5

EXPERIMENTAL RESULTS

Now microscopic pulses would be bouncing through the complex circuitry of the unit, probing for possible failures, testing the myriads of components to see that they all lay within their specific tolerances. This had been done, of course, a score of time before the unit had ever left the factory; but that was two years ago, and more than a half a billion miles away. It was often impossible to see how solid-state electronic components could fail; yet they did.

"Circuit fully operational", reported HAL after only ten seconds. In that time, he carried out as many tests as a small army of human inspectors.

—Arthur C. Clarke - 2001. A Space Odyssey, 1968

Summary

This chapter briefly introduces TRANSYT, the CAD/CAV tool which incorporates the implementation of the verification methodology presented in Chapter 4. The presentation is carried out through a number of experiments that illustrate the applicability of the approach and the basic capabilities of the tool.

In order to introduce the notation and basic notions of the symbolic analysis of a system, details on the mapping of transition systems onto boolean algebras are provided. Additionally, the input format of the tool is briefly introduced in order to ease the reading of the chapter.

The experiments start with a small example that illustrates the need for forward unfolding of the state space of a system in order to achieve the timing analysis required to prove or disprove a given property. Next, the verification of quasi-speed-independent asynchronous circuits in which complex-gate decompositions have been performed, is illustrated. Both a small example and a complete set of benchmarks are analyzed. Finally, the verification of relative timing assumptions in timed asynchronous circuits is illustrated by analyzing a bus controller.

Along the chapter, the capabilities of TRANSYT in order to model timed PNs, timed STGs and digital circuits in terms of binary-encoded TTSs are also illustrated.

5.1 A brief introduction to TRANSYT

The verification methodology presented in Chapter 4 has been fully integrated into a CAD/CAV experimental tool called TRANSYT. In short, the tool uses conventional symbolic BDD-based [Bry86] techniques for state representation and reachability analysis combined with the relative timing-based approach for verification.

TRANSYT can handle systems modeled by means of transition systems, being them untimed (TS), timed (TTS) or lazy (LzTS) transition systems. The systems are specified with the native format of the tool, called `tsif` [PPa]. The tool also allows the specification and manipulation of complex systems by using modularity and hierarchy constructs incorporated to the basic model for transition systems. Synchronization and variable sharing mechanisms are also provided to support the communication between modules.

Apart of transition systems, TRANSYT can also handle timed PNs and timed STGs specified with the `astg` format [CKK+97], and digital circuits specified with the `blif` format [SSL+92]. These types of systems are automatically translated by the tool into equivalent transition systems for internal manipulation.

The user interface of TRANSYT is based on a console-type interactive shell. The different commands can be typed-in by the user or read from command files. The tool can run also in non-interactive mode by providing a command file when the tool is launched. The analysis of the systems can be done using both textual and graphical interfaces, including the specification of the system and the outputs produced by certain commands.

Despite of the relative timing-based verification functionality that we illustrate in this and the next chapter, TRANSYT also provides a number of other features. For example, different algorithms for complete or partial state space traversal are provided, as well as fast symbolic simulation, bug-hunting and guided-search algorithms [GA98, YD98]. Currently, the reachability analysis engine is capable of computing both the untimed state space and the timed state space of a system under the relative timing paradigm. In both cases, symbolic techniques that rely on BDDs are used for efficiency. In the exact timed state space of a system, states must be labeled with integer or real values (see Chapter 3) that capture explicitly the precise instants at which the states are visited. Hence, symbolic techniques based on BDDs cannot be easily applied. Although the exact time state space of a system cannot be computed with TRANSYT, it is not required to support our relative timing-based verification approach. Nevertheless, we plan to incorporate representation mechanisms and traversal algorithms for exact timed reachability in the future.

Some of the features of TRANSYT are illustrated in the following sections, however the full set of features provided by the tool is beyond the scope of this thesis. The reader is referred to [PPb] for more details on the functionalities of TRANSYT.

To conclude this brief presentation of the tool, just say that the resulting tool suite is composed of about 84000 lines of ANSI C code, not counting the BDD package and

other libraries. Around 34000 lines of code correspond to the implementation of the verification methodology described in Chapter 4. Currently, the tool is only available for 32-bit Unix/Linux systems, although it could be ported to other Unix/Linux or Windows systems without too much effort.

The following section introduces some basic notions on boolean algebras and how they can be used to model transition systems. This introduction is completed with a brief review of the modeling capabilities of the `tsif` format used in TRANSYT.

5.1.1 Representation of LzTSs with boolean algebras

In order 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. Such functions are represented in TRANSYT using BDDs [Bry86].

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 (often called the *switching 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 with BDDs. 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, which shows that a boolean function can be represented in terms of its cofactors. That is:

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

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:

$$\exists_{v_i} f = f_{v_i} + f_{\overline{v_i}} \qquad and \qquad \forall_{v_i} f = f_{v_i} \cdot f_{\overline{v_i}} \;\; .$$

In order 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.

On the other hand, it 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.

The interested reader is referred to [Bro90] for a complete introduction to boolean algebras.

Representation of LzTSs

Let $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathsf{EnR} \rangle$ be a LzTS. According to the above discussion, 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.

Each state $\mathsf{s} \in S$ can be represented by means of an *encoding function* $\mathcal{Q} : S \rightarrow \mathbb{B}^n$, such that $n \geq \lceil log_2(\mid S \mid) \rceil$. 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, characteristic functions provide, in general, compact and efficient representations.

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 $A = \langle S, \Sigma, T, \mathsf{s}_0, \mathsf{EnR} \rangle$. In such a way that v'_i is the next-state variable corresponding to the current-state variable v_i , and vice versa. Thus, the usual definition of LzTS can be extended to contain $\mathcal{V}$ and $\mathcal{V}'$, *i.e.* $A = \langle \mathcal{V}, \mathcal{V}', S, \Sigma, T, \mathsf{s}_0, \mathsf{EnR} \rangle$. Now, given an event $\mathsf{e} \in \Sigma$ we can represent its enabling region, its firing region and its transitions relation, by means of the following characteristic functions:

- $EF(\mathsf{e}) : \mathbb{B}^n \rightarrow \mathbb{B}$ such that $EF(\mathsf{e}) = 1$ for all the states (encoded using $\mathcal{V}$) belonging to the enabling region of e , *i.e.* $\mathsf{EnR}(\mathsf{e})$.

- $FF(\mathsf{e}) : \mathbb{B}^n \rightarrow \mathbb{B}$ such that $FF(\mathsf{e}) = 1$ for all the states (encoded using $\mathcal{V}$) belonging to the firing region of e , *i.e.* $\mathsf{FR}(\mathsf{e})$.

- $TR(\mathsf{e}) : \mathbb{B}^{2n} \rightarrow \mathbb{B}$ such that $TR(\mathsf{e}) = 1$ for all the relations $(\mathsf{s}_1, \mathsf{s}_2)$ such that there is a transition of event e , $\mathsf{s}_1 \xrightarrow{\mathsf{e}} \mathsf{s}_2 \in T$. 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 will write $EF'(\mathsf{e})$ and $FF'(\mathsf{e})$, respectively. Also, when the sets of variables in a transition relation are interchanged we will write $TR(\mathsf{e})^{-1}$.

Figure 5.1 (a) shows a simple LzTS. The states of the system can be encoded using at least three (current-state) boolean variables, *i.e.* $\mathcal{V} = \{v_1, v_2, v_3\}$. Figure 5.1 (b) shows the same LzTS but encoded using an arbitrary binary encoding of the states. Thus, given a set of states $P = \{\mathsf{s}_0, \mathsf{s}_1, \mathsf{s}_2\}$, its characteristic function will be: $\mathcal{X}_P = \overline{v_0}\ \overline{v_1}\ \overline{v_2} + \overline{v_0}\ v_1\ \overline{v_2} + \overline{v_0}\ v_1\ v_2 = \overline{v_0}\ (v_1 + \overline{v_2})$. Similarly, $EF(\mathsf{c}) = v_2\ \overline{v_3}$ and $FF(\mathsf{c}) = \overline{v_1}\ v_2\ \overline{v_3}$.

In order to encode the transition relations, corresponding next-state variables, $\mathcal{V}' = \{v'_1, v'_2, v'_3\}$, are also required. Thus, with the given encoding we have that $TR(\mathsf{b}) = \overline{v_1}\ \overline{v_2}\ \overline{v_3}\ \overline{v'_1}\ v'_2\ \overline{v'_3} + v_1\ \overline{v_2}\ \overline{v_3}\ v'_1\ v'_2\ \overline{v'_3}$.

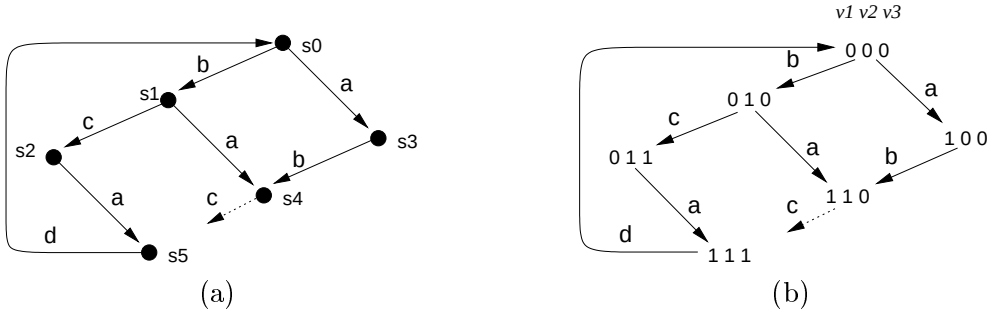


Figure 5.1 A simple LzTS (a) and the same LzTS but binary-encoded (b).

Finding the set of states, $P_2 \in S$, that can be reached after the firing of a given event, $\mathsf{e} \in \Sigma$, from any of the states in another set, $P_1 \in S$, is reduced to compute:

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

Thus, in the example, in order to compute the states reached after firing event b from state s_0 we just have: $\exists_{v_1,v_2,v_3} \; (\; TR(\mathsf{b}) \cdot \mathcal{X}_{\{\mathsf{s}_0\}} \;) = \exists_{v_1,v_2,v_3} \; (\; \overline{v_1} \; \overline{v_2} \; \overline{v_3} \; \overline{v'_1} \; v'_2 \; \overline{v'_3} \;) \; = \overline{v'_1} \; v'_2 \; \overline{v'_3}$. As expected, this corresponds to state s_1 but encoded using next-state variables.

5.1.2 TRANSYT input format

The TRANSYT input format, called `tsif` (see [PPa] for a complete description), is intended to be a basic and simple low-level format used to describe most types of transitions systems. The basic model is augmented to accept modular and hierarchical systems that are coordinated by some synchronization mechanism. The behavior of each coordinated subsystem is defined by means of a detailed specification of its events.

Transition systems are often used to model highly concurrent systems which suffer of the well-known state explosion problem. Clearly, in those cases it is not viable an explicit enumerative declaration of all the states of the system, thus some form of symbolic modeling mechanism must be used. The `tsif` format requires the transition system to be modeled by a boolean algebra, as described in the previous section. Each individual state is assigned a unique binary code in the algebra, but no explicit representation of the relation between the states and their encoding is required. Thus, the states of the system can be represented by means of boolean characteristic functions, and the transition relations of the events can be described by means logic relations.

Flat systems

The basic elements required for the description of a flat transition system (*i.e.* without hierarchy) are the following:

- A set of boolean variables used to describe the state of the system and its environment (if any).
- A set of *labels* that capture the different operations that can be performed by the system (*e.g.* signals in a digital circuit).
- A set of *events* for each label. The transition relation of each event is specified by means of a boolean relation that describes how the current state of the system is modified into the next state each time the event is executed.
- The initial state of the system. Since it is given in terms of a boolean equation, it is not restricted to a single state.

Variables. There exist three types of variables: input, output and internal variables. Internal and output variables are used to describe the internal and the visible behavior of the system, respectively. Whereas input variables are used to describe the state of the environment. The variable declaration consists of three elements: the variable type, followed by the `VARS` keyword and by the list of declared variables ended by a semicolon. Namely:

```
{INPUT|OUTPUT|INTERNAL} VARS <list_of_variables> ;
```

Besides the current-state variables, TRANSYT requires an associated set of next-state variables in order to specify, for example, the transition relations. Next-state variables can be either specified explicitly in the `tsif` format or leave TRANSYT create them internally. User defined next-state variables can have any desired name. However, internally created next-state variables will share the name of the corresponding current-state variable but with the `NS` operator. For example, the current-state variable `var1` will be assigned a next-state variable named `NS(var1)` .

Labels. There exist four different types of *labels*: input, output, internal and dummy. Input labels correspond to operations executed by the environment. Output (internal) labels correspond to operations executed by the system and that can (cannot) be observed by the environment. Dummy labels correspond to instantaneous (zero delay) operations and are provided to ease the modeling of certain complex systems. The label declaration consists of three elements: the variable type, followed by a `LABELS` keyword and by the list of declared labels ended by a semicolon. Namely:

```
{INPUT|OUTPUT|INTERNAL|DUMMY} LABELS <list_of_labels> ;
```

Events. Since the same operation (label) may need to be executed in quite different circumstances, a second level of detail is provided by means of a set of *events* associated to each label. For example, a label could model a signal of a circuit, whereas the events of the label could model the different signal switches.

```
TS example INTERLEAVED

INTERNAL VARS v1 v2 v3;
INTERNAL LABELS a b c d;

EVENT a a
EQN TR v1' NS(v1) (v2 = NS(v2)) (v3 = NS(v3));
END

EVENT b b
EQN TR (v1 = NS(v1)) v2' NS(v2) v3' NS(v3)';
END

EVENT c c
EQN TR (v1 = NS(v1)) v2 NS(v2) v3' NS(v3);
EQN FF v1' v2 v3';
END

EVENT d d
EQN TR v1 v2 v3 NS(v1)' NS(v2)' NS(v3)';
END

EQN ISTATE v1' v2' v3';

END
```

Figure 5.2 TRANSYT input file for the LzTS of Figure 5.1.

An event declaration consists of five elements: the `EVENT` keyword is followed by the name of the event and the name of the label to which it is associated; then a number of boolean equations (`EQN` keyword) can be specified to declare the transition relation of the event (`TR` keyword), failure conditions (`FAIL` keyword), etc. The equations are expressed as a list of logic functions, each one ended with a semi-colon. Each equation accepts logic operators such as negation (`!` or `'`), addition (`+`), product (`*`, or simply by joining terms), equivalence (`=`) or difference (`<>`), as well as parenthesis. The list of equations ends with the `END` keyword. Namely:

```
EVENT <event_name> <label_name>
<list_of_equations>
END
```

Finally, a set of different equations can be specified for the transition system, such as failure conditions (`FAIL` keyword) or the initial states of the system (`ISTATE` keyword). The equation for the set of initial states is mandatory.

Figure 5.2 shows the `tsif` specification of the binary-encoded LzTS of Figure 5.1 (b). The explanation about the `FF` equation associated to event `c` can be found in the following section devoted to timed systems.

Finally, remark that the basic TS model is augmented in TRANSYT to allow the description of complex systems as hierarchical structures of coordinated subsystems. Two general mechanisms are provided to implement the inter-system communication:

- Synchronized execution of events with common labels between multiple systems, but without any data exchange (pure *rendez vous*).
- Sharing of boolean variables between systems, but without synchronization.

All the examples that follow in the chapter are modeled as flat transition systems. Hence we do not enter into the details of the specification of hierarchical modular systems. The interested reader is referred to [PPa].

Timed systems

The `tsif` format allows to specify both absolute and relative timing information. Absolute timing information is incorporated by specifying a delay interval for each event of the system. Relative timing information is incorporated by explicitly distinguishing between the enabling and the firing of the events.

The TTS formalism is supported by allowing the specification of a minimum and maximum delay values (or just a typical fixed delay) to each event in the system. Specifying a typical delay implies that the maximum and minimum delays are equal. A maximum delay with value zero implies a minimum delay with the same value. If no delay information is provided for an event, it is assumed to have unbounded delays. That is, the minimum delay is zero and the maximum delay is unbounded (but finite). Absolute delay values must be specified inside the scope of the `EVENT` declaration. The specification of delay information must be in the following format:

```
{ DELAY: [MAX=<max_delay>; MIN=<min_delay>; TYP=<typ_delay>;] }
```

Examples of `tsif` files where delay information is specified can be found in Sections 5.2, 5.3 and 5.4.

The LzTS formalism is supported by allowing the specification of characteristic equations for the enabling and firing functions of an event. Both the enabling (`EF` keyword) and the firing (`FF` keyword) functions must be specified inside the scope of the `EVENT` declaration. If no `EF` is specified for an event, it is automatically extracted from the transition relation of the event. If no `FF` is specified, it is assumed to be equal to its `EF`.

See the specification of the firing function of event `c` in the `tsif` file of Figure 5.2.

Properties

TRANSYT supports the automatic verification of safety properties specified by the characteristic function of the corresponding failure condition. Each property is specified as a boolean proposition that can pose conditions on the value of the current-state variables as well as the next-state variables. Failure conditions that only depend on current-state variables are called *state* conditions because they define properties on the reachable states of the system. Failure conditions that depend on a mixture of current and next-state vari-

ables are called *transition* conditions, because they define conditions on potential transitions from reachable states of the system. To tool automatically detects the the type of the failure condition by inspecting the boolean equation that defines it.

Failure conditions can be specified with equations in the `tsif` file or directly provided from the TRANSYT command-shell (`add_fail` command). When specified in the `tsif` file the `FAIL` keyword is used. Examples of both cases are shown in the following sections.

