1

INTRODUCTION

The real value of formal methods lies not in their ability to eliminate doubt, but in their capacity to focus and circumscribe it.

—John Rushby - [Rus93]

Summary

This chapter introduces the generalities of the use of formal methods in the design and analysis of complex systems. Special attention is paid to the formal verification problem and its differentiation from simulation-based methods to prove the correctness of a design. The main approaches in the area of formal verification are also reviewed.

The chapter concludes with an overview of the motivations behind the work presented in the thesis, together with the main contributions.

1.1 Introduction

Continuous advances in electronics and software engineering have driven the increase in size and functionality of systems into unprecedented levels of complexity. As a consequence the probability of introducing design error has increased considerably. This fact, combined with the ubiquity of such systems in our current lives makes necessary the development of techniques that help to reduce the probability of failures.

Formal methods appear as a promising tool in such context. They bring the formalization and reasoning power of mathematics and logic into system design. Thus, they can help in systematizing the early specifications, providing appropriate abstract models of systems, and allowing the development of automatic techniques for the analysis of such models. Currently, tools for the automatic synthesis of circuits, or the formal verification of real-time systems, to cite some, exist both in academia and in industry. Moreover, they are gaining acceptance in this latter context.

This thesis relies on the use of formal methods to contribute to *the formal verification of systems whose correct behavior depends on timing issues.* Formal verification, although it is not a mainstream research topic, is getting increasing attention from industry due to several reasons.

In contrast to simulation, formal verification consists in building a mathematically-based proof that a system (implementation) behaves according to a given specification. For the check, all possible behaviors of the system must be taken into consideration, leading to the well-known state-explosion problem. In systems with a finite number of states, this problem is often alleviated by using symbolic techniques to implicitly enumerate all reachable states. Abstraction methods are also a common technique used to reduce the complexity of the model, by hiding those implementation details that are irrelevant to the properties begin verified.

The correctness of timed systems depends on the actual response times of the system and not only on its functional behavior. Therefore, time becomes an essential dimension in the verification problem and the complexity issue is exacerbated. For example, the problem of computing the language of a timed system modeled as a timed automata has been proved to be PSPACE-complete.

This thesis proposes a novel formal verification approach that extends the applicability of the conventional methods based on symbolic reachability analysis to timed systems. A major issue is the use of *relative timing*, which instead of considering exact delay separations, considers the *effect* of delays in a system in terms of relative ordering of events (*e.g.* a happens before b). This leads to the model of *lazy transition systems* which allows to represent the time domain in an efficient way, without increasing complexity when dealing with untimed systems.

1.2 Formal methods

In recent years, hardware and software systems have experimented a continuous growth in size and functionality. Due to this increase of complexity the probability to introduce design errors has also increased considerably. Often, such systems are used in applications where a failure has unacceptable consequences. Errors in electronic commerce systems, communication networks, traffic control systems, medical instruments, etc. may be the cause of important loss of money, time, or even human lives. One well-known example is the error found in the divider unit of the *Pentium* microprocessor in the fall of 1994, which correction and replacement costed *Intel Corp.* about 475 million US dollars [Pet97]. Another famous case is the launch failure of the *Ariane 5* rocket, which exploded 37 seconds after lit-off on June 1996 due to a software error that interpreted the flight altitude as a 16-bit integer when it was meant to be a 64-bit real [Lio96]. A long list of similar incidents related to failures in computers and electronic systems can be found in [Neu].

A major goal of engineering is to provide mechanisms that allow the construction of reliable systems despite of their complexity. One way of achieving this goal is to use *formal methods*, which involve mathematically-based languages, techniques and tools for the *modeling*, *specification*, *design* and *verification* of systems. The use of formal methods for the specification of a system requires certain precision, due to which ambiguities can be avoided along the design process. Also, the strict syntax and semantics provided by the formalisms often forces the designers to achieve a deep understanding of the system, in such a way that the relevant features are properly captured. On the other hand, the formal nature of the analysis on the resulting implementation provides an objective point of view about the degree of correctness of the system with respect to the original specification. Altogether provides a systematic approach for the correct construction of systems, which is suitable for its automatization by means of CAD/CAV tools.

