\mathsf{b})$. This approach is less efficient than ours because:

- Requires to compute (global) nodal states, while only "local" nodal states are necessary (*i.e.* states in which both events become enabled simultaneously while some other event can remain enabled from a previous state).
- It processes nodal states one by one. This is feasible since global nodal states are usually few, but is clearly inappropriate when using local nodal states which are much more common.
- Only the case when both event become simultaneously enabled is considered, although the case where event a is already enabled could be also incorporated.

pairwise pruning In this case they compute pairs of events that can become simultaneouly enabled and have appropriate delay bounds. In such case a LzCES is built and is composed. However this still could be improved by allowing to be already enabled the fast event, as in previous option. Moreover this requires the introduction of one new Boolean variable (two considering the corresponding next-state variable) because an actual composition is performed. Thus the pruning demands bigger transformation of the TRs than in our static analysis and can produce state splitting.

groupwise pruning Basically the same as the previous option, but to reduce the necessity of new variables, instead of pairs of events, groups of events are tested. This way, bigger LzCES are used, which can reduce the number of compositions. However, this approach is two-fold as in some cases it can produce more local refinements, which instead increases the number of compositions. Moreover, the computational cost is higher because more events are involved and the resulting LzCES is bigger.

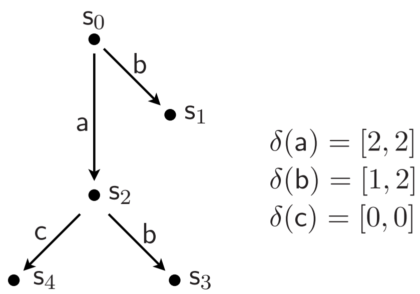


Figure A.14. TTS in which zero delay pruning fails. It will disable the firing of event b in s_2 when it should be allowed to fire.

Appendix B

Proofs

That which needs to be proved cannot be worth much.

—Friedrich Nietzsche

B.1 Proof of Lemma 5.6

We will proceed by proving each one of the properties. Because the enabling compatible mapping mp is defined over traces that allow repeated events, while mp_e is defined for distinct traces, it will be necessary to go back and forth between the two representations. We will use the following notation:

$$\theta_1 = E_1 \xrightarrow{\mathsf{e}_1} E_2 \cdots , \; \theta_2 = E'_1 \xrightarrow{\mathsf{e}'_1} E'_2 \cdots , \; \theta_1^d = E_1^d \xrightarrow{\mathsf{e}_1^d} E_2^d \cdots \text{ and } \theta_2^d = E_1^{d'} \xrightarrow{\mathsf{e}_1^{d'}} E_2^{d'} \cdots .$$

For any event e (or e') of trace θ_1 (or θ_2), we write as e^d (or $\mathsf{e}^{d'}$) the corresponding distinct event of θ_1^d (or θ_2^d). Although an event of a trace with repeated events might have several occurrences in a distinct trace, distinct event e^d is unique if some enabling E_i in θ_1 in which e appears is provided as then $\exists \mathsf{e}^d \in E_i^d : \mathsf{e}^d = \langle \mathsf{e}, \alpha \rangle$.

Property (5.4) is that for all events $\mathsf{e}^{d'}$ of θ_2^d that have a trigger in the trace, if $\delta^l(\mathsf{e}) > 0$ then $\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}^{d'}, \theta_2^d)) = \mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}^{d'}), \theta_1^d)$. Assume $\mathsf{e}^{d'}$ has been triggered by $\mathsf{e}_i^{d'}$ *i.e.* $\mathsf{e}_i^{d'} = \mathrm{trigger}(\mathsf{e}^{d'}, \theta_2^d)$. This also implies that e'_i triggers e in θ_2. Let $\mathsf{e}_j^d = \mathsf{mp}_e^{-1}(\mathsf{e}_i^{d'})$, we know from the enabling compatible mapping definition that $\mathsf{mp}(j) = i$ and that $\mathsf{mp}(j+1) = i+1$ by the firing rule and the fact that $\mathsf{e}_i = \mathsf{e}'_j$. By the projection rule we know that $E_j \supseteq E_i$ and $E_{j+1} \supseteq E_{i+1}$. As $\mathsf{e} \in E_{i+1}$ because e'_i triggered e, then $\mathsf{e} \in E_{j+1}$. As $\mathsf{mp}(j+1) = \mathsf{mp}(j) + 1$ by the enabling preservation rule we have that $E_{j+1} \setminus (E_j \setminus \{\mathsf{e}_j\}) \supseteq \mathrm{LB}(E'_{i+1} \setminus (E'_i \setminus \{\mathsf{e}'_i\}))$. As $\delta^l(\mathsf{e}) > 0$ and by definition of trigger $\mathsf{e} \notin E'_i$ or $\mathsf{e}'_i = \mathsf{e}$ in any of both cases $\mathsf{e} \notin E'_i \setminus \{\mathsf{e}'_i\}$ thus $\mathsf{e} \in \mathrm{LB}(E'_{i+1} \setminus (E'_i \setminus \{\mathsf{e}'_i\}))$. Consequently $\mathsf{e} \in E_{j+1}$ and either $\mathsf{e} \notin E_j$ or $\mathsf{e}_j = \mathsf{e}$. In both cases e_j will trigger e in θ_1 which implies that e_j^d triggers e^d in θ_1^d, hence $\mathsf{e}_j^d = \mathsf{mp}_e^{-1}(\mathsf{e}_i^{d'})$ which gives $\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_1^d) = \mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}, \theta_2^d))$. Q.E.D.

Property (5.5) is that for all events $\mathsf{e}^{d'}$ of θ_2^d that have a trigger in the trace, if $\delta^l(\mathsf{e}) = 0$ then $\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}^{d'}), \theta_1^{s,d}) \leq_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}^{d'}, \theta_2^d))$. Previous discussion applies as well with the only difference that as $\delta^l(\mathsf{e}) = 0$ now expression $\mathrm{LB}(\ldots)$ will not contain e for sure, hence we only have this result from the projection rule: $\mathsf{e} \in E_{j+1}$ (so $\mathsf{e}^d \in E_{j+1}^d$). Thus whatever event triggers $\mathsf{mp}_e^{-1}(\mathsf{e}^{d'})$ in θ_1^d, it must be at most e_j^d or any of the previously fired events, so $trigger(\mathsf{mp}_e^{-1}(\mathsf{e}^{d'}), \theta_1)$ must be some event e_l of θ_1^s (the standarized version of θ_1) with $l \leq j$, consequently $\mathsf{e}_l \leq_{\mathsf{fo}} \mathsf{e}_j$ (as we are using an standarized version e_l can be the special startup event ϵ) or, what is the same, $\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_1^{s,d}) \leq_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}, \theta_2^d))$. Q.E.D.

Property (5.6) is proved using mainly the initialization rule. If $\mathrm{trigger}(\mathsf{e}, \theta_2^{s,d}) = \epsilon$ then e has to be enabled in E'_1. By the initialization rule we know that $\mathsf{mp}(k+1) = 1$ thus $\mathsf{mp}_e^{-1}(\mathsf{e}_1^{d'}) = \mathsf{e}_{k+1}^d$. By the projection rule, $\mathsf{e}^{d'} \in E_1^{d'}$ which implies $\mathsf{e} \in E_{k+1}$. Initialization rule also obliges that $E_k \cap (\mathrm{LB}(E'_1) \setminus \{\mathsf{e}_k\}) = \emptyset$ or $\mathrm{LB}(E'_1) = \emptyset$ depending on whether $k \geq 1$ or $k = 0$. The latter case however is not possible as $\delta^l(\mathsf{e}) > 0$ and $\mathsf{e} in E'_1$ thus $\mathrm{LB}(E'_1) \neq \emptyset$. Consequently we do only have one scenario: $E_k \cap (\mathrm{LB}(E'_1) \setminus \{\mathsf{e}_k\}) = \emptyset$. As $\mathsf{e} \in \mathrm{LB}(E'_1)$ it can only be that either $\mathsf{e} \in E_k$ and $\mathsf{e}_k = \mathsf{e}$ or that $\mathsf{e} \notin E_k$. In any of the two cases e_k^d will trigger e^d in θ_1^d. Q.E.D.

Property (5.7) is proved in a similar style. By the initialization property $\mathsf{mp}_e^{-1}(\mathsf{e}_1^{d'}) = \mathsf{e}_{k+1}^d$. By the projection rule, $\mathsf{e}^{d'} \in E_1^{d'}$ which implies $\mathsf{e} \in E'_1$ and $\mathsf{e} \in E_{k+1}$. However this time with $\delta^l(\mathsf{e}) = 0$, it could be triggered by e_k or by some other previous event in θ_1, in any case $\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}^{d'}), \theta_1^{s,d}) <_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\mathsf{e}_1^{d'})$ (standarized trace is used again in order to have a defined trigger for all events in θ_1 because it can be the case that e is already enabled in E_1 hence its trigger would be considered the startup event ϵ). Q.E.D.

Property (5.8) is that for any two fired events $\mathsf{e}_i^{d'}, \mathsf{e}_j^{d'}$ of θ_2^d, we have that

$$\mathsf{e}_i^{d'} <_{\mathsf{fo}} \mathsf{e}_j^{d'} \Rightarrow \mathsf{mp}_e^{-1}(\mathsf{e}_i^{d'}) <_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\mathsf{e}_j^{d'}).$$

If $\mathsf{e}_i^{d'}, \mathsf{e}_j^{d'}$ then $\mathsf{e}'_i <_{\mathsf{fo}} \mathsf{e}'_j$ and $i < j$ by the definition of the $<_{\mathsf{fo}}$ relation. Now let $k, r \in \mathbb{N}$ so that $\mathsf{mp}(k) = i$ and $\mathsf{mp}(r) = j$, then as mp is a monotonically increasing function we have that $k < r$ if $i < j$. Q.E.D.

Property (5.9) is that for all disabled events $\mathsf{e}^{d'}$ of θ_2^d, if $\mathsf{mp}_e^{-1}(\mathsf{e}^{d'})$ is disabled in θ_1^d then $\mathsf{mp}_e^{-1}(\mathrm{disabler}(\mathsf{e}^{d'}, \theta_2^d)) \leq_{\mathsf{fo}} \mathrm{disabler}(\mathsf{mp}_e^{-1}(\mathsf{e}^{d'}), \theta_1^d)$ Assume $\mathsf{e}^{d'}$ has been disabled by $\mathsf{e}_i^{d'}$ *i.e.* $\mathsf{e}_i^{d'} = \mathrm{disabler}(\mathsf{e}^{d'}, \theta_2^d)$. This also implies that e'_i disables e in θ_2. Let $\mathsf{e}_j^d = \mathsf{mp}_e^{-1}(\mathsf{e}_i^{d'})$, we know from the enabling compatible mapping definition that $\mathsf{mp}(j) = i$ and that $\mathsf{mp}(j+1) = i+1$ by the firing rule and the fact that $\mathsf{e}_i = \mathsf{e}'_j$. By the projection rule we know that $E_j \supseteq E_i$ and $E_{j+1} \supseteq E_{i+1}$. As $\mathsf{e} \notin E_{i+1}$ because e'_i disabled e, then e can either be present or not in E_{j+1}. Before this point by the projection rule it must be enabled, as we know that it is disabled in θ_1^d, then the actual disabler of $\mathsf{mp}_e^{-1}(\mathsf{e}^{d'})$ must be an event e_k^d with $k \geq j$, thus $\mathsf{mp}_e^{-1}(\mathrm{disabler}(\mathsf{e}^{d'}, \theta_2^d)) \leq_{\mathsf{fo}} \mathrm{disabler}(\mathsf{mp}_e^{-1}(\mathsf{e}^{d'}), \theta_1^d)$. Q.E.D.

Finally, Property (5.10) proof is similar to the previous one with the only difference that now there is a fired event $\mathsf{e}_k^d = \mathsf{mp}_e^{-1}(\mathsf{e}^{d'})$ with $k > j$. The value of k cannot be less than j because of the projection rule as $\mathsf{e}^{d'}$ appears in θ_2^d it cannot be fired in θ_1^d while it is enabled in θ_2^d as this will violate the enabling preservation rule. It cannot be equal because the disabler of an event, by definition, cannot be the event itself. Hence the unique possibility is that $k > j$, which implies that $\mathsf{mp}_e^{-1}(\mathrm{disabler}(\mathsf{e}^{d'}, \theta_2^d)) <_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\mathsf{e}^{d'})$. Q.E.D.

B.2 Proof of Theorem 5.7

To ease the reading we will use the following shorthands $G = \mathrm{graph}(\mathrm{g_0LP}(\theta))$ and $G' = \mathrm{graph}(\mathrm{g_0LP}(\theta'))$. The graphs are constructed using the distinct versions of the traces, named θ^d and $\theta^{d'}$. In order to reason more uniformly we will use the standarized versions of the distinct traces that we will name as θ_s^d and $\theta_s^{d'}$. There is no problem in doing so because the graphs associated to distinct traces and to standarized traces are completely isomorphic and the only difference is that node 0 is now named $\mathrm{ft}(\epsilon)$ in the graphs of standarized traces.

Our objective is to show that for all representative possible negative cycles in G' there is a representative possible negative cycle with lower weight in G. To achieve our objective we will show that for any weighted arc $\mathrm{ft}(\mathsf{e}'_i) \xrightarrow{w'} \mathrm{ft}(\mathsf{e}'_j)$ in G' there is a path from node $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e}'_i))$ to node $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e}'_j))$ with weight $w \leq w'$. The latter will be proved by all arcs in G' induced by an event of $\theta^{d'}$ according to Lemma 5.2 and Definition 5.12.

Using standarized traces, in which all events have a trigger in the trace except special event ϵ, Table 5.1 used Lemma 5.2 reduces to the second column which has only three cases. From these three cases, two of them can be merged seamlessly: the case in which an event is enabled until the end of a finite trace can be interpreted as a particular case of disabling, in which the event is disabled by the last fired event of the trace. It is straightforward to see that the resulting equations are identical in both situations. Hence from now on we will assume that in our traces all events enabled until the end are changed by the corresponding disablings. This allows to reduce the discussion to only two cases: either the event is fired or it is disabled.

In this context next table relates several situations arising in a trace θ_1 with the arcs they induce in $\mathrm{graph}(\mathrm{gLP}(\theta_1))$ (assuming the graph is built using the standarized version of the distinct trace of θ_1):

Situation	Arcs in $\mathrm{graph}(\mathrm{gLP}(\theta_1))$
$\mathsf{e} <_{\mathsf{fo}} \mathsf{e}'$ in θ_1^d	$\mathrm{ft}(\mathsf{e}') \xrightarrow{0} \mathrm{ft}(\mathsf{e})$
e fires in θ_1^d	$\mathrm{ft}(\mathrm{trigger}(\mathsf{e})) \xrightarrow{\delta^u(\mathsf{e})} \mathrm{ft}(\mathsf{e})$ $\mathrm{ft}(\mathsf{e}) \xrightarrow{-\delta^l(\mathsf{e})} \mathrm{ft}(\mathrm{trigger}(\mathsf{e}))$
e disabled in θ_1^d	$\mathrm{ft}(\mathrm{trigger}(\mathsf{e})) \xrightarrow{\delta^u(\mathsf{e})} \mathrm{ft}(\mathrm{disabler}(\mathsf{e}))$

Table B.1. Table of arcs induced in $\mathrm{graph}(\mathrm{gLP}(\theta_1))$ in several scenarios.

Applying Table B.1 to θ' we obtain all the arcs of G' which have to be derived[1].