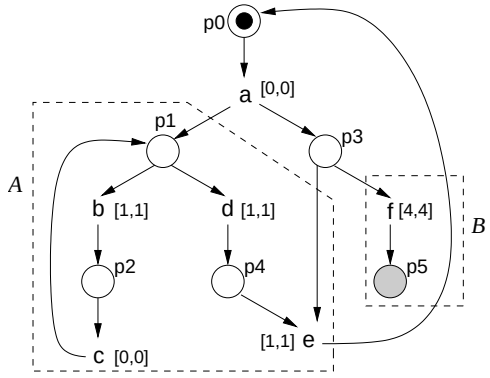
A failure condition can be associated to different objects in a transition system: the system itself, a label or an event. The semantics of the condition depends on the type of the object to which it is associated. In a case of a TS, a state condition can characterize failures at any reachable state of the system, whereas a transition condition can characterize failures due to the enabling of a transition at any reachable state of the system. Failure conditions for a TS can be specified at any point between the `TS` and the last `END` keywords. State conditions associated to labels (or events) can characterize failures at any reachable state in which the label (or the event) is enabled. Transition conditions associated to labels (or events) can characterize failure situations caused by a firable transition of the label (or the event) from any reachable state of the system. Failure conditions for events can be specified at any point inside the `EVENT` scope. Failure conditions for labels can only be specified at the label declaration.

Finally, remark that TRANSYT also supports several built-in failure conditions for commonly used properties in the analysis of concurrent systems. Using built-in conditions avoids the explicit specification of characteristic functions for the properties. Since this type of conditions are not used in the subsequent sections, we skip the details here and refer the reader to [PPa] for details.

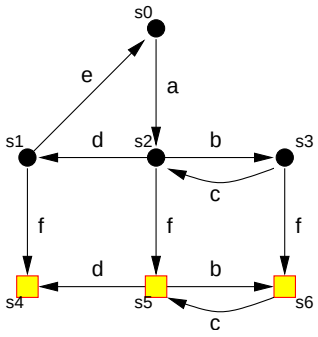
The remaining contents of this chapter are organized as follows. Section 5.2 develops a small example that illustrates the need for forward unfolding of the state space of a system in order to achieve the timing analysis required to prove or disprove a given property. The section also illustrates the capabilities of TRANSYT in order to model a timed PN as a binary-encoded TTS. Section 5.3 describes the details of the verification of quasi-speed-independent asynchronous circuits in which complex-gate decompositions have been performed. The section also illustrates the way a timed STG and a digital circuit are handled in TRANSYT, and how certain crucial properties for verification are modeled. Finally, Section 5.4 illustrates the use of our verification methodology for the verification of relative timing assumptions in timed asynchronous circuits.

5.2 An example with forward unfolding

Section 4.6 dealt with the convergence issues of the proposed verification methodology. Such issues arise from the fact that the refinement procedure performs an unfolding of the state space in order to separate those traces which are enabling-compatible with the timing



(a)



(b)

```
#Yoneda's example
TS yoneda INTERLEAVED

INTERNAL VARS p0 p1 p3 p2 p4 p5;
INTERNAL LABELS a b c d e f;

EVENT a a
EQN TR p0 NS(p0)' p1' NS(p1) p3' NS(p3);
{DELAY: [TYP=0;]}
END

#Fair process A: events b, c, d and e
EVENT b b
EQN TR p1 NS(p1)' p2' NS(p2);
{DELAY: [TYP=1;]}
END

EVENT c c
EQN TR p1' NS(p1) p2 NS(p2)';
{DELAY: [TYP=0;]}
END

EVENT d d
EQN TR p1 NS(p1)' p4' NS(p4);
{DELAY: [TYP=1;]}
END

EVENT e e
EQN TR p0' NS(p0) p3 NS(p3)' p4 NS(p4)';
{DELAY: [TYP=1;]}
END

#Unfair process B: event f
EVENT f f
EQN TR p3 NS(p3)' p5' NS(p5);
{DELAY: [TYP=4;]}
END

#Initial state: "only p0 is marked"
EQN ISTATE p0 p1' p3' p2' p4' p5';

#Failure condition: "p5 is marked"
EQN FAIL p5;

END
```

(c)

Figure 5.3 *Yoneda*'s example: (a) timed PN, (b) untimed state space and (c) TRANSYT input file.

analysis, and those traces which are not. However, that is not the only source of unfolding of the state space. As was discussed in Section 4.6, in order to perform an accurate-enough timing analysis, the critical cycles of the state space involved in the analysis might need to be *unrolled*. Moreover, depending on the characteristics of the system, the number of unrollings could be very high. In our methodology, such unrollings require a series of *forward unfoldings* of certain regions of the state space, which may drastically affect the efficiency of the verification. Although a pathological example that requires an enormous number of forward unfoldings in order to prove or disprove a given property, can be easily generated by hand, we believe that they are not likely to appear in practice. The claim is supported by the fact that none of the circuits verified in this and the next chapter show such a pathological behavior.

This section develops an example that illustrates the above ideas, *i.e.* the need for forward unfolding of the state space of a system in order to achieve the timing analysis

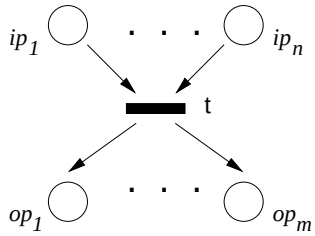


Figure 5.4 Transition of an PN with its input and output places.

required to prove or disprove certain property. The example is due to Tomohiro Yoneda, who suggested it during some discussions we had at the *6th International Conference on Advanced Research in Asynchronous Circuits and Systems (ASYNC'2000)*. The example models, in a simplified way, the behavior of a system where two processes, say A and B, compete for a shared resource. If process A takes the resource, it is fair and releases it after some bounded amount of time. However, process B retains the resource forever, making the resource no longer available for process A. The correct behavior of the system depends on the timing properties of processes A and B.

The system was originally modeled by means of the timed Petri net of Figure 5.3 (a). The shared resource is represented by place p_3. The unfair process B is modeled by transition f, so that if the resource is taken, place p_5 gets marked forever. The fair process A is modeled by transitions b, c, d and e, being e the transition that represents the allocation of the shared resource for process A. Finally, fixed delays are associated to the transitions.

5.2.1 Model of a timed PN

Although TRANSYT supports a timed PN as the input specification of a system [1] and translates it into a TTS, we show here how to model timed PN directly using TRANSYT native format.

In order to model a timed safe PN using TRANSYT input format, the state of the corresponding TS can be specified using one boolean state variable for each place of the PN in a similar way as in [PRCB94]. The variable is set when the place holds a token and reset otherwise. Thus, for every transition of the PN, a transition relation in the corresponding TS must be specified, in which the variables for the input (output) places of the PN transition are set (reset) in order for the transition to become enabled, and the variables become reset (set) after the transition is executed. To illustrate this idea, Figure 5.4 depicts a portion of a PN where a transition, t, and all its input ($ip_1, \ldots, ip_n$) and

[1] The `read_pn` command of TRANSYT reads PNs and STGs specified using the `astg` format of PETRIFY [CKK+97].

output $(op_1, \ldots, op_m)$ places are shown. Hence, for transition t, the following transition relation will be specified:

$$TR(\mathsf{t}) = \prod_{j=1}^{n} ip_j \cdot \prod_{j=1}^{m} \overline{op_j} \cdot \prod_{j=1}^{n} \overline{NS(ip_j)} \cdot \prod_{j=1}^{m} NS(op_j)$$

where $NS(\mathsf{x})$ represents the value of variable x in the next state of the system, after the transition relation has been executed. Thus, for example, transition a of the PN in Figure 5.3 (a) will be modeled by the following equation:

$$TR(\mathsf{a}) = p_0 \cdot \overline{p_1} \cdot \overline{p_3} \cdot \overline{NS(p_0)} \cdot NS(p_1) \cdot NS(p_3)$$

Similarly, the initial state of the TS, corresponding to the initial marking of the PN, is also specified by a boolean equation. In the equation, the variables for the marked places are set, whereas the variables for the unmarked places are reset. Thus, the initial marking of the PN in Figure 5.3 (a) has a corresponding initial state as: $p_0 \cdot \overline{p_1} \cdot \overline{p_2} \cdot \overline{p_3} \cdot \overline{p_4} \cdot \overline{p_5}$.

Notice that the encoding of a PN using boolean variables as described above, assumes the PN to be safe, *i.e.* each place can hold at most one token at any state of the system. Although the details are beyond the scope of this thesis, say that TRANSYT also allows non-safe PNs as specifications, which are automatically translated into TSs using appropriate encoding mechanisms (see [PCP99] for details).

With all the above considerations, Figure 5.3 (c) shows the `tsif` file (`yoneda.ts`) corresponding to the PN of Figure 5.3 (a). Since no communication with other systems is required, only internal variables and labels are declared. The set of variables is used to specify the boolean equations that define the behavior of the system. Such behavior is specified by means of a transition relation for each individual transition of the PN. In order to keep things separated, one internal label is declared for each transition of the PN and one event containing the actual transition relation is defined for each label. Also, each event specifies the delay bounds associated to the corresponding transition in the timed PN. In this case, a fixed *typical* delay is specified. The file concludes with the specification of the initial values of all the state variables, and an equation that specifies the failure condition. The result is a simple state condition which is activated whenever place p_5 gets marked.

What follows is the result of starting a session with TRANSYT for the verification of the system. First, the `tsif` file is read into the tool (`read_ts` command). Then, all the reachable states are computed (`traverse` command) and those states which satisfy the given failure condition are annotated. Finally, the `print_fails` command prints the fail conditions and the actual failure states (`-s` flag) detected. The three cubes shown correspond to failure states s_5, s_6 and s_4 of Figure 5.3 (b), respectively. Recall that in TRANSYT, the negation operator is indicated by a *prime* symbol.

```
$ transyt

TRANSYT 1.6.3 (compiled mon may  6 16:17:42 CEST 2002 on linux) running at minkar
By E.Pastor (enric@ac.upc.es) and M.A.Penya (marcoa@ac.upc.es)
Dept. of Computer Architecture (UPC)
Copyright (c)1998-2002 Universitat Politecnica de Catalunya
Welcome to the interactive version.

ts > read_ts yoneda.ts

ts:: Opening TS file 'yoneda.ts'.
ts:: Transition System 'yoneda' successfully read.

ts > traverse

ts:: Traversing system 'yoneda' using atom-partitioned TR.
ts:: End of Traversal with depth : 3
ts:: Final reached states: 7 Fail states: 3
ts:: Number of TR applications: 24 of which 10 useful
ts:: Time =     0.00 sec for the fix-point computation.
ts:: Time =     0.00 sec for the traverse.

ts > print_fails -s

ts:: Fail conditions for TS 'yoneda'.
ts:: Condition #0 defined (on states) in the TSI model
ts:: with equation:
ts:: EQN FAIL p5;
ts:: Detected #3 failure states
ts:: p0' p1 p3' p2' p4' p5 + p0' p1' p3' p2 p4' p5 + p0' p1' p3' p2' p4 p5
```

For illustrative purposes, the reachability analysis and the set of failure states has been computed before starting the verification process. Actually, this step can be avoided since the verification algorithms can perform partial state space analysis in order to discover failures, prove if they exist in the timed domain or not, and incrementally refine the state space.

5.2.2 Verification

Figure 5.3 (b) depicts the (untimed) state space of the system which is the starting point for the verification process. Failure states where event f has fired and place p_5 is marked are drawn as squares.

Before actually going for the verification process, let us briefly analyze the behavior of the system in the timed domain. Thus, assuming that the system is in its initial state (s_0) at instant 0, it can be seen that the firing time of the first occurrence of event f is fixed at time 4. On the contrary, the firing time of the first occurrence of event e depends on how many of the loops formed by events b and c are completed before d fires and triggers e. For example, if d fires right after a, event e is enabled at time 1. Therefore, e fires before f and prevents the failure. Similarly, if b fires before d right after a, but not after c, the run $s_0 \xrightarrow{a} s_2 \xrightarrow{b} s_3 \xrightarrow{c} s_2 \xrightarrow{d} s_1 \xrightarrow{e} s_0$ is time-feasible since the firing of e happens at time 3. Conversely, if the $b - c$ loop happens twice, the firing of e will happen at time 4, which conflicts with that of f. Moreover, if the $b - c$ loop is produced

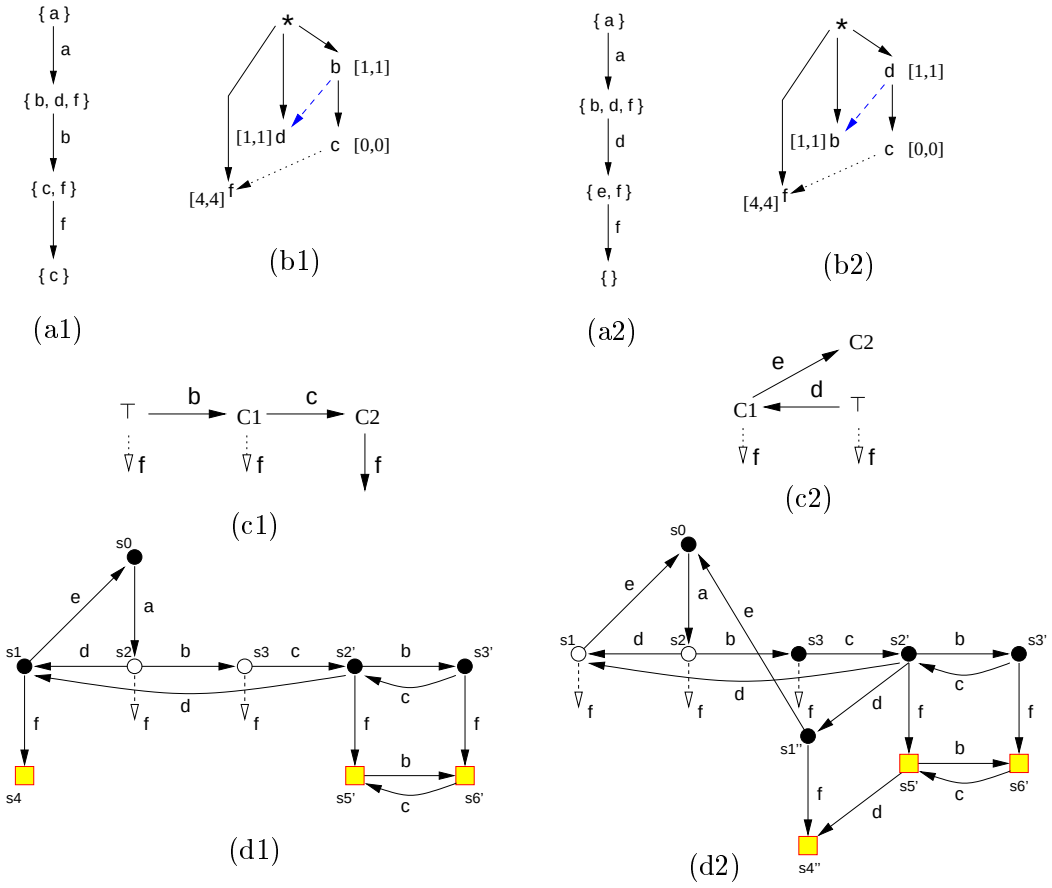


Figure 5.5 *Yoneda*'s example: first (left) and second (right) refinements.

more than twice, the system will always end up in a failure state. In consequence, no guarantee of correct behavior can be given for the system.

The verification process, as implemented in TRANSYT, requires five iterations to discover a counterexample trace that proves the incorrectness of the system. Figures 5.5 and 5.6 depict the four refinements of the state space before the counterexample is found. For each refinement four pictures are provided: (a) the untimed failure trace, (b) the LzCES obtained from the trace after timing analysis, (c) the LzTS corresponding to the GRC of the LzCES, and (d) the resulting LzTS after the enabling-compatible product of (c) with the LzTS obtained in the previous refinement. The asterisk drawn as the root event of the LzCESs represents a generic event that enables, at a nodal state, the events connected to it. Therefore, the delays of such events can be set to their respective min-max bounds. Moreover, the timing analysis for the resulting CES will apply to any portion of the state space where those events get enabled simultaneously, no matter who is their actual trigger

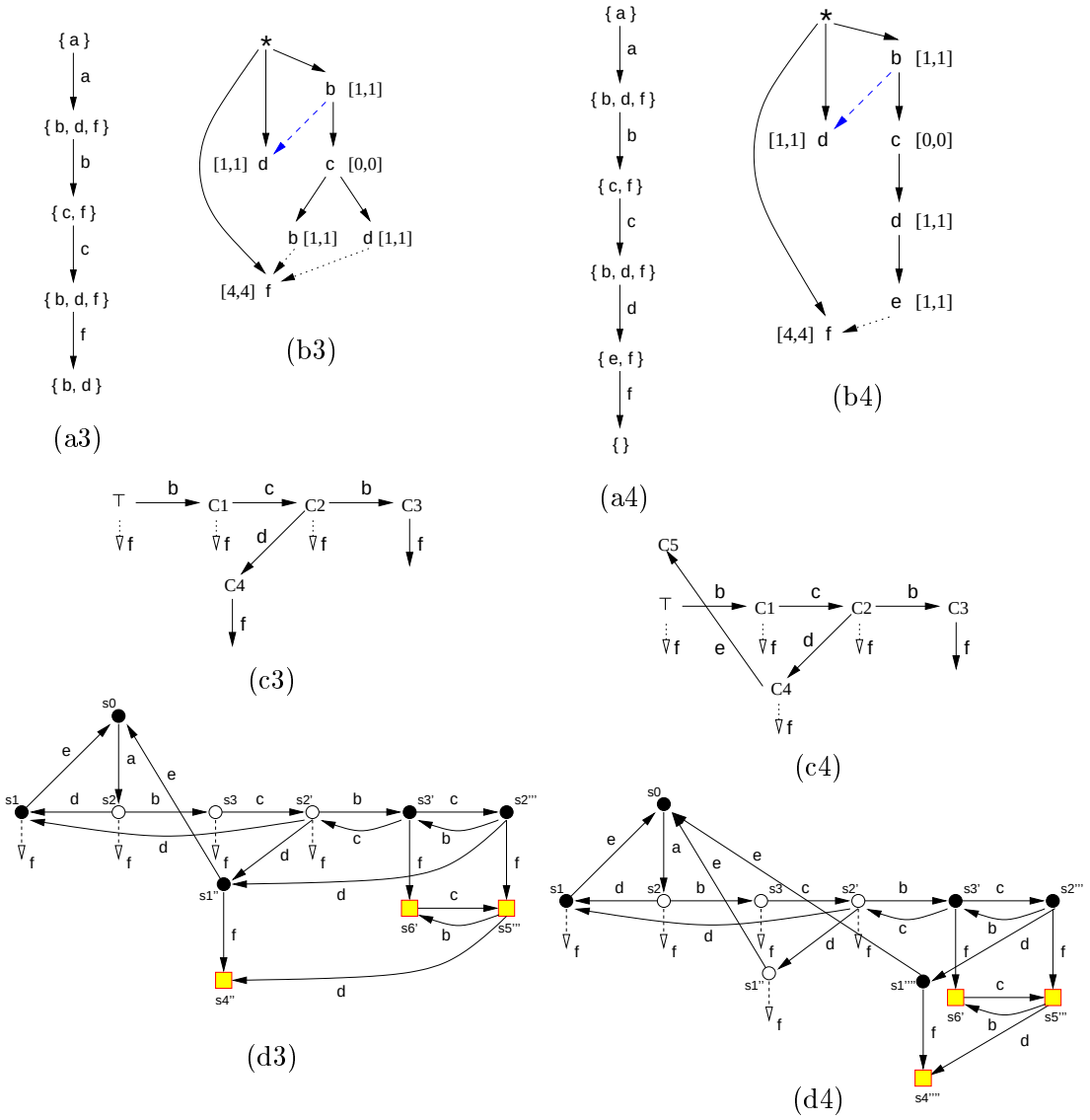


Figure 5.6 *Yoneda*'s example: third (left) and forth (right) refinements.

event. Hence the asterisk. Finally, Figure 5.7 depicts the counterexample trace found in the fifth iteration, which concludes the verification process.