Formal methods *per se* do not guarantee the construction of systems of better quality. In order to benefit from formal methods, appropriate formalisms for the specific application domain must be chosen. Also, if CAD/CAV tools are used one must remember that they can be buggy and comparisons between more than one tool can be mandatory. Moreover, the results of the formal analysis to check the correctness of the results obtained along the design, must be properly interpreted, and this is not always an easy or evident task.

Finally, certain techniques such as formal verification often cannot be applied to a system as a whole, due to size/complexity considerations. Hence, a compromise is usually required between the size of the system and an adequate level of abstraction which allows the successful application of formal methods. When too complex systems are involved such that a complete formalization becomes intractable, the verification approach is used only for the most critical parts of the system. Moreover, high-level reasoning techniques

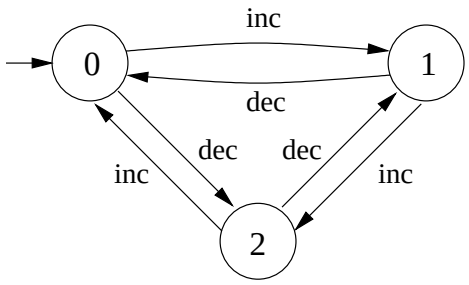

Figure 1.1 Automaton modeling a modulo 3 counter.

often come into play in order to allow dealing with the complexity issue at a higher level of abstraction.

1.2.1 Formal methods in the design process

Formal methods can be used along the complete design flow, from the early stages where the requirements are still being captured, until the latter stages where the system is yet implemented and full details are available.

Formal modeling consists in selecting a mathematical representation expressive enough to formalize a particular application, and powerful enough to explore and reason about the behavior of the system. This often requires the translation from a non-mathematical model, such as data-flow diagrams, pseudo-code, English text, etc., into formal models that include, among others: *process algebras* [BW90], *Petri nets* [Pet81], *transition systems* [Arn94], or *timed automata* [AD96]. The choice of one or another depends on the expressiveness power and the level of abstraction required for the application. Some of such modeling formalisms are discussed in more detail in Chapters 2 and 3.

EXAMPLE 1.1 *Informally, an* automaton *is a machine that evolves from one* state *to another under the action of* transitions*. For example, a module 3 counter can be modeled by an automaton with three states, one per counter value, and transitions that reflect the possible actions on the counter,* i.e. *increment or decrement its value (see Figure 1.1).*

Notice that details such as if the counter is implemented by a software program or by a sequential circuit, for example, are abstracted away. Therefore, if the counter is finally implemented as a circuit, with a so abstract model, it will be impossible to reason about facts like if there is a short-circuit in a stack of transistors implementing a flip-flop, for example. However, the model may suffice if we want to check, for instance, if the counting process gets stuck after counting up to 2. ■ 1.1

Formal specification covers the process of describing a system and its desired properties. The specification describes how the system is expected to work in the given environment.

The specification avoids unnecessary details and provides a general-enough description that can be adapted to system changes later on. This requires the use of a language with a mathematically defined syntax and semantics, which must be related to the chosen formal model. The kind of properties specified may include functional behavior, interface, timing behavior, performance, etc. Formal specifications may serve as a sound communication mechanism between the people involved along the life cycle of a system: customers, designers, implementer, testers, and so on. Examples of formal specification languages include Z [Spi88], CCS [Mil89], CSP [Hoa85], temporal logic [Pnu81, CE81], LOTOS [ISO89], etc. Some of them are more focused on the system description, whereas others are more suitable for the specification of properties. A good survey of the successful use of formal specifications in a variety of areas can be found in [CW96].

Example 1.1 (cont.) *We may want to formulate a property that states that the modulo 3 counter of Figure 1.1 is free of deadlocks,* i.e. *it cannot end up stuck in any state. For example, such property is generally stated using the temporal logic CTL [CE81] by the formula* $AG\ EX$ true *which can be read as: "whatever the state reached may be (*AG*), there will exist an immediate successor state (*EX true*)". See Section 3.4.1 for more details on CTL.*

■ 1.1

Formal analysis refers to techniques that can be used to calculate and explore the system behavior, and to verify properties of it. The main topic in this area of research is *formal verification*, in which two main approaches have traditionally coexisted. Namely, those approaches based in proof-theoretic automated deduction, such as *theorem proving* [GMW79]; and those based in finite state methods and state exploration, such as *model checking* [CGP00]. More details on both approaches are given in Section 1.3.2.