For all arcs in G' $\mathrm{ft}(\mathsf{e}') \xrightarrow{0} \mathrm{ft}(\mathsf{e})$ coming from the first scenario we have to prove that

$$\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e}')) \xrightarrow{w} \mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e})) \text{ with } w \leq 0$$

exists in G. This is so because of Property (5.8) as $\mathsf{e} <_{\mathsf{fo}} \mathsf{e}'$ in $\theta_s^{d'}$ implies $\mathsf{mp}_e^{-1}(\mathsf{e}) <_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\mathsf{e}')$ in $\theta_s^{d'}$ which produces the expected arcs in G as shows Table B.1.

Next we will check for arcs $\mathrm{ft}(\mathrm{trigger}(\mathsf{e})) \xrightarrow{\delta^u(\mathsf{e})} \mathrm{ft}(\mathsf{e})$ in G'. This corresponds to the firing of e in $\theta^{d'}$. We have to prove that $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}, \theta_s^{d'}))) \xrightarrow{\delta^u(\mathsf{e})} \mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e}))$. By Properties (5.4), (5.5), (5.6) and (5.7) we know that always $\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_s^d) \leq_{\mathsf{fo}} \mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}, \theta_s^{d'}))$. This implies that there is a path (which can be empty if both nodes are the same node)

$$\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}, \theta_s^{d'}))) \xrightarrow{0} \mathrm{ft}(\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_s^d))$$

with value 0. Now in G there is also $\mathrm{ft}(\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_s^d)) \xrightarrow{\delta^u(\mathsf{e})} \mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e}))$, thus there is a path from $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}, \theta_s^{d'})))$ to $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e}))$ with cost $\delta^u(\mathsf{e})$ as required.

Now there is one type of arc remaining in the scenario where e is fired in $\theta^{d'}$ which corresponds to arcs $\mathrm{ft}(\mathsf{e}) \xrightarrow{-\delta^l(\mathsf{e})} \mathrm{ft}(\mathrm{trigger}(\mathsf{e}, \theta_s^{d'}))$ in G'. To prove this we will split the proof in two cases. First one is when $\delta^l(\mathsf{e}) > 0$ and the second one is when $\delta^l(\mathsf{e}) = 0$. For the first case by application of Table B.1 to the firing of $\mathsf{mp}_e^{-1}(\mathsf{e})$ in θ^d we have arc $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e})) \xrightarrow{-\delta^l(\mathsf{e})} \mathrm{ft}(\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_s^d))$ in G, then by Properties (5.4) and (5.6) we have $\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}), \theta_s^d) = \mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}, \theta_s^{d'}))$ this automatically implies that the arc $ft(\mathsf{mp}_e^{-1}(\mathsf{e})) \xrightarrow{-\delta^l(\mathsf{e})} \mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e}, \theta_s^{d'})))$ will be present

[1] Graph G' is derived from SPTE $\mathrm{g_0LP}(\theta')$ while the arcs in the table correspond to the arcs in the graph derived from SPTE $\mathrm{gLP}(\theta')$, however this is not a problem since the only difference is that all delay bounds which are 0 have been substituted by its actual value (*i.e.* 0). We will use the knowledge of delay values along the proof, so although we start from $\mathrm{graph}(\mathrm{gLP}(\theta'))$ we will use the fact that some delays are 0 to achieve our goal, that is precisely to use G'.

in G. In the second case ($\delta^l(\mathsf{e}) = 0$) the arc that we have to find in G is simply

$$\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e}))\xrightarrow{0}\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e},\theta_s^{d'})))$$

and this is trivially true because $\mathrm{trigger}(\mathsf{e},\theta_s^{d'})<_{\mathsf{fo}}\mathsf{e}$ implies $\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e},\theta_s^{d'}))<_{\mathsf{fo}}\mathsf{mp}_e^{-1}(\mathsf{e})$ by Property (5.8), which produces the expected arc in G.

Finally, we have to check all the cases in which e is disabled in $\theta^{d'}$. By Table B.1 we know there is an arc $\mathrm{ft}(\mathrm{trigger}(\mathsf{e},\theta^{d'}))\xrightarrow{\delta^u(\mathsf{e})}\mathrm{ft}(\mathrm{disabler}(\mathsf{e},\theta^{d'}))$ in G'. We have to prove that $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e},\theta^{d'})))\xrightarrow{\delta^u(\mathsf{e})}\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{disabler}(\mathsf{e},\theta^{d'})))$ exists in G. A disabled event in $\theta^{d'}$ can be either disabled or fired in θ^d. If disabled, in G we already have

$$\mathrm{ft}(\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}),\theta^d))\xrightarrow{\delta^u(\mathsf{e})}\mathrm{ft}(\mathrm{disabler}(\mathsf{mp}_e^{-1}(\mathsf{e}),\theta^d)).$$

As before by Properties (5.4), (5.5), (5.6) and (5.7) we know that, under any circumstance,

$$\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}),\theta_s^d)\leq_{\mathsf{fo}}\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e},\theta_s^{d'}))$$

so there is a path (can be empty if both nodes are the same node)

$$\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e},\theta_s^{d'})))\xrightarrow{0}\mathrm{ft}(\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}),\theta_s^d))$$

with value 0. At this point we already have a path

$$\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e},\theta_s^{d'})))\rightsquigarrow ft(disabler(\mathsf{mp}_e^{-1}(\mathsf{e}),\theta^d))$$

with cost $\delta^u(\mathsf{e})$.

It remains only to show that there is a path

$$\mathrm{ft}(\mathrm{disabler}(\mathsf{mp}_e^{-1}(\mathsf{e}),\theta^d))\rightsquigarrow\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{disabler}(\mathsf{e},\theta^{d'})))$$

with cost 0 and this is shown by Property (5.9). Finally in the case event $\mathsf{mp}_e^{-1}(\mathsf{e})$ is fired instead of disabled in θ^d we have arc $\mathrm{ft}(\mathrm{trigger}(\mathsf{mp}_e^{-1}(\mathsf{e}),\theta^d))\xrightarrow{\delta^u(\mathsf{e})}\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e}))$. Using the same reasoning as before we end up with a path from $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{trigger}(\mathsf{e},\theta_s^{d'})))$ to $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e}))$ with cost $\delta^u(\mathsf{e})$. It remains only to show that there is a path from $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathsf{e}))$ to $\mathrm{ft}(\mathsf{mp}_e^{-1}(\mathrm{disabler}(\mathsf{e},\theta^{d'})))$ with cost 0 and this is shown by Property (5.10).

B.3 Proof of Lemma 5.8

We start the prove by showing that, if θ' can be derived from θ by applying any number of the allowed transformations, then $\theta\geq_{\mathsf{ec}}\theta'$. This part of the proof is by induction on the number of transformations applied.

The base case is when no transformation has been yet applied, in which case we have that $\theta\geq_{\mathsf{ec}}\theta$, which is trivially true.

The hypothesis step is, given a trace θ'' such that $\theta\geq_{\mathsf{ec}}\theta''$, we will show that applying any of the transformations to θ'' we obtain a trace θ' such that $\theta''\geq_{\mathsf{ec}}\theta'$, hence by the transitivity of the enabling compatible relation we would have $\theta\geq_{\mathsf{ec}}\theta'$.

We proceed by cases. For prefix removal it is easy to show that $\theta''\geq_{\mathsf{ec}}\theta'$ verifies because it only requires minimal corrections in the identity enabling compatible mapping. The only thing one must be careful is to satisify the initialization rule which deals with lower bounded events, but this is solved by promoting to conservative events the distinct events that were already enabled in E_1 and are still enabled in E_2. Note that this transformation can only reduce a trace to a degenerated trace with a single enabling, but can never produce the empty trace.

The modification of last enabling transformation trivially satisfies the property because last enabling is not checked in the enabling compatible mapping hence it does not even require a

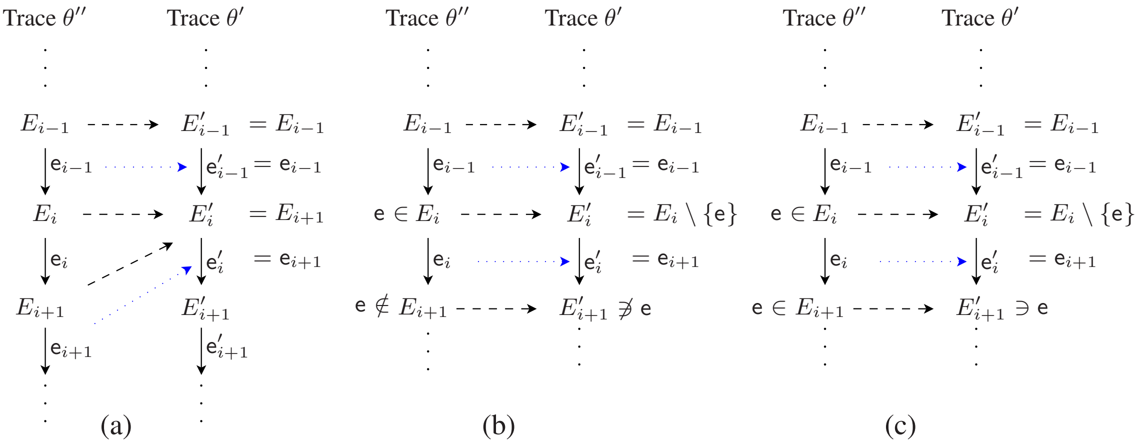


Figure B.1. Enabling compatible mappings between θ'' and θ' after different transformations: (a) θ' is the result of supressing the firing of a sink event in θ''. (b) θ' is the result of advancing the disabling of an event e in θ''. (c) θ' is the result of delaying the enabling of an event e with $\delta^l(\mathsf{e}) = 0$ in θ''.

change in the enabling compatible mapping, the same case as the promotion to conservative event transformation. The three remaining transformations are not so evident and we will illustrate them using Figure B.1.

In subfigure (b) we show the case when we advance the disabling of an event. The mapping will remain the same. As e_i disables e in θ'', then $\mathsf{e} \neq \mathsf{e}_i$, so it can be removed from E_i without creating an inconsistency. Also the projection rule is trivially satisfied because $E_i \supseteq E_i \setminus \{\mathsf{e}\}$.

In subfigure (c) we have the situation in which we delay the enabling of an event e with $\delta^l(\mathsf{e}) = 0$. The identity mapping works as before for similar reasons. However this time we must also guarantee that the enabling preservation rule will be satisfied for the event that now enables e, but this is sure because as $\delta^l(\mathsf{e}) = 0$ the event does not belong to $\mathrm{LB}(E_{i+1} \setminus (E_i \setminus \{\mathsf{e}, \mathsf{e}i\}))$.

Finally subfigure (a) shows the last transformation: the supression of the firing of a sink event. Here the enabling compatible mapping will be $\forall k : 1 \leq k \leq i, \mathsf{mp}(k) = k$ and $\forall k : i+1 \leq k \leq n, \mathsf{mp}(k) = k-1$. Because the sink event e_i does not trigger nor disable any event, we know that $E_{i+1} = E_i \setminus \{\mathsf{e}_i\}$. Hence projection rule is satisfied for E_i and E_{i+1} as well as the enabling preservation rule that states that if $\mathsf{mp}(i+1) = \mathsf{mp}(i)$ then $\mathsf{e}_i \notin E'_{\mathsf{mp}(i)}$ as $E'_{\mathsf{mp}(i)} = E'_i = E_{i+1} = E_i \setminus \{\mathsf{e}_i\}$.

The proof in the other sense, consists in proving that *only if* θ' can be derived from θ by applying any number of the allowed transformations, then $\theta \geq_{\mathsf{ec}} \theta'$. This is done by construction. Given two traces θ and θ' such that $\theta \geq_{\mathsf{ec}} \theta'$ we will show the steps needed to obtain θ' from θ by applying the transformations.

First step is prefix removal based on the value of k on the enabling compatible mapping (this value is related to where the fragment of θ that is used in the mapping starts). Second step is to remove events in the suffix until the last fired event of θ' is reached. To remove a suffix a little *trick* must be used: first the last enabling must be changed so that the current last fired event of θ becomes a sink event, and then it can be removed with the corresponding transformation.

After this, any conservative event of θ' that is not already conservative in θ is promoted to conservative. Then the enabling of all non lower bounded events such that become enabled later in θ' than in θ are delayed accordingly (they can only be delayed due to the properties of Lemma 5.6). After that, the disablings are also advanced if necessary (disablings can only be advanced also by the properties of Lemma 5.6). For events not appearing in θ' and not firing in θ this implies to use the advance enabling transformation up to the point where they disappear from θ.

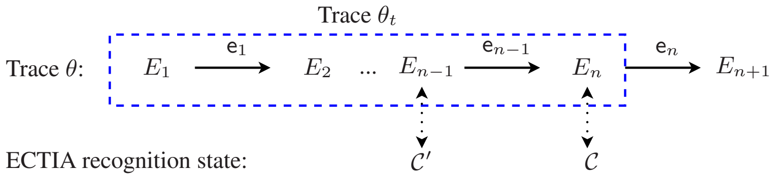


Figure B.2. Trace θ, its fragment θ_t, and their corresponding ECTIA recognition state.

Then all events that do not fire in θ', thus do not advance the mapping, but fire in θ are removed from θ starting by the end of the trace by using the remove sink event transformation. After this they either disappear from θ or become disabled. In the latter case another round of advancing delays might be required in order to mimick the behavior of θ'. Fially, the procedure ends with a modification of the last enabling of θ to match it with θ' the last enabling of θ' if it was different.

B.4 Proof of Theorem 6.1

We will prove this theorem by induction on the length of the checked trace θ.

Base case (length=1)

For trace length equal to one we have that $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2$. According to this, we have to check which is the recognition state $\mathcal{C}$ for the degenerated trace fragment E_1. There are only two possibilities, either it is $\perp$ or $\mathcal{C}_1$. If it is $\perp$, then this renders false the *if* condition of the theorem, hence it is trivially true. If $\mathcal{C} = \mathcal{C}_1$ it is required that all events in $\mathcal{E}(\mathcal{C}_1)$ have a lower delay bound equal to 0 and that $E_1 \supseteq \mathcal{E}(\mathcal{C}_1)$. Because θ is a trace generated by a run of A starting at some initial state, all events in E_1 are conservative according to what is exposed in §5.3.2.Hence both sets of events satisfy the projection rule of the enabling compatible mapping (Definition 5.16).

Now fired event e_1 can either belong to $\mathcal{E}(\mathcal{C}_1)$ or to $E_1 \setminus \mathcal{E}(\mathcal{C}_1)$. In the first case $\mathrm{traces}(\mathcal{C}_1, \mathsf{e}_1) = \{\mathcal{E}(\mathcal{C}_1) \xrightarrow{\mathsf{e}_1}\}$ which will allow the following enabling compatible mapping: $\mathtt{mp}(1) = 1, \mathtt{mp}(2) = 2$. In the second case, then $\mathrm{traces}(\mathcal{C}_1, \mathsf{e}_1) = \{\mathcal{E}(\mathcal{C}_1)\}$ and the mapping is $\mathtt{mp}(1) = 1, \mathtt{mp}(2) = 1$ which is a special case of a mapping against a degenerated trace fragment that consists simply in an enabling without event firing. In both cases a mapping is possible hence trace θ is enabling compatible with the unique trace present in set $\mathrm{traces}(\mathcal{C}_1, \mathsf{e}_1)$.