The first refinement is depicted in the left of Figure 5.5. The failure is given by the firing of f in state s_3 after having fired a and b (a1). The timing analysis on the CES (b1) reveals that c, and therefore b, must fire before f. The LzTS (c1) corresponding to the GRC of the LzCES in (b1) is composed with the original TS of the system. The LzTS (d1) is obtained, where f has become lazy in states s_2 and s_3. However, states s_2 and s_3 can be also reached after subsequent firings of the loop formed by events

b and c. This makes that the performed timing analysis cannot apply for subsequent occurrences of states s_2 and s_3 in the timed domain. As a consequence, an unfolding of the state space is produced by the enabling-compatible product. Hence, states s'_2, s'_3, s'_5 and s'_6 in the resulting LzTS, will require further analysis in later iterations of the verification process. Remark, that those states where the enabling-compatibility applies are represented by white dots. Also, those states unfolded are annotated with as many primes as the number of the refinement that produced them. That is, s'_2 is produced by the first refinement, whereas s'''_2 is produced by the third refinement.

The next three refinements continue to prune and unfold the different parts of the state space of the system. In particular, the loop formed by events b and c is progressively unrolled, so that the enabling of event e is postponed more and more. As a result, when state s''''_1 is reached in the LzTS of Figure 5.6 (d4), event e becomes enabled whereas f is already enabled since state s_2. The trace in Figure 5.7 (a) depicts this situation, where f fires and disables e, thus producing the failure. The trace is timing-consistent with the delays of the events and therefore exists in the timed domain of the system. The enabling intervals and the firing times of the events in the trace are shown for clarity. Also, Figure 5.7 (b) depicts the LzCES obtained from the complete trace. The only timing relation in the LzCES (d must fire before f) is already given in the trace.

Notice that the LzCESs of Figures 5.5 (b1) and (b2), and those in Figures 5.6 (b3) and (b4), include the disabling relations that appear in the respective traces. Such disablings are not relevant for the timing analysis and therefore they can be simply removed from the LzCESs without affecting the later enabling-compatible product. The disabling relations are included here just for illustrative purposes. TRANSYT automatically removes them when they are irrelevant for the timing analysis.

What follows is the textual output produced by TRANSYT during the verification session. Brief information is given about the process of each refinement performed. More extensive information is stored in an browsable HTML file which contains links to the different graphical objects produced by the tool during the process. In this case, the options `-VwriteTrace1`, `-VwriteTES1`, `-VwriteGRC1` and `-VwriteSTD` indicate that DOT[2] files must be generated at each iteration for the failure trace, the timed event structure, the corresponding GRC, and the resulting state space after the refinement, respectively. The remaining options stand for the method to use in order to generate the failure traces (`-VfailTrace2` option specifies a partial traversal using chaining), and the methods to use for building the simplest possible event structures (`-AfilterTedges` and `-AfailGuided` options). More details on the options of the `tverif` command can be found in Appendix C.

[2] DOT is text-based format for specifying graphs, developed by AT&T research labs. Tool for editing and displaying the graphs are also included in the pack.

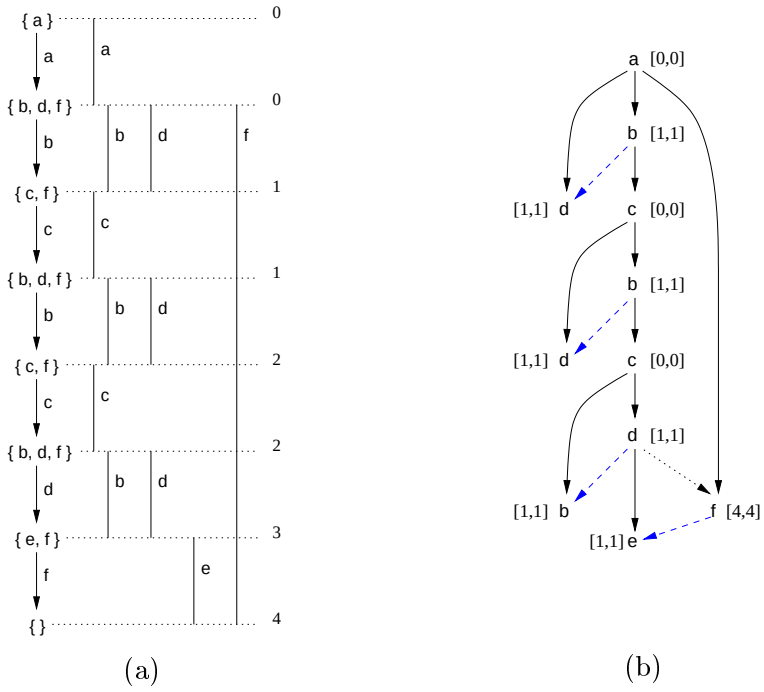


Figure 5.7 *Yoneda*'s example: (a) counterexample trace proving incorrectness annotated with enabling intervals and firing times; (b) corresponding LzCES.

```
ts > tverif -HTML -VwriteTrace1 -VwriteTES1 -VwriteGRC1 -VwriteSTD \
-AfilterTedges -AfailGuided -VfailTrace2

ts:: Starting verification iteration 1.
ts:: Searching a failure trace
ts:: Try to build timed ES from trace by "escape fail" criterion ... Succeeded
ts:: Time-compliance: escape fail.
ts:: Reachability analysis of the ES ... 4 markings visited
ts:: Composing GRC with the TS. 0+1 encoding vars required...
ts:: Timing constraints successfully applied.
ts:: Traversing the system....
ts:: Number of untimed states reached: 9
ts:: Checking fail conditions....
ts:: Number of fail states detected: 3
ts:: End of iteration 1.

ts:: Starting verification iteration 2.
ts:: Searching a failure trace
ts:: Try to build timed ES from trace by "escape fail" criterion ... Succeeded
ts:: Time-compliance: escape fail.
ts:: Reachability analysis of the ES ... 4 markings visited
ts:: Composing GRC with the TS. 0+1 encoding vars required...
ts:: Timing constraints successfully applied.
ts:: Traversing the system....
ts:: Number of untimed states reached: 10
ts:: Checking fail conditions....
ts:: Number of fail states detected: 3
ts:: End of iteration 2.
```

```
ts:: Starting verification iteration 3.
ts:: Searching a failure trace
ts:: Try to build timed ES from trace by "escape fail" criterion ... Succeeded
ts:: Time-compliance: escape fail.
ts:: Reachability analysis of the ES ... 8 markings visited
ts:: Composing GRC with the TS. 1+1 encoding vars required...
ts:: Timing constraints successfully applied.
ts:: Traversing the system....
ts:: Number of untimed states reached: 11
ts:: Checking fail conditions....
ts:: Number of fail states detected: 3
ts:: End of iteration 3.

ts:: Starting verification iteration 4.
ts:: Searching a failure trace
ts:: Try to build timed ES from trace by "escape fail" criterion ... Succeeded
ts:: Time-compliance: escape fail.
ts:: Reachability analysis of the ES ... 7 markings visited
ts:: Composing GRC with the TS. 1+1 encoding vars required...
ts:: Timing constraints successfully applied.
ts:: Traversing the system....
ts:: Number of untimed states reached: 12
ts:: Checking fail conditions....
ts:: Number of fail states detected: 3
ts:: End of iteration 4.

ts:: Starting verification iteration 5.
ts:: Searching a failure trace
ts:: The failure trace found is time-feasible.
ts:: Verification FAILS after 5 iterations.
ts:: End of iteration 5.
```

The overall verification process takes less than one second of CPU time in a $866MHz$ Pentium-III computer running Linux.

5.2.3 Discussion

In this example, the unfolding mechanism has lead to discover a suitable timing analysis that demonstrates the incorrectness of the system. In other cases, the unfoldings are necessary to achieve an exact-enough timing analysis to prove the non-existence of a given failure trace in the timed domain. In general, when this type of forward unfoldings are required, the actual number of unfoldings depends, among other factors, on the delays of the events involved in the timing analysis. In the example, the number of unfoldings needed to demonstrate the incorrectness of the system is in direct dependence on the delay of event `f`. Therefore, a pathological example can be easily built by increasing the delay of `f` sufficiently as to make the number of refinements of the state space too big to be handled by TRANSYT. However, we believe that these cases do not arise that often in practice.

Despite of the results related to the unfolding mechanism in the verification process, this section has briefly illustrated the capabilities of TRANSYT in order to model a timed PN as a binary-encoded TTS. Also, some fundamental commands of the tool have been introduced through the examples.

5.3 Verification of complex-gate decompositions in speed-independent circuits

This section illustrates the verification of the correctness of complex-gate decompositions in quasi-speed-independent asynchronous circuits. Additionally, the section also illustrates the way STGs and digital circuits are handled in TRANSYT, and how certain crucial properties for verification are modeled using boolean equations.

5.3.1 Speed-independent circuits

Several formalisms and methodologies have been proposed in recent years for the design and analysis of asynchronous control circuits (see [Mye01, SF01] for complete surveys on the topic). In particular, a lot of research has been carried out around the *speed-independent* paradigm [MB59, Dil89a, BM92, CKK+02].

Speed-independent circuits work under the *input-output* mode of operation, assuming the *unbounded gate delay* model. On one hand, the input-output mode of operation allows the environment of the circuit to change again after a circuit output, with no assumption about the stabilization of the internal signals of the circuit. Thus, speed-independent circuits are faster than those designed under the *fundamental mode* of operation [Huf54]. On the other hand, the pessimistic unbounded delay model for the gates allows a robust (*i.e. hazard-free*) operation regardless of the actual delays of the gates implementing the circuit. The interested reader is referred to [CKK+02] for a precise characterization of the speed-independence property, the associated design style, etc.

A speed-independent circuit is specified in terms of the behavior observed in the communication between the circuit and its environment. Such behavioral specifications are often given in terms of STGs (see Section 2.5). Both, speed-independent circuits and STGs have become very popular in the community of asynchronous circuits researchers. As a consequence, several logic synthesis tools have been developed, being PETRIFY [CKK+97] the most advanced of them.

The synthesis process requires the specification STG to satisfy certain properties. Once they are ensured, reachability analysis is performed to obtain the equivalent state graph from which boolean equations for each non-input signal can be derived. Finally, the equations can be implemented, for example, in terms of atomic complex-gates (*i.e.* one complex-gate per non-input signal). Often, the actual implementation of the circuit in terms of complex-gates is not feasible, since it is difficult to find such gates in traditional technology libraries. Therefore, the designer must face the decomposition of the complex-gates into structures of simpler logic gates (see [Bur96, KCKL99] for more details of the difficulties involved). Unfortunately, the decomposition process might introduce violations of the conditions that guarantee the correct operation of the circuit, may be a source for

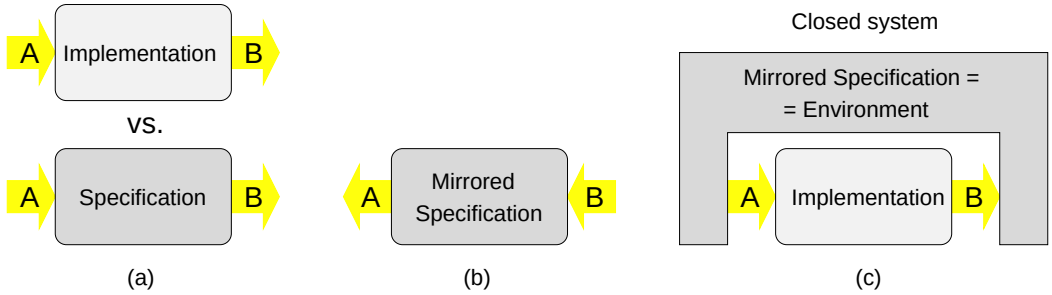


Figure 5.8 Verification scheme: (a) specification and implementation, (b) the specification is mirrored and (c) closed system used for verification.

hazards, etc. In these cases, however, the circuit can often operate correctly if certain timing assumptions on the delays of the gates are taken into account.

5.3.2 Experimental set-up

The experiments described in this section have been performed on a set of well-known STG behavioral specifications of academic-size asynchronous circuits. A speed-independent complex-gate circuit implementation of each specification has been obtained by using PETRIFY. Then, the complex gates have been decomposed and mapped into a library with only 2-input gates (NAND2, NOR2 and inverters). Finally, conventional decomposition methods for synchronous circuits have been applied for technology mapping, using the `map` command in SIS [SSL+92]. As a consequence, after decomposition the resulting circuits not hazard-free under the unbounded gate delay model. However, the circuits were expected to operate correctly if appropriate delay bounds were provided for the gates.

Consequently, a delay interval $[d-\varepsilon, d+\varepsilon]$ is assigned to each circuit gate, where $d = 1$ for inverters, $d = 3$ for NAND2 and NOR2 gates, and ε represents a 10% variation over the value of d, *i.e.* $\varepsilon = d/10$. We will consider also that the communication of the circuit with the environment is slower than the internal behavior of the circuit itself. Thus, the events produced by the environment are assumed to be 10 times slower than those produced by an inverter, *i.e.* $d = 10$. In summary, the delays for the experiments were set to: $[0.9, 1.1]$ for inverters, $[2.7, 3.3]$ for NAND2 and NOR2 gates and $[9, 11]$ for the environment. It is important to remark, that the actual values of the delay bounds do not influence the performance of the verification algorithm, as often happens in other verification approaches (see [CY91] for details).

Two properties were considered for verification on each circuit (see Section 5.3.6). One of the properties is the *input-output conformance* of the circuit with respect to the original specification. That is, the circuit always accepts the events produced by the environment, and vice versa. The other property is the *absence of hazards* in the circuit, which is

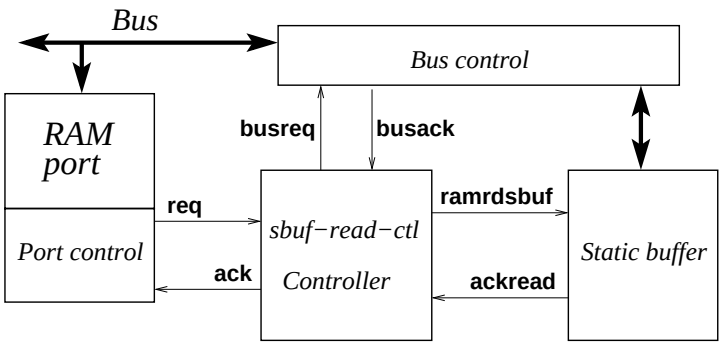


Figure 5.9 Input-output interface of the `sbuf-read-ctl` controller.

modeled in terms of signal persistency. Although absence of hazards is a desirable property of the internal behavior of a circuit, it does not guarantee correct input-output operation according to the specification. On the other hand, the circuit may conform with the specification but may also have internal hazards that do not propagate to the outputs. For the experiments, both properties were verified with the given delays.

The composition of the environment (mirrored specification) with the circuit defines the transition system that must be used for verification (see Figure 5.8). The specification is mirrored such that the input labels and variables are turned into output labels and variables, and vice versa. Then, the mirrored specification can be interpreted as the environment that exercises the inputs and listens to the outputs of the circuit implementation.

5.3.3 The `sbuf-read-ctl` controller

This section introduces the details of the modeling of a circuit example and its specification given as an STG, using TRANSYT. The circuit chosen to illustrate this section is a small asynchronous controller of the classical HP's Post Office benchmark suite [CDS93], commonly used by the asynchronous circuits community. The Post Office is the communication component for a distributed memory multiprocessor, that uses a buffering protocol to interface with the memory. The implementation of the control-circuitry requires 95 asynchronous finite-state machines, from which we have chosen the one called `sbuf-read-ctl`.

Figure 5.9 depicts the input-output interface of the `sbuf-read-ctl` controller. Its function is to control the data transfer from the static buffer to a port of the RAM through a shared bus. The behavior of the controller is basically as follows [Ste02]. First, it waits until a data request (req+) arrives from the control circuitry of the RAM port. Then the controller requests (ramrdsbuf+) the data packet to the static buffer, which acknowledges (ackread+) the request when the data is ready to be sent across the bus. Thus, the controller asks for bus mastership (busreq+ and busack+) and the data is placed on the

bus to be consumed by the RAM. Then, the bus mastership is released (busreq− and busack−), the buffer is freed (ramrdsbuf−) and the RAM port is acknowledged (ack+). Finally, the return-to-zero of all the lines is produced.

Figure 5.10 (a) depicts an STG that summarizes the intended behavior of the controller. In the STG, transitions corresponding to input signals are underlined and arcs between two transitions are assumed to hold an implicit place, which is not drawn. However, all places are explicitly named for better correspondence with the TRANSYT input file in Figure 5.10 (b). Also remark that signals y0_sbufreadctl and y1_sbufreadctl are not part of the original specification. They correspond to internal state signals required for the implementation of the controller as a speed-independent circuit [Ste02].