Example 1.1 (cont.) *Using the appropriate mechanism, for example model-checking, it can be demonstrated that in the automata of Figure 1.1 which models a modulo 3 counter, every state satisfies the deadlock freeness property stated above.*

■ 1.1

A typical design flow for concurrent systems consists of an iterative process in which both CAD/CAV tools and also the designer are involved. First, the process of verifying the specification is aimed at checking whether the system will not exhibit undesired behaviors. Then, the synthesis process generates an implementation of the system, using the primitives provided by some sort of *library*. The library may consist of different objects depending on the particular application: a set of logic gates to implement digital circuits, a set of assembler instructions of a given microprocessor, etc. Once the implementation is generated, the designer may want to prove if the *functionality* of the implementation is equivalent to that initially specified, under certain equivalence criteria. Despite of the

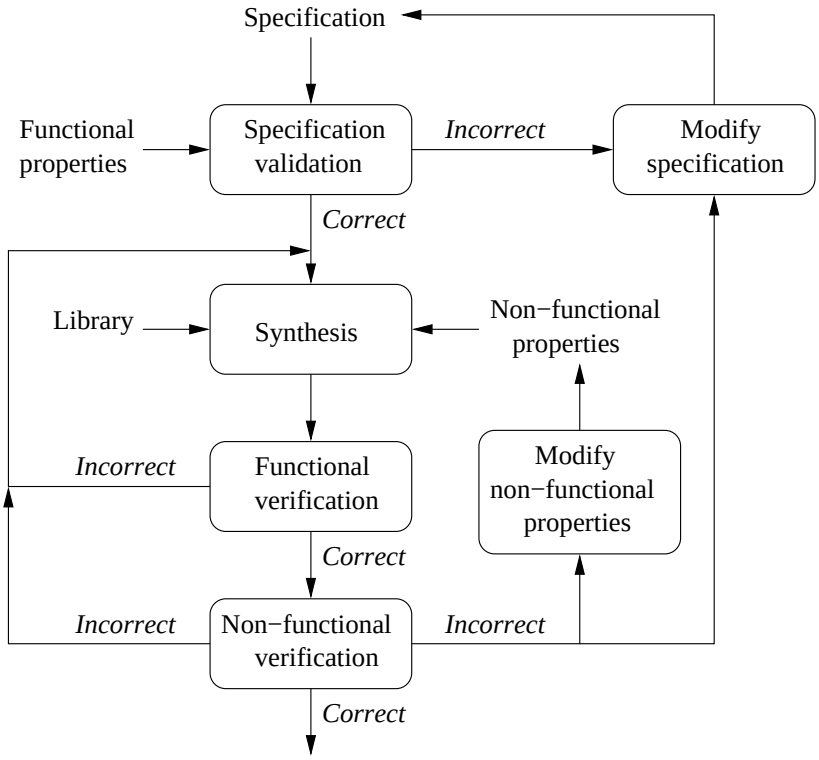

Figure 1.2 Iterative design flow.

equivalence with the specification, it is often required that the system satisfies other *non-functional* constraints. Requirement such as a particular response time, a limit in the amount of memory used by a program, etc. may be desirable at this point. In order to finally obtain a correct system that satisfies both functional and non-functional constraints, it may be required to modify the specification, resynthesizing the system, etc. Therefore, the whole process leads to an iterative design flow as that depicted in Figure 1.2.

The contributions of this thesis are focused on the verification of functional properties. The properties may depend on timing aspects of the system and/or of the environment in which it operates. The developed methods can be applied also to the verification of general functional properties and the validation of certain aspects of the specifications.

1.3 Formal verification

Although more insightful details about the verification of timed systems are given in Chapter 3, this section provides some fundamentals about the formal verification problem in general.

1.3.1 Verification versus simulation

In order to check if a system implementation behaves according to its specification or satisfies certain properties, all possible behaviors of the system must be taken into consideration.

Nowadays, the most common approach for design verification is still computer-aided simulation. In simulation, input patterns are created which reflect typical or critical execution traces, the implementation is excited with such patterns, and the output is compared to that expected according to the specification. In case of sequential circuits, for example, all possible input combinations in every possible state must be analyzed. Since the number of required input patterns increases exponentially with the number of inputs and the number of states of the circuit, the approach is impractical even for circuits of moderate size. In consequence, the number of input patterns must be reduced and some design errors may remain undetected. Although simulation is the most intuitive approach for checking the correct behavior of a system, and is important for discovering failures quickly, it is not satisfactory when too complex designs need to be extensively analyzed.