Induction step

Assume that theorem holds for $\theta_t = E_1 \xrightarrow{\mathsf{e}_1} \ldots \xrightarrow{\mathsf{e}_{n-1}} E_n$, we will show that it works for $\theta = \theta_t \xrightarrow{\mathsf{e}_n} E_{n+1}$. We know that there is a trace θ'_t in $traces(\mathcal{C}', \mathsf{e}_n)$ such that $\theta_t \geq_{\mathsf{ec}} \theta'_t$, where $\mathcal{C}'$ is the configuration after θ_t has been checked by the ECTIA up to enabling E_{n-1} (see Figure B.2). Let $\mathtt{mp}_t$ be the mapping that makes θ_t enabling compatible with θ'_t and let $\mathcal{C}$ be the configuration reached by an arc of the ECTIA $\mathcal{C}' \xrightarrow{\langle \mathsf{S}_s, \Sigma', \mathsf{S}_d \rangle} \mathcal{C}$ such that transition $E_{n-1} \xrightarrow{\mathsf{e}_{n-1}} E_n$ matches that arc (*i.e.* $\mathrm{SE}(E_{n-1}) \subseteq \mathsf{S}_s$, $\mathsf{e}_{n-1} \in \Sigma'$ and $\mathrm{SE}(E_n) \subseteq \mathsf{S}_d$). There are only four possibilities for $\mathcal{C}$, namely $\perp, \mathcal{C}_1, \mathcal{C}'$ and a configuration $\mathcal{C}_i$ for which $\mathcal{C}'$ is a predecessor configuration.

- If $\mathcal{C} = \perp$ the *if* condition of the theorem is false, hence the theorem itself is trivially true.
- If $\mathcal{C} = \mathcal{C}_1$ then the discussion is the same as in the base case of the proof where the initial recognition state of the ECTIA is $\mathcal{C}_1$. The only additional check is that if $\mathcal{C} = \mathcal{C}_1$ (and different than $\mathcal{C}'$) then an enter or a reenter arc of the ECTIA has been taken by $E_{n-1} \xrightarrow{\mathsf{e}_{n-1}} E_n$. In both cases the enter condition of ECTIA guarantees that the requirements of enabling compatibility about lower bounded events will be satisfied. Namely that a lower bounded event in θ' can only

be matched by a lower bounded event in θ. This is guaranteed because enter condition obliges all lower bounded events to be not enabled in $\mathcal{C}'$ or that only one of them is enabled but that this event is precisely the fired event (*i.e.* e_{n-1}) so that the one enabled in $\mathcal{C}_1$ must be considered also newly enabled.

- If $\mathcal{C} = \mathcal{C}'$, then the arc $\mathcal{C}' \overset{\langle \mathsf{S}_s, \Sigma', \mathsf{S}_d \rangle}{\longrightarrow} \mathcal{C}'$ is a *transition keeping the mapping* of the ECTIA. This enforces that $E_{n-1} \xrightarrow{\mathsf{e}_{n-1}} E_n$ satisfies $E_{n-1} \supseteq \mathcal{E}(\mathcal{C}')$, $E_n \supseteq \mathcal{E}(\mathcal{C}')$ and $\mathsf{e}_{n-1} \in \Sigma \setminus \mathcal{E}(\mathcal{C}')$ (thus $\mathsf{e}_{n-1} \notin \mathcal{E}(\mathcal{C}')$). Assume last assigned value in mp_t is $\mathsf{mp}_t(n) = m$. Now there are two cases:

  - If $\mathsf{e}_n \in \mathcal{E}(\mathcal{C}')$ then $\mathrm{traces}(\mathcal{C}', \mathsf{e}_n) = \{\theta'_e \mathsf{e}_n \mid \forall \theta'_e \in \mathrm{traces}(\mathcal{C}')\}$ in which case the mapping mp, which is a copy of mp_t in which $\mathsf{mp}(n+1) = m+1$ is added, is an enabling compatible mapping. This is so because last added value satisfies the firing rule ($\theta' = \theta'_t \xrightarrow{\mathsf{e}_n}$ and last fired event of θ is also e_n), the completeness rule (as m was the length of θ'_t and now $\theta' = \theta'_t \xrightarrow{\mathsf{e}_n}$ whose length is $m+1$), the projection rule (as $E_n \supseteq \mathcal{E}(\mathcal{C}')$) and the enabling preservation rule from E_{n-1} to E_n due to the fact that $\mathsf{e}_{n-1} \notin \mathcal{E}(\mathcal{C}')$.
  - If $\mathsf{e}_n \notin \mathcal{E}(\mathcal{C}')$ then $\mathrm{traces}(\mathcal{C}', \mathsf{e}_n) = \mathrm{traces}(\mathcal{C}')$ in which case the mapping mp, which is a copy of mp_t in which $\mathsf{mp}(n+1) = m$ is added, is an enabling compatible mapping. This is so because last added value satisfies the firing rule ($\theta' = \theta'_t$ and last fired event of θ could not be fired in the end of θ'_t because $\mathsf{e}_n \notin \mathcal{E}(\mathcal{C}')$), the completeness rule (as m was the length of θ'_t and now $\theta' = \theta'_t$ whose length is also m), the projection rule (as $E_n \supseteq \mathcal{E}(\mathcal{C}')$) and the enabling preservation rule from E_{n-1} to E_n due to the fact that $\mathsf{e}_{n-1} \notin \mathcal{E}(\mathcal{C}')$.

- If $\mathcal{C} = \mathcal{C}_i$, then the arc $\mathcal{C}' \overset{\langle \mathsf{S}_s, \Sigma', \mathsf{S}_d \rangle}{\longrightarrow} \mathcal{C}_i$ is a *transition advancing the mapping* of the ECTIA. This enforces that $E_{n-1} \xrightarrow{\mathsf{e}_{n-1}} E_n$ satisfies $E_{n-1} \supseteq \mathcal{E}(\mathcal{C}')$, $E_n \supseteq \mathcal{E}(\mathcal{C}_i)$ and $\mathsf{e}_{n-1} \in \mathcal{E}(\mathcal{C}')$. Assume last assigned value in mp_t is $\mathsf{mp}_t(n) = m$. Now there are two cases:

  - If $\mathsf{e}_n \in \mathcal{E}(\mathcal{C}_i)$ then $\mathrm{traces}(\mathcal{C}_i, \mathsf{e}_n) = \{\theta'_e \mathsf{e}_n \mid \forall \theta'_e \in \mathrm{traces}(\mathcal{C}_i)\}$, as $\mathcal{C}'$ is a predecessor configuration of $\mathcal{C}_i$ then $\mathrm{traces}(\mathcal{C}_i)$ includes the traces coming from $\mathcal{C}'$ and firing e_{n-1}, hence $\theta'_t \in \mathrm{traces}(\mathcal{C}_i)$. The mapping mp, which is a copy of mp_t in which $\mathsf{mp}(n+1) = m+1$ is added, is an enabling compatible mapping. This is so because last added value satisfies the firing rule ($\theta' = \theta'_t \xrightarrow{\mathsf{e}_n}$ and last fired event of θ is also e_n), the completeness rule (as m was the length of θ'_t and now $\theta' = \theta'_t \xrightarrow{\mathsf{e}_n}$ whose length is $m+1$), the projection rule (as $E_n \supseteq \mathcal{E}(\mathcal{C}_i)$) and the enabling preservation rule from E_{n-1} to E_n due to the fact that the transitions advancing the mapping require that all events that should become newly enabled (including e_{n-1} if it is selfenabling) are disabled in $\mathcal{C}'$.
  - If $\mathsf{e}_n \notin \mathcal{E}(\mathcal{C}_i)$ then $\mathrm{traces}(\mathcal{C}_i, \mathsf{e}_n) = \mathrm{traces}(\mathcal{C}_i)$. As shown before $\theta'_t \in \mathrm{traces}(\mathcal{C}_i)$. Thus the mapping mp, which is a copy of mp_t in which $\mathsf{mp}(n+1) = m$ is added, is an enabling compatible mapping. This is so because last added value satisfies the firing rule ($\theta' = \theta'_t$ and last fired event of θ could not be fired in the end of θ'_t because $\mathsf{e}_n \notin \mathcal{E}(\mathcal{C}_i)$), the completeness rule (as m was the length of θ'_t and now $\theta' = \theta'_t$ whose length is also m), the projection rule (as $E_n \supseteq \mathcal{E}(\mathcal{C}_i)$) and the enabling preservation rule from E_{n-1} to E_n for the same reasons as exposed in previous case.

B.5 Proof of Theorem 6.2

If $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \ldots E_n \xrightarrow{\mathsf{e}_n} E_{n+1}$ and θ_s is one of its suffix then, $\theta_s = E_{k+1} \xrightarrow{\mathsf{e}_k+1} \ldots \xrightarrow{\mathsf{e}_n} E_{n+1}$ for some $k \geq 0$. As it is enabling compatible with some $\theta' = E'_1 \xrightarrow{\mathsf{e}'_1} \ldots \xrightarrow{\mathsf{e}'_m} E'_{m+1}$ such that $\theta' \in traces(\mathcal{C}_i)$ this implies that an enabling compatible mapping mp exists between both. As it is *fully* enabling compatible, then $\mathsf{mp}(k+1) = 1$ and $\mathsf{mp}(n+1) = m+1$. We proceed by induction on the length of θ'.

Base case (length=0)

If θ' is a degenerated trace formed by a single enabling, then the unique configuration it can belong is $\mathcal{C}_1$ as set $traces(\mathcal{C}_i)$ always contains traces with one fired event more than the traces of its predecessor configurations.

As $\theta' = E'_1$ then $\forall i : k+1 \leq i \leq n+1,\ \mathsf{mp}(i) = 1$, this implies that $\forall i : k+1 \leq i \leq n,\ E_i \supseteq E'_1$ and $\forall i : 1 \leq i \leq n+1,\ \mathsf{e}_i \notin E'_1$. If $\theta_s = \theta$ then initial recognition state is directly $\mathcal{C}_1$ (because $k = 0$ and by initialization rule $\mathrm{LB}(E'_1) = \emptyset$) and we are done. Otherwise let $\mathcal{C}$ be the recognition state of θ before $E_k \xrightarrow{\mathsf{e_k}} E_{k+1}$ has been checked. The ECTIA construction guarantees that there is always a reenter arc from each configuration and transition $E_k \xrightarrow{\mathsf{e_k}} E_{k+1}$ can take this arc (otherwise θ_s would not be fully enabling compatible with θ') so initial recognition state in E_1 of θ_s must contain $\mathcal{C}_1$ (but is not guaranteed that it is *uniquely* $\mathcal{C}_1$).

Now all transitions in θ_s can take keep arcs of $\mathcal{C}_1$, so $\mathcal{C}_1$ must belong to recognition state after all θ_s has been checked.

Induction step

If length of θ' is greater than 0, then $\theta' = E'_1 \xrightarrow{\mathsf{e}'_1} \ldots E'_m \xrightarrow{\mathsf{e}'_m} E'_{m+1}$ for $m \geq 1$. Let $\theta'' = E'_1 \xrightarrow{\mathsf{e}'_1} \ldots \xrightarrow{\mathsf{e}'_{m-1}} E'_m$ and $\mathcal{C}_j$ be the configuration such that $\theta'' \in \mathrm{traces}(\mathcal{C}_j)$. The existence of such configuration is guaranteed by the fact that $\theta' \in \mathrm{traces}(\mathcal{C}_i)$ and the definition of the traces function. Precisely because of this definition we know that $\mathcal{C}_j$ must be one of the predecessor configurations of $\mathcal{C}_i$ such that ECTIA arc $\mathcal{C}_j \xrightarrow{\langle \mathsf{S}_s, \Sigma', \mathsf{S}_d \rangle} \mathcal{C}_i$, with $\mathsf{e}'_m \in \Sigma'$, exists.

As θ_s is fully enabling compatible with θ', then there is a prefix of θ_s (name it θ_p) such that θ_p is fully enabling compatible with θ''. Trace fragment θ_p can be obtained by simply clipping θ_s at enabling E_x, where $\mathsf{mp}(x) = m$. Let θ_1 be a trace obtained from θ by removing suffix θ_s and appending θ_p (*i.e.* clipping θ at enabling E_x). Then θ_p is a suffix of θ_1, and is fully enabling compatible with θ'' such that $\theta'' \in \mathrm{traces}(\mathcal{C}_j)$. By the induction hypothesis, the recognition state of the ECTIA must include $\mathcal{C}_j$ after θ_1 has been checked.

Now let us have a look at the clipped part of θ_s. As $\theta_s = E_{k+1} \xrightarrow{\mathsf{e_{k+1}}} \ldots \xrightarrow{\mathsf{e_{x-1}}} E_x \xrightarrow{\mathsf{e_x}} \ldots \xrightarrow{\mathsf{e_n}} E_{n+1}$ the clipped part is formed by $\theta_s = E_x \xrightarrow{\mathsf{e_x}} \ldots E_{n-k+1} \xrightarrow{\mathsf{e_{n-k+1}}} E_{n-k+2}$. As $\mathsf{mp}(x) = m$ and $\mathsf{mp}(n+1) = m+1$ then at some y such that $x \leq y \leq n$ we must have that $\mathsf{mp}(y+1) = \mathsf{mp}(y) + 1$. This forces that $\forall i : x \leq i < y, \mathsf{e}_i \notin E'_m$ and $\forall i : x \leq i \leq y, E_i \supseteq E'_m$. If after parsing θ_1 (which ends in E_x) in the ECTIA we are in $\mathcal{C}_j$, from E_x to E_y we can take *keep* arcs of $\mathcal{C}_j$, hence at E_y recognition state will still be $\mathcal{C}_j$.

As $\mathsf{mp}(y+1) = \mathsf{mp}(y) + 1$ then $E_{y+1} \setminus (E_y \setminus \{\mathsf{e}_y\}) \supseteq \mathrm{LB}(E'_{m+1} \setminus (E'_m \setminus \{\mathsf{e}'_m\}))$ by the enabling preservation rule as well as $E_{y+1} \supseteq E'_{m+1}$ and $E_y \supseteq E'_m$ by the projection rule. This is satisfied by the advance arc $\mathcal{C}_j \xrightarrow{\langle \mathsf{S}_s, \Sigma', \mathsf{S}_d \rangle} \mathcal{C}_i$, with $\mathsf{e}'_m \in \Sigma'$, hence at E_{y+1} recognition state will be $\mathcal{C}_i$. As $\forall i : y < i \leq n+1, \mathsf{mp}(i) = m+1$ then we will remain in configuration $\mathcal{C}_i$ for the same reasons we exposed before for $\mathcal{C}_j$. Thus when θ has been completely checked by the ECTIA, current recognition state will include $\mathcal{C}_i$.

B.6 Proof of Lemma 7.1

First of all we will show that the conditions in Definition 7.7 are necessary *i.e.* if any of them is violated then for sure the CESA will not be able to generate trace θ.

- If $\Sigma \subset \Sigma_\theta$ then events in $\Sigma_\theta \setminus \Sigma$ will never be able to be enabled.
- If $\exists i : 1 < i \leq n+1,\ \exists \mathsf{e} \in E_i : \mathsf{e} \notin E_{i-1},\ \mathsf{e}_{i-1} \not\prec \mathsf{e}$ then event e, which is not enabled in E_{i-1}, can never become enabled in E_i although it should.
- If a $\prec$ relation outside the set $\{\mathsf{e}_j \prec \mathsf{e} \mid \mathsf{e}_i \prec \mathsf{e} \wedge 1 \leq j < i-1\}$ is added, it means that either the relation $\mathsf{e}_j \prec \mathsf{e}$ was added to an event that had no causal predecessor (*i.e.* $\nexists i : \mathsf{e}_i \prec \mathsf{e}$) or

that a new causal relation is established with an event that fires *after* its trigger in the trace. In both cases the event e would becomes enabled later than it should.

