RELATIVE TIMING BASED VERIFICATION OF CONCURRENT SYSTEMS

MARCO ANTONIO PEÑA BASURTO
Licenciat en Informàtica, Technical University of Catalonia, Barcelona, Spain 1993
Licenciat amb grau en Informàtica, Technical University of Catalonia, Barcelona, Spain 1995

Department of Computer Architecture
Technical University of Catalonia
Barcelona (Spain), February, 2003

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor en Informàtica
To Nuria and Pau
Para mí no hay emoción comparable a la que produce la actividad creadora, tanto en ciencia como en arte, literatura u otras ocupaciones del intelecto humano. Mi mensaje, dirigido sobre todo a la juventud, es que si sienten inclinación por la ciencia, la sigan, pues no dejará de proporcionarles satisfacciones inigualables. Cierto es que abundan los momentos de desaliento y frustración, pero éstos se olvidan pronto, mientras que las satisfacciones no se olvidan jamás.

—Severo Ochoa

It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and hardware design, is unsound and clumsy because the people who do it do not have any clear understanding of the fundamental principles underlying their work. Most of the abstract mathematics and theoretical work is sterile because it has no contact with the real computing.

—Christopher Strachey
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xvii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>xix</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Formal methods</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Formal methods in the design process</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Formal verification</td>
<td>6</td>
</tr>
<tr>
<td>1.3.1 Verification versus simulation</td>
<td>7</td>
</tr>
<tr>
<td>1.3.2 Main approaches to formal verification</td>
<td>8</td>
</tr>
<tr>
<td>1.4 Formal verification of timed systems</td>
<td>10</td>
</tr>
<tr>
<td>1.5 Overview of the contributions</td>
<td>11</td>
</tr>
<tr>
<td>1.6 Structure of the thesis</td>
<td>14</td>
</tr>
<tr>
<td>2 MODELS FOR CONCURRENT SYSTEMS</td>
<td>17</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>18</td>
</tr>
<tr>
<td>2.2 Transition systems</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Timed transition systems</td>
<td>21</td>
</tr>
<tr>
<td>2.4 Lazy transition systems</td>
<td>24</td>
</tr>
<tr>
<td>2.5 Petri nets</td>
<td>27</td>
</tr>
<tr>
<td>2.5.1 Labeled Petri nets</td>
<td>31</td>
</tr>
<tr>
<td>2.6 Conclusions</td>
<td>33</td>
</tr>
<tr>
<td>3 VERIFICATION OF TIMED SYSTEMS</td>
<td>35</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>36</td>
</tr>
<tr>
<td>3.2 Quantitative timing information</td>
<td>37</td>
</tr>
<tr>
<td>3.3 Timed automata</td>
<td>39</td>
</tr>
<tr>
<td>3.4 Timed specifications</td>
<td>42</td>
</tr>
<tr>
<td>3.4.1 Temporal logic</td>
<td>42</td>
</tr>
<tr>
<td>3.4.2 Timed temporal logic</td>
<td>45</td>
</tr>
</tbody>
</table>
3.5 Verification of timed systems
  3.5.1 Clock regions
  3.5.2 Region automata
  3.5.3 Zone automata
  3.5.4 Difference-bound matrices
  3.5.5 Discussion
3.6 Petri net-based methods
3.7 Conclusions

4 VERIFICATION WITH RELATIVE TIMING
  4.1 Introduction
    4.1.1 Relative Timing
  4.2 Overview
  4.3 Trace semantics
    4.3.1 Traces and languages
    4.3.2 Trace-based verification
    4.3.3 Enabling compatibility
  4.4 Event structures
    4.4.1 Timing analysis on event structures
  4.5 Enabling-compatible product
    4.5.1 State-based representation of a CES
    4.5.2 Refining the reachability space by timing constraints
  4.6 Verification methodology
    4.6.1 Iterative refinement
    4.6.2 Off-line timing analysis of failures
    4.6.3 Incorporation of relative timing constraints
    4.6.4 Back-annotation
    4.6.5 Correctness
    4.6.6 Convergence
  4.7 Conclusions

5 EXPERIMENTAL RESULTS
  5.1 A brief introduction to TRANSYT
    5.1.1 Representation of LzTs with boolean algebras
    5.1.2 TRANSYT input format
  5.2 An example with forward unfolding
    5.2.1 Model of a timed PN
    5.2.2 Verification
    5.2.3 Discussion
  5.3 Verification of complex-gate decompositions in speed-independent circuits
    5.3.1 Speed-independent circuits
    5.3.2 Experimental set-up
    5.3.3 The sbuf-read-ctl controller
5.3.4 Model of an STG 115
5.3.5 Model of the circuit 117
5.3.6 Specification of properties 118
5.3.7 Verification 121
5.3.8 Results and discussion 125
5.4 Verification of relative timing assumptions 127
  5.4.1 Synthesis of asynchronous circuits with relative timing assumptions 128
  5.4.2 The VME bus controller 131
  5.4.3 Models and properties 134
  5.4.4 Verification 137
  5.4.5 Discussion 142
5.5 Conclusions 144