Using PETRIFY, a speed-independent circuit implementation of the specified behavior can be obtained. Figure 5.11 (a) shows the circuit obtained using a complex-gate per each non-input signal. Then, each complex-gate has been decomposed into structures of 2-input gates applying traditional methods for synchronous circuits. The SIS tool has been used for that purpose, which has generated the circuit of Figure 5.11 (b). As yet discussed above, the decomposition process might have introduced hazards that can cause circuit malfunction. However, if appropriate timing conditions are met due to the delays of the gates, the circuit might still operate properly in the specified environment. Thus the verification task will consist in checking such correct behavior and searching for a set of timing assumptions that guarantee it, provided the delays associated to the gates of the circuit.

Although TRANSYT supports both STGs (`astg` format) and circuit descriptions (`blif` format [SSL+92]) as input formats, and translates them automatically into TTSs, we show next how to model both types of systems using the `tsif` format.

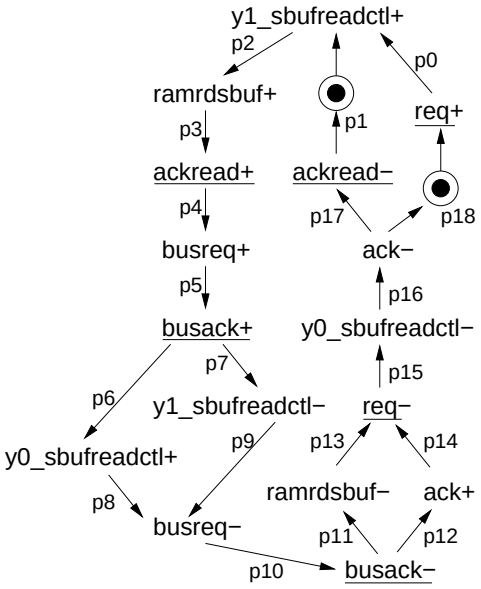
5.3.4 Model of an STG

In order to model an STG with TRANSYT input format, a similar mechanism to that presented for PNs in Section 5.2.1 can be used. The only actual difference is that when modeling an STG, the values of the signals must be also considered to properly model the state of the system. Thus, besides the variables for the places, one boolean state variable is used for each signal of the STG. Such variable is set if the signal has a high-level value and reset otherwise. Hence, for every transition of the STG, a transition relation of the TS must be specified in which the variables for the input (output) places of the STG transition are set (reset) in order for the transition to become enabled, and the variables become reset (set) after the transition is executed. Moreover, the variable for the signal value must switch according to the sense of the STG transition. For example, transition busack+ of the STG in Figure 5.10 (a) will be modeled by the following equation:

$$TR(\mathsf{busack}+) = p_5 \cdot \overline{p_6} \cdot \overline{p_7} \cdot \overline{\mathsf{busack}} \cdot \overline{NS(p_5)} \cdot NS(p_6) \cdot NS(p_7) \cdot NS(\mathsf{busack})$$

```
#HP's Post Office sbuf-read-ctl specification
TS sbuf-read-ctl INTERLEAVED

INPUT VARS req ackread busack;
OUTPUT VARS ack ramrdsbuf busreq y1_sbufreadctl y0_sbufreadctl;
INTERNAL VARS p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 p15 p16 p17 p18;

INPUT LABELS req ackread busack;
OUTPUT LABELS ack ramrdsbuf busreq y1_sbufreadctl y0_sbufreadctl;

#Behavior of the input signals
EVENT req- req
EQN TR p13 NS(p13)' p14 NS(p14)' p15' NS(p15) req NS(req)';
END

EVENT req+ req
EQN TR p0' NS(p0) p18 NS(p18)' req' NS(req);
END

EVENT ackread+ ackread
EQN TR p3 NS(p3)' p4' NS(p4) ackread' NS(ackread);
END

EVENT ackread- ackread
EQN TR p1' NS(p1) p17 NS(p17)' ackread NS(ackread)';
END

EVENT busack- busack
EQN TR p10 NS(p10)' p11' NS(p11) p12' NS(p12) busack NS(busack)';
END

EVENT busack+ busack
EQN TR p5 NS(p5)' p6' NS(p6) p7' NS(p7) busack' NS(busack);
END

#Behavior of the output signals
EVENT ack- ack
EQN TR p16 NS(p16)' p17' NS(p17) p18' NS(p18) ack NS(ack)';
END

EVENT ack+ ack
EQN TR p12 NS(p12)' p14' NS(p14) ack' NS(ack);
END

EVENT ramrdsbuf- ramrdsbuf
EQN TR p11 NS(p11)' p13' NS(p13) ramrdsbuf NS(ramrdsbuf)';
END

EVENT ramrdsbuf+ ramrdsbuf
EQN TR p2 NS(p2)' p3' NS(p3) ramrdsbuf' NS(ramrdsbuf);
END

EVENT busreq- busreq
EQN TR p8 NS(p8)' p9 NS(p9)' p10' NS(p10) busreq NS(busreq)';
END

EVENT busreq+ busreq
EQN TR p4 NS(p4)' p5' NS(p5) busreq' NS(busreq);
END

EVENT y1_sbufreadctl- y1_sbufreadctl
EQN TR p7 NS(p7)' p9' NS(p9) y1_sbufreadctl NS(y1_sbufreadctl)';
END

EVENT y1_sbufreadctl+ y1_sbufreadctl
EQN TR p0 NS(p0)' p1 NS(p1)' p2' NS(p2) y1_sbufreadctl' NS(y1_sbufreadctl);
END

EVENT y0_sbufreadctl- y0_sbufreadctl
EQN TR p15 NS(p15)' p16' NS(p16) y0_sbufreadctl NS(y0_sbufreadctl)';
END

EVENT y0_sbufreadctl+ y0_sbufreadctl
EQN TR p6 NS(p6)' p8' NS(p8) y0_sbufreadctl' NS(y0_sbufreadctl);
END

#Initial state
EQN ISTATE CONJUNCTIVE
p0' p1 p2' p3' p4' p5' p6' p7' p8' p9' p10' p11' p12' p13' p14' p15' p16' p17' p18;
req' ackread' busack' ack' ramrdsbuf' busreq' y1_sbufreadctl' y0_sbufreadctl';
END
```

(b)



(a)

Figure 5.10 (a) STG specification of the `sbuf-read-ctl` controller and (b) TRANSYT input file.

Similarly, in the boolean equation that specifies the initial state of the TS, the variables for the signals will be set to the initial values of the signals according to the STG.

With all the above considerations, Figure 5.10 (b) shows the `sbuf-read-ctl.g.ts` file corresponding to the STG of Figure 5.10 (a). Input and output variables are declared for each input and output signal of the specified circuit. Internal variables are declared for the places of the STG. The whole set of variables is then used to specify the boolean equations that define the behavior of the system. Such behavior is specified by means of a transition relation for each individual transition of the STG. Thus, one label is declared for each signal, whereas one event containing the actual transition relation is defined for each transition of the signal. Finally, the initial values of all the variables are specified.

Notice that the state signals y0_sbufreadctl and y1_sbufreadctl have been modeled by means of two output labels. They could have been modeled as internal labels, since they do not correspond to any observable behavior of the controller. However, we have made them visible on purpose, so that we can perform an stricter verification process.

5.3.5 Model of the circuit

In order to model a gate-level circuit, a straightforward procedure is followed. A variable and a label are used for each signal of the circuit. Each variable encodes the value of a signal, whereas the events associated to the label specify the signal switches according to the boolean functions implemented by the logic gates driving such signal. For example, two transition relations can be specified for the events produced by the complex-gate driving signal busreq in Figure 5.11 (a), as follows:

$$TR(\mathsf{busreq}+) = (\overline{\mathsf{y0\_sbufreadctl}} \cdot \mathsf{busreq} + \mathsf{ackread} \cdot \mathsf{y1\_sbufreadctl}) \cdot \overline{\mathsf{busreq}} \cdot NS(\mathsf{busreq})$$

and

$$TR(\mathsf{busreq}-) = (\mathsf{y0\_sbufreadctl} + \overline{\mathsf{busreq}}) \cdot (\overline{\mathsf{ackread}} + \overline{\mathsf{y1\_sbufreadctl}}) \cdot \mathsf{busreq} \cdot \overline{NS(\mathsf{busreq})}$$

Notice that the part of the equations inside the parenthesis correspond to the excitation conditions for a positive and a negative signal switch of the complex-gate, respectively.

Additionally, appropriate delay intervals can be specified for each event according to the criteria discussed above. Although it is possible to assign different delay intervals to each individual event of a label, in this case, the same delay bounds are assigned for all the changes of a given signal, no matter if they correspond to a rising or a falling transition.

Figure 5.12 depicts the resulting `tsif` file (`sbuf-read-ctl.m2.blif.ts`) for the quasi-speed-independent circuit implementation of Figure 5.11 (b). Notice that the behavior specified for the input signals allows them to switch freely. What this actually means is that there will exist an environment system responsible of driving the input signals of the circuit and setting the input state variables to appropriate values, according to the

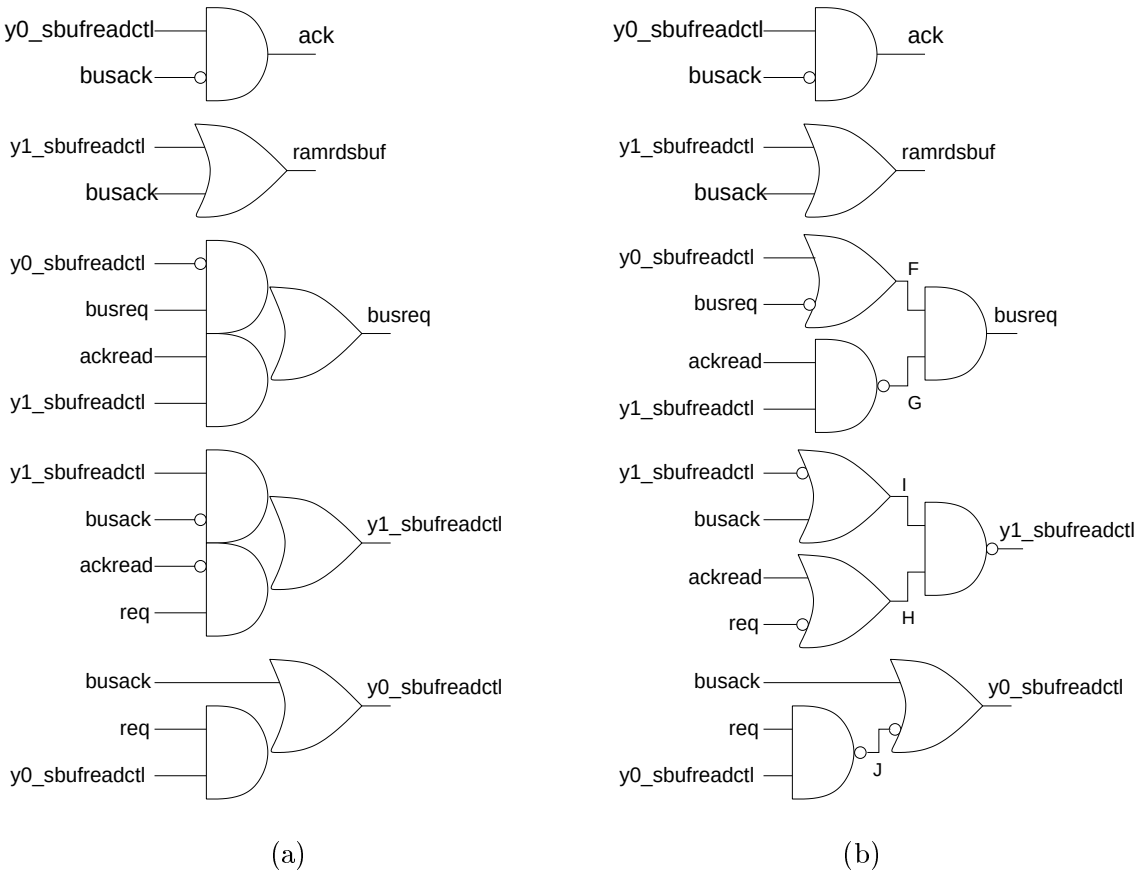


Figure 5.11 Two implementations of the `sbuf-read-ctl` controller: (a) complex-gate speed-independent and (b) after decomposition into structures of 2-input gates.

specification. Such system will be no other than that for the specification in Figure 5.10, but mirrored, *i.e.* inputs become outputs and vice versa (see the experimental set-up in Figure 5.8).

5.3.6 Specification of properties

According to the experimental set-up described above, both input-output conformance of the circuit with respect to the specification, and the absence of hazards in the circuit need to be considered. In TRANSYT, a property or invariant to be verified is specified in terms of its negated, *i.e.* as a failure condition expressed by a boolean formula.

Input-output conformance. Fail conditions to check input-output conformance are computed for those labels which are *synchronized* in order to build the closed system for verification (see Figure 5.8). More precisely, the fail conditions are only computed for those

```
#HP's Post Office sbuf-read-ctl circuit implementation
TS sbuf-read-ctl_net INTERLEAVED

INPUT VARS req ackread busack;
OUTPUT VARS ack ramrdsbuf busreq y1_sbufreadctl y0_sbufreadctl;
INTERNAL VARS F G H I J;

INPUT LABELS req ackread busack;
OUTPUT LABELS ack ramrdsbuf busreq y1_sbufreadctl y0_sbufreadctl;
INTERNAL LABELS F G H I J;

#Circuit behavior
EVENT rise ack
EQN TR NS(ack) ack' (busack' y0_sbufreadctl);
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT fall ack
EQN TR NS(ack)' ack (busack' y0_sbufreadctl)';
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT rise ramrdsbuf
EQN TR NS(ramrdsbuf) ramrdsbuf' (y1_sbufreadctl + busack);
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT fall ramrdsbuf
EQN TR NS(ramrdsbuf)' ramrdsbuf (y1_sbufreadctl + busack)';
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT rise busreq
EQN TR NS(busreq) busreq' (G' + F');
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT fall busreq
EQN TR NS(busreq)' busreq (G' + F')';
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT rise y1_sbufreadctl
EQN TR NS(y1_sbufreadctl) y1_sbufreadctl' (I' + H');
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT fall y1_sbufreadctl
EQN TR NS(y1_sbufreadctl)' y1_sbufreadctl (I' + H')';
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT rise y0_sbufreadctl
EQN TR NS(y0_sbufreadctl) y0_sbufreadctl' (J' + busack);
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT fall y0_sbufreadctl
EQN TR NS(y0_sbufreadctl)' y0_sbufreadctl (J' + busack)';
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

#Internal gates
EVENT rise F
EQN TR NS(F) F' (y0_sbufreadctl + busreq');
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT fall F
EQN TR NS(F)' F (y0_sbufreadctl + busreq')';
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT rise G
EQN TR NS(G) G' (ackread' + y1_sbufreadctl');
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT fall G
EQN TR NS(G)' G (ackread' + y1_sbufreadctl')';
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT rise H
EQN TR NS(H) H' (ackread + req');
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT fall H
EQN TR NS(H)' H (ackread + req')';
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT rise I
EQN TR NS(I) I' (y1_sbufreadctl' + busack);
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT fall I
EQN TR NS(I)' I (y1_sbufreadctl' + busack)';
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT rise J
EQN TR NS(J) J' (req' + y0_sbufreadctl');
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

EVENT fall J
EQN TR NS(J)' J (req' + y0_sbufreadctl')';
{DELAY: [ MIN= 2.7; MAX= 3.3; ]}
END

#The environment changes freely
#and has big delay
EVENT switch req
EQN TR NS(req)=req';
{DELAY: [ MIN= 9; MAX= 11; ]}
END

EVENT switch ackread
EQN TR NS(ackread)=ackread';
{DELAY: [ MIN= 9; MAX= 11; ]}
END

EVENT switch busack
EQN TR NS(busack)=busack';
{DELAY: [ MIN= 9; MAX= 11; ]}
END

#Initial state
EQN ISTATE CONJUNCTIVE
req' ackread' busack' ack' ramrdsbuf' busreq';
F G H I J y1_sbufreadctl' y0_sbufreadctl';
END
```

Figure 5.12 TRANSYT input file for the circuit of Figure 5.11 (b).

cases in which an output label of the implementation system is synchronized with an input label of the environment. Intuitively, the condition identifies as an incorrect behavior the fact that some event on an output label x may be produced by the implementation in a given state, but the environment is not ready to process such event. In other words, the implementation system is producing an event which was not expected according to the specification system.

More formally the condition for the synchronized label x (IMP.x in the circuit and ENV.x in the environment) to cause an input-output conformance failure in a given state can be posed as:

$$Fail(\mathsf{x}) = FF(\mathsf{IMP.x}) \cdot \overline{FF(\mathsf{ENV.x})}$$

Where IMP.x and ENV.x are the corresponding *local* versions of label x in the implementation system (IMP.x is an output) and in the environment system (ENV.x is an input), respectively. In practice, the condition identifies as failure situations all those states where a change of the circuit's output signal IMP.x is not expected by the environment. That is, although IMP.x can fire, ENV.x cannot.

Failure conditions for input-output conformance can be automatically computed by TRANSYT.

Signal persistency. The presence of a potential hazard at the output of a gate is modeled in terms of non-persistency. That is, a label is persistent if once some of its events becomes enabled to fire, it cannot be disabled by the firing of any of the events of another label. Non-persistent labels may result in undesired hazards at the corresponding circuit signals, therefore the failure condition only applies to the non-input labels of the implementation system. Although this type of condition is mainly used for circuit analysis, it can be useful also in other contexts where the undesired disabling of a given event must be considered.

The following condition specifies the fact that the firing of label x *disables* some other label y, *i.e.* x induces non-persistency to y:

$$Fail(\mathsf{x}) = TR(\mathsf{x}) \cdot FF(\mathsf{x}) \cdot \bigcup_{\mathsf{y}} \left(EF(\mathsf{y}) \cdot \overline{EF'(\mathsf{y})} \right)$$

where TR is a transition relation, EF and FF are respectively enabling and firing functions expressed in terms of current-state variables, and EF' is a firing function expressed in terms of next-state variables. Finally, y is any non-input label of the (implementation) system, different from x.

The resulting condition identifies as failure situations all those attempts to execute label x from a state where another non-input label y was ready to fire, and the firing of x leads to a state where y it is no longer ready to fire.