An alternative to simulation is formal verification, which consists in building a mathematically-based proof that a system (implementation) behaves according to a given specification. Often, some simulation-based methods are also called "verification". To distinguish them from verification, the prefix "formal" is used to differentiate between both methods. The following example, taken from [Gor89], illustrates the fundamental difference between simulation and formal verification.

Example 1.2 *The goal is to show that the expression* $(x+1)^2 = x^2 + 2x + 1$ *holds,* i.e. *that both sides of the equation lead to the same result for all possible input values:*

A simulation based approach would check the equation using concrete values for x *as:*

x	$(x+1)^2$	x^2+2x+1
0	1	1
1	4	4
2	9	9
3	16	16
9	100	100
67	4624	4624
...	...	...

However, as long as the equality must hold for all numbers – not even restricted to the subset of natural numbers as in the above table – simulation is not capable of establishing the validity of the equation.

In contrast, a formal mathematical proof can do exactly this by applying mathematical transformation rules as it is shown in the following table:

1.	$(x+1)^2 = (x+1)(x+1)$	*definition of square*
2.	$(x+1)(x+1) = (x+1)x + (x+1)1$	*definition of distributivity*
3.	$(x+1)^2 = (x+1)x + (x+1)1$	*substitution of 2. in 1.*
4.	$(x+1)1 = x+1$	*neutral element 1*
5.	$(x+1)x = xx + 1x$	*distributivity*
6.	$(x+1)^2 = xx + 1x + x + 1$	*substitution of 4. and 5. in 3.*
7.	$1x = x$	*neutral element 1*
8.	$(x+1)^2 = xx + x + x + 1$	*substitution of 7. in 6.*
9.	$xx = x^2$	*definition of square*
10.	$x + x = 2x$	*definition of* $2x$
11.	$(x+1)^2 = x^2 + 2x + 1$	*substitution of 9. and 10. in 8.*

■ 1.2

In simulation, a complete model of the system is used, however only a partial verification is achievable. In contrast, in formal verification a partial model of the appropriate abstraction level is used, and a complete proof can be obtained provided that model. As a consequence, it is often the case that both approaches are combined. Fast simulation may be used to discover simple or expected failures in the early stages of a design, whereas formal verification may be used to discover unusual or exotic failures in critical parts of the system.

Finally, recall that in general, in order to be able to perform a formal analysis and even be able to automate it, the specification, the implementation and the correctness relation must be in a form which allows a rigorous formal treatment.

1.3.2 Main approaches to formal verification

As cited above, there are two major approaches to formal verification, namely theorem proving and model checking. This section gives some general details on how these approaches work.

In theorem proving, both the system (implementation) and the properties (specification) are expressed as formulas in some mathematical logic. The logic is based in a set of axioms and provides a set of inference rules. Then, the approach consists in finding a proof of a given correctness relation between the implementation and the specification, following the axioms and the inference rules of the logic (see Figure 1.3). Therefore, it can deal directly with infinite state spaces, since no explicit state space exploration is required. However, the high complexity of the algorithms involved makes theorem proving applicable in practice only to moderate size or to particularly well-suited systems.

The proofs can be constructed automatically, although often require manual interaction of experts on the underlying logic and proof mechanisms. As a consequence, the process may become slow and often error-prone. In contrast, in the process of building the proof, the user achieves deep knowledge of the details of the system and the properties it must satisfy.

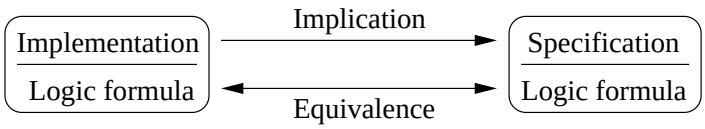

Figure 1.3 The theorem proving approach.

Theorem proving methods have not yet achieved widespread use outside universities. However, there are a number of representative theorem provers, such as HOL [GM93] or PVS [ORSS94], which have been used successfully in several domains.