6 COMPOSITIONAL VERIFICATION 147
  6.1 Introduction 148
  6.2 The IPCMOS architecture 149
    6.2.1 IPCMOS pipelines 149
    6.2.2 Strobe circuit 152
    6.2.3 Reset circuit 153
    6.2.4 Valid circuit 155
    6.2.5 The environment modules 155
    6.2.6 About complexity 156
  6.3 Compositional verification 156
    6.3.1 Framework 158
  6.4 Verification of IPCMOS pipelines 159
    6.4.1 Verification strategy 159
    6.4.2 Abstractions 161
    6.4.3 Assume-guarantee verification 162
  6.5 Verification of a stage 165
    6.5.1 Modeling CMOS circuits 165
    6.5.2 Modeling IPCMOS circuits 169
    6.5.3 Verification results 170
  6.6 Conclusions 173

7 CONCLUSIONS 175
  7.1 Introduction 176
  7.2 Contributions 177
  7.3 Future research 179

A TIMING ANALYSIS 183
  A.1 Introduction 184
  A.2 Timing analysis on acyclic graphs 186
ON THE ENABLING-COMPATIBLE PRODUCT

B.1 Enabling-compatible product
  B.1.1 State-based representation of a CES
  B.1.2 Refining the reachability space by timing constraints

B.2 Symbolic representation
  B.2.1 Encoding of a LzTS
  B.2.2 Encoding the state space of a LzCES

B.3 Computation of the new transition relations
  B.3.1 Transitions entering the timed domain
  B.3.2 Staying inside the timed domain: no synchronization
  B.3.3 Staying inside the timed domain: synchronization
  B.3.4 Transitions re-entering the timed domain
  B.3.5 Exiting or staying outside the timed domain
  B.3.6 New transition relation
  B.3.7 Lazy events
  B.3.8 Initial state

C VERIFICATION-RELATED COMMANDS

C.1 Failure analysis
  C.1.1 The add_fail command
  C.1.2 The check_fails command
  C.1.3 The print_fails command

C.2 Analysis of delay relations

C.3 The uverif command

C.4 The tverif command
  C.4.1 Output
  C.4.2 Construction of the failure trace
  C.4.3 Construction of the LzCES
  C.4.4 Timing analysis
  C.4.5 Miscellaneous
  C.4.6 Summary of the tverif command

REFERENCES

GLOSSARY OF SYMBOLS
# LIST OF FIGURES

1 INTRODUCTION
   1.1 Automaton modeling a modulo 3 counter. 4
   1.2 Iterative design flow. 6
   1.3 The theorem proving approach. 9
   1.4 The model checking approach. 10

2 MODELS FOR CONCURRENT SYSTEMS 17
   2.1 An example of transition system. 19
   2.2 An example of timed transition system. 21
   2.3 Portion of the timed state space of a TTS. 23
   2.4 An example of lazy transition system. 25
   2.5 Relations among the main notions related to transition systems. 27
   2.6 An example of Petri net. 28
   2.7 Reachability graph of a Petri net 31
   2.8 Petri net of Figure 2.6 with labeled transitions. 32

3 VERIFICATION OF TIMED SYSTEMS 35
   3.1 Three representations of time. 38
   3.2 An example of timed automaton. 40
   3.3 A simple automaton with atomic propositions. 43
   3.4 Regions for two clocks. 47
   3.5 Region automaton. 49
   3.6 Zone automaton. 50

4 VERIFICATION WITH RELATIVE TIMING 53
   4.1 Relative timing in the synthesis of circuits. 56
   4.2 Example of verification with relative timing. 59
   4.3 Example of verification with relative timing (first iteration). 60
   4.4 Example of verification with relative timing (second iteration). 61
   4.5 From traces to language refinement. 63
   4.6 Relation between runs and traces. 64
   4.7 Enabling and disabling in a trace. 65
4.8  Circuit with a potential disabling. 67
4.9  Enabling-compatible and non-enabling-compatible mapping. 68
4.10 Symmetric vs. asymmetric disabling. 71
4.11 Timing analysis over a CES and resulting lazy CES. 74
4.12 Graphs of reachable configurations and enableings. 78
4.13 Step by step refinement by enabling-compatible product. 82
4.14 Flow of the verification methodology. 83
4.15 Main algorithm of the relative timing-based verification approach. 84
4.16 Algorithm for the derivation of a $L_zCES$ from a trace. 84
4.17 Generation of the sufficient shortest suffix of a trace. 85
4.18 Example of a nodal and a not nodal point. 89

5  EXPERIMENTAL RESULTS 93
5.1 A binary-encoded $L_zTS$. 98
5.2 TRANSYXT input file for the $L_zTS$ of Figure 5.1. 100
5.3 Yoneda’s example. 103
5.4 Transition of an PN with its input and output places. 104
5.5 Yoneda’s example: first and second refinements. 107
5.6 Yoneda’s example: third and forth refinements. 108
5.7 Yoneda’s example: counterexample trace proving incorrectness. 110
5.8 System vs. specification verification scheme. 113
5.9 Input-output interface of the sbuf-read-ctl controller. 114
5.10 STG specification of the sbuf-read-ctl controller. 116
5.11 Two implementations of the sbuf-read-ctl controller. 118
5.12 TRANSYXT input file for the circuit of Figure 5.11 (b). 119
5.13 Three refinements for the sbuf-read-ctl controller. 124
5.14 An example of relative timing assumptions. 129
5.15 VME bus controller. 132
5.16 Timing assumptions for the synthesis of the VME bus controller. 133
5.17 Implementation of the VME bus controller with timing assumptions. 134
5.18 TRANSYXT input files for the VME bus controller. 135
5.19 First four refinements for the VME bus controller. 141
5.20 Last four refinements for the VME bus controller. 143