- If $\exists i : 1 \leq i < n+1,\ \exists \mathsf{d} \in E_i : \mathsf{d} \notin E_{i+1} \wedge \mathsf{d} \neq \mathsf{e}_i,\ \mathsf{e}_i \not\triangleright \mathsf{d}$ then event d can not be disabled in E_{i+1} if it was enabled in E_i.
- If a relation $\mathsf{a} \triangleright \mathsf{b}$ between the events in Σ_θ (*i.e.* $\mathsf{a}, \mathsf{b} \in \Sigma_\theta$) outside the set $\{\mathsf{e} \triangleright_\mu \mathsf{d} \mid \mathsf{d} \in E_i \wedge \mathsf{d} \notin E_{i+1} \wedge \mathsf{d} \neq \mathsf{e}_i\}$ or the set $\{\mathsf{e} \triangleright \mathsf{d} \mid \mathsf{e}, \mathsf{d} \in E_{n+1}\}$ is added then this means that, by the CESA definition, b cannot be an already disabled event of the trace, otherwise two concurrent events will disable it. Obviously cannot be neither a fired event of the trace as we would have a contradiction. Then the only option is that $\mathsf{b} \in E_{n+1}$, but then a cannot have fired or otherwise $\mathsf{b} \notin E_{n+1}$. Hence $\mathsf{a} \in E_{n+1}$. Then both events should belong to the set E_{n+1}, but this is excluded because they should not belong to the set $\{\mathsf{e} \triangleright \mathsf{d} \mid \mathsf{e}, \mathsf{d} \in E_{n+1}\}$, hence we have another contradiction.
- For events in Σ_P if none of their causal predecessors is in the set $E_{n+1} \cup \Sigma_P$ then it must be some of the events fired in the trace (disabled events cannot have causal successors). Moreover if it does not exist an event that disables it that is fired in the trace *before* one of its causal predecessors, then the event in Σ_P will become enabled in the trace although it does not belong to Σ_θ.

Now we will show that the conditions in Definition 7.7 are sufficient *i.e.* if all of them hold then for sure the CESA will be able to generate trace θ.

- For events in Σ_θ:
  - Definition 7.7 ensures that $\forall \mathsf{e} \in \Sigma_\theta$ its trigger in the trace belongs to the set of their causal predecessors. Additionally any event fired before the trigger can also be part of this set. Hence the event can only become enabled at the point in the trace where it does.
  - Definition also ensures that, if an event is disabled, its disabler in the trace belongs to the set of disablers of the event. This guarantees that the event will be disabled at the expected point in the trace. Additionally events not fired neither disabled (*i.e.* they remain enabled until E_{n+1}) can disable events of E_{n+1} as the observability of the disabling will be after the last state of the trace.
- For events in Σ_P:
  - If some of their causal predecessor is in set $E_{n+1} \cup \Sigma_P$ then they cannot be enabled along θ, as expected.
  - If an event disables them before being enabled (*i.e.* $\exists i,\ \mathsf{e}_i \triangleright \mathsf{d} \wedge \exists \mathsf{e} \in \Sigma : \mathsf{e} \prec \mathsf{d} \wedge (\mathsf{e} = \mathsf{e}_j \wedge i < j \vee \mathsf{e} \in E_{n+1} \cup \Sigma_P)$), we say they are *prevented* and cannot appear in θ, too.
  - Relations between events in Σ_P cannot affect the enablings/disablings of events in Σ_θ, hence the generation of the trace cannot be affected by any number of arbitrary relations of this kind.

B.7 Proof of Lemma 8.1

We need to introduce an auxiliary lemma first. The lemma proves that the timing consistency of a trace is not altered if additional events are added, but only in the case that the added events are allowed to fire at any possible position.

LEMMA B.1
Given a distinct trace $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \ldots E_n \xrightarrow{\mathsf{e}_n} E_{n+1}$, let be θ' a trace equal to θ but in which an arbitrary distinct event e is added to some E_k of θ. Let be Θ the set of traces formed by

considering all possible firing positions of e *in* θ'. *Then* $\mathrm{sol}(\mathrm{LP}(\theta)) \neq \emptyset \Leftrightarrow \mathrm{sol}(\mathrm{LP}(\Theta)) \neq \emptyset$ *and, for any two events* e_i *and* e_j *fired in* θ, *we have that* $\mathrm{mst}_\theta(\mathsf{e}_i, \mathsf{e}_j) = \mathrm{mst}_\Theta(\mathsf{e}_i, \mathsf{e}_j)$. ■ B.1

Proof: *Although the added event* e *can have any imaginable delay, there are only a limited number of possible scenarios:*

- *If* θ *is not timing consistent, then* $\mathrm{sol}(\mathrm{LP}(\theta)) = \emptyset$ *and all maximal separation times are undefined. As any trace* $\theta' \in \Theta$ *satisfies that* $\mathrm{LP}(\theta')$ *has more conditions than* $\mathrm{LP}(\theta)$, *then no* θ' *has solution and* $\mathrm{sol}(\mathrm{LP}(\Theta)) = \emptyset$.
- *If* θ *is timing consistent, then* $\mathrm{mst}_\theta(\mathsf{e}_i, \mathsf{e}_j)$ *is defined as both events are required to be fired. Assume* $(\tau_1, \tau_2, \ldots, \tau_n) \in \mathrm{sol}(\mathrm{LP}(\theta))$ *is the set of values assigned to the firing times of the events (*τ_1 *to* $\mathrm{ft}(\mathsf{e}_1)$, τ_2 *to* $\mathrm{ft}(\mathsf{e}_2)$, ... *) such that* $\mathrm{mst}_\theta(\mathsf{e}_i, \mathsf{e}_j) = \mathrm{ft}(\mathsf{e}_j) - \mathrm{ft}(\mathsf{e}_i) = \tau_j - \tau_i$. *We will show that it exists a trace* $\theta' \in \Theta$ *such that* $\mathrm{sol}(\mathrm{LP}(\theta'))$ *contains at least one set of values that contains all the previous timestamps.*

  Trace θ' *is equal to* θ *except that at some point the extra event* e *is fired. Its firing does not disable any of the events of* θ *hence all timestamps are also a partial solution of* $\mathrm{sol}(\mathrm{LP}(\theta'))$, *the only remaining thing is to determine the timestamp assigned to* $\mathrm{ft}(\mathsf{e})$ *which must belong to the set* $[\mathrm{ft}(\mathsf{e}_{k-1}) + \delta^l(\mathsf{e}), \mathrm{ft}(\mathsf{e}_{k-1}) + \delta^u(\mathsf{e})]$ *as event* e *is first enabled in enabling* E_k. *The exact point of* θ' *at which this firing must occur is easy to compute: the sequence* $\tau_1, \tau_2, \ldots$ *is monotonically increasing. Let be* τ_e *the timestamp assigned to* $\mathrm{ft}(\mathsf{e})$. *The order in the trace of* e *is determined by the position that* τ_e *can occupy in the sequence of timestamps while preserving its monotonicity.*

  This proves that $\mathrm{sol}(\mathrm{LP}(\Theta)) \neq \emptyset$ *and that* $\mathrm{mst}_\theta(\mathsf{e}_i, \mathsf{e}_j) = \mathrm{mst}_\Theta(\mathsf{e}_i, \mathsf{e}_j)$, *since* τ_j *and* τ_i *are the same as in the solution of* θ.

We define predicate $\mathrm{derive}(\Theta, \theta)$ to be true if set Θ can be obtained from θ by iteratively applying the procedure described in the previous lemma.

The proof in the $\boxed{\Leftarrow}$ sense is now easy using the previous lemma. The idea is that any trace $\theta \in \mathrm{expand}(\Theta_{\mathcal{C}})$ has a set Θ' such that $\mathrm{derive}(\Theta', \theta)$ and $\Theta' \subseteq \mathrm{traces(CS)}$. This immediately gives that $\mathrm{mst}_{\Theta'}(\mathsf{a}, \mathsf{b}) \leq \mathrm{mst}_{\mathrm{traces(CS)}}(\mathsf{a}, \mathsf{b})$, which, by previous lemma, is the same as $\mathrm{mst}_{\mathrm{expand}(\Theta_{\mathcal{C}})}(\mathsf{a}, \mathsf{b}) \leq \mathrm{mst}_{\mathrm{traces(CS)}}(\mathsf{a}, \mathsf{b})$. So if $\mathrm{mst}_{\mathrm{traces(CS)}}(\mathsf{a}, \mathsf{b}) < 0$, $\mathrm{mst}_{\mathrm{expand}(\Theta_{\mathcal{C}})}(\mathsf{a}, \mathsf{b})$ must also be.

The proof in the $\boxed{\Rightarrow}$ is more complex. First of all we must explain what does it mean that a set of traces Θ_p, expressible by a partial order $<_p$, is timing inconsistent. In terms of $\mathrm{LP(CS)}$ this means that the equations induced by the partial order and the equations in $\mathrm{LP(CS)}$ are contradictory. Since both elements in isolation are not contradictory, then contradiction can only come from the interaction between both. A useful way to see it, is to think of $\mathrm{LP(CS)}$ as generating an order between the events. If this order contradicts the order in $<_p$, then all the traces of Θ_p are timing inconsistent.

If $\mathrm{mst}_{\mathrm{expand}(\Theta_{\mathcal{C}})}(\mathsf{a}, \mathsf{b}) < 0$, then all traces in $\mathrm{expand}(\Theta_{\mathcal{C}})$ such that $\mathsf{a} < \mathsf{b}$ are timing inconsistent. Let $<_p$ be the partial order of the set of traces in $\mathrm{expand}(\Theta_{\mathcal{C}})$ and firing a before b. This partial order is contradictory with $\mathrm{LP(CS)}$. This means that $\mathrm{LP(CS)}$ generates an ordering that contradicts some of the elements in the partial order $<_p$. This elements can be classified in two types. The partial order of all the traces in $\mathrm{expand}(\Theta_{\mathcal{C}})$ and $\mathsf{a} < \mathsf{b}$. If the contradicted element belongs to the first class, then there is no timing consistent trace reaching $\mathcal{E}(\mathcal{C})$, which contradicts one of the lemma's assumptions. On the other hand if contradicts $\mathsf{a} < \mathsf{b}$, then it means that there is no trace in $\mathrm{traces(CS)}$ in which $\mathsf{a} < \mathsf{b}$, consequently $\mathrm{mst}_{\mathrm{traces(CS)}}(\mathsf{a}, \mathsf{b}) < 0$.

Q.E.D.

B.8 Proof of Lemma 8.2

To prove this theorem an additional number of lemmas have to be established first.

Lemma B.2 (Disabling elimination)

Given a distinct trace $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \cdots E_n \xrightarrow{\mathsf{e}_n} E_{n+1}$ in which there is a single disabling in which event e_i disables event b, lets consider the set of traces Θ, that will contain all the traces θ' equal to θ with the only difference that in θ' event e_i does not disable event b, instead event b is fired later than e_i. Then timing-consistency of θ is exactly the same as Θ. Formally,

$$\mathrm{sol}(\mathrm{LP}(\theta)) \neq \emptyset \Longleftrightarrow \mathrm{sol}(\mathrm{LP}(\Theta)) \neq \emptyset$$

■ B.2

Proof: *First of all let us characterize $\mathrm{LP}(\Theta)$ and $\mathrm{LP}(\theta)$. As Θ contains traces that have the same triggers, we will only have conjunction of equations in $\mathrm{LP}(\Theta)$. In fact the sequence order equations will be the same as in $\mathrm{LP}(\theta)$ (all traces in Θ have the same firing order of events others than b), with the addition that, as formerly disabled event b is now fired after e_i we must add equation $\mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{b})$ to $\mathrm{LP}(\Theta)$. The other difference between $\mathrm{LP}(\Theta)$ and $\mathrm{LP}(\theta)$ is that the latter contains the equation associated with the disabling of b, that is $\mathrm{ft}(\mathsf{e}_i) - \mathrm{ft}(\mathrm{trigger}(\mathsf{b})) \leq \delta^u(\mathsf{b})$, while in $\mathrm{LP}(\Theta)$ this equation is substituted by the equation of the firing of b, that is $\delta^l(\mathsf{b}) \leq \mathrm{ft}(\mathsf{b}) - \mathrm{ft}(\mathrm{trigger}(\mathsf{b})) \leq \delta^u(\mathsf{b})$. To summarize, lets call Λ the set of common equations between $\mathrm{LP}(\Theta)$ and $\mathrm{LP}(\theta)$. Then we have that:*

$$\mathrm{LP}(\theta) = \Lambda \wedge \mathrm{ft}(\mathsf{e}_i) - \mathrm{ft}(\mathrm{trigger}(\mathsf{b})) \leq \delta^u(\mathsf{b})$$
$$\mathrm{LP}(\Theta) = \Lambda \wedge \mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{b}) \wedge \delta^l(\mathsf{b}) \leq \mathrm{ft}(\mathsf{b}) - \mathrm{ft}(\mathrm{trigger}(\mathsf{b})) \leq \delta^u(\mathsf{b})$$

$\boxed{\Leftarrow}$ *First of all we will show that if $\mathrm{LP}(\Theta)$ is timing consistent, then $\mathrm{LP}(\theta)$ will also be. This is easy to show, as $\mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{b})$ and $\mathrm{ft}(\mathsf{b}) - \mathrm{ft}(\mathrm{trigger}(\mathsf{b})) \leq \delta^u(\mathsf{b})$ implies that $\mathrm{ft}(\mathsf{e}_i) - \mathrm{ft}(\mathrm{trigger}(\mathsf{b})) \leq \delta^u(\mathsf{b})$. Hence $\mathrm{LP}(\Theta) \supseteq \mathrm{LP}(\theta)$* i.e. *all equations of $\mathrm{LP}(\theta)$ also appear in $\mathrm{LP}(\Theta)$, so if it is timing consistent, then $\mathrm{LP}(\theta)$ must also be.*

$\boxed{\Rightarrow}$ *In this sense the proof is a little bit longer. We want to show that if $\mathrm{LP}(\theta)$ is timing consistent, then $\mathrm{LP}(\Theta)$ also is. If $\mathrm{LP}(\theta)$ is timing consistent means that there is no contradiction in Λ, thus we must only show that adding equations $\mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{b})$ and $\delta^l(\mathsf{b}) \leq \mathrm{ft}(\mathsf{b}) - \mathrm{ft}(\mathrm{trigger}(\mathsf{b})) \leq \delta^u(\mathsf{b})$ to Λ does not create any contradiction. First thing we notice is that adding only one of these equations cannot cause any problem, as there will be only one equation bounding the value of $\mathrm{ft}(\mathsf{b})$ and a valid value for it can always be found. Thus, the only possibility to introduce a contradiction is that there is a contradiction between them. However, this is only possible if $\mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{b})$ and $\mathrm{ft}(\mathsf{b}) \leq \delta^u(\mathsf{b}) + \mathrm{ft}(\mathrm{trigger}(\mathsf{b}))$ give unsatisfiable bound for $\mathrm{ft}(\mathsf{b})$. But this is only possible if $\mathrm{ft}(\mathsf{e}_i) > \delta^u(\mathsf{b}) + \mathrm{ft}(\mathrm{trigger}(\mathsf{b}))$ which is equal to $\mathrm{ft}(\mathsf{e}_i) - \mathrm{ft}(\mathrm{trigger}(\mathsf{b})) > \delta^u(\mathsf{b})$, but we know that $\mathrm{ft}(\mathsf{e}_i) - \mathrm{ft}(\mathrm{trigger}(\mathsf{b})) \leq \delta^u(\mathsf{b})$ is true because $\mathrm{LP}(\theta)$ is consistent, thus, there is no possible source of contradiction between the two new equations added to Λ, so $\mathrm{LP}(\Theta)$ will also be timing consistent.*

We can apply Lemma B.2 repeatedly to reduce the checking of a trace θ, with an arbitrary number of disablings, to the checking of a set of traces without disablings. Let us call this set as Θ_d. Using an identical approach as in Lemma B.1 we can also prove that $\mathrm{mst}_\theta(\mathsf{e}_i, \mathsf{e}_j) = \mathrm{mst}_{\Theta_d}(\mathsf{e}_i, \mathsf{e}_j)$. Now consider each one of the traces in Θ_d, they also belong to $\mathrm{traces}(\mathrm{CS}_{\mathrm{TZ}})$. If we select one of them, let $\mathcal{E}(\mathcal{C}')$ be the enabling reached by it in the corresponding GRE. We can use a minor variation of Lemma 8.1 (that uses a CESAR without disablings instead of a CES) to prove Lemma 8.2.