Failure conditions for induced non-persistency can be automatically computed by TRANSYT.

Although these and other failure conditions can be explicitly specified in the TRANSYT input file or through the command line, the verification engine is also able to compute them automatically just by specifying certain options when building the closed system for verification (see the `uverif` command below). In this example, we will leave TRANSYT to compute such failure conditions.

5.3.7 Verification

In the verification session, the specification (`sbuf-read-ctl`) and the circuit implementation (`sbuf-read-ctl_net`) models are read first. Then, the closed system for verification is built (`uverif` command with the `-Vclose` and `-Vnotdestroy` options). Internally, the specification model is mirrored and the failure conditions are automatically computed by default (see Appendix C for more details).

What follows is an excerpt of the textual output produced by TRANSYT for these an other commands of the verification session.

```
ts > read_ts sbuf-read-ctl.g.ts
. . .

ts > read_ts sbuf-read-ctl.m2.blif.ts
. . .

ts > uverif -Vclose -Vnotdestroy sbuf-read-ctl sbuf-read-ctl_net
ts:: Mirroring specification ...
ts:: Building closed system ...
ts:: Building automatic CONFORMANCE fail conditions  ...
ts:: Building automatic PERSISTENCY fail conditions  ...
ts >

ts > traverse
. . .

ts > print_fails
ts:: Fail conditions for label 'IMP.F'.
ts:: Condition #0 defined (on transitions)
ts:: with equation:
ts:: busreq NS(busreq) y0_sbufreadctl' IMP.F NS(IMP.F)' IMP.G + busreq' IMP.F' NS(IMP.F)
(NS(busreq) NS(IMP.G)' + NS(busreq)' NS(IMP.G))
ts:: Detected #1 failure states
. . .
ts:: Fail conditions for label 'y1_sbufreadctl'.
ts:: Condition #0 defined (on states)
ts:: with equation:
ts:: y1_sbufreadctl IMP.H IMP.I (busreq' + ramrdsbuf' + ack + busack' + ackread' + req') +
y1_sbufreadctl' (IMP.I' + IMP.H') (y0_sbufreadctl + busreq + ramrdsbuf + ack + busack +
ackread + req')
ts:: Detected #4 failure states
. . .
```

Next, the system is traversed detecting 16 failures from a total of 74 untimed states. Also, part of the textual output of the `print_fails` command is shown: induced signal

Signal	*Failure type*	*Initial*	*It.1*	*It.2*	*It.3*
F	Induces non-persistency to busreq	1	1	-	-
I	Induces non-persistency to y1_sbufreadctl	4	-	-	-
J	Induces non-persistency to y0_sbufreadctl	4	4	4	-
req	Induces non-persistency to J	1	1	1	-
busack	Induces non-persistency to I	4	-	-	-
y0_sbufreadctl	Induces non-persistency to F	6	6	-	-
busreq	Non-conformance	1	1	-	-
y1_sbufreadctl	Non-conformance	4	-	-	-
y0_sbufreadctl	Non-conformance	4	4	4	-

Table 5.1 Failure situations in sbuf-read-ctl along the verification.

persistency condition for circuit's internal label F (1 failure situation detected) and conformance condition for synchronized label y1_sbufreadctl (4 failure situations detected). The first three columns of Table 5.1 summarize the failure situations detected for the different signals. Notice that the total amount of failure situations (the sum of the column named *Initial* is 29) is bigger than that reported after the traversal of the system (16). This reason is because the same state or transition may cause a property violation due to more that one failure condition.

Although in general, the actual number of failure situations to deal with is bigger that that reported after the traversal of the system, it also happens that some of the failure situations are consequence of other failures. That is, a cascade effect is produced as long as the effect of a failure is propagated through the system under verification. As a consequence, it is often the case that when a failure situation is eliminated along the verification process, other failure situations are eliminated as well. On the other hand, the `-VextendTrace` option of the `tverif` command (see Appendix C for details) allows to include more that one failure situation in a single failure trace, if it is possible. In such a way, several failure situations can be directly refined in a single iteration. The former *side effect* and the latter optimization, help to reduce the number of iterations required by the verification process.

The same options for the `tverif` command as those used in Section 5.2.2, have been used for verification. The verification process needs three iterations in order to prove the correctness of the quasi-speed-independent circuit implementation of the `sbuf-read-ctl` controller against the original specification. Hence, although the circuit is not speed-independent, the delays in the gates and the assumption of a slow environment (see Section 5.3.2), guarantee the correct operation of the circuit. Table 5.1 summarizes the

evolution of the number of failure situations along the three refinements. Also Figure 5.13 depicts the failure trace and the corresponding LzCES used in each iteration. The resulting LzTSs obtained after each refinement are not shown for space reasons.

In the first iteration of the verification process, the trace of Figure 5.13 (a1) is generated. In the trace, the firing of I− causes the disabling of y1_sbufreadctl−. Therefore, the state in which I− fires is one of the four failure states indicated in the second row of Table 5.1. The failure situation arises since after the rising of signal H and being I high, the gate driving y1_sbufreadctl (see Figure 5.11 (b)) is excited to produce a falling transition. However, the transition is prevented by the (long enabled) fall of signal I. Clearly, the failure would not exist if I− occurred before H+. In this sense, the timing analysis derived from the causal relations extracted from the trace reveals that when I− is concurrently enabled with ramrdsbuf+ (which triggers the input ackread+), I− actually occurs earlier than ackread+ (see the LzCES of Figure 5.13 (b1)), and consequently before H+. Therefore, the failure is proved inexistent and the verification process continues. As a side result of the refinement, the non-persistency of I induced by busack and the non-conformance of y1_sbufreadctl have been also removed (see *Ite. 1* in Table 5.1).

The second iteration analyzes a persistency violation of the falling transition of signal F, induced by an early rising transition of y0_sbufreadctl right after busack+ (see the failure trace in Figure 5.13 (a2)). Looking at the circuit in Figure 5.11 (b), the disabling of F− is clear since a high value of y0_sbufreadctl will prevent the OR gate driving F from falling. The failure would not exist if F− was allowed to happen before y0_sbufreadctl+ fires. As a consequence of the timing analysis on the events of the failure trace, F− is faster than the input busack+ when both are triggered simultaneously (see the LzCES in Figure 5.13 (b2)). Therefore, since busack+ triggers y0_sbufreadctl+, the failure cannot happen. The refinement of the state space has removed also the non-persistency on busreq (induced by F) and the subsequent non-conformance (see *Ite. 2* in Table 5.1).

In the third and last iteration, a persistency violation on y0_sbufreadctl induced by J is analyzed. Notice that, although the potential firing of y0_sbufreadctl− would also cause a conformance violation, it is not *observable* by the trace since such transition never occurs along it. The resulting failure trace is too long and it is only depicted partially in Figure 5.13 (a3). However, the beginning of the trace (up to the firing of G−) is the same than that of the trace of the second iteration. Looking at the circuit of Figure 5.11 (b), the disabling of y0_sbufreadctl− due to the falling transition of J is obvious. For the failure to be avoided, J− should have occurred before busack− in such a way that y0_sbufreadctl keeps at a high value, and does not get enabled to fall after busack−. Precisely, this condition is discovered by the timing analysis (see the LzCES of Figure 5.13 (b3)). As a consequence of the subsequent refinement, all the remaining failure situations are removed from the system. This concludes the verification process, which proves the

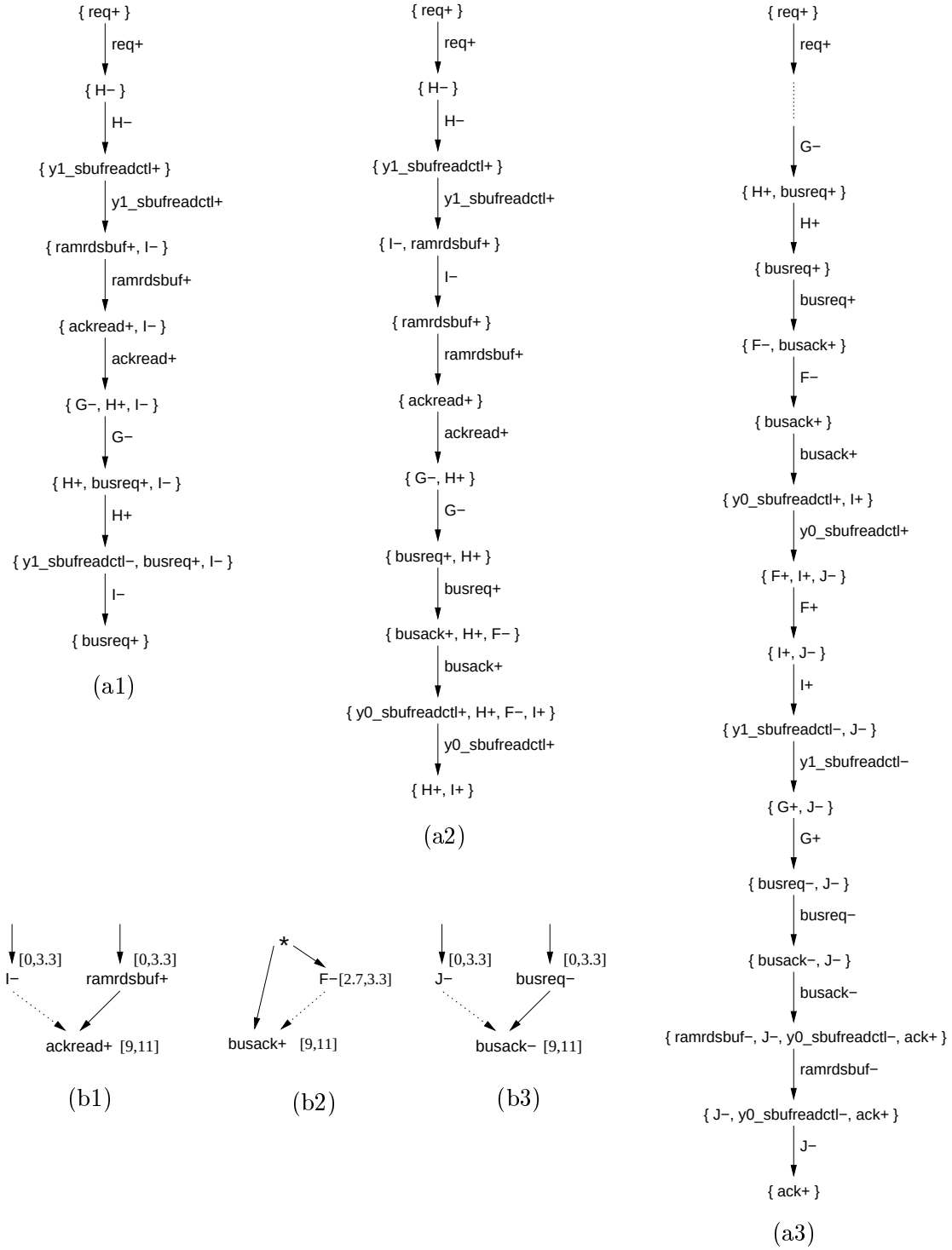


Figure 5.13 Three refinements for the verification of the `sbuf-read-ctl` controller: failure traces and corresponding LzCESs.

Circuit	Σ	S	G	S_u	S_f	TC	C	CPU
`sbuf-read-ctl`	8(5)	19	10	74	16	3	Y	1
`rcv-setup`	5(2)	14	6	78	34	2	N	1
`alloc-outbound`	9(5)	21	11	82	20	4	Y	3
`ebergen`	5(3)	18	9	83	22	1	N	1
`mp-forward-pkt`	8(5)	22	8	186	70	8	Y	5
`dff`	3(1)	14	6	255	164	6	N	3
`half`	4(2)	14	7	227	133	1	N	1
`chu133`	7(4)	24	9	288	204	2	N	1
`converta`	5(3)	18	12	408	244	10	N	12
`nowick`	6(3)	20	10	510	292	4	Y	3
`chu150`	6(3)	26	8	520	339	3	N	1
`sbuf-send-ctl`	8(5)	27	13	1592	1081	18	N	54
`vme`	6(3)	24	12	1736	1460	21	Y	30
`rpdtf`	5(1)	22	8	2612	1841	2	N	2
`tsend-bm`	9(4)	40	12	3880	2999	3	N	46
`sbuf-send-pkt2`	9(5)	28	13	4544	4044	19	Y	103
`sbuf-ram-write`	12(7)	64	15	14016	12362	34	N	415
`ram-read-sbuf`	11(6)	39	16	19328	17488	36	Y	550
`mr1`	9(5)	190	16	21076	11574	29	Y	317
`mr0`	11(6)	302	20	727304	642291	2	N	48
`trimos-send`	9(6)	336	24	$2.1\ 10^6$	$1.8\ 10^6$	1	N	127
`mmu`	8(4)	174	22	$5.6\ 10^6$	$5.2\ 10^6$	3	N	480

Table 5.2 Experimental results for the verification of asynchronous circuits.

correct behavior of the circuit according to the given specification and the properties imposed.

The overall verification process takes less than one second of CPU time in a $866MHz$ Pentium-III computer running Linux.

5.3.8 Results and discussion

Table 5.2 reports the results obtained in the verification of a set of asynchronous circuits to which complex-gate decompositions, similar to those described for the `sbuf-read-ctl` example, were applied. The experimental set-up described in Section 5.3.2 and a verification procedure similar to that of Section 5.3.7 has been used for all the benchmarks.

In the table, columns Σ and S contain, the total number of signals of the circuits (the number of non-input signals are shown in parenthesis) and untimed states of the corresponding specification, respectively. Columns G and S_u indicate the number of gates and the number of untimed states of the circuit. Column S_f indicates the number of untimed failure states. Column TC indicates the number of event structures (timing constraints) generated for timing analysis. This corresponds to the number of iterations of the main verification algorithm presented in Chapter 4 (see Figure 4.15). The column C indicates whether the circuit is proved correct or not for the aforementioned properties. Finally, CPU times obtained in a $866MHz$ Pentium-III computer with 1GB of memory running Linux are given in seconds. Although it is not explicitly mentioned in the table, we want to remark that the peak memory usage for most of the benchmarks keeps below 265MB.

It can be observed that the synchronous decomposition of the complex-gates of the speed-independent implementations produces a large amount of failure states (see column S_f in the table). Namely, in average, about 65% of the untimed states of each circuit correspond to failures. Although the number of untimed states of the circuits is not too big, the number of failure situations makes the verification very hard in some cases.

Most specifications were marked graphs, *i.e.* choice-free Petri nets. However, the transition system obtained after the composition with the circuit to build the closed system for verification, in some cases manifested a great variety of causality relations among the events (conjunctive, disjunctive, and complex combinations of both) produced by the functionality of the gates. This fact, complicates the verification process in some cases. Those where several LzCESs must be generated in order to cover all the causality relations that lead to a given failure situation.

The results show that systems with more than 10^6 untimed states could be verified in reasonable CPU times. The computational cost of the verification algorithm highly depends on the number of timing constraints required to refine the untimed state space. Some heuristics to improve the strategies to select adequate event structures will be explored in the future. On the other hand, the memory requirements keep reasonable in all cases (below 256MB for most benchmarks).

The three largest examples were proved to be incorrect. Only few iterations were required to find an erroneous trace. On the other hand, some circuits required a lot of timing constraints to prove its correctness (*e.g.* `ram-read-sbuf`, `sbuf-ram-write` and `mr1`). We believe that many of these constraints can be redundant and simplified when considering the complete set of constraints as a whole. This is left for future optimizations.

It is important to notice that the experiments have been performed using the generic verification methodology presented in Chapter 4, without any tunning or specific strategies to cope with digital circuits. For example, hierarchical verification using automatic

abstractions of sets of gates into complex ones (see *e.g.* [RCP95]) could have improved the results significantly.

For comparison, say that the same set of benchmarks was used in [BJMY02] to experiment with the tool OPENKRONOS, which extends the tool KRONOS [BDM+98] with BDD support for efficient state representation. The results, although promising, show that OPENKRONOS was not able to cope with the bigger circuits such as `mr1`, `mr0`, `trimos-send` and `mmu`. As a matter of example, the biggest circuit OPENKRONOS could verify was the `ram-read-sbuf` controller, requiring 826 seconds of CPU time on a SUN Ultrasparc 10 with 2GB of memory.

Despite of the above verification results, this section has also illustrated the way an STG and a digital circuit can be modeled in TRANSYT. Also, a discussion is provided on how to model certain crucial properties for verification. That is, the input-output conformance of a system with respect to a specification; and signal persistency, which captures the presence of hazards in a circuit, for example. The boolean equations needed to model such properties in TRANSYT can be computed automatically by the tool itself.

We want also to remark that TRANSYT supports both timed STGs (`astg` format) and digital circuit descriptions (`blif` format) as input formalisms. The tool is able to automatically map them into binary-encoded TTSs using similar procedures to those presented in Sections 5.3.4 and 5.3.5.

5.4 Verification of relative timing assumptions

Speed-independent circuits typically require a lot of circuitry to properly implement the event-detection mechanisms that make possible their correct operation regardless of the delays of the gates. Moreover, the delay model they rely on is sometimes too conservative about the temporal behavior of the environment of the circuit, and also about the physical details of the implementation of the gates. On the other hand, it has been shown [CKK+98] that by taking delay information into account, certain behaviors covered by the speed-independent implementation cannot actually exist. As a consequence, the size of the circuits can be reduced (see Example 4.1) at the cost of considering a set of timing assumptions. However, the property of speed-independence may be lost due to the optimizations. That is, the circuit may not operate correctly for any possible delay of the gates, and it is crucial to know under which assumptions the circuit will behave properly.

This section illustrates how the methodology presented in Chapter 4 can be used for the verification of relative timing assumptions in timed asynchronous circuits. That is, to check whether the assumptions derived by the synthesis process are actually met in the circuits when the delay information is taken into account. Moreover, it is shown how the set of sufficient timing constraints used by TRANSYT along the proofs, are actually very close to those imposed by the synthesis. Notice that this verification is not sufficient in

order to guarantee the correct operation of the circuit according to the specification. For that, input-output conformance with the specification should be verified as in the previous section.