6  COMPOSITIONAL VERIFICATION 147
6.1 General block-level interlocking scheme. 149
6.2 Linear IPCMOS architecture. 150
6.3 Detailed 2-stage IPCMOS pipeline and waveform of its behavior. 151
6.4 Two-phase handshake mechanism. 152
6.5 The $strobe$ circuit in detail. 153
6.6 The $reset$ circuit in detail. 154
6.7 The valid circuit in detail. 155
6.8 STGs modeling the pulse-based behavior of the \textit{IN} and \textit{OUT} modules. 155
6.9 Assume-guarantee verification using abstractions. 157
6.10 Pipeline verification using abstractions. 160
6.11 STGs modeling the abstractions $A_{in}$ and $A_{out}$. 161
6.12 Scheme of the guarantee part of the verification. 163
6.13 LzCES used to prove correctness of the \textit{strobe switch} circuit. 170

A TIMING ANALYSIS 183
A.1 Classes of timing analysis problems. 184
A.2 Timing analysis on an acyclic graph. 187

C VERIFICATION-RELATED COMMANDS 199
C.1 An example of DOT file for a failure trace. 205
C.2 An example of DOT file for a LzCES. 207
C.3 An example of DOT file for the lazy state space of a system. 208
# List of Tables

5 Experimental Results 93
  5.1 Failure situations in \texttt{sbuff-read-ctl} along the verification. 122
  5.2 Experimental results for the verification of asynchronous circuits. 125
  5.3 Failure situations in the VME bus controller along the verification. 139

6 Compositional Verification 147
  6.1 Summary of the results for the 5 steps of the verification. 164
  6.2 Model of the \textit{strobe} circuit. 166
  6.3 Model of the \textit{strobe switch} circuit. 166
  6.4 Model of the \textit{reset} circuit. 167
  6.5 Model of the \textit{reset switch} circuit. 167
  6.6 Model of the \textit{valid} circuit. 167
ABSTRACT

The thesis presents a new theory and methodology for the formal verification of safety properties in timed systems. The correct operation of such systems not only depends on a set of functional properties but also on certain assumptions about the delays of the components of the system and the response times of the environment in which the system operates. The verification of this type of systems typically involves several computationally hard problems. In particular, the combinatorial state explosion problem becomes exacerbated by the time dimension.

The theory that supports the proposed verification approach extends the conventional BDD-based symbolic methods to the verification of timed systems, modeled by means of timed transition systems. The theory is based on the relative timing paradigm, which instead of considering exact time differences in the occurrence of events, considers the effect of delays in terms of relative orderings between events. For example, in order to guarantee that a race is not propagated in a digital circuit, it is often sufficient to check that certain signal switches before another, instead of identifying the exact instants of time in which both signals switch. Moreover, the timing information does not need to be computed for the overall system, but only locally for the part of the system involved in the proof or disproof of a given property. This is possible thanks to a crucial observation, that the set of executions of a transition system can be covered by a set of partial orders. As a consequence, only a subset of the events of the system is involved in the proof of a property and the timing analysis can be carried out very efficiently.

Conventional methods for the verification of timed systems rely on the computation of the exact timed state space of the system as the first step of the analysis. Although efficient techniques have been devised to overcome the complexity issue (e.g. difference bound matrices), symbolic methods cannot be easily applied. Thus, the combinatorial time-state explosion problem often limits the applicability of such methods to moderate-size systems.

Instead, the approach proposed in the thesis relies on an incremental refinement of the untimed state space of the system, so that timing information is incorporated as soon as it is needed. The timing information is derived by an efficient off-line timing analysis over small sets of events. The refined state space is captured under the model of lazy transition
system, which allows an efficient representation of the timed domain using conventional symbolic methods. As a consequence, the approach can be potentially applied to bigger systems or to systems with more level of detail, than those that can be handled by similar methods for the verification of timed systems. Moreover, the 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.

A key feature of the proposed verification approach is that not only proves or disproves the correctness of a timed system. If the system is correct the set of relative timing relations used for the proof are provided. Such 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 it can be used as valuable back-annotation information along the design process. This feature, which allows to bridge the gap between verification and design, constitutes another differential aspect of our verification approach when compared to other equivalent verification methods.

The verification approach has been fully implemented in an experimental CAV tool called TRANSYT. The tool can handle hierarchical and distributed modular systems which can inter-operate by a variety of communication mechanisms. TRANSYT has successfully proved its functionality as well as the validity of the overall verification approach, by verifying a number of timed asynchronous circuits with up to more than $10^6$ untimed states. The experiments cover, for example, the verification of: complex-gate decompositions in quasi-speed-independent asynchronous circuits, delay-reset domino circuits, pulse-based systems, circuits optimized for speed using timing assumptions, etc. Additionally, compositional verification methods have been combined with the 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 the IPCMOS architecture. It is a scalable pipelined architecture which is aimed to the interconnection of different clock zones in a system.