Model checking relies on building a finite model of a system and checking that the desired property holds in that model (see Figure 1.4). In *temporal model checking* [CE81, QS81] specifications are expressed in a *temporal logic* [Pnu81] and systems are modeled as finite state transition systems. An efficient search procedure is used to *check* if the transition system is a *model* for the specification. Other approaches use automata for both the specification and the system model. Then, the system is compared to the specification to determine if its behavior *conforms* to that of the specification. Different notions of conformance have been explored, such as *language inclusion* [Kur94], *refinements* [CPS93, Ros94], *observational equivalence* [CPS93], etc.

In contrast to theorem proving, model checking techniques are completely automatic. The check is performed by an exhaustive state space exploration which requires the use of specific algorithms and data structures to handle large state spaces. When the model checking algorithms fail to prove a given property, they are able to produce a counterexample, which indicates how is possible for the system to violate the specification. Counterexamples often correspond to subtle design errors and therefore can be used for debugging the system.

The main drawback of model checking is the so-called *state explosion* problem, which refers to the exponential blow up of the number of states of a system, such that it exceeds the available resources of a computer. Several approaches have been used so far to alleviate this problem. These include low-level techniques such as: symbolic representations of the state space [McM93] using binary decision diagrams (BDDs) [Bry86], partial order reductions [KP88, Pel96], etc. But also techniques that work at a higher level, such as: *assume guarantee* reasoning to exploit the modularity of the system [Pnu84], *abstractions* that remove irrelevant details for a particular analysis [Mel88], use of *symmetries* [CJEF96, ES96] and *induction* [BSV94, VK98] for systems with certain degree of regularity such as pipelines, etc. However, except for certain well-suited examples, only systems with about one hundred state variable can be handled as a whole. Clearly, this is far from the sizes of the current integrated circuits and microprocessors, for example. The verification problem

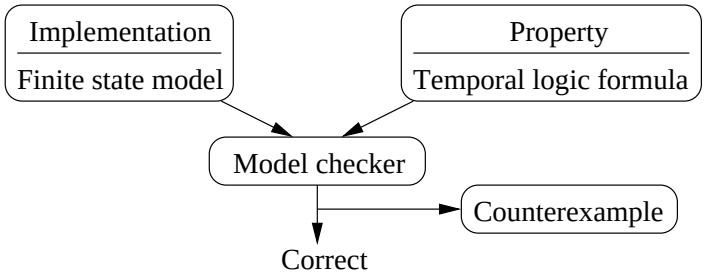

Figure 1.4 The model checking approach.

becomes much more difficult in the case of timed systems, because timing information must be taken into account when building the state space (see Chapter 3).

Several successful model checkers can be found nowadays, including: SMV [McM93] which was the first model checker to use BDDs allowing symbolic analysis; SPIN [Hol97] that takes advantage of partial orders for the verification of distributed algorithms; HSIS [ABC+94] which combines model checking with language inclusion; KRONOS [Yov97] and UPAAL [BLL+95] for the verification of real-time systems using timed automata; COSPAN [AK95] which verifies real-time systems by checking inclusion between ω-automata; HYTECH [HHWT97] which allows to perform parametrized analysis, *i.e.* to determine the values of design parameters for which a linear hybrid automaton satisfies a temporal-logic requirement; and MOCHA [AHM+98] for modular verification of heterogeneous systems modeled by reactive modules.

The success of all these tools developed at universities, combined with the intensified need for formal methods has attracted the interest of the industry. As a result, internal tools have been developed (inside Motorola, Intel, IBM, etc.) and some commercial tools are also available (FORMALCHECK from Lucent Technologies, RULEBASE from IBM, INSIGHT from Crysalis Design, etc.).

Finally recall that there is no ideal verification approach which is powerful enough for all proof tasks and which, at the same time, allows completely automated proofs. Moreover, the choice of the best suited approach strongly depends on the actual verification problem.

1.4 Formal verification of timed systems

In systems whose correctness depends on a proper timing a quantitative notion of time must be incorporated both into the system models and also into the specification formalisms. Since time constitutes an additional source of complexity, the way it is represented has a crucial impact on the size of the resulting timed state space. Two main approaches exist for that purpose: *discrete-time* and *continuous-time*.