5.4.1 Synthesis of asynchronous circuits with relative timing assumptions

PETRIFY allows the synthesis of hazard-free asynchronous circuits from STGs under certain *relative timing assumptions*. The assumptions refer to the specific ordering of events with respect to other events in the timed domain. In contrast, *absolute timing assumptions* rely on the specification of time intervals about the occurrence or enabling of the events.

Three types of relative timing assumptions are allowed: *difference* assumptions, *simultaneity* assumptions and *early enabling* assumptions. All of them rely on the differentiation between the concept of enabling region an that of firing region (see the concept of LzTS in Chapter 2). Whereas in speed-independent circuits both concepts are the same (an event can fire as soon as it is enabled), they become different when timing information is considered. For an excellent coverage on the topic, refer to [CKK+02].

The example in Figure 5.14 will illustrate the following discussion on the types of timing assumptions that can be considered for synthesis. Figure 5.14 shows a portion of a simple STG (a) and its corresponding untimed state space (b), where the enabling region coincides with the firing region for all the events.

Difference assumptions

Given two concurrent (*i.e.* simultaneously enabled and not in conflict) events a and b, a *difference assumption* denoted by $\mathsf{a} < \mathsf{b}$ indicates that a will always fire before b. In terms of separation times between events, $\mathsf{a} < \mathsf{b}$ is given when $Sep_{max}(\mathsf{a}, \mathsf{b}) < 0$, *i.e.* the upper bound on the difference between the firing times of a and b is negative.

In a LzTS, $\mathsf{a} < \mathsf{b}$ is represented by a *concurrency reduction* of b with respect to a, such that b is only firable in those states where a has already fired. As a consequence, those states where a and b are simultaneously enabled can be removed from the enabling region of b.

In the example of Figure 5.14, a difference assumption such as $\mathsf{a} < \mathsf{d}$ causes the removal of the arc $\mathsf{s}_1 \xrightarrow{\mathsf{d}} \mathsf{s}_5$. As a consequence state s_5 is unreachable and event d becomes lazy since $\mathsf{FR(d)} = \{\mathsf{s}_2, \mathsf{s}_3, \mathsf{s}_4\}$ and $\mathsf{EnR(d)} = \mathsf{FR(d)} \cup \{\mathsf{s}_1\}$ (see Figure 5.14 (c)). During the synthesis process, PETRIFY can consider state s_1 as a "*don't care*" for the enabledness of event d, which provides a source for logic optimization.

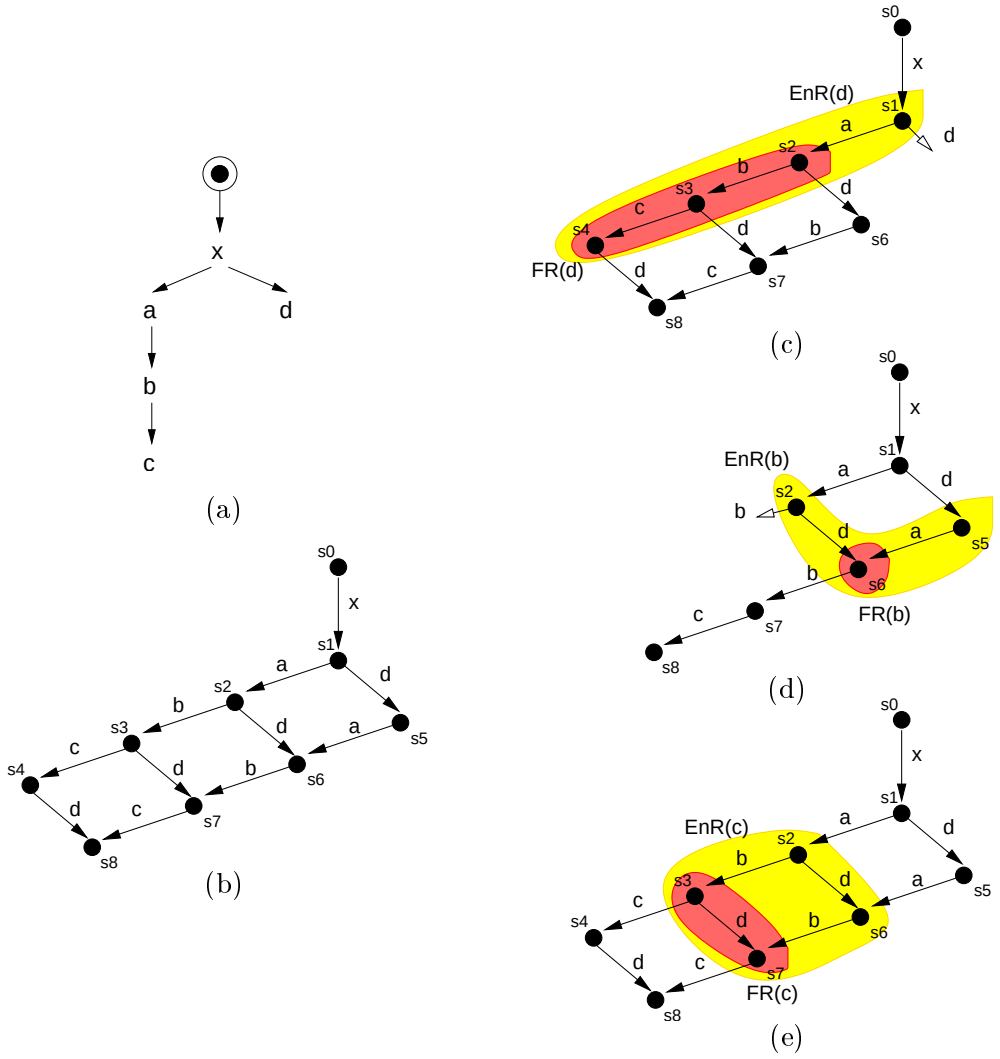


Figure 5.14 An example of relative timing assumptions. (a) A portion of a simple STG and (b) its corresponding untimed state space. LzTSs where: (c) $a < d$, (d) $a = d@b$ and (e) $c > b$.

Although difference assumptions are the mainly used timing assumptions (see [MM93] for example), they do not fully express the lazy behavior of signals. Hence the following types of relative timing assumptions.

Simultaneity assumptions

Under the *burst-mode* of operation [Now93], the transitions at the outputs of a circuit appear as simultaneous from the point of view of the (slow) environment of the circuit. Under the more restrictive *fundamental mode* of operation [Huf54], the input signals must

also change simultaneously. In contrast, *simultaneity assumptions* propose a sort of *local fundamental mode* with respect to particular groups of transitions.

The simultaneity assumption is a relative notion, defined on a set of events $E_s = \{e_1, \ldots, e_k\}$ with respect to a reference event a, which is triggered by some of the events in E_s. Under the assumption for a, the skew on the firing times of the events in E_s is not distinguishable. In terms of separation times: $\forall e_i, e_j \in E_s, |Sep_{max}(e_i, e_j)| < \delta^l(a)$. The practical consequence of this assumption, denoted by $e_1 = e_2 \ldots = e_k@a$, in a LzTS is that event a will not fire in any of the states where some $e_i \in E_s$ is still enabled, until all the events in E_s have fired. Moreover, the system would produce the same observable behavior if a was triggered by its original trigger, or by all the events in E_s.

In the example of Figure 5.14, a simultaneity assumption such as $a = d@b$ affects the LzTS in two ways (see Figure 5.14 (d)):

- States s_3 and s_4 become unreachable since although b is already enabled by the firing of a, d has not fired yet. Clearly, if $|Sep_{max}(d, a) < \delta^l(b)|$ (from the simultaneity assumption) and $|Sep_{max}(a, b) < 0|$ (from the causality between a and b), implies that $|Sep_{max}(d, b) < 0|$, *i.e.* d must fire before b.

- Additionally, $\mathsf{EnR}(b)$ can be safely extended to include state s_5 indicating that b could have been also triggered by d. The observable behavior of the system will remain unchanged thanks to its timing properties.

Figure 5.14 (d) depicts the resulting LzTS where the enabling and firing regions of b are highlighted. Notice that the possibility of extending the enabling region of the reference event allows further logic optimizations, which are not possible if only difference assumptions are considered.

Early enabling assumptions

Simultaneity exploits the laziness between concurrent events. *Early enabling* assumptions generalize this idea to ordered events. Assume event a triggers event b and the implementation of b is slow compared to that of a, *i.e.* $\delta^u(a) < \delta^l(b)$. Therefore, the enabling of b could be simultaneous to that of a and the proper ordering of a before b will be ensured by the timing properties of the logic implementing both events. The practical consequence of this assumption, denoted by $b > a$, in the LzTS is that the enabling region of b can be safely expanded to cover also the enabling region of a.

In the example of Figure 5.14, an early enabling assumption such as $c > b$ results in the possibility of expanding $\mathsf{EnR}(c)$ to include s_2 and s_6. Thus, c could have been triggered by a but the timing relation between b and c guarantee that c will not fire until b has fired. Figure 5.14 (e) depicts this effect, where $\mathsf{EnR}(c)$ and $\mathsf{FR}(c)$ are highlighted.

The above relative timing assumptions are key to perform timing optimizations in the synthesis process implemented by PETRIFY. Whereas difference assumptions are mainly used to remove unreachable states in the timed domain, simultaneity and early enabling assumptions provide a source for optimizations of the logic by choosing appropriate lazy behaviors between sets of signals.

Notice that the above assumptions rely on certain properties on the delays of the logic implementing each signal of the circuit. However, accurate delays may not be known until the synthesis process has completed. As a consequence, verification to ensure the validity of the assumptions is required. Moreover, when the timing assumptions do not hold, either the circuit is resynthesized without the invalid assumptions, or the delays of the circuit components are adapted to satisfy them.

5.4.2 The VME bus controller

This section introduces the example we will use to illustrate the verification of relative timing assumptions in the following sections.

Figure 5.15 (a) shows the I/O interface of a VME bus controller that controls the communication of a device with the bus through a data transceiver (signal D). At the side of the device there is a pair of handshake signals that follow a four-phase protocol (LDS and LDTACK). At the side of the bus there are two input signals (DSr and DSw) that follow a four-phase protocol with the output signal DTACK. DSr and DSw indicate the beginning of a READ and WRITE cycle, respectively. The timing diagram corresponding to a READ cycle is depicted in Figure 5.15 (b).

An STG describing the complete behavior of the controller is shown in Figure 5.15 (c). The choice places model the non-determinism of the environment, which can choose to initiate a READ or a WRITE cycle after the completion of the previous cycle. Notice also that some signal transitions, *e.g.* LDS+, have multiple instances in the STG. The indexes 1 and 2 have been used to distinguish events in the READ and WRITE cycles, respectively.

The timing assumptions used by PETRIFY for the synthesis of the controller come from three different sources: the assumption of a slow environment, the assumption of a slow bus control logic, and the intervention of the designer.

The assumption of a slow environment considers that the response time of the environment is long enough to allow the circuit to complete its internal activity, *i.e.* firing of enabled non-input signals. Thus, the inputs of the circuit are assumed to have a delay in the range $[k, \infty)$, where k is large enough to allow the circuit to stabilize after a change at its inputs. This general assumption gives a lot of margin for PETRIFY to automatically derive other timing assumptions.

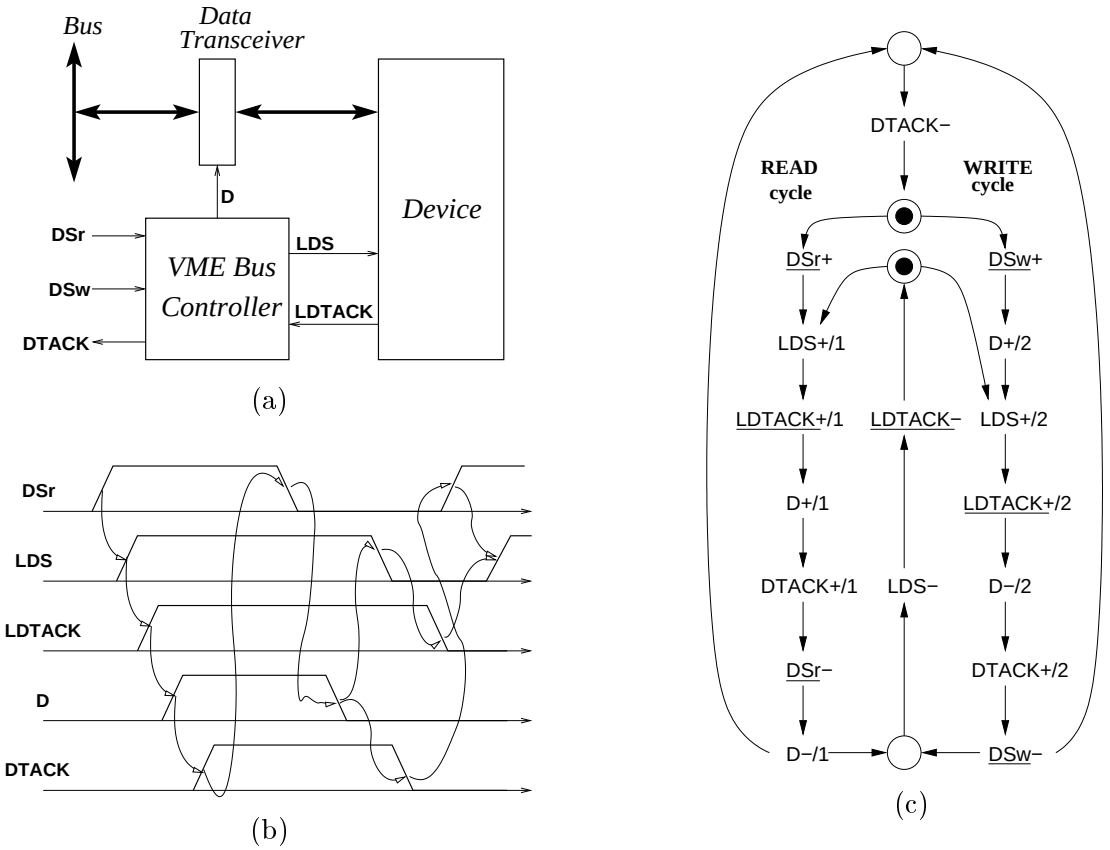


Figure 5.15 VME bus controller: (a) input-output interface, (b) waveform for the READ cycle and (c) STG specifying the full behavior of the controller.

On the other hand, looking at the specification of the controller in Figure 5.15, it can be seen that the return-to-zero of the protocols at both sides of the controller (bus and device) is done concurrently. However, it could be realistic to assume that the circuitry at the side of the bus is slow enough, such that any new request for a read or write cycle (DSr+ or DSw+) will never arrive at the controller before the handshake with the device has been completed. This corresponds to two difference assumptions, LDTACK− < DSr+ and LDTACK− < DSw+, which can be enforced to PETRIFY.

Finally, the analysis of some preliminary implementations of the circuit shows that events LDS− and DTACK−, which are concurrent in the specification, will always occur ordered in the timed domain since the logic driving LDS is simpler than that for DTACK. Thus, the designer may force an evident concurrency reduction such that LDS− will always fire before DTACK−.

The resulting set of timing assumptions, including those derived automatically by PETRIFY, are summarized in Figure 5.16. Figure 5.17 depicts the resulting circuit implemen-

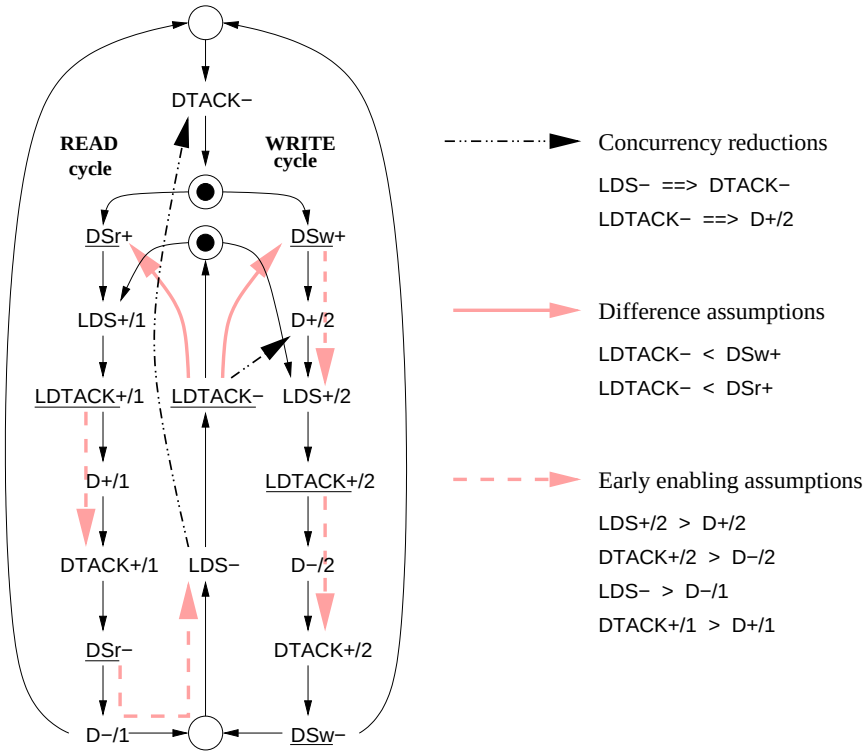


Figure 5.16 Timing assumptions for the synthesis of the VME bus controller.

tation. The concurrency reductions indicate additional causality relations enforced by the resulting logic, but do not correspond to actual timing assumptions. The two difference assumptions denote the assumed firing ordering of concurrent events of the environment due to the slow response time of the bus control logic. Hence, they can be considered as satisfied. On the other hand, the early enabling assumptions must be verified for the circuit to be correct. The validity of all the assumptions relies on the fact that the delay of D is smaller than that of LDS and DTACK. However, the gate driving D is much more complex than the gates driving the other two signals (see Figure 5.17), and the delay of D is expected to be bigger. As a consequence, two delay elements are added to the circuit in order to solve the above unrealistic timing assumptions. $d1$ should ensure that LDS+/2 > D+/2 and LDS− > D−/1 hold, whereas $d2$ should ensure that DTACK+/1 > D+/1 and DTACK+/2 > D−/2 hold.