Thanks to the rather theoretical nature of the proposed verification approach, its potential applicability covers a wider range of systems than those cited above, such as: custom transistor-level circuits that exploit the technology limits for performance, complex digital structures where synchronization is a crucial issue (e.g. dynamic MOS), asynchronous and GALS-type systems, real-time systems, etc.
ACKNOWLEDGMENTS

If I have seen further than others, it is because I was standing upon the shoulders of giants.

—Isaac Newton - Letter to a friend, 1676

The first person that deserves my acknowledgment is my supervisor Jordi Cortadella. He introduced me to the world of asynchronous circuits, Petri nets, formal verification, etc. back in 1993 when I was still an undergraduate student. His deep insight into the subject of this thesis provided me a lot of helpful suggestions. Without his guidance and kind encouragement this thesis would have never been possible.

I am specially indebted to my other supervisor, Enric Pastor. His continuous support and friendship have helped me to overcome the (technical and personal) difficulties during the critical phases of this work.

My gratitude also to Alex Kondratyev and Alexander Smirnov for their contributions to the theoretical soundness and the practical implementation of this work, respectively. And to Luciano Lavagno, Alex Yakovlev and Alexander Taubin, for their kindness hosting me in respective visits to the Politecnico di Torino, the University of Newcastle and the University of Aizu. The numerous insightful discussions with them about different research topics have contributed this thesis in a number of ways.

Thanks to the other members of the CAD/VLSI group at the Department of Computer Architecture of the Technical University of Catalonia —Rosa Badia, Fermín Sánchez and Josep Carmona— and former members —Oriol Roig and Enric Musoll. Along these years they have provided me with the kind of environment that makes work a much more pleasant experience.

Thanks also to the reviewers at Department of Computer Architecture —Rosa Badía, Antonio González and Antonio Juan Hormigo— and the external reviewers —Abelardo Pardo and Supratik Chakraborty. They read carefully the preliminary versions of this thesis giving me valuable suggestions on the contents and the presentation of this work. And the members of the thesis committee for the effort they put into judging this thesis.
On the institutional and industry side I would like to acknowledge: the Ministry of Science and Technology of Spain under contracts CICYT-TIC 95-0419, CICYT-TIC 98-0410-CO-01 and CICYT-TIC 2001-2476-CO-03; the AcId Working Group under contracts ESPRIT-7225, ESPRIT-21949 and IST-1999-29119; and Intel Corporation. They all are gratefully acknowledged for funding this research.

On the personal side my family deserves infinite gratitude: my father Antonio, my mother Adoración and my brother Jose. Thanks a lot for your efforts on infusing me the values of a good education and supporting all my studies along the years.

And the most effusive thanks to Nuria. During all this time she has been my best friend, providing lots of emotional support and love, and patiently suffering the numerous moments of solitude she has been forced to because of my work. This thesis is dedicated to her and our beloved son Pau.

Finally I would like to express my thanks to everyone I have not cited above but has helped me, directly or not, in the long way until this thesis has been completed. Thanks a lot to you all.
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 - [Rus03]

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 cost $475 million US dollars [Pet97]. Another famous case is the launch failure of the Ariane 5 rocket, which exploded 37 seconds after lift-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
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.

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 \( \forall G \exists X \text{ true} \) which can be read as: “whatever the state reached may be \((AG)\), there will exist an immediate successor state \((EX \text{ true})\)” 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
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:

<table>
<thead>
<tr>
<th>(x)</th>
<th>((x + 1)^2)</th>
<th>(x^2 + 2x + 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>67</td>
<td>4624</td>
<td>4624</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
</tr>
</tbody>
</table>

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:
\begin{center}
\begin{tabular}{|l|l|}
\hline
1. & \((x + 1)^2 = (x + 1)(x + 1)\) \textit{definition of square} \\
2. & \((x + 1)(x + 1) = (x + 1)x + (x + 1)1\) \textit{definition of distributivity} \\
3. & \((x + 1)^2 = (x + 1)x + (x + 1)1\) \textit{substitution of 2. in 1.} \\
4. & \((x + 1)1 = x + 1\) \textit{neutral element 1} \\
5. & \((x + 1)x = xx + 1x\) \textit{distributivity} \\
6. & \((x + 1)^2 = xx + 1x + x + 1\) \textit{substitution of 4. and 5. in 3.} \\
7. & \(1x = x\) \textit{neutral element 1} \\
8. & \((x + 1)^2 = xx + x + x + 1\) \textit{substitution of 7. in 6.} \\
9. & \(xx = x^2\) \textit{definition of square} \\
10. & \(x + x = 2x\) \textit{definition of 2x} \\
11. & \((x + 1)^2 = x^2 + 2x + 1\) \textit{substitution of 9. and 10. in 8.} \\
\hline
\end{tabular}
\end{center}

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.

\subsection*{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.
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 it is possible for the system to violate the specification. Countereamples 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
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 [Ho197] that takes advantage of partial orders for the verification of distributed algorithms; HSIS [ABC+94] which combines model checking with language inclusion; KRONOS [Yow97] 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 $\omega$-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.
1.5 Overview of the contributions

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. [BCM92]).