Formalisms based on the *discrete-time* notion map time onto the integer domain. They require to discretize time by choosing a fixed *time quantum*, so that the separation of two events in the timed domain is always a multiple of such quantum. In *continuous-time* models a non-negative real value is associated to each event of the system and to each reachable state, so that the exact bounds on the actual delays between the events can be expressed. The main advantage of discrete-time is that the timing analysis and timed state space exploration techniques are generally simpler than their counterparts for continuous-time. The main drawback is that determining the time quantum *a priori* may not be easy and therefore may compromise the accuracy of the model.

It has been mentioned above that the verification of concurrent systems typically suffers from the well known state-explosion problem. In systems with a finite number of states, this problem is often alleviated by using symbolic techniques to implicitly enumerate all reachable states [Bur92]. Abstraction methods are also a common technique used to reduce the complexity of the model, by hiding those implementation details that are irrelevant to the properties begin verified [Mel88]. However, when time becomes an essential dimension in the verification problem, complexity is drastically increased. The correctness of timed systems depends on the actual values of event delays and not only on its functional behavior. Typically, timing behavior is specified by a set of delays that determine the time duration between the initiation and the completion of an event. This is the valid model for the gates in a circuit, for example, in which gate delays denote the time between the enabledness of the gate and the actual change at the output.

Most approaches for the verification of timed systems rely on the construction of the timed reachability space. The problem is PSPACE-hard [AD94] since the number of timed states is infinite. Therefore, typical model checking algorithms are no longer applicable. Also, in order to overcome the complexity, finite representations of the timed state space must be provided. Although many techniques have been devised to alleviate the state-explosion problem and the additional complexity due to the time dimension, spectacular improvements in the resulting representations and algorithms are unlikely. In consequence, other high-level techniques (*e.g.* abstraction, compositional reasoning, induction, etc.) appear as the more promising ones for future developments in this area of research. Nevertheless, several methodologies and tools exist for the verification of timed systems.

1.5 Overview of the contributions

This thesis proposes a novel verification approach that extends the applicability of the conventional methods based on symbolic reachability analysis to timed systems. The approach is based on two fundamental facts:

- The observation that the set of traces of a transition system can be covered by a set of marked graphs. This reduces the verification problem to that of: the timing analysis over small sets of events from which timing constraints that prove the correctness or incorrectness of a system can be derived; and the incorporation of such constraints into the system along an incremental refinement process.

- The use of *relative timing* [SGR99] to represent the time domain in an efficient way. When considering precise delay bounds in timed systems, the complexity blow-up often causes synthesis and verification to become intractable problems, even for small systems. Instead, relative timing considers the *effect* of delays in a system in terms of relative ordering of events (*e.g.* a happens before b).

The verification approach can be briefly summarized as follows. Rather than calculating the exact timed state space, the verification approach performs an *off-line* timing analysis on a set of event structures [NPW81] that covers the traces leading to system failures. This timing analysis is efficiently performed by using McMillan and Dill's algorithm [MD92]. The resulting timing constraints are incorporated to the system in the form of relative timing information along a series of iterative refinements of the original untimed state space. Finally, if some of the traces leading to failure situations cannot be proved to be timing-inconsistent, then the system is incorrect and the failure trace is a counterexample.

Due to the incremental incorporation of timing information along the verification, our approach works with over-approximations of the actual timed state space of the system. Being the completely untimed state space used as starting point the roughest approximation possible. This fact allows the efficient verification of safety properties but makes impossible the verification of liveness properties, for example. For safety properties, it is enough to prove that no "undesired" situations (states) are reachable by the system. If "undesired" states do not appear in the over-approximations, they will neither appear in the exact timed state space, but not vice versa. Therefore, the verification can produce "false-negatives" but never "false-positives", *i.e.* it is conservative for safety properties. On the contrary, for liveness properties it must be proved that some "desired" situation is actually reachable. For that kind of proof, the exact timed state space (or an under-approximation for conservativeness) must be computed.

The idea of using event structures for timing analysis was already proposed in [KBS02]. However, no algorithm was presented that can handle a general class of transition systems for verification.

The approach presented here, not only verifies the correctness of the system with respect to a set of given properties, but also provides as back-annotation a set of timing constraints sufficient to prove correctness. This information is crucial in frameworks in which synthesis

and verification are iteratively invoked to design systems that must meet functional and non-functional constraints.