With respect to an equivalent speed-independent implementation optimized to minimize the delay, the circuit of Figure 5.17 represents almost a 50% area reduction. Also, if compared to an speed-independent implementation optimized for area, the circuit with timing assumptions still requires about a 20% less area, and has a reduction in the response time of about another 30% [CKK+02].

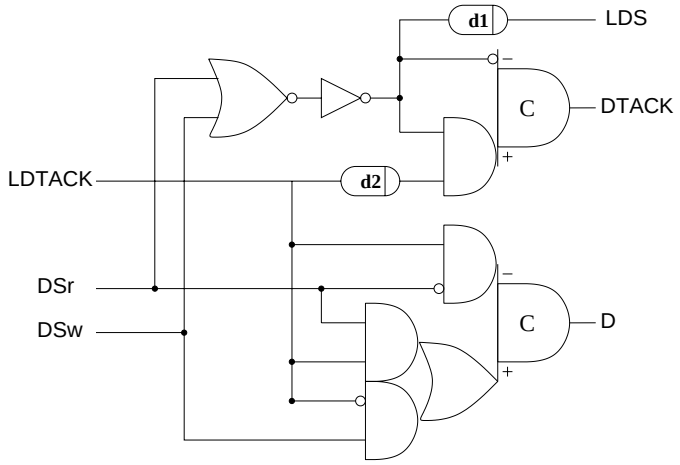


Figure 5.17 Implementation of the VME bus controller with timing assumptions. Generalized C-elements are used for signals DTACK and D.

5.4.3 Models and properties

In order to verify the correct behavior of the circuit implementation shown in the previous section, models for the specification and the circuit must be built including the appropriate delays. The (mirrored) specification will serve as the environment of the circuit using the same verification scheme than that depicted in Figure 5.8.

Specification and circuit models

The procedures presented in Sections 5.3.4 and 5.3.5 are used here for building the corresponding TRANSYT models for the STG of Figure 5.15 (c) and the circuit implementation of Figure 5.17, respectively. Figure 5.18 shows the resulting input files. The model for the circuit deserves a few comments.

Two internal signals have been added to the model, B_LDS and A_LDTACK, in order to represent the two internal nodes *before* the delay element d_1 and *after* the delay element $d2$, respectively. Thus, both B_LDS and A_LDTACK are fed to the gate driving DTACK according to the discussion in the previous section.

Fixed delays have been specified for each event in the model. For signals B_LDS, DTACK and D, the delay values have been taken directly from the estimations provided by PETRIFY after the synthesis process. Conversely, the delay values for LDS and A_LDTACK come from a simple manual timing analysis of the possible values for the delay elements d_1 and d_2, according to the discussion in the previous section. Recall that these delay elements are required to satisfy the timing assumptions where signal D must be faster than LDS and DTACK, respectively. In this sense, observe that the

```
#VME bus controller: specification
TS vme INTERLEAVED

INPUT VARS dsr dsw ldtack;
OUTPUT VARS dtack lds d;
INTERNAL VARS p0 p1 p2 p3 p4 p5 p2 p3 p6 p7 p8 p9 p10;
INTERNAL VARS p11 p12 p13 p14;

INPUT LABELS dsr dsw ldtack;
OUTPUT LABELS dtack lds d;

# Read cycle
EVENT dsr+ dsr
EQN TR p13 NS(p13)' p0' NS(p0) dsr' NS(dsr); END

EVENT lds+/1 lds
EQN TR p14 NS(p14)' p0 NS(p0)' p1' NS(p1) lds' NS(lds); END

EVENT ldtack+/1 ldtack
EQN TR p1 NS(p1)' p2' NS(p2) ldtack' NS(ldtack); END

EVENT d+/1 d
EQN TR p2 NS(p2)' p3' NS(p3) d' NS(d); END

EVENT dtack+/1 dtack
EQN TR p3 NS(p3)' p4' NS(p4) dtack' NS(dtack); END

EVENT dsr- dsr
EQN TR p4 NS(p4)' p5' NS(p5) dsr NS(dsr)'; END

EVENT d-/1 d
EQN TR p5 NS(p5)' p2' NS(p2) p3' NS(p3) d NS(d)'; END

# Write cycle
EVENT dsw+ dsw
EQN TR p13 NS(p13)' p6' NS(p6) dsw' NS(dsw); END

EVENT d+/2 d
EQN TR p6 NS(p6)' p7' NS(p7) d' NS(d); END

EVENT lds+/2 lds
EQN TR p14 NS(p14)' p7 NS(p7)' p8' NS(p8) lds' NS(lds); END

EVENT ldtack+/2 ldtack
EQN TR p8 NS(p8)' p9' NS(p9) ldtack' NS(ldtack); END

EVENT d-/2 d
EQN TR p9 NS(p9)' p10' NS(p10) d NS(d)'; END

EVENT dtack+/2 dtack
EQN TR p10 NS(p10)' p11' NS(p11) dtack' NS(dtack); END

EVENT dsw- dsw
EQN TR p2' NS(p2) p3' NS(p3) p11 NS(p11)' dsw NS(dsw)'; END

# Return to zero
EVENT lds- lds
EQN TR p2 NS(p2)' p12' NS(p12) lds NS(lds)'; END

EVENT dtack- dtack
EQN TR p13' NS(p13) p3 NS(p3)' dtack NS(dtack)'; END

EVENT ldtack- ldtack
EQN TR p14' NS(p14) p12 NS(p12)' ldtack NS(ldtack)'; END

#Initial state
EQN ISTATE CONJUNCTIVE
p0' p1' p2' p3' p4' p5' p2' p3' p6' p7' p8' p9' p10';
p11' p12' p13 p14 dsr' dsw' ldtack' dtack' lds' d';
END
```

```
#VME bus controller: implementation with
#relative timing assumptions
TS vme_net INTERLEAVED

INPUT VARS dsr dsw ldtack;
OUTPUT VARS dtack lds d;
INTERNAL VARS B_lds A_ldtack;

INPUT LABELS dsr dsw ldtack;
OUTPUT LABELS dtack lds d;
INTERNAL LABELS B_lds A_ldtack;

#Output signals
EVENT dtack- dtack
EQN TR B_lds' dtack NS(dtack)';
{DELAY: [ TYP= 16.0; ]} END

EVENT dtack+ dtack
EQN TR B_lds A_ldtack dtack' NS(dtack);
{DELAY: [ TYP= 23.96; ]} END

EVENT B_lds- B_lds
EQN TR dsr' dsw' B_lds NS(B_lds)';
{DELAY: [ TYP= 24.96; ]} END

EVENT B_lds+ B_lds
EQN TR (dsr + dsw) B_lds' NS(B_lds);
{DELAY: [ TYP= 19.58; ]} END

EVENT lds- lds
EQN TR B_lds' lds NS(lds)';
{DELAY: [ TYP= 5.0; ]} END

EVENT lds+ lds
EQN TR B_lds lds' NS(lds);
{DELAY: [ TYP= 12.0; ]} END

EVENT d- d
EQN TR ldtack dsr' d NS(d)';
{DELAY: [ TYP= 29.08; ]} END

EVENT d+ d
EQN TR (ldtack' dsw + ldtack dsr) d' NS(d);
{DELAY: [ TYP= 31.33; ]} END

#Input signals change freely
EVENT dsr- dsr
EQN TR dsr NS(dsr)';
{DELAY: [ TYP= 64.0; ]} END

EVENT dsr+ dsr
EQN TR dsr' NS(dsr);
{DELAY: [ TYP= 64.0; ]} END

EVENT dsw- dsw
EQN TR dsw NS(dsw)';
{DELAY: [ TYP= 64.0; ]} END

EVENT dsw+ dsw
EQN TR dsw' NS(dsw);
{DELAY: [ TYP= 64.0; ]} END

EVENT ldtack- ldtack
EQN TR ldtack NS(ldtack)';
{DELAY: [ TYP= 64.0; ]} END

EVENT ldtack+ ldtack
EQN TR ldtack' NS(ldtack);
{DELAY: [ TYP= 64.0; ]} END

EVENT A_ldtack- A_ldtack
EQN TR ldtack' A_ldtack NS(A_ldtack)';
{DELAY: [ TYP= 0.0; ]} END

EVENT A_ldtack+ A_ldtack
EQN TR ldtack A_ldtack' NS(A_ldtack);
{DELAY: [ TYP= 8.0; ]} END

#Initial state
EQN ISTATE
dsr' dsw' ldtack' dtack' lds' d' B_lds' A_ldtack';
END
```

Figure 5.18 TRANSYT input files for the specification (left) and circuit (right) of the VME bus controller.

falling transition of A_LDTACK is not delayed at all (fixed delay of 0 time units), since nothing in the above timing analysis required that. Finally, input events have a delay which is twice the delay of the slowest gate of the circuit. This corresponds to a *slow environment* assumption made by PETRIFY during the synthesis.

Properties

Despite of the above two models for the circuit and the specification, properties must be incorporated into the circuit model such that failure situations are raised when some of the relative timing assumptions are violated. Recall that we are not interested in checking the correct operation of the circuit according to the specification, but if the assumptions derived by the synthesis process are actually met in the circuit when the delay information is taken into account. Thus, let us analyze the three types of assumptions in order to derive appropriate failure conditions.

A difference timing assumption $\mathsf{a} < \mathsf{b}$ for two concurrent events a and b, denoted by $\mathsf{a} < \mathsf{b}$, indicates that a will always fire before b. As a consequence, any transition of b from a state where a is also enabled would violate the assumption. This situation can be characterized by the following condition, which should be incorporated into the model:

$$Fail(\mathsf{a} < \mathsf{b}) = EF(\mathsf{a}) \cdot FF(\mathsf{b})$$

This condition, associated to event b, will identify as a failure any potential firing of b in those states where a is also enabled.

In case of a simultaneity timing assumption, say $\mathsf{a} = \mathsf{b}@\mathsf{c}$, despite of the possibility of extending the enabling region of c to include states from the enabling region of a (b) if b (a) is the actual trigger of c, c is also assumed to fire always later than both a and b. Therefore, a failure condition similar to that for a difference timing assumption must be specified. However, in this case, c should not fire, not only in those states where a and b are simultaneously enabled, but also in those states where a (b) has already fired but b (a) is still enabled. The following failure condition captures this idea:

$$Fail(\mathsf{a} = \mathsf{b}@\mathsf{c}) = [\ EF(\mathsf{a}) \cdot EF(\mathsf{b})\ +\ EF(\mathsf{a}) \cdot SF(\mathsf{b})\ +\ SF(\mathsf{a}) \cdot EF(\mathsf{b})\ ]\ \cdot\ FF(\mathsf{c})$$

where $SF(\mathsf{a})$ characterizes the set of states where a has already fired and it is not enabled. More precisely, $SF(\mathsf{a}) = \exists_{NS(v_i)} TR^{-1}(\mathsf{a})$ for all current-state variables v_i. The condition covers all the states of the concurrency *diamond* for a and b.

Notice that if a (b) is the actual trigger of c, the condition of c firing after a (b) is automatically satisfied. However, the condition must be checked for both events since during the synthesis process the enabling region of c could have been extended to cover states where only b (a) has fired, but not a (b).

In case of a more general assumption involving groups of simultaneous events and groups of reference events, the corresponding failure condition can be still computed easily, although its formulation becomes a bit more complicated.

Finally, in an early enabling timing assumption such as $\mathsf{a} > \mathsf{b}$, despite of the possibility of extending the enabling region of b, the logic must ensure that a is slower than b, and hence it cannot fire until b has already fired. Again, this fact is captured by the following failure condition:

$$Fail(\mathsf{a} > \mathsf{b}) = EF(\mathsf{b}) \cdot FF(\mathsf{a})$$

which invalidates all transitions of a from states where b is also enabled.

In general, given an early enabling timing assumption such as $\mathsf{a} > \mathsf{b}_1 > \cdots > \mathsf{b}_n$, the failure condition would be such as:

$$Fail(\mathsf{a} > \mathsf{b}_1 > \cdots > \mathsf{b}_n) = (\bigcup_{i=1}^{n} EF(\mathsf{b}_i)) \cdot FF(\mathsf{a})$$

Apart from the previous failure conditions regarding the timing assumptions, input-output conformance of the circuit with respect to the environment must be checked too. Finally, we will enforce persistency conditions to all the non-input signals of the circuit. Although persistency is not mandatory in order to ensure a correct observable behavior of the circuit, it is always a desirable property in asynchronous circuits, where each signal transition (*e.g.* an undesired *glitch*) can be eventually propagated. Both failure conditions are automatically computed by TRANSYT when the closed system for verification is built (see Section 5.3.7).

5.4.4 Verification

Once the models for the specification STG and the circuit implementation are built, and the correctness properties are characterized using boolean equations, TRANSYT can be used to carry out the verification process in a similar way to that shown for the previous experiments.

The specification model (`vme`) is read first followed by the circuit implementation model (`vme_net`). The corresponding input files `vme.g.ts` and `vme.blif.delays.ts` are shown in Figure 5.18.

Then, the closed system for verification is built using the `uverif` command. The resulting system is called `C[M[vme]][vme_net]`, as the (C)losing of the (M)irrored specification `vme` and the circuit implementation `vme_net`. The failure conditions for persistency and input-output conformance are automatically computed by default. The closed system is traversed producing a total of 45 untimed states where 16 of them correspond to failure situations. What follows is an excerpt of the textual output produced by TRANSYT at the beginning of the verification session.

```
ts > read_ts vme.g.ts
. . .

ts > read_ts vme.blif.delays.ts
. . .

ts > uverif -HTML -Vclose -Vnotdestroy vme vme_net
. . .

ts > traverse
ts:: Traversing system 'C[M[vme]][vme_net]' using atom-partitioned TR.
ts:: End of Traversal with depth : 17
ts:: Final reached states: 45 Fail states: 16
ts:: Number of TR applications: 180 of which 68 useful
ts:: Time =     0.00 sec for the fix-point computation.
ts:: Time =     0.00 sec for the traverse.

ts > flatten -prj vme_flat C[M[vme]][vme_net]
ts:: Flattening system 'C[M[vme]][vme_net]' ...
ts:: Order computed visiting 25 states
ts:: Time =     0.00 sec for the order computation.
ts:: Done
ts:: Time =     0.05 sec for the flattening process.

ts > add_fail EFAIL (dsw+,dsw) EQN EF(ldtack-,ldtack)*FF(dsw+,dsw);
ts:: Adding fail condition for event 'dsw' of label 'dsw+'

ts > add_fail EFAIL (dsr+,dsr) EQN EF(ldtack-,ldtack)*FF(dsr+,dsr);
ts:: Adding fail condition for event 'dsr' of label 'dsr+'

ts > add_fail EFAIL (lds+,lds) EQN EF(d+,d)*FF(lds+,lds);
ts:: Adding fail condition for event 'lds' of label 'lds+'

ts > add_fail EFAIL (dtack+,dtack) EQN EF(d-,d)*FF(dtack+,dtack);
ts:: Adding fail condition for event 'dtack' of label 'dtack+'

ts > add_fail EFAIL (lds-,lds) EQN EF(d-,d)*FF(lds-,lds);
ts:: Adding fail condition for event 'lds' of label 'lds-'

ts > add_fail EFAIL (dtack+,dtack) EQN EF(d+,d)*FF(dtack+,dtack);
ts:: Adding fail condition for event 'dtack' of label 'dtack+'

ts > traverse
. . .
```

Next, the closed system is flattened to produce a new single monolithic system called `vme_flat`. In `vme_flat` each pair of synchronized labels of the interface of the circuit and the environment, is replaced by a single internal label with the same name. The transition relation for the events of the new label is formed by the product of the transition relations corresponding to the two synchronized events of the circuit and the specification. Regarding the delays, only one of the two original events can have delay information specified, which becomes the delay information of the new event. The result is a simpler system, without hierarchy, that keeps all the information required for verification. For detailed information on the `flatten` command refer to [PPb].

Then, failure conditions for the two difference timing assumptions and the four early enabling timing assumptions summarized in Figure 5.16 are added to the flat system for verification using the `add_fail` command. The failure conditions are derived as discussed

Signal	*Failure type*	*Initial*	*It.1*	*It.2*	*It.3*	*It.4*	*It.5*	*It.6*	*It.7*	*It.8*
DSw	LDTACK- < DSw+	1	1	1	1	1	1	-	-	-
DSr	LDTACK- < DSr+	1	1	1	1	-	-	-	-	-
LDS	LDS+/2 > D+/2	3	2	2	2	2	2	-	-	-
DTACK	DTACK+/2 > D-/2	1	1	1	1	1	1	1	-	-
LDS	LDS- > D-/1	1	1	1	1	1	-	-	-	-
DTACK	DTACK+/1 > D+/1	4	5	5	5	3	3	2	1	-
B_LDS	Ind. non-persistency to DTACK	2	2	2	2	-	-	-	-	-
A_LDTACK	Ind. non-persistency to DTACK	5	7	7	7	7	7	5	2	-
LDTACK	Ind. non-persistency to D	4	4	4	4	2	2	2	1	-
DTACK	Non-conformance	12	14	14	14	10	10	7	3	-
LDS	Non-conformance	5	4	4	4	4	3	-	-	-
D	Non-conformance	4	2	2	2	-	-	-	-	-

Table 5.3 Failure situations in the VME bus controller along the verification.

in Section 5.4.3. The system is traversed and the failure conditions are checked. The third column in Table 5.3 summarizes the failure situations detected in the untimed state space of the system.