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 [SRC00]. 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 [Pmu84] 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 TRANSYRT tool related to the verification of timed systems, which implement the presented verification approach.
MODELS FOR CONCURRENT SYSTEMS

The best material model of a cat is another, or preferably the same, cat.
—Arturo Rosenblueth - Philosophy of Science, 1945

A theory has only the alternative of being right or wrong. A model has a third possibility: it may be right, but irrelevant.
—Manfred Eigen - The Physicist’s Conception of Nature, 1973

Summary

This chapter introduces the fundamentals of the models used for the specification, synthesis and verification of systems in the subsequent chapters. In particular, transition systems and Petri nets are introduced as models for untimed systems. Other classes of transition systems, such as timed transition systems and lazy transition systems are introduced as the models used for timed systems.

Transition systems and Petri nets provide an abstract view of the events and states of a system, without considering any binary encoding. In some cases, such as for the logic synthesis of circuits, or simply in order to achieve efficient implementations of verification algorithms, such encoding is required. The encoding allows the symbolic manipulation of a system using compact representations and efficient algorithms based on BDDs, for example. Chapter 5 provides details on the binary encoding of transition systems.
2.1 Introduction

As it will be seen in Chapter 4, the proposed verification approach uses *timed transition systems* to model the timed systems under verification. The approach, however, does not manipulate the exact timed state space of the system. Instead, the untimed state space, modeled by a *transition system*, is used as the starting point of the verification approach. Then, an incremental refinement of the untimed state space with relative timing information is carried out, thus leading to the use of *lazy transitions systems*. This chapter presents the fundamental concepts and notation related to the different types of transition systems involved in the verification approach.

On the other hand, *Petri nets* and its interpretation as *signal transition graphs* are often used to model asynchronous digital circuits and other concurrent systems. As a consequence, a number of verification methods have been developed under such formalisms (see Section 3.6). Moreover, in Chapter 5 some illustrative examples of the verification of timed systems are presented, which are originally specified with *Petri nets* and *signal transition graphs*. For this reason, we have considered appropriate to introduce the fundamentals of such modeling formalisms at the end of this chapter.

2.2 Transition systems

Transition systems (TS) are a formalism used to describe systems of concurrent processes [Arn94]. The formalism, although mathematically simple, can model most of the properties of such systems, and so can be used to study their semantics. Several theoretical tools based on transition systems have been developed, including equivalence relations with other formalisms (formal languages, Petri nets, etc.). The usefulness of these theoretical tools is supported by the existence of a variety of software tools.

Intuitively, a *transition system* consists of the set of possible states of a system, and a set of transitions that the system can produce in order to change from one state to another. In comparison to event-based models such as Petri nets, transition systems offer a view of a system at a lower level of abstraction.

**Definition 2.1 (Transition System)**

A transition system (TS) [NRT92] is a quadruple $A = (S, \Sigma, T, s_0)$, where $S$ is a non-empty set of states, $\Sigma$ is a non-empty alphabet of events, $T \subseteq S \times \Sigma \times S$ is a transition relation, and $s_0$ is the initial state.

The elements of $T$ are called transitions and are indistinctly denoted by $s \stackrel{e}{\rightarrow} s'$ or by $(s, e, s')$.

An event $e$ is enabled at state $s$ if $\exists s \stackrel{e}{\rightarrow} s' \in T$. We will denote by $E(s)$ the set of events enabled at state $s$. The firing region of event $e$ is defined as $FR(e) = \{s \in S \mid e \in E(s)\}$, i.e. the set of states where $e$ is enabled.  

2.1
A TS is finite if \( S \) and \( \Sigma \) are finite. A TS is called deterministic if for each state \( s \) and each event \( e \) there is at most one state \( s' \) such that \( s \xrightarrow{e} s' \). In the sequel, only finite transition systems will be considered. Moreover, no multiple arcs should exist between any pair of states, i.e. \( s \xrightarrow{e} s' \in T \wedge s \xrightarrow{e'} s'' \in T \Rightarrow e = e' \).

**Example 2.1** A TS can be represented by an arc-labeled directed graph. A simple example of a TS is shown in Figure 2.1. States are represented as dots, transitions are the directed arcs between the states, and the events are the labels of the transitions. That is, \( S = \{s_0, \ldots, s_7\} \), \( \Sigma = \{t_1, \ldots, t_7\} \) and \( T = \{s_0 \xrightarrow{t_1} s_1, s_1 \xrightarrow{t_4} s_4, \ldots\} \). As an example, the firing region of event \( t_5 \) is \( \text{FR}(t_5) = \{s_2, s_6\} \).

**Definition 2.2 (Run)**

A run of a transition system \( A = (S, \Sigma, T, s_0) \) is a sequence of transitions \( \rho = s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \cdots \), such that \( s_1 = s_0 \) and \( \forall i \geq 1 : s_i \xrightarrow{e_i} s_{i+1} \in T \). Event \( e_i \) is said to fire at step \( i \) of the run.

With an abuse of notation, the expressions \( s_i \in \rho \), \( s_i \xrightarrow{e_i} s_{i+1} \in \rho \), \( s_i \xrightarrow{e_i} \in \rho \), \( s_{i+1} \in \rho \), etc., will be often used to denote the fact that different fragments of a sequence belong to a run.

Several ordering relations between the events of a TS can be defined.