B.9 Proof of Theorem 8.3

Now we can start the proof of the theorem. Let $\mathrm{CS}_{\mathrm{TZ}} = \langle \Sigma, \prec, \emptyset, <_{\mathrm{CS}} \rangle$ be the CESAR associated to the timing zone TZ to which the enabling $\mathcal{E}(\mathcal{C})$ belongs. Let $\Theta_{\mathcal{C}}$ be the set of all traces reaching enabling $\mathcal{E}(\mathcal{C})$. For each outgoing event from $\mathcal{E}(\mathcal{C})$ *i.e.* $\forall \mathsf{e} \in \mathcal{E}(\mathcal{C})$, consider the set of traces Θ_{e}, formed from Θ by firing event e at the end of each one of the traces. Let $<_{\mathsf{e}}$ be the partial order satisfied by all traces in Θ_{e}.

Using disabling elimination (Lemma B.2) and Lemma B.1 we obtain a set of traces, that we denote as $\mathrm{tes}(\Theta_{\mathsf{e}})$, that is a subset of $\mathrm{twd}(\mathrm{CS_{TZ}})$ such that $\mathrm{sol}(\mathrm{LP}(\Theta_{\mathsf{e}})) \neq \emptyset \Leftrightarrow \mathrm{sol}(\mathrm{LP}(\mathrm{tes}(\Theta_{\mathsf{e}}))) \neq \emptyset$.

If set Θ_{e} includes some timing consistent trace, then $\mathrm{sol}(\mathrm{LP}(\Theta_{\mathsf{e}})) \neq \emptyset$. This means that also $\mathrm{sol}(\mathrm{LP}(\mathrm{tes}(\Theta_{\mathsf{e}}))) \neq \emptyset$ and, in set $\mathrm{tes}(\Theta_{\mathsf{e}})$, there is also a timing consistent trace. As $\mathrm{tes}(\Theta_{\mathsf{e}}) \subseteq \mathrm{twd}(\mathrm{CS_{TZ}})$ the timing consistent trace also belongs to set $\mathrm{twd}(\mathrm{CS_{TZ}})$. This trace satisfies ordering $<_{\mathsf{e}}$. Given a set of traces A, $\mathrm{mst}_A(\mathsf{a}, \mathsf{b})$ can never be below 0 if event a appears before b in some timing consistent trace of set A. We already know that there is a timing consistent trace in $\mathrm{twd}(\mathrm{CS_{TZ}})$ such that satisfies partial order $<_{\mathsf{e}}$ so no $<_{\mathrm{lzy}}$ relation can be derived that contradicts $<_{\mathsf{e}}$. Consequently, at least one trace in Θ_{e} satisfies the $<_{\mathrm{lzy}}$ relation.

If set Θ_{e} has only timing inconsistent traces, then $\mathrm{sol}(\mathrm{LP}(\Theta_{\mathsf{e}})) = \emptyset$. This means that also $\mathrm{sol}(\mathrm{LP}(\mathrm{tes}(\Theta_{\mathsf{e}}))) = \emptyset$ and in set $\mathrm{tes}(\Theta_{\mathsf{e}})$ also all traces are timing inconsistent. The only possibility to render all trace in set $\mathrm{tes}(\Theta_{\mathsf{e}})$ timing inconsistent is that some event ordering of the common partial order is inconsistent. If the inconsistent ordering of the event takes place between events related (directly or indirectly) by timing zone restrictions (*i.e.* an ordering that should be satisfied by all traces of $\mathrm{CS_{TZ}}$), then no trace in $\mathrm{twd}(\mathrm{CS_{TZ}})$ is timing consistent and all maximal separation times are undefined. On the other hand if the inconsistent ordering (for instance $\mathsf{a} < \mathsf{b}$) does not affect the whole set of traces in $\mathrm{twd}(\mathrm{CS_{TZ}})$, then timing consistent traces in the set must verify that $\mathrm{mst}_{\mathrm{twd}(\mathrm{CS_{TZ}})}(\mathsf{a}, \mathsf{b}) < 0$, hence $\mathsf{b} < \mathsf{a}$ will form part of the $<_{\mathrm{lzy}}$ relation and all traces in Θ_{e} will do not satisfy relation $<_{\mathrm{lzy}}$.

B.10 Proof of Theorem 10.1

Let $\rho = \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \xrightarrow{\mathsf{e}_2} \ldots \mathsf{s}_n \xrightarrow{\mathsf{e_n}}$ be a run of LzTS A, and $\theta_\rho = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \ldots E_n \xrightarrow{\mathsf{e_n}}$ be its corresponding trace. Let $A \odot \mathrm{EC} = \langle S', \Sigma, T', \mathsf{s}'_0, \mathrm{FR}' \rangle$ be the resulting LzTS of the composition of A with ECTRA EC. Let be θ be a trace equal to θ_ρ but without the last fired event *i.e.* $\theta = E_1 \xrightarrow{\mathsf{e}_1} E_2 \xrightarrow{\mathsf{e}_2} \ldots E_n$ Then by Theorem 6.2 we know that for each configuration $\mathcal{C}$ in EC such that there is a suffix $\theta_s = E_k \xrightarrow{\mathsf{e_k}} \ldots E_n$ of θ satisfying that θ_s is fully enabling compatible with some of the traces in $\mathrm{traces}(\mathcal{C})$, there would be a state $(\mathsf{s}_n, \mathcal{C})$ in S'. This means that several different suffixes of θ_ρ are fully enabling compatible with traces of different configurations, then there will be multiple $(\mathsf{s}_n, \mathcal{C})$ in S' (and at least two states with $\mathcal{C} \neq \perp$). We know by Theorem 6.1 that if we have $(\mathsf{s}_n, \mathcal{C})$ in S' then θ_ρ (which has an additional event e_n fired) will be enabling compatible with some of the traces in $\mathrm{traces}(\mathcal{C}, \mathsf{e}_n)$.

By the ECTRA construction from a LzCES we know that if all traces in $\mathrm{traces}(\mathcal{C}, \mathsf{e}_n)$ are timing inconsistent, then all arcs $\mathcal{C} \xrightarrow{\langle \mathsf{S}_s, \{\mathsf{e}_n\}, \mathsf{s}_d \rangle}$ of EC will be declared as lazy *i.e.* $\mathcal{C} \notin \mathrm{FR}(\mathsf{e}_n)$ in EC. As $\mathrm{traces}(\mathcal{C}, \mathsf{e}_n) \subseteq \Theta$ then $\theta_\rho \in \mathsf{EnComp}(\mathcal{L}(A), \Theta)$. First of all as composition $A \odot \mathrm{EC}$ is a type of product it should be clear that $\mathcal{L}(A \odot \mathrm{EC}) \subseteq \mathcal{L}(A)$, so only traces are removed. As removed traces by composition require that one of the composed arcs to be lazy, and we are considering only the valid behavior of A (so all the arcs of the traces we are considering are not lazy), then the lazyness of the composed arcs only depends on the lazyness of the arcs of EC. Thus it is not guaranteed that θ_ρ is removed unless there for all configuration $\mathcal{C}$ such that $\exists \theta' \in \mathrm{traces}(\mathcal{C}, \mathsf{e}_n) : \theta_\rho \geq_{\mathsf{ec}} \theta'$ we have that $\mathcal{C} \notin \mathrm{FR}(\mathsf{e}_n)$. This is surely satisfied if there is a unique configuration $\mathcal{C}$ such that $\exists \theta' \in \mathrm{traces}(\mathcal{C}, \mathsf{e}_n) : \theta_\rho \geq_{\mathsf{ec}} \theta'$ which corresponds to the case where the ECTRA is deterministic, or that no indeterministic arc is ever taken in the composition. In this case all traces in $\mathcal{L}(A)$ that are enabling compatible with some of the traces in Θ will be removed, thus $\mathcal{L}(A \odot \mathrm{EC}) = \mathcal{L}(A) \setminus \mathsf{EnComp}(\mathcal{L}(A), \Theta)$. On the other hand if indeterministic arcs of the ECTRA are taken, trace removal is not guaranteed (because depends on the lazyness of event e_n in all involved configurations as we have seen), we have that $\mathcal{L}(A \odot \mathrm{EC}) \supseteq \mathcal{L}(A) \setminus \mathsf{EnComp}(\mathcal{L}(A), \Theta)$.

B.11 Proof of Theorem 11.2

This theorem could have been proved using Theorem 8.3, but here we will use a different approach.

Given a run of the zone, we can establish a mapping between the arcs and states of the run and the arcs and configurations of the GRC of the CESAR. We will denote this relationship by $\mathsf{GRC}_{map}(\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b) = \mathcal{C}_a \xrightarrow{\mathsf{e}} \mathcal{C}_b$.

$\boxed{\Leftarrow}$ The proof in this sense is very easy, as a timing consistent trace exists trough $\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b$, and the trace itself is a word of LCS then for any two events e_i and e_j such that $i < j$ it must be true that the maximum separation time from e_i to e_j must be greater or equal to 0, thus no lazy relation like $\mathsf{e}_j <_{lzy} \mathsf{e}_i$ can be established by the maximum separation time algorithm. Consequently all the arcs of the trace will satisfy all lazy relations in $<_{lzy}$, so they would not be lazy arcs themselves.

$\boxed{\Rightarrow}$ To prove this sense we will show that if there is no timing consistent trace through that arc, then we would have found a lazy relation in the timing analysis of the CES that would have rendered the arc lazy according to the definition of firing region of an event in Definition 7.3.

First we will show that the system of equations associated to the timing analysis of CS, that is $\Lambda(\mathrm{CS})$ is equivalent to the timing consistency checking of all the traces of CS. In fact this is straightforward to prove when no disablgins are involved, as different sets of triggers induce the max contstraints of $\Lambda(\mathrm{CS})$. In the presence of disablings we can get rid of them by using Lemma B.2.

Now that this link has been established we derive the system of equations that express the timing consistency of all the traces through the arc $\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b$. We will denote such system as $LP(\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b)$. We obtain such system by restricting $\Lambda(\mathrm{CS})$ with precedences between events that ensure that all traces (and only these) crossing the arc will satisfy them. Let be $\mathcal{C}_a \xrightarrow{\mathsf{e}} \mathcal{C}_b$ the corresponding arc in the GRC of CS of $\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b$ *i.e.* $\mathsf{GRC}_{map}(\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b) = \mathcal{C}_a \xrightarrow{\mathsf{e}} \mathcal{C}_b$. The precedences that we must add are of the form $\forall \mathsf{e}_i, \mathsf{e}_j : \mathsf{e}_i \in \mathcal{C}_a \wedge \mathsf{e}_j \notin \mathcal{C}_a, \mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{e}_j)$ or $\forall \mathsf{e}_i : \mathsf{e}_i \notin \mathcal{C}_b, \mathrm{ft}(\mathsf{e}) \leq \mathrm{ft}(\mathsf{e}_i)$. We identify the set formed by all these equations as Λ_{arc}. Then $\mathrm{LP}(\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b) = \Lambda(\mathrm{CS}) \wedge \Lambda_{arc}$.

If all traces through the arc are inconsistent, then $\mathrm{LP}(\mathsf{s}_a \xrightarrow{\mathsf{e}} \mathsf{s}_b)$ must also be inconsistent. However, $\Lambda(\mathrm{CS})$ cannot be inconsistent on its own, or otherwise we could not have obtained a LzCES, and Λ_{arc} can neither be as it only expresses a partial order between events (it only enforces event e to happen after all fired events up to $\mathcal{C}_a$, and before all the events fired after it). As a result the inconsistency must come from the interaction between $\Lambda(\mathrm{CS})$ and Λ_{arc}. This means that $\Lambda(\mathrm{CS})$ implies some precedence between events that contradicts some of the equations in Λ_{arc}. Remeber that precedences implied by $\Lambda(\mathrm{CS})$ are the ones that conform the lazy relation $<_{\mathrm{lzy}}$. It is straightforward to show that a lazy relation $<_{\mathrm{lzy}}$ that contradicts Λ_{arc} renders the arc $\mathcal{C}_a \xrightarrow{\mathsf{e}} \mathcal{C}_b$ lazy.

B.12 Counterexample for the proof of Lemma 12.5

In this section we complete the proof of Lemma 12.5 by providing a counterexample TTS A such that $A \in$ BNCTS but $A \notin$ BRIFTS. The counterexample is depicted in Figure B.3 (a). This system contains an infinite set of failure traces that can be obtained by repeatedly unfolding the loop formed by events x and y. Subfigure (b) contains one of such traces with only one unfolding. In (c) we show a parametrized representation of the associated graph to any of these traces. We will use the term θ_i to refer to the trace in which the extensible fragment is unfolded i times.

System A belongs to class BNCTS because it is a correct system (*i.e.* all failure traces are timing inconsistent) and in all failure traces there is a negative cycle of, at most, four arcs. Although the graph of (c) does not contain all the arcs of $\mathrm{graph}(\mathrm{LP}^*(\theta_i))$ for readability (see §12.6), the negative cycle marked in red is the unique simple negative cycle in the graph. However, it is straightforward that the set of the preserved events for the corresponding timing inconsistent reduced trace contains all the unfolded events, hence the length of the reduced trace can be enlarged at will. Consequently there is no bound for reduced traces and A does not belong to BRIFTS.

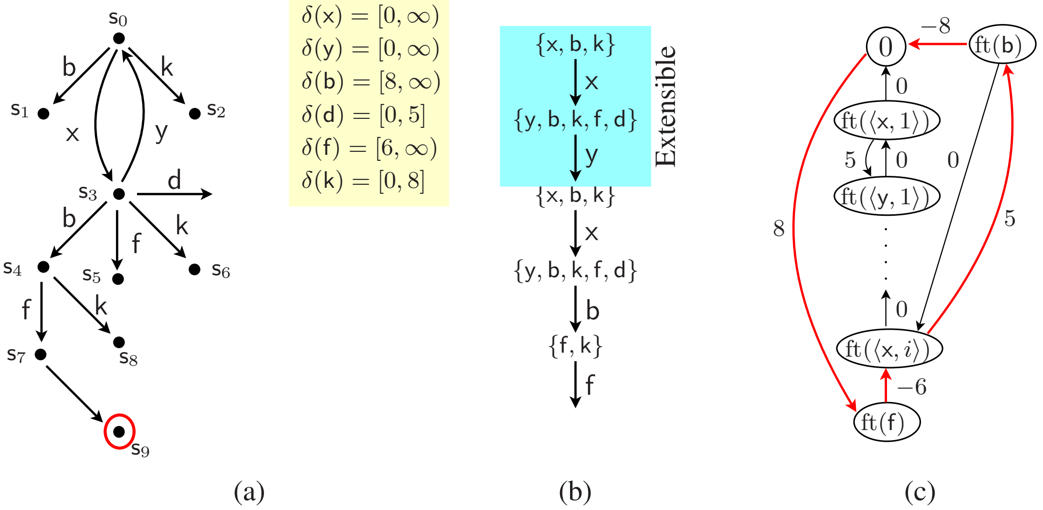


Figure B.3. TTS that belongs to class BNCTS but not to class BRIFTS. (a) Zeno TTS. (b) Trace that has a fragment that can be arbitrarily enlarged (blue box). (c) Associated graph to the trace where the extensible fragment has been repeated $i-1$ times.