The same options for the `tverif` command as those used in previous sections have been used for verification. The verification process needs eight refinements of the original untimed state space in order to prove the input-output conformance of the circuit with respect to the original specification, the absence of hazards in the circuit, and that all the relative timing assumptions are met. As we expected the delay information guarantees the correct operation of the circuit in the timed domain. Table 5.3 summarizes the evolution of the number of failure situations along the eight iterations. What follows is the textual output produced by TRANSYT for the first and the last iterations. The eight failure traces and the corresponding LzCESs are shown in Figure 5.19 and Figure 5.20.

```
ts > tverif -HTML -VwriteTrace1 -VwriteTES1 -AfilterTedges -AfailGuided -VfailTrace2

ts:: Starting verification iteration 1.
ts:: Searching a failure trace
ts:: Try to build timed ES from trace by "escape fail" criterion ... Succeeded
ts:: Time-compliance: escape fail.
ts:: Reachability analysis of the ES ... 4 markings visited
ts:: Composing GRC with the TS. 0+1 encoding vars required...
ts:: Timing constraints successfully applied.
ts:: Traversing the system....
ts:: Number of untimed states reached: 51
ts:: Checking fail conditions....
ts:: Number of fail states detected: 19
ts:: End of iteration 1.
```

```
. . .

ts:: Starting verification iteration 8.
ts:: Searching a failure trace
ts:: Try to build timed ES from trace by "escape fail" criterion ... Failed
ts:: Try to build timed ES from trace by "guided trace contradiction" criterion ... Succeeded
ts:: Time-compliance: contradict trace.
ts:: Reachability analysis of the ES ... 5 markings visited
ts:: Composing GRC with the TS. 0+1 encoding vars required...
ts:: Timing constraints successfully applied.
ts:: Traversing the system....
ts:: Number of untimed states reached: 29
ts:: Checking fail conditions....
ts:: No fail states detected.
ts:: All properties are satisfied.
ts:: Verification SUCCEEDED after 8 iterations.
ts:: End of iteration 8.
```

In the first iteration of the verification process, the failure trace of Figure 5.19 (a1) is generated. In the trace, LDS+ occurs after DSw+ but before D+, which causes a violation of the early enabling timing assumption LDS+/2 > D+/2. Moreover, this situation also corresponds to a violation of the input-output conformance according to the specification of the WRITE cycle. The failure would not exist in the timed domain if D+ is proved to be faster than LDS+ after DSw+ has fired. This is exactly what is discovered by the timing analysis on the events of the trace, and is captured by the LzCES of Figure 5.19 (b1). Actually, the LzCES expresses a more general fact, that no matter who triggers simultaneously D+ and B_LDS+, B_LDS+ will always occur before D+, and D+ will occur before LDS+. The refinement of the state space (see *It. 1* in Table 5.3) removes one state related to the violation of the timing assumption and one state related to the non-conformance of LDS+. As a side effect of the refinement, two failure states where D+ would cause a conformance violation are also removed. Moreover, certain splitting is produced in the states around failure conditions induced by DTACK and A_LDTACK, due to the need of distinguishing those traces which conform to the timing analysis and those which do not conform. Hence some of the failure situations are replicated, causing the total number of failure situations to keep constant or even to increase (see Table 5.3).

In the second iteration, the failure trace of Figure 5.19 (a2) is generated. It reflects a violation of the early enabling timing assumption DTACK+/1 > D+/1 since DTACK+ occurs before D+ in the trace. Moreover, this ordering of the events violates the input-output conformance with respect to the specification of the READ cycle. The timing analysis demonstrates that D+ actually fires before DTACK+, such that the failure behavior cannot occur in the timed domain. See the corresponding LzCES in Figure 5.19 (b2).

The third iteration resembles the previous one, but due to a violation of the timing assumption DTACK+/2 > D−/2, which also corresponds to a violation of the input-output conformance with respect to the specification of the WRITE cycle. See the failure trace in Figure 5.19 (a3) and the resulting LzCES in Figure 5.19 (b3), which proves that the failure trace does not exist in the timed domain, since D− actually fires before DTACK+.

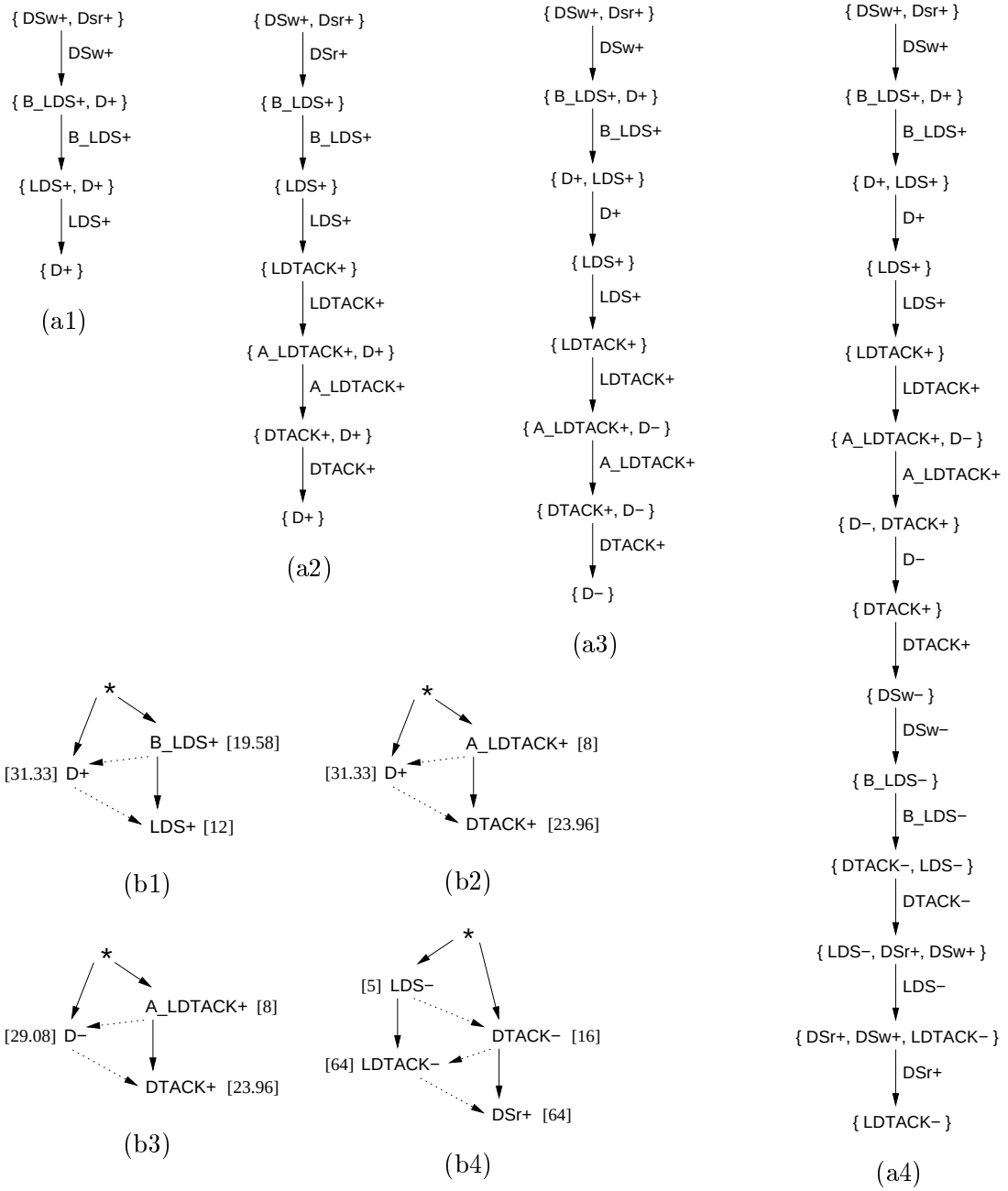


Figure 5.19 First four refinements for the verification of the VME bus controller: failure traces and corresponding LzCESs.

In the fourth and sixth iterations, the violations of the difference timing assumptions $\mathsf{LDTACK}- < \mathsf{DSr}+$ (see Figure 5.19 (a4)) and $\mathsf{LDTACK}- < \mathsf{DSw}+$ (see Figure 5.20 (a6)) are tackled, respectively. Both situations are proved non-existent in the timed domain,

thanks to the assumption of a slow environment with respect to the gates of the circuit. In this sense, notice the 64 delay units of the input signals LDTACK, DSr and DSw, against the smaller delay of the circuit gates in the LzCESs of Figures 5.19 (b4) and 5.20 (b6).

The fifth iteration deals with a situation similar to that of the first iteration, but related to the READ cycle. In the failure trace (see Figure 5.20 (a5)), LDS– fires before D– , thus violating the timing assumption LDS– > D–/1 and also the input-output conformance in the return-to-zero phase of the READ cycle. The LzCES obtained after the timing analysis (see Figure 5.20 (b5)) reflects the same causality relations than those of the first refinement, but for the negative transitions of the signals.

The last two iterations of the verification process, tackle the non-persistency of the rising transitions of signal DTACK induced by the fall of signal A_LDTACK. In particular, the seventh iteration deals with the potential hazard due to the disabling of DTACK+/2 in the WRITE cycle, whereas the last iteration deals with the disabling of DTACK+/1 in the READ cycle. See the corresponding failure traces in Figure 5.20 (a7) and Figure 5.20 (a8), respectively. The failures appear because B_LDS+ excites DTACK to rise too early. That is, before the falling transition of A_LDTACK occurs, which prevents DTACK from rising. Such rising transition of DTACK would cause an input-output conformance violation at the beginning of the operation (READ or WRITE) cycle. The timing analysis proves that A_LDTACK falls faster than B_LDS can rise and trigger DTACK (see the resulting LzCESs in Figure 5.20 (b7) and Figure 5.20 (b8)). Therefore, both failure situations are proved not to exist in the timed domain.

The overall verification process, which proves the absence of hazards, the input-output conformance with respect to the specification, and that all the relative timing assumptions hold, takes less than two seconds of CPU time in a $866MHz$ Pentium-III computer running Linux.

5.4.5 Discussion

This section has illustrated how the verification methodology presented in Chapter 4 can be used for the verification of relative timing assumptions in timed asynchronous circuits.

An overview of the different types of timing assumptions is provided and illustrated through the synthesis of the VME bus controller using the logic synthesis tool PETRIFY. The timing assumptions are modeled by means of boolean equations so that they can be checked using TRANSYT. In fact, the tool has been used to prove the correctness of all the timing assumptions used during the synthesis of the controller.

Currently, the boolean equations needed to model the relative timing assumptions must be specified by hand, either in some of the input files or through the command line of the tool. However, it would be desirable that the equations could be automatically derived by

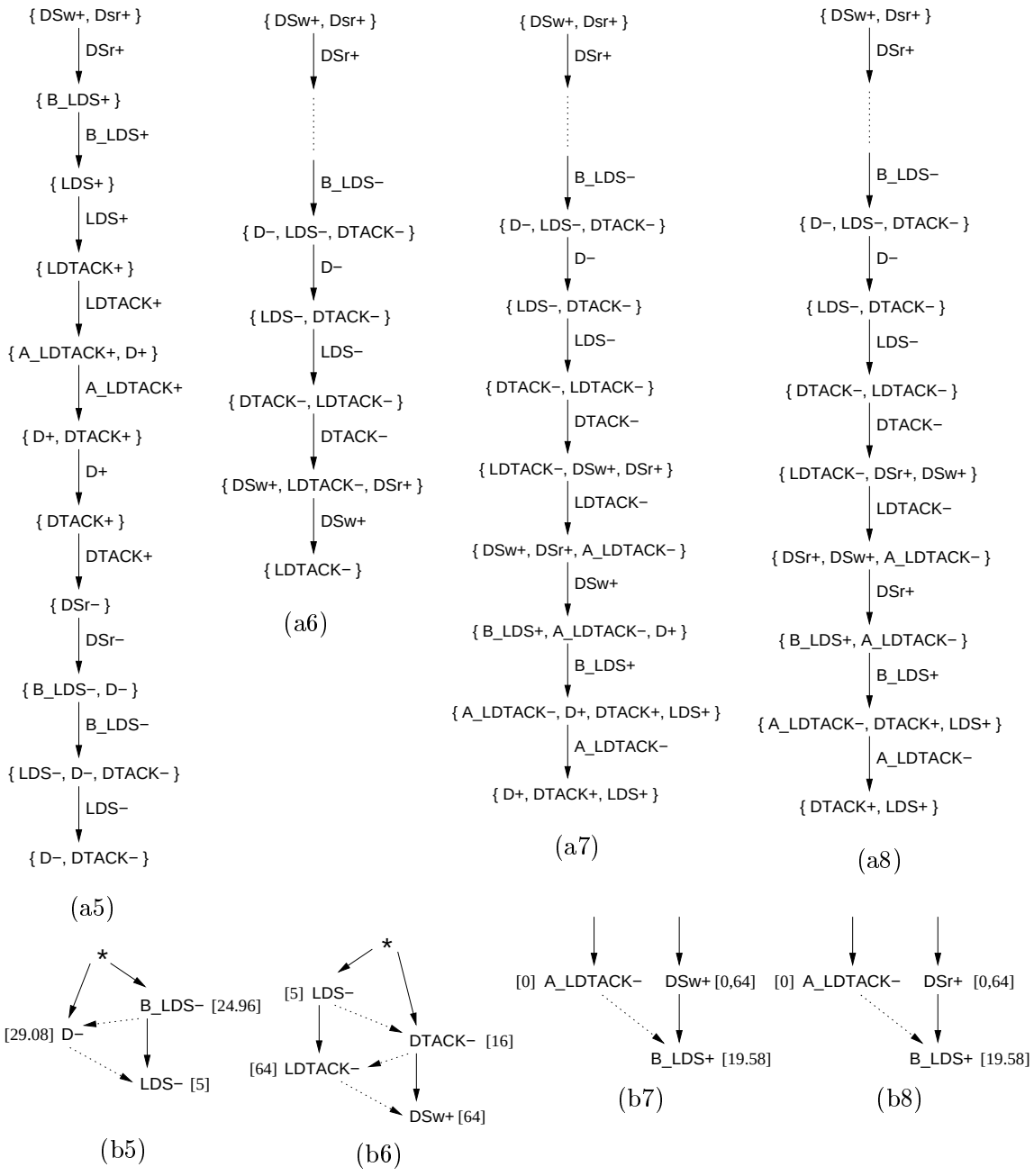


Figure 5.20 Last four refinements for the verification of the VME bus controller: failure traces and corresponding LzCESs. The four traces start with the same common prefix up to the firing of B_LDS− .

the tool, so that the user just needed to specify the actual timing assumptions, which is generally simpler and more intuitive. We hope this feature can be easily incorporated to TRANSYT in the near future.

5.5 Conclusions

In this chapter several features of the TRANSYT tool have been reviewed through the analysis of a number of experiments. The applicability of the verification methodology and its implementation in the tool has been proved by verifying two types of timed asynchronous circuits.

Some fundamentals on the symbolic representation of transition systems with boolean algebras have been provided first. Boolean functions are represented in TRANSYT using BDDs [Bry86]. It is well known that some intermediate computations along the reachability analysis, for example, can cause an exponential blow-up in the size of the data structures that handle the BDDs. Nevertheless, BDDs generally provide a compact and efficient representation.

The representation based on BDDs is only suitable for untimed state spaces or timed state spaces under the relative timing paradigm. If the exact timed state space of a system needed to be analyzed, other types of representations should be used (*e.g.* difference bound matrices). The computation of the exact timed state space could be a desirable feature of the tool. Although only for moderate-size systems, it would allow the direct comparison of verification methods for timed systems, for example.

A short introduction to the `tsif` format used by TRANSYT is also provided, covering the basics needed to understand the examples in the chapter. `tsif` is a simple text-based low-level format to describe binary-encoded untimed, lazy and timed transition systems. A system is modeled by specifying the boolean functions and relations that characterize the behavior of its events. The format also provides constructs for the modeling of hierarchy and communication mechanisms in modular concurrent systems. The expressiveness of the format has been illustrated along the chapter, by modeling timed PNs and STGs as well as digital circuits.

The iterative refinement of the verification methodology performs an unfolding of the state space in order to separate those traces which are enabling-compatible with the timing analysis and those traces which are not. Moreover, in some cases, in order to perform an accurate-enough timing analysis, the critical cycles of the system behavior must be *unrolled*. This yields to the necessity of a forward unfolding of certain regions of the state space. The number of such unfoldings depends basically in the delays associated to the events involved in the timing analysis. Hence, pathological cases which require an unmanageable number of unfoldings could be easily built. Our experience shows that such extreme cases do not arise often in practice, since none of the systems analyzed exhibit such undesired behavior.

Complex-gate decompositions are often required to build speed-independent circuits with conventional libraries of logic gates. Although the decomposition breaks the speed-independence property, the behavior of the resulting circuits can still be correct if appro-

priate delays are chosen for the gates. A general experimental set-up for the verification has been provided in which the required properties (input-output conformance and persistency) have been modeled in terms of boolean functions. A number of quasi-speed-independent asynchronous circuits have been verified showing that the amount of failure situations induced by the complex-gate decomposition is very high. Although this fact makes the verification process very hard, circuits with more than 10^6 untimed states have been verified in reasonable CPU times and with small memory requirements.

The use of timing assumptions in the synthesis of circuits often yields to significant reductions of the area requirements and improvements in the response time of the circuits. However, the timing assumptions must be proved correct in order to guarantee the proper behavior of the circuit. Relative timing assumptions are commonly used in the synthesis of asynchronous circuits, due to its conceptual simplicity but expressiveness power. The chapter reviews several types of relative timing assumptions and shows how they can be modeled in terms of boolean equations. The resulting properties can then be verified by TRANSYT. Currently, the equations must be specified by hand. We expect the tool could compute them automatically, thus allowing the designer to just deal with the actual timing assumptions, which are generally simpler and more intuitive.

All the above experiments have been performed without any specific of optimization for the different systems handled: timed PNs, timed STGs, digital circuits, etc. On the contrary, just a direct translation from the corresponding models into TTSs has been performed, and the generic algorithms of Chapter 4 have been used. For example, a possible source of optimization for circuits could have been the use of hierarchical verification techniques, based on the automatic abstraction of sets of gates in a circuit into complex ones.

On the other hand, an explosion in the size of the BDDs used to represent the transition relations is produced as the number of refinements increases (see Appendix B for details). The main reason for the explosion is the fact that each transition relation is split into several pieces for each condition of the enabling-compatible product. Each piece is manipulated and then the new transition relation is built by joining the different pieces. Although the enabling-compatible product provides a simple mechanism for the iterative refinement, it complicates the transition relations at each iteration. As a result, some large systems with complex causality relations cannot be verified due to memory requirements. To alleviate this problem, partitioned transition relations could be used. Partitions would correspond to the different pieces in which a transition relation is split for the composition. We plan to incorporate this improvement in the near future, which hopefully would allow us to handle larger and more complex systems for verification.

Despite of the aforementioned improvements, and considering the experimental nature of the TRANSYT tool, we want to remark that the results obtained look promising when compared to similar approaches for the verification of timed systems.