**Definition 2.3 (Triggering, Conflict, Concurrency)**

Given a transition system \( A = (S, \Sigma, T, s_0) \) and two events \( e_1, e_2 \in \Sigma \):

(a) \( e_1 \) triggers \( e_2 \) if the firing of \( e_1 \) enables \( e_2 \), i.e. if \( \exists s_1 \xrightarrow{e_1} s_2 \in T \), such that \( s_1 \notin \text{FR}(e_2) \) and \( s_2 \in \text{FR}(e_2) \).
(b) \( e_1 \) disables \( e_2 \) if \( \exists s_1 \xrightarrow{e_1} s_2 \in T \), such that \( s_1 \in FR(e_2) \) and \( s_2 \notin FR(e_2) \). \( e_1 \) and \( e_2 \) are in conflict if either \( e_1 \) disables \( e_2 \) or \( e_2 \) disables \( e_1 \).

(c) \( e_1 \) and \( e_2 \) are concurrent if there is a state in which both events are enabled and the firing of one of them does not disable the other: \( \exists s \in FR(e_1) \cap FR(e_2) : s \xrightarrow{e_1} s_1 \in T \land s \xrightarrow{e_2} s_2 \in T \Rightarrow \exists s_3 \in S : s_1 \xrightarrow{e_2} s_3 \in T \land s_2 \xrightarrow{e_1} s_3 \in T \).

The following definition captures the notion of enabling of an event at a given state of a run, and the conditions for such event to keep enabled along the run until it fires or is disabled by the firing of another event.

**Definition 2.4 (Enabling interval)**

Let \( A = (S, \Sigma, T, s_0) \) be a TS and let \( \rho = s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \cdots \) be a run of \( A \). Given an event \( e \) and a state \( s_i \in \rho \) such that \( s_i \in FR(e) \), FirstEnabled(\( \rho, s_i, e \)) is defined as the state \( s_j, j \leq i \), such that:

- \( j \leq k \leq i \Rightarrow s_k \in FR(e) \) ( \( e \) is continuously enabled between \( s_j \) and \( s_i \))
- \( j > 0 \Rightarrow s_{j-1} \notin FR(e) \) ( \( e \) is not enabled before \( s_j \))

The sequence \( s_j \xrightarrow{e_j} \cdots \xrightarrow{e_i} s_i \) is called the enabling interval of \( e \) with respect to \( s_i \).

Notice that this definition imposes nothing about the necessity of event \( e \) to actually fire in \( \rho \). Therefore, given \( s_j \xrightarrow{e_j} \cdots s_i \xrightarrow{e_i} s_{i+1} \in \rho \) and FirstEnabled(\( \rho, s_i, e \)) = \( s_j \) such that \( s_{i+1} \notin FR(e) \), \( e \) actually fires if \( e = e_i \), whereas \( e \) is disabled by the firing of \( e_i \) otherwise. As an example, consider the run \( \rho = s_0 \xrightarrow{12} s_2 \xrightarrow{16} s_0 \xrightarrow{15} s_7 \cdots \) of the TS in Figure 2.1. In this case we have that FirstEnabled(\( \rho, s_0, s_5 \)) = \( s_2 \).

The transitive closure of the transition relation \( T \) is called the reachability relation between states and is denoted by \( \mathcal{T}^+ \). In other words, state \( s' \) is reachable from state \( s \) if there is a run between both, denoted by \( s \xrightarrow{\rho} s' \), or simply \( s \xrightarrow{\mathcal{T}} s' \) if the run is not relevant.

The set of states reachable through all possible runs of the system is given by the following definition:

**Definition 2.5 (Reachable states)**

Given a TS \( A = (S, \Sigma, T, s_0) \), the set of reachable states from a state \( s \) is recursively defined as:

\[
Reach(s, T) = \{s\} \cup \bigcup_{s\xrightarrow{\mathcal{T}} s' \in T} Reach(s', T)
\]

Henceforth, it is assumed that \( S = Reach(s_0, T) \) for any TS.
2.3 Timed transition systems

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

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

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

\[\delta(a) \in [1,2]\]
\[\delta(b) \in [1.5,2]\]
\[\delta(c) \in [3,4]\]
\[\delta(d) \in [1.1.5]\]
\[\delta(\theta) \in [1,1.5]\]
**Definition 2.6 (Timed Transition System)**

A timed transition system (TTS) [HMP92a] is a triple \( A = (A^-, \delta', \delta^u) \), where \( A^- = (S, \Sigma, T, s_0) \) is a TS called the underlying transition system, \( \delta' : \Sigma \to \mathbb{R}^+ \) and \( \delta^u : \Sigma \to \mathbb{R}^+ \cup \{ \infty \} \) respectively associate a minimum and a maximum delay bound to each event, such that \( \forall e \in \Sigma : \delta'(e) \leq \delta^u(e) \).

Min-max delay ranges are represented by \([d, D]\), or by \([d, \infty)\) if the maximum delay is unbounded. \(\blacksquare\)

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

The TS \( A^- \) captures the behavior of the system in the untimed domain, i.e. without considering the delay information. The behavior of the system in the timed domain is obtained by associating to each state a time stamp that indicates when the state was reached. Thus, the same state can be visited at different time instants depending on its prehistory, and the time stamps must comply with the delay bounds specified for each event (see Definition 2.7). Conversely, a state can never be reached in the timed domain if its incoming transitions are never fired due to their respective delay constraints. Time is always relative to that of the initial state of the system, and must monotonically increase along the firing sequences.

Because of the time dimension, the computation of the exact timed state space of a system has been proved to be a PSPACE-complete problem [AD90] and demonstrated to be a highly complex task in several contexts such as real-time systems [AD94, HMP91] and asynchronous circuits [DKMW92, Bur92, HB94, MP95, SY96, VdL96].