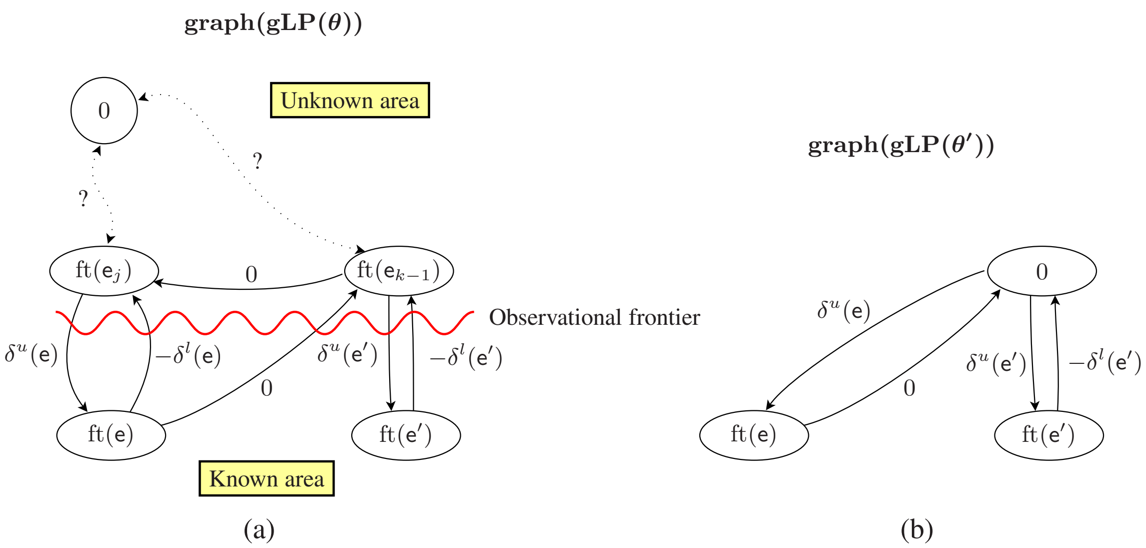


Figure B.4. Associated graphs of: (a) trace θ, and (b) its suffix trace θ'.

B.13 Modeling uncertainty in timing consistency checking

Assume we have a trace $\theta = E_1 \xrightarrow{e_1} E_2 \xrightarrow{e_2} \ldots E_{k-1} \xrightarrow{e_{k-1}} E_k \xrightarrow{e_k} E_{k+1} \ldots$ but that we can only observe its behavior starting from enabling E_k, that is the observed trace is $\theta' = E_k \xrightarrow{e_k} E_{k+1} \ldots$. In the associated graph to $\mathrm{gLP}(\theta)$ used to check the timing consistency (see Figure B.4) there is many information not available. For all events whose trigger fires after E_k, the two graphs $\mathrm{gLP}(\theta)$ and $\mathrm{gLP}(\theta')$ will be isomorphic, so we must only worry about events that were enabled by some event previous to e_k. We consider two different possibilities: assume $e \in E_k$ and $\mathrm{trigger}(e, \theta) = e_j$ with $j < k-1$ as well as $e' \in E_k$ and $\mathrm{trigger}(e, \theta) = e_{k-1}$. Although many information is not available some facts about $\mathrm{gLP}(\theta)$ are known: if $e_j <_{fo} e_{k-1}$, then there must

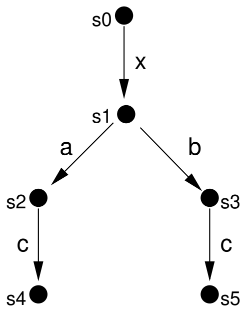


Figure B.5. TTS showing that $\mathrm{ft}(\mathsf{a}) \leq \mathrm{ft}(\mathsf{b})$ must be used to express *logical* precedence.

exist an arc $\mathrm{ft}(\mathsf{e}_{k-1}) \xrightarrow{0} \mathrm{ft}(\mathsf{e}_j)$ and since $\mathrm{trigger}(\mathsf{e}, \theta) = \mathsf{e}_j$ the usual delay arcs between $\mathrm{ft}(\mathsf{e})$ and $\mathrm{ft}(\mathsf{e}_j)$ must also be present.

In all the previous discussion, we have assumed that events e and e' are fired in θ, but the discussion is similar in the other cases (if they are disabled or remain enabled until the end). As there are arcs $\mathrm{ft}(\mathsf{e}_j) \xrightarrow{\delta^u(\mathsf{e})} \mathrm{ft}(\mathsf{e})$ and $\mathrm{ft}(\mathsf{e}_{k-1}) \xrightarrow{0} \mathrm{ft}(\mathsf{e}_j)$, then arc $\mathrm{ft}(\mathsf{e}_{k-1}) \xrightarrow{\delta^u(\mathsf{e})} \mathrm{ft}(\mathsf{e})$ is also available that is not lost when all events in the unknown part of the trace are removed (with the exception of node $\mathrm{ft}(\mathsf{e}_{k-1})$ that is transformed into node 0 in $\mathrm{gLP}(\theta'))$. For the case of event e' conversion is even easier because of the conversion of $\mathrm{ft}(\mathsf{e}_{k-1})$ into node 0.

This illustrates the fact that events that are enabled before the firing of e_k should be considered conservative. As far as initially enabled events concerns, we assume that they are enabled by an unknown sequence of firings hence we cannot assume that they are enabled by e_k and this implies to set their lower delay to 0.

B.14 Why linear constraints are translated as $\leq$ and not $<$?

This is a rather philosophical question. In equation 8.3 we add inequality $\mathrm{ft}(\mathsf{e}_i) \leq \mathrm{ft}(\mathsf{e}_j)$ to model the fact that e_i happens before e_j. Obviously if $\mathrm{ft}(\mathsf{e}_i) = \mathrm{ft}(\mathsf{e}_j)$ events happen at the same time and *logical* order might be any of the two possible (but interleaving semantics enforce that one of them happens, logically, before the other, even if they have the same firing time). Why then not to use inequation $\mathrm{ft}(\mathsf{e}_i) < \mathrm{ft}(\mathsf{e}_j)$? The reason lies in the definition of a valid run in a timed TS (Definition 2.6). We will illustrate this point with an example.

In Figure B.5 we can see a TTS. Assume the following delays for the events: $\delta(\mathsf{x}) = [0]$ and $\delta(\mathsf{a}) = \delta(\mathsf{b}) = \delta(\mathsf{c}) = [1]$. There are only two possible runs in the system, namely $\mathsf{s}_0 \xrightarrow{\mathsf{x}} \mathsf{s}_1 \xrightarrow{\mathsf{a}} \mathsf{s}_2 \xrightarrow{\mathsf{c}} \mathsf{s}_4$ and $\mathsf{s}_0 \xrightarrow{\mathsf{x}} \mathsf{s}_1 \xrightarrow{\mathsf{b}} \mathsf{s}_3 \xrightarrow{\mathsf{c}} \mathsf{s}_5$. Both runs are timing consistent according to Definition 2.6 with the following timestamps: $\tau_1 = 0, \tau_2 = 1, \tau_3 = 2$. This example shows also the reason why we cannot consider runs with different logical order but same firing time as equal: in our example both a and b must fire at time 1, but the resulting behaviour of one happening logically before the other is different depending on the event that appears first. This might be counterintuitive, but is needed to keep the theory coherent if interleaved semantics are used.

However, imagine now state s_4 is a failure state, then if we build a CES from the trace reaching it we will have to incorporate the disabling $\mathsf{a} \triangleright \mathsf{b}$. When doing the timing analysis of this CES we will have to model that a happens before b. If we translate this as $\mathrm{ft}(\mathsf{a}) < \mathrm{ft}(\mathsf{b})$ the timing analysis is inconsistent. Moreover doing the same thing for the other trace has the same result! However this enters in contradiction with the fact that a timing consistent run should always be analyzable by timing analysis of a CES. The problem is that using the strict inequation we elliminate one situation in which one of the events happens *logically* before the other, which is an allowed behaviour of the theory. On the other hand if we use the non-strict inequation, timing analysis is consistent.

Please note that when retrieving the relative ordering of events from a timing analysis on a CES we take the opposite approach, because only a strict inequality of the firing times can imply a logical precedence between two events (as we have already seen if they can have the same firing time they can happen logically in whatever order).

Appendix C

Additional examples

Given the strong connections between several parts of this work, when the material is lineraly presented it is sometimes difficult to provide good examples that help to justify some of the decisions taken along the discussion, mainly because the consequences only appear in later stages of the work that has not been yet presented at the moment where the decision is introduced. This appendix provides some illustrative examples of elements that are better understood when the rest of the material has been presented and provide a good insight of the reasons behind some key elements of this work.

C.1 The need of event structures with (asymmetric) conflict

In Definition 5.4 disablings of events in traces were introduced, but this phenomenon is not properly modelled by the CESs that do not allow conflict to be represented (for instance CES of Definition 7.1). However we need to conveniently include such feature into our CES model because enabling compatibility of traces takes it into account (see *enabling preservation* rule in Definition 5.16). Without a proper modelling of the disabling phenomenon the enabling compatible product will be, in some cases, unable to falsify known timing-inconsistent traces as shown in Figure C.1.

In this example the firing of event a disables b and viceversa. The event structure of subfigure (d) does not model this phenomenon. Consequently the composition with the ECTRA of subfigure (f) produces a LzTS, shown in subfigure (g), in which the original failure trace is not removed, violating one of the first principles of iterative refinement methods. The reason because the trace is not removed is precisely the lack of disabling in the ECTRA: configuration $\mathcal{C}_2$ expects that events b, c and d are enabled. However in the original trace of subfigure (c), after the firing of a, the events enabled are only c and d and the *projection* condition of the enabling compatible mapping definition cannot be satisfied. As a result the arc $\{\mathsf{a},\mathsf{b},\mathsf{c}\}\xrightarrow{\mathsf{a}}\{\mathsf{c},\mathsf{d}\}$ goes to the *nothing recognised* configuration $\perp$ which prevents the timing inconsistent arcs to be removed.

C.2 Effects of a careless timing analysis

In Figure C.2 we can see a system for which the preliminary verification methodology produces a CESAR (subfigure (d)) that is equal to the one shown in Figure 8.2. If one performs timing analysis without considering timing zones, then precedence $\mathsf{a} < \mathsf{b}$ could have been derived (subfigure (e)). But this clearly produces inconsistent results when composed with the original system (subfigure (f)).

A correct timing analysis needs two elements: a correct methodology to derive the timing precedences (as we have seen in our previous example) and the right CESAR to be analyzed. In Figure C.3 (a) we give an example of system in which all its traces do also belong to the CESAR

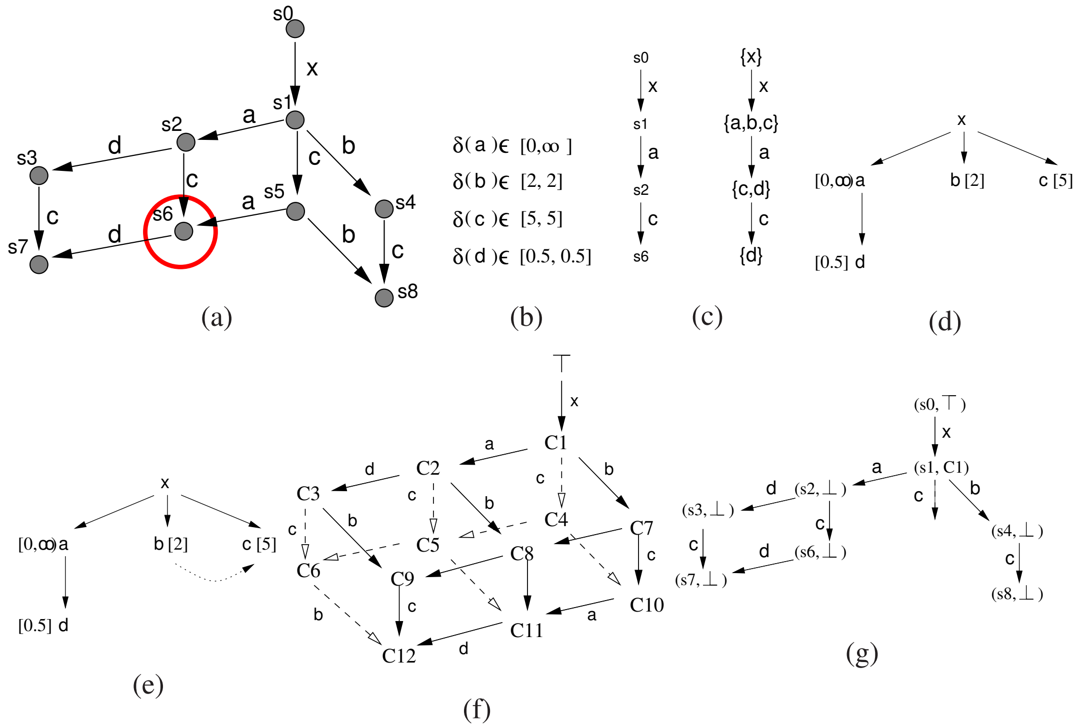


Figure C.1. Example of verification with relative timing: (a,b) TTS and delay intervals. Circled state s_6 is designed to be a failure state. (c) A possible run to this failure state and its corresponding trace. They are timing inconsistent. (d) Event structure derived from the previous trace in which no disabling is present. (e) Event structure annotated with timing arcs (LzCES). (f) Simplified depiction of the ECTRA induced by the LzCES. (g) Classical LzTS obtained after composition. The original failure trace is not removed from the system!

of subfigure (b). However this CESAR is not representative enough to prove that state s_{13} is not reachable through a timing consistent trace, as proved in (c) and (d).

If the lack of convergence of concurrent paths is considered during CESAR construction, then a restriction is added that allows to successfully complete the analysis (the resulting CESAR is shown in subfigure (e)).

C.3 Search failure trace method constraints

In this example we will show how the iterative refinement methodology can become divergent if the search failure trace method used does not return a failure trace whose length is bounded by some integer k with respect from the shortest failure trace present in the system. In Figure C.4 we can see a TTS that is a small variation of the system of Figure 12.6 that has been extended with one unbounded delay event y and one transition $s_0 \xrightarrow{y} s_7$. If state s_7 is declared as a failure state then the system is clearly incorrect and the run $s_0 \xrightarrow{y} s_7$ is the shortest failure trace as well as the shortest (and unique) timing consistent failure trace.