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

The resulting verification algorithms have been fully implemented in the CAV tool TRANSYT. The applicability of the approach and the functionality of the tool have been proved by verifying a number of timed asynchronous circuits [PCKP00].

The work on verification is completed by tackling the verification of a complex timed system, namely the IPCMOS architecture [SRC+00]. The IPCMOS circuit is a controller for asynchronous scalable architectures (such as pipelines, meshes, etc.) that can operate at frequencies of up to 4GHz thanks to a pulse-driven protocol for the communication with the environment. The correctness of the system highly depends on the delays of the internal gates and the environment. The verification has been carried out by combining the core verification algorithm outlined above, together with the use of assume-guarantee reasoning [Pnu84] to perform a hierarchical verification by means of abstractions [Mel88], and the use of mathematical induction to prove the correctness of infinite-state systems. As a result, it has been proved the correctness of an IPCMOS pipeline regardless of the number of stages that conform it [PCSP02].

The key features of the presented work on the verification of timed systems can be summarized by the following topics:

- The use of relative timing allows to avoid the computation of the exact timed state space of the system, which is a common practice of model checking methods for timed systems. Instead in the proposed approach, the timed behavior of events is captured by means of partial orders that represent simple facts as if an event happens before another, *i.e.* relative temporal relations.

- As a consequence of the previous topic, the state space of the system can be represented and managed using symbolic methods with proved efficiency such as BDDs. This allows a natural extension of traditional symbolic model checking techniques for untimed systems into the timed systems domain of application.

- No global timing analysis is done for the whole system. Instead, the timing analysis is performed locally for a set of failure traces that are covered by a marked graph. Therefore, only a subset of the events of the system is involved and the timing analysis can be carried out very efficiently.

- Although timed systems provide delays for all the events in the system, often many of the constraints imposed by such delays are not required for the correctness of

the system. Because of the iterative nature of the proposed verification approach, timing information is only considered in an *on-demand* basis, as long as it is required to prove the infeasibility in the timed domain of a set of failure traces.

- As a result of the previous topic, the untimed state space of the system is refined incrementally as long as new timing information is taken into account. This incremental nature of the approach provides a good way to obtain at least partial results even on systems for which complete solutions could be too complex to compute.
- The proposed verification approach not only proves or disproves the correctness of the system with respect to a set of properties. If the system is correct the algorithm provides the set of relative timing relations used for the proof. Those relations constitute a set of sufficient timing constraints that guarantee the correctness of the system. On the other hand, if the system is incorrect, a counterexample failure trace is provided. The most important aspect of all this feedback is that can be used as valuable back-annotation information along a design process.
- The verification approach has been fully implemented into the CAV tool TRANSYT. The tool has proved its functionality as well as the validity of the overall verification approach, by verifying a set of different types of timed asynchronous circuits with up to more than 10^6 untimed states.
- Compositional verification methods have been combined with our basic verification approach in order to tackle the size/complexity issues involved in the verification of complex timed systems. Thus, abstractions, assume-guarantee reasoning and mathematical induction have been used to prove the correctness of a scalable pipelined architecture.

1.6 Structure of the thesis

The rest of this document is organized as follows.

Chapter 2 introduces the fundamentals of the different formal models used in the subsequent chapters. Models such as Petri nets and several types of transition systems are described, together with some of their basic properties.

Chapter 3 introduces general background on the formal verification of timed systems and reviews the significant previous work on this area of research. Special attention is paid to the verification using timed automata, since the currently most successful methods and tools are based on them.

In Chapter 4 the main theoretical aspects of the relative timing-based verification approach for timed system, presented in this thesis are introduced. Examples of the applicability of the developed methodology are shown in Chapter 5, where different flavors of asynchronous circuits are verified. Chapter 6 presents a complex case study in which

the basic verification approach is combined with assume-guarantee reasoning by means of abstractions, and mathematical induction. The result is the successful verification the IPCMOS architecture.

Chapter 7 summarizes the conclusions and contributions of this work and outlines some open areas for future research.

Additionally, some appendixes are included.

First, Appendix A analyzes the problem of determining the time separation between the events of a system. An algorithm for timing analysis on acyclic graphs is described in detail.

Then, Appendix B provides implementation details of one of the key parts of the verification methodology.

And finally, Appendix C introduces the commands in the TRANSYT tool related to the verification of timed systems, which implement the presented verification approach.