The following definition characterizes which sequences of states of the system are actually feasible when the specified delay bounds are taken into consideration.

**Definition 2.7 (Timing-consistent run)**

Let \( A = (A^-, \delta', \delta^u) \) be a TTS and let \( \rho = s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \cdots \) be a run of \( A^- \). \( \rho \) is timing-consistent with \( A \) if a sequence \( \tau_1 \leq \tau_2 \leq \cdots \) of monotonically increasing real-valued time stamps can be assigned to the states in \( \rho \) such that:

\[ \forall s_i \xrightarrow{e_i} s_{i+1} \in \rho \text{ and } \forall e \in E(s_i) \text{ such that } \text{FirstEnabled}(\rho, s_i, e) = s_j : \]

1. \( e = e_i \Rightarrow \delta'(e) \leq \tau_{i+1} - \tau_j \) i.e. an event cannot fire before its minimum delay has elapsed, and
2. \( \tau_{i+1} - \tau_j \leq \delta^u(e) \) i.e. an event cannot remain enabled after its maximum delay has elapsed. \(\blacksquare\)
The previous definition characterizes those runs which are possible according to the delay bounds associated to the events of the system. The time stamp \( \tau_i \) is assigned to state \( s_{i+1} \) and corresponds to the firing time of event \( e_i \) along \( \rho \). Similarly, \( \tau_j \) corresponds to the enabling time. Thus, the firing time of an event only depends on its enabling time plus certain delay amount within the given bounds. Moreover, the disabling of an event must be produced before its maximum delay has elapsed, otherwise it should have fired already.

A timing-consistent run can be represented as a sequence of transitions of the form: \( (s_1, \tau_1)^{e_1} (s_2, \tau_2)^{e_2} \cdots \). Where for all \( i > 0 \): each pair \((s_i, \tau_i)\) is called a timed state with \( s_i \in S \) and \( \tau_i \in \mathbb{R}^+ \); and \( s_i \xrightarrow{e_i} s_{i+1} \in T \). Also \( \tau_1 = 0 \) and \( \lim_{i \to \infty} \tau_i = \infty \). We will use this notation to represent the timed runs of a TTS when it eases the reading.

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

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

To illustrate the behavior of the system in the timed domain assume the execution starts at the timed state \((s_0, 0)\). The system remains at such state until the minimum delay of a elapses. Then the system can either fire a or wait until b becomes firable. 0.5 time
units later. Thus, \( a \) can fire between time \( 0 + \delta'(a) = 1 \) and time \( 0 + \delta''(a) = 2 \), whereas \( b \) can fire between time \( 0 + \delta'(b) = 1.5 \) and time \( 0 + \delta''(b) = 2 \). Assume \( a \) fires at time 1.5 leading to the timed state \((s_1,1.5)\). Now \( b \) is still enabled and can fire, while \( e \) and \( c \) become newly enabled. \( e \) cannot fire until time \( 1.5 + \delta'(e) = 2.5 \), whereas \( c \) is much slower and cannot fire until \( 1.5 + \delta'(c) = 4.5 \). Therefore, if for example \( b \) fires at time 1.5 leading to the timed state \((s_1,1.5)\), it will be followed by \( e \) and \( c \), leading for example to the timed states \((s_5,2.5)\) and \((s_{10},4.5)\) successively. The execution continues by firing \( d \), and so on.

None of the states \( s_5, s_6, s_7, s_8, s_9 \) or \( s_{11} \) is reachable in the portion of the timed domain shown in the figure. For example, state \( s_9 \) can not be reached from \( s_3 \) by firing \( c \), since \( c \) is slower than \( a \), even in the case that both events were enabled at the same instant. In fact, only the following three runs are potentially timing-consistent if appropriate time stamps are assigned to each state: \( s_0 \xrightarrow{a} s_1 \xrightarrow{e} s_2 \xrightarrow{b} s_3 \xrightarrow{c} s_10 \xrightarrow{d} s_{12} \ldots \), \( s_0 \xrightarrow{a} s_1 \xrightarrow{e} s_2 \xrightarrow{c} s_10 \xrightarrow{d} s_{12} \ldots \) and \( s_0 \xrightarrow{b} s_3 \xrightarrow{a} s_4 \xrightarrow{e} s_5 \xrightarrow{c} s_10 \xrightarrow{d} s_{12} \ldots \).

For example, the following timed run is timing-consistent:

\[
(s_0, 0) \xrightarrow{a} (s_1, 1) \xrightarrow{b} (s_1, 1.5) \xrightarrow{e} (s_5, 2) \xrightarrow{c} (s_{10}, 4) \xrightarrow{d} (s_{12}, 5) \ldots
\]

Conversely, the following timed run is not timing-consistent:

\[
(s_0, 0) \xrightarrow{a} (s_1, 1) \xrightarrow{b} (s_3, 2) \xrightarrow{e} (s_5, 3.5) \xrightarrow{c} (s_{10}, 5) \xrightarrow{d} (s_{12}, 5) \ldots
\]

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

To conclude this section, we want to remark that different notions of timed transition systems with timing constraints for the discrete-time paradigm have been also proposed by several authors [PH88, Ost90, HMP91]. Similarly, clocked transition systems were defined in [MP96] as an alternative model for real-time systems, where time is explicitly represented by means of a set of timers (often called clocks) which increase uniformly whenever time progresses, and can be set to specific values by the firing of transitions. This type of transition systems is inspired by a more commonly used model called time automata which is analyzed in detail in Chapter 3.