Now if the search failure method finds first run $s_0 \xrightarrow{x} s_1 \xrightarrow{e} s_3 \xrightarrow{a} s_4$, and after composing still always find the failure trace that goes through failure arc $s_3 \xrightarrow{a} s_4$ which we already know it is timing inconsistent (see the discussion in Figure 12.6 on page 197), the method will always find timing inconsistent traces of increasing length and will never terminate. This cannot happen if the search failure trace method gives always a failure trace within a bounded interval from the shortest failure trace as at some point the next fail run going through arc $s_3 \xrightarrow{a} s_4$ will have a length outside

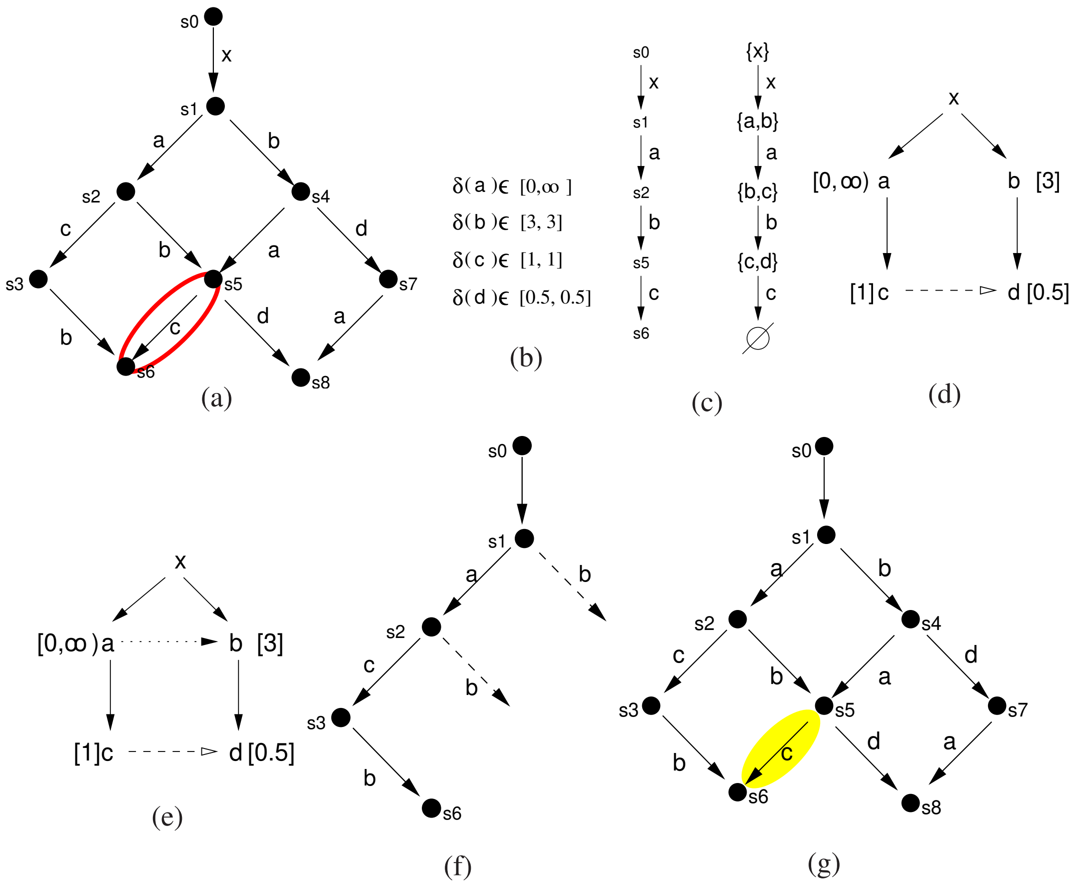


Figure C.2. Example of timing analysis problem: (a,b) TTS and delay intervals. Circled arcs $s_5 \xrightarrow{c} s_6$ is designed to be a failure arc. (c) A possible run to this failure state and its corresponding trace. They are timing inconsistent. (d) Event structure derived from the previous trace in which disabling of d by c is present. (e) Event structure annotated with timing arcs. (f) Classical LzTS obtained after composition. Many timing consistent traces are removed accidentally from the system!! (g) Real timed state space of the system (shaded arcs are unfireable). A correct refinement should never produce a smaller state space.

the allowed bound and the run found could only be $s_0 \xrightarrow{y} s_7$ which is timing consistent and the analysis will terminate.

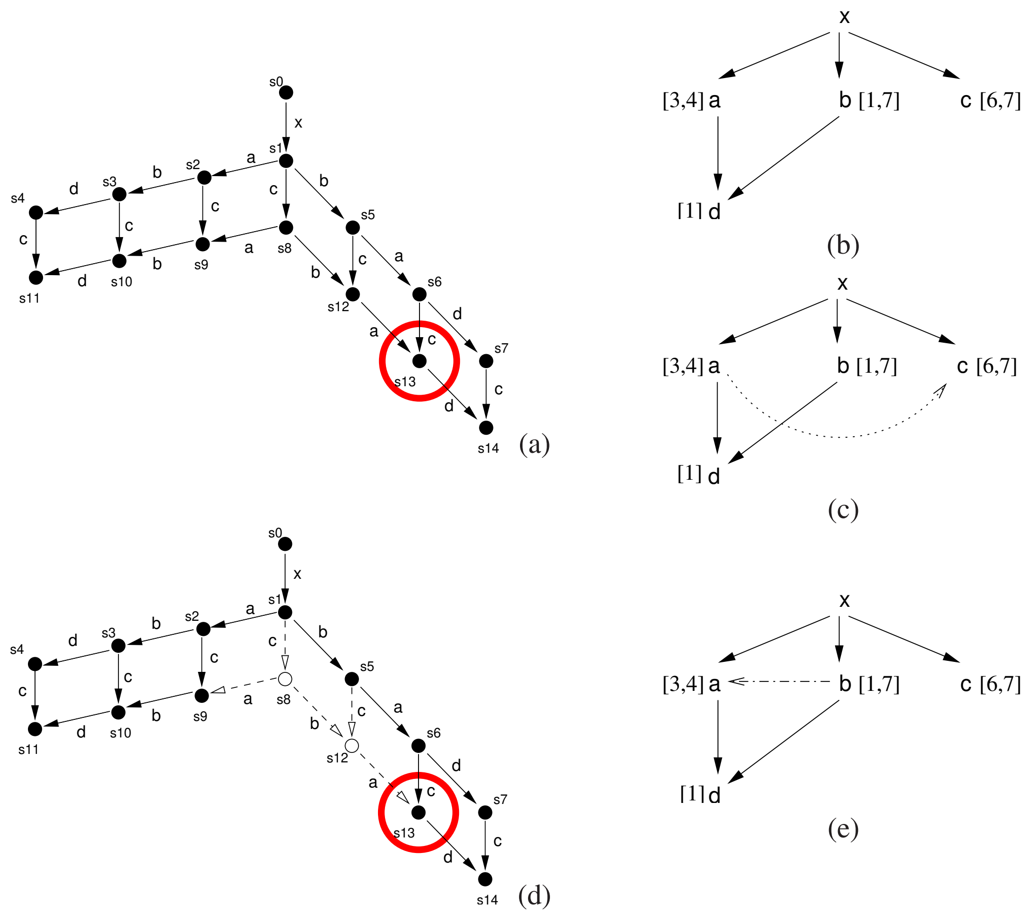
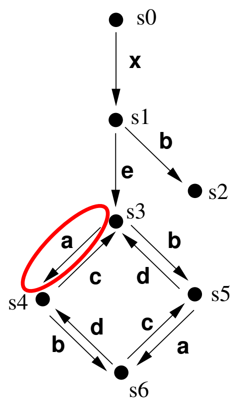


Figure C.3. (a) Untimed state space of a TTS. (b) CESAR that expresses the causality in all the paths present in the system, with annotated delays for the events. Its zone in s_0 is formed only by states $\{s_0, s_1, s_2, s_5, s_8\}$ because of the lack of convergence of concurrent paths. (c) LzCES obtained after timing analysis, only showing that a must happen before c. (d) LzTS showing the refinement produced by the LzCES. There is still a failure trace, but timing consistency checking shows it is inconsistent! (e) CESAR that should have been used in order to prune the inexistant failure.



$\delta(x) = [0, \infty)$
$\delta(y) = [0, \infty)$
$\delta(a) = [2]$
$\delta(b) = [2]$
$\delta(c) = [2]$
$\delta(d) = [2]$
$\delta(e) = [1]$

Figure C.4. TTS for which any failure trace search method that does not return a failure trace within a bounded interval from the shortest failure trace can make the methodology divergent.

References

[AAB00] Aurore Annichini, Eugene Asarin, and Ahmed Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Computer Aided Verification*, pages 419–434, 2000.

[ABABL03] Luca Aceto, Patricia Bouyer, no Augusto Burgue and Kim G. Larsen. The power of reachability testing for timed automata. *Theor. Comput. Sci.*, 300(1-3):411–475, 2003.

[ABH+97] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S. Rajamani. Partial-order reduction in symbolic state space exploration. In *Proceedings International Workshop on Computer Aided Verification*, volume 1254 of *LNCS*, pages 340–351. Springer-Verlag, 1997.

[ABK+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In O. Maler, editor, *Proceedings of HART'97*, volume 1201 of *LNCS*, pages 346–360. Springer-Verlag, 1997.

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[ACD90] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking for real-time systems. In *Proceedings of the 5th annual IEEE Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, 1990.

[ACD+92] R. Alur, C. Courcoubetis, D.L. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Real-Time Systems Symposioum*, pages 157–166, 1992.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[ACGM04] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, and Marco Maratea. A sat-based decision procedure for the boolean combination of difference constraints. In *SAT*, 2004.

[AD90] R. Alur and D.L. Dill. Automata for modeling real-time systems. In *Automata, Languages and Programming: Proceedings of the 17th ICALP*, volume 443 of *LNCS*, pages 322–335. Springer-Verlag, 1990.

[AD94] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AH89] R. Alur and T.A. Henzinger. A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[AH90] R. Alur and T.A. Henzinger. Real-Time Logics: Complexity and Expressiveness. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 390–401. IEEE Computer Society Press, 1990.

[AH97] R. Alur and T. A. Henzinger. Modularity for timed and hybrid systems. In *International Conference on Concurrency Theory*, volume 1243 of *LNCS*, pages 74–88. Springer-Verlag, 1997.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Taşiran. MOCHA: Modularity in model checking. In *Proceedings International Workshop on Computer Aided Verification*, LNCS, pages 521–525. Springer-Verlag, 1998.

[AIKY92] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximations. In *Proceedings International Workshop on Computer Aided Verification*, volume 663 of *LNCS*, pages 137–150. Springer-Verlag, 1992.

[AK83] S. Aggarwal and R.P. Kurshan. Modeling elapsed time in protocol specification. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification*, volume III, pages 51–62. North-Holland, 1983.

[AK95] R. Alur and R.P. Kurshan. Timing Analysis in COSPAN. In *Hybrid Systems III: Verification and Control*, volume 1066, pages 220–231. Springer-Verlag, October 1995.

[Alu98] R. Alur. Timed automata, 1998. In NATO ASI Summer School on Verification of Digital and Hybrid Systems. Available at http://www.cis.upenn.edu/∼alur/Nato97.ps.gz.

[AMP98] Eugene Asarin, Oded Maler, and Amir Pnueli. On discretization of delays in timed automata and digital circuits. In *CONCUR '98: Proceedings of the 9th International Conference on Concurrency Theory*, pages 470–484, London, UK, 1998. Springer-Verlag.

[Arn94] A. Arnold. *Finite Transition Systems*. Prentice-Hall, 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.

[AT92] Rajeev Alur and Gadi Taubenfeld. Results about fast mutual exclusion. In *IEEE Real-Time Systems Symposium*, pages 12–22, 1992.

[AV05] Maria V. Andreeva and Irina Virbitskaite. Timed equivalences for timed event structures. In Malyshkin [Mal05], pages 16–26.

[Bal95] Felice Balarin. *Iterative methods for formal verification of digital systems*. PhD thesis, Berkeley, CA, USA, 1995.

[Bal00] Paolo Baldan. *Modelling Concurrent Computations: from Contextual Petri Nets to Graph Grammars*. PhD thesis, University of Pisa, 2000.

[BAPM81] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. In *Eighth Annual Symposium on Principles of Programming Languages*, pages 164–176. ACM Press, 1981.

[BC94] Gérard Boudol and Ilaria Castellani. Flow models of distributed computations: three equivalent semantics for ccs. *Inf. Comput.*, 114(2):247–314, 1994.

[BCL+94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BCM98] Paolo Baldan, Andrea Corradini, and Ugo Montanari. An event structure semantics for P/T contextual nets: Asymmetric event structures. *Lecture Notes in Computer Science*, 1378:63–??, 1998.

[BCM01] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Contextual petri nets, asymmetric event structures, and processes. *Inf. Comput.*, 171(1):1–49, 2001.

[BD91] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[BGO+04] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The if toolset. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 237–267. Springer, 2004.

[BJMY02] M. Bozga, H. Jianmin, O. Maler, and S. Yovine. Verification of asynchronous circuits using timed automata. In *Workshop on Theory and Practice of Timed Systems*, 2002.

[BLL+95] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPAAL – a tool suite for automatic verification of real-time systems. In *Proccedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, LNCS. Springer-Verlag, 1995.

[BLN03] Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A tool for BDD-based verification of real-time systems. In W. A. Hunt and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, LNCS 2725, pages 122–125. Springer-Verlag, Berlin, 2003.

[BLP$^+$99] Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 341–353, London, UK, 1999. Springer-Verlag.

[BM00] O. Bournez and O. Maler. On the representation of timed polyhedra. In *Proceedings International Conference on Automata, Languages and Programming (ICALP)*, volume 1853 of *LNCS*, pages 793–807. Springer-Verlag, 2000.

[BMH99] W. Belluomini, C.J. Myers, and H.P. Hofstee. Verification of delayed-reset domino circuits using ATACS. In *Proceedings International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, 1999.

[BMPY97] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In O. Grumberg, editor, *Proceedings International Workshop on Computer Aided Verification*, volume 1254 of *LNCS*, pages 179–190. Springer-Verlag, 1997.

[BMT99] Marius Bozga, Oded Maler, and Stavros Tripakis. Efficient verification of timed automata using dense and discrete time semantics. In *CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 125–141, London, UK, 1999. Springer-Verlag.

[BN03] D. Beyer and A. Noack. Can decision diagrams overcome state space explosion in real-time verification? In *Proc. FORTE*, LNCS 2767, pages 193–208. Springer, 2003.

[BR03] S. Barner and I. Rabinovitz. Efficient symbolic model checking of software using partial disjunctive partitioning. In Subramanian Iyer, Debashis Sahoo, Christian Stangier, Amit Narayan, and Jawahar Jain, editors, *Proceedings of CHARME'03*, volume 2860 of *LNCS*, pages 35–50. Springer-Verlag, 2003.

[Bro90] F.M. Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Kluwer Academic Publishers, 1990.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[BS91] J. Brzozowski and C. Seger. Advances in asynchronous circuit theory part ii: Bounded inertial delay models, mos circuits, design techniques. *Bulletin of the European Association for Computer Science*, 43(3):199–263, 1991.

[BS95] J. Brzozowski and C. Seger. *Asynchronous Circuits*. Monographs in Computer Science. Springer-Verlag, 1995.

[BSV93] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. A verification strategy for timing-constrained systems. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 151–163, London, UK, 1993. Springer-Verlag.

[BSV95] F. Balarin and A.L. Sangiovanni-Vincentelli. An iterative approach to verification of real-time systems. *Formal Methods in System Design*, 6:67–95, January 1995.

[Bur89] J.R. Burch. Combining CTL, trace theory and timing models. In *Proceedings of the First Workshop on Automatic Verication Methods for Finite State Systems*, volume 407, pages 197–212. LNCS, 1989.

[Bur92] J.R. Burch. Delay models for verifying speed-dependent asynchronous circuits. In *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, March 1992.

[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[CC05] Robert Clarisó and Jordi Cortadella. Verification of concurrent systems with parametric delays using octahedra. In *Proc. 5th International Conference on Application of Concurrency to System Design (ACSD'05)*. IEEE Computer Society Press, June 2005.

[CDY99] S. Chakraborty, D.L. Dill, and K.Y. Yun. Min-max timing analysis and an application to asynchronous circuits. *Proceedings of the IEEE*, 87(2):332–346, February 1999.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CFT98] A. Caprara, M. Fischetti, and P. Toth. Algorithms for the set covering problem, 1998.

[CGP00] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 2000.

[CK98] P. Crescenzi and V. Kann. A compendium of np optimization problems, 1998.

[CKK+97] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.

[CKK+98] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Lazy transition systems: application to timing optimization of asynchronous circuits. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, November 1998.

[CKK+02] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis for Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002.

[CM90] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *ICCAD*, pages 126–129, 1990.

[CMB90] O. Coudert, J.C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. *Proceedings International Workshop on Computer Aided Verification*, pages 75–84, 1990.

[CY91] C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. In *Proceedings International Workshop on Computer Aided Verification*, volume 575 of *LNCS*, pages 399–409. Springer-Verlag, 1991.

[DD90] Philippe Darondeau and Pierpaolo Degano. Event structures, causal trees, and refinements. In Branislav Rovan, editor, *MFCS*, volume 452 of *Lecture Notes in Computer Science*, pages 239–245. Springer, 1990.

[DGV93] Pierpaolo Degano, Roberto Gorrieri, and Sebastiano Vigna. On relating some models for concurrency. In *TAPSOFT '93: Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 15–30. Springer-Verlag, 1993.

[DHWT91] D.L. Dill, A.J. Hu, and H. Wong-Toi. Checking for language inclusion using simulation relations. In *Proceedings International Workshop on Computer Aided Verification*, volume 575 of *LNCS*, pages 255–265. Springer-Verlag, 1991.

[Dil89] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer-Verlag, 1989.

[DKMW92] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Verification of asynchronous interface circuits with bounded wire delays. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 188–195, November 1992.

[Dub05] R. S. Dubtsov. Real-time event structures and scott domains. In Malyshkin [Mal05], pages 42–48.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[EG94] E.A. Walkup and G. Boriello. Interface Timing Verification with Application to Synthesis. In *31st ACM/IEEE Design Automation Conference (DAC)*, San Diego, CA, 1994. San Diego Convention Center.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leuven, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.

[EMSS90] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Proceedings International Workshop on Computer Aided Verification*, volume 531 of *LNCS*, pages 136–145. Springer-Verlag, 1990.

[God96] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[Gre93] M. R. Greenstreet. STARI: A technique for high-bandwidth communication. Technical Report TR-394-92, PhD Thesis, 1993.

[Gup92] A. Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in System Design*, 1:151–238, 1992.

[GV94] A. Puri A. Gollu and P. Varaiya. Discretization of timed automata. In *Proceedings of the 33rd IEEE conferene on decision and control*, pages 957–958, 1994.

[GW91] P. Godefroid and P. Wolper. A partial approach to model checking. In *Sixth Annual Symposium on Logic in Computer Science*, pages 406–415. IEEE Computer Society Press, 1991.

[HB94] H. Hulgaard and S.M. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *Proceedings International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11, November 1994.

[HBBM97] Youpyo Hong, Peter A. Beerel, Jerry R. Burch, and Kenneth L. McMillan. Safe BDD minimization using don't cares. In *Design Automation Conference*, pages 208–213, 1997.

[Hen90] T.A. Henzinger. Half-Order Modal Logic: How to Prove Real-Time Properties. In *Proceedings of the 9th Annual Symposium on Principles of Distributed Computing*, pages 281–296. ACM Press, 1990.

[HKT93] P. W. Hoogers, H. C. M. Kleijn, and P. S. Thiagarajan. Local event structures and petri nets. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 462–476. Springer-Verlag, 1993.

[HLP90] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit Clock Temporal Logic. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 402–413. IEEE Computer Society Press, 1990.

[HMP91] T.A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 353–366, 1991.

[HMP92a] T.A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *REX Workshop. Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 226–251. Springer-Verlag, 1992.

[HMP92b] T.A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proceedings International Conference on Automata, Languages and Programming (ICALP)*, volume 623 of *LNCS*, pages 545–558. Springer-Verlag, 1992.

[HNSY92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406. IEEE Computer Scienty Press, 1992.

[Hop71] John E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[HU79] John E. Hopcroft and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Adison-Wesley Publishing Company, Reading, Massachusets, USA, 1979.

[JM86] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transaction on Software Engineering*, 12(9):890–904, 1986.

[KBS02] H. Kim, P.A. Beerel, and K. Stevens. Relative timing based verification of timed circuits and systems. In *Proceedings International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 115–124, April 2002.

[KOY+01] Tomoya Kitai, Yusuke Oguro, Tomohiro Yoneda, Eric Mercer, and Chris Myers. Partial order reduction for timed circuit verification based on level oriented model. *IEICE transactions on information and systems*, 86(12):2601–2611, 20031201.

[Koy90] R. Koymans. Specifying real-time properties with metric temporal logic. In *LNCS*, volume 443, pages 255–299. Springer-Verlag, 1990.

[Kro99] T. Kropf. *Introduction to Formal Hardware Verification*. Springer-Verlag, 1999.

[Kur94] R.P. Kurshan. *Computer–Aided Verification of Coordinated Processes – An Automata Theoretic Approach*. Princeton University Press, 1994.

[KYM05] Tomoya Kitai, Tomohiro Yoneda, and Chris Myers. Failure trace analysis of timed circuits for automatic timing constraints derivation. *IEICE - Trans. Inf. Syst.*, E88-D(11):2555–2564, 2005.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[Lan92] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, Department of Computer Science, University of Twente, November 1992.

[Lan93] Rom Langerak. Bundle event structures: a non-interleaving semantics for lotos. In *FORTE '92: Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 331–346. North-Holland, 1993.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, July 1959.

[Les95] Ted Leslie. Efficient approaches to subset construction. Master's thesis, Computer Science, University of Waterloo, 1995.

[LLPY03] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Compact data structures and state-space reduction for model-checking real-time systems. *Real-Time Syst.*, 25(2-3):255–275, 2003.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *12th annual Symposium on Principles of Programming Languages*, pages 97–107. ACM Press, 1985.

[LPY95] K.G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 76–89, 1995.

[Mal05] Victor E. Malyshkin, editor. *Parallel Computing Technologies, 8th International Conference, PaCT 2005, Krasnoyarsk, Russia, September 5-9, 2005, Proceedings*, volume 3606 of *Lecture Notes in Computer Science*. Springer, 2005.

[Maz88] A. Mazurkiewicz. Basic notions of trace theory. In J. W. Baker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 285–363. Springer-Verlag, 1988.

[McM92] K. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings International Workshop on Computer Aided Verification*, volume 663 of *LNCS*, pages 164–177. Springer-Verlag, 1992.

[MD92] K.L. McMillan and D.L. Dill. Algorithms for interface verification. In *Proceedings of the IEEE International Conference on Computer Design*, October 1992.

[MP95a] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In P.E. Camurati and H. Eveking, editors, *Proceedings of CHARME'95*, volume 987 of *LNCS*, pages 189–205. Springer-Verlag, 1995.

[MP95b] Z. Manna and A. Pnuelli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[MP96] Z. Manna and A. Pnueli. Clocked transition systems. Technical Report STAN-CS-TR-96-1566, Computer Science Department, Stanford University, April 1996.

[MV01] Nataly S. Moskaljova and Irina Virbitskaite. On the category of event structures with dense time. In Rusins Freivalds, editor, *FCT*, volume 2138 of *Lecture Notes in Computer Science*, pages 287–298. Springer, 2001.

[MVF00] J. Muttersbach, T. Villigers, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proceedings International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 52–59, 2000.

[NMA+02] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, N. Jain, and O. Maler. Verification of timed automata via satisfiability checking. In *Proc. Formal Techniques in Real-Time and FaultTolerant Systems FTRTFT'02, volume 2469 of Lecture Notes in Computer Science*, pages 225–244, 2002.

[NMY03] Curtis A. Nelson, Chris J. Myers, and Tomohiro Yoneda. Efficient verification of hazard-freedom in gate-level timed asynchronous circuits. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 424, Washington, DC, USA, 2003. IEEE Computer Society.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains. *Theoretical Computer Science*, 13:85–108, 1981.

[NRT92] M. Nielsen, G. Rozenberg, and P. S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96(1):3–33, 1992.

[OCVSV98] Arlindo L. Oliveira, Luca P. Carloni, Tiziano Villa, and Alberto L. Sangiovanni-Vincentelli. Exact minimization of binary decision diagrams using implicit techniques. *IEEE Transactions on Computers*, 47(11):1282–1296, 1998.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[Ost90] J.S. Ostroff. *Temporal Logic for Real-Time Systems*. Research Studies Press / Wiley, 1990.

[PCKP00] M.A. Peña, J. Cortadella, A. Kondratyev, and E. Pastor. Formal verification of safety properties in timed circuits. In *Proceedings International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11, Eilat, Israel, April 2000.

[Peñ03] M.A. Peña. *Relative Timing-Based Formal Verification of Complex Timed Systems*. PhD thesis, Departament of Computer Architecture, Technical University of Catalonia, Barcelona (Spain), 2003.

[Pet62] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn, Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 3, 1962. English translation, "Communication with Automata", Griffiss Air Force Base, Technical Report RADC-TR-65-377, vol. 1, Suppl. 1, 1966.

[PH88] A. Pnueli and E. Harel. Applications of Temporal Logic to the Specification of Real Time Systems. In M. Joseph, editor, *Formal Techniques in Real Time and Fault Tolerant Systems*, volume 331 of *LNCS*, pages 84–98. Springer-Verlag, 1988.

[Plo96] A. Plotnikov. Polynomial-time partition of a graph into cliques, 1996.

[Plo00] Anatoly Plotnikov. An efficient algorithm for the minimum clique partition problem, 2000.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[PP95] G. Michele Pinna and Axel Poigné. On the nature of events: another perspective in concurrency. In *MFPS '92: Selected papers of the meeting on Mathematical foundations of programming semantics*, pages 425–454. Elsevier Science Publishers B. V., 1995.

[PP03] E. Pastor and M.A. Peña. Efficient hybrid reachability analysis for asynchronous concurrent systems. Technical Report RR-2003/6, DAC/UPC, January 2003.

[PS07a] Esteban Pavese and Fernando Schapachnik. Relaxed clock difference diagrams for timed automata model checking. Technical report, Depto. de Computación, FCEyN, UBA, nov 2007.

[PS07b] Wojciech Penczek and Maciej Szreter. Sat-based unbounded model checking of timed automata. In *ACSD '07: Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, pages 236–237, Washington, DC, USA, 2007. IEEE Computer Society.

[RCP95] O. Roig, J. Cortadella, and E. Pastor. Hierarchical gate-level verification of speed-independent circuits. In *Asynchronous Design Methodologies*, pages 129–137. IEEE Computer Society Press, May 1995.

[RE88] G. Rozenberg and J. Engelfriet. Elementary net systems. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models. Advances in Petri Nets 1988*, volume 1491 of *LNCS*, pages 12–121, 1988.

[RM94] T.G. Rokicki and C.J. Myers. Automatic verification of timed circuits. In David L. Dill, editor, *Proceedings International Workshop on Computer Aided Verification*, volume 818 of *LNCS*, pages 468–480. Springer-Verlag, 1994.

[RSG$^+$99] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: An asynchronous instruction length decoder. In *Proceedings International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70, April 1999.

[S.B78] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6), June 1978.

[SB97] R.H. Sloan and U. Buy. Stubborn sets for real-time Petri nets. *Formal Methods in System Design*, 11(1):23–40, July 1997.

[SB03] Sanjit A. Seshia and Randal E. Bryant. Unbounded, fully symbolic model checking of timed automata using boolean methods. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 154–166. Springer, 2003.

[SGR99] K. Stevens, R. Ginosar, and S. Rotem. Relative timing. In *Proceedings International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, April 1999.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[SP04a] M. Sole and E. Pastor. Evaluating symbolic traversal algorithms applied to asynchronous concurrent systems. Technical Report RR-2004/07, UPC/DAC, January 2004.

[SP04b] Marc Solé and Enric Pastor. Evaluating symbolic traversal algorithms applied to asynchronous concurrent systems. Technical Report RR-2004/08, UPC/DAC, January 2004.

[SRC$^+$00] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous Interlocked Pipelined CMOS Circuits Operating at $3.3 - 4.5$GHz. In *IEEE International Solid-State Circuits Conference*, pages 292–293, 2000.

[Str98] Karsten Strehl. Using interval diagram techniques for the symbolic verification of timed automata. Technical Report 53, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, July 1998.

[SW96] Sauerhoff and Wegener. On the complexity of minimizing the OBDD size for incompletely specified functions. *IEEETCAD: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15, 1996.

[SY96] A. Semenov and A. Yakovlev. Verification of asynchronous circuits using time Petri-net unfolding. In *Proceedings ACM/IEEE Design Automation Conference*, 1996.

[Tho81] W. Thomas. A combinatorial approach to the theory of ω-automata. *Information and Computation*, 48:261–283, 1981.

[TKY$^+$98] S. Taşiran, S. Khatri, S. Yovine, R.K. Brayton, and A. Sangiovanni-Vincentelli. A timed-automaton-based method for accurate computation of circuit delay in the presence of cross-talk. In *Proceedings of Formal Methods in Computer-Aided Design*, LNCS. Springer-Verlag, 1998.

[Tod93] Gaetano Borriello Tod Amon, Henrik Hulgaard, Steven M. Burns. Algorithm for exact bounds on the time separation of events in concurrent systems. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 166–173, 1993.

[VdJL96] E. Verlind, G. de Jong, and B. Lin. Efficient partial enumeration for timing analysis of asynchronous systems. In *Proceedings ACM/IEEE Design Automation Conference*, 1996.

[VG04] Irina Virbitskaite and Natalya S. Gribovskaja. Open maps and observational equivalences for timed partial order models. *Fundam. Inform.*, 60(1-4):383–399, 2004.

[vGP04] Rob J. van Glabbeek and Gordon D. Plotkin. Event structures for resolvable conflict. In Jirí Fiala, Václav Koubek, and Jan Kratochvíl, editors, *MFCS*, volume 3153 of *Lecture Notes in Computer Science*, pages 550–561. Springer, 2004.

[Wal95] Elizabeth A. Walkup. *Optimization of Linear Max-Plus Systems with Application to Timing Analysis*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, 1995.

[Wan01] Farn Wang. Red: Model-checker for timed automata with clock-restriction diagram. Technical Report 2001-014, Dept. of Information Technology, Uppsala University, Sweden, August 2001.

[Wan04] Farn Wang. Efficient verification of timed automata with bdd-like data structures. *Int. J. Softw. Tools Technol. Transf.*, 6(1):77–97, 2004.

[Wat93] Bruce W. Watson. A taxonomy of finite automata construction algorithms. Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993.

[Win87] Glynn Winskel. Event structures. In *Proceedings of an Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986-Part II*, pages 325–392. Springer-Verlag, 1987.

[WT94] Howard Wong-Toi. Symbolic approximations for verifying real-time systems. Technical report, Stanford, CA, USA, 1994.

[WZP03] Bozena Woźna, Andrzej Zbrzezny, and Wojciech Penczek. Checking reachability properties for timed automata via sat. *Fundam. Inf.*, 55(2):223–241, 2003.

[YD99] K. Y. Yun and A. E. Dooply. Pausible clocking based heterogeneous systems. *IEEE Transactions on VLSI Systems*, 7(4):482–487, 1999.

[Yov97] S. Yovine. KRONOS: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.

[YR99] Tomohiro Yoneda and Hiroshi Ryu. Timed trace theoretic verification using partial order reduction. In *ASYNC '99: Proceedings of the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 108, Washington, DC, USA, 1999. IEEE Computer Society.

[YSSC93] T. Yoneda, A. Shibayama, B. Schlingloff, and E.M. Clarke. Efficient verification of parallel real-time systems. In *Proceedings International Workshop on Computer Aided Verification*, LNCS, pages 321–332. Springer-Verlag, 1993.

[YWCI94] Ti-Yen Yen, Wayne Wolf, Al Casavant, and Alex Ishii. Efficient algorithms for interface timing verification. In *Proceedings of the conference on European design automation*, pages 34–39. IEEE Computer Society Press, 1994.

[ZMM03] Hao Zheng, Eric Mercer, and Chris J. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(9):1138–1153, 2003.

[Zub80] W. M. Zuberek. Timed petri nets and preliminary performance evaluation. In *Proc. of the 7th Annual Symposium on Comp. Architecture, May 6–8, 1980, La Baule, France*, pages 88–96, 1980. NewsletterInfo: 7.