### 2.4 Lazy transition systems

When time becomes an essential magnitude in a model for concurrent systems, a complexity dimension is added to the derived analysis and verification methods. In those formalisms and analysis mechanisms where time is an explicit magnitude, such complexity is specially noticeable when computing the timed state space of the system. For example, the problem of reachability in timed automata is proved to be PSPACE-hard [AD94], where the exponentiality depends on the number of clocks and on the encoding of the maximum values that can be taken by the clocks (see Section 3.3). This complexity makes the analysis of systems with a moderate amount of untimed states almost impractical.
Several approaches have been devised to represent timed states in a succinct form, e.g. [BM00]. However, the incorporation of the timed domain in the representation of the states hampers an efficient representation of large state spaces with BDDs [Bry86]. Even the discretization of time [HMP91] poses serious problems when the number of clocks or the constants of the timing constraints are large. An interesting approach to face this complexity problem was proposed in [AK95], where the clocks used during the analysis of the system and their accuracy, are determined dynamically upon demand. In this way, only that timing information relevant to the analysis emerges during the calculation of the reachable states.

In some cases, however, rather than calculating the exact time intervals in which each state can be visited by any valid run of the system, the only information required is to know whether each state is visited by some timing-consistent run and what are the enabling conditions for every visited state. In other words, only the set of reachable states in the timed domain and the transition relations for every event are required. This information can be represented by abstracting absolute time information out of the model. The abstraction leads to the definition of a new computational model called lazy transition systems [CKK+98], in which timing information is represented in terms of the notion of laziness. This notion explicitly distinguishes among the enabling and the firing of an event, assuming certain implicit delay between them. Formally:
**Definition 2.8 (Lazy Transition System)**

A lazy transition system (LzTS) $[CKK^+98]$ is a five-tuple $A = \langle S, \Sigma, T, s_0, \text{EnR} \rangle$, where $\langle S, \Sigma, T, s_0 \rangle$ is a TS, and the function $\text{EnR} : \Sigma \to 2^S$ defines the enabling region of each event. \hfill \Box 2.8

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

**Example 2.2 (cont.)** Figure 2.4 shows the lazy transition system corresponding to the timed transition system in Figure 2.2. Analyzing the delays, it can be proved that event $c$ is always slower to fire than events $a$, $b$ and $e$. Therefore, $c$ becomes lazy in those states where it is concurrently enabled with those faster events. Thus, although $c$ is enabled in states $\text{EnR}(c) = \{s_1, s_2, s_3, s_4, s_5\}$, it can only fire in $\text{FR}(c) = \{s_6\}$, once $a$, $b$ and $e$ have already fired. \hfill \Box 2.2

Notice that a TS is just a particular case of LzTS in which both enabling and firing regions coincide for all the events. Thus, the notion of enabling interval (see Definition 2.4) is naturally extended to lazy transition systems by considering $\text{EnR}(e)$ instead of $\text{FR}(e)$ for the enabledness of event $e$. Similarly, the ordering relations defined between the events of a TS (see Definition 2.3) can be extended for LzTSs as follows:

**Definition 2.9 (Triggering, Conflict, Concurrency)**

Given a lazy transition system $A = \langle S, \Sigma, T, s_0, \text{EnR} \rangle$ and two events $e_1, e_2 \in \Sigma$:

(a) $e_1$ triggers $e_2$ if the firing of $e_1$ enables $e_2$, i.e. if $\exists s_1 \xrightarrow{e_1} s_2 \in T$, such that $s_1 \not\in \text{EnR}(e_2)$ and $s_2 \in \text{EnR}(e_2)$.

(b) $e_1$ disabling $e_2$ if $\exists s_1 \xrightarrow{e_1} s_2 \in T$, such that $s_1 \in \text{EnR}(e_2)$ and $s_2 \not\in \text{EnR}(e_2)$.

$e_1$ and $e_2$ are in conflict if $e_1$ disables $e_2$ or $e_2$ disables $e_1$.

(c) $e_1$ and $e_2$ are concurrent if $\text{EnR}(e_1) \cap \text{EnR}(e_2) \neq \emptyset$ and they are not in conflict, and $\exists s \in \text{FR}(e_1) \cap \text{FR}(e_2) : s \xrightarrow{e_1} s_1 \in T \land s \xrightarrow{e_2} s_2 \in T \Rightarrow \exists s_3 \in S : s_1 \xrightarrow{e_1} s_3 \in T \land s_2 \xrightarrow{e_2} s_3 \in T$.

\hfill \Box 2.9

Notice that the second condition for concurrency is analogue of the non-conflict requirement, but applies to the FR rather than to the EnR.

The use of LzTSs enables to reason in terms of partial orders of events, i.e. in terms of the so-called relative timing paradigm [SGR99], which is much more intuitive than
defining absolute time separations between pairs of events (see also Chapter 3). Moreover, whereas absolute timing information requires complex techniques to represent the space of reachable timed regions or states [Aku98] (e.g. using difference bound matrices, time polyhedra, etc.), the generation of the reachable state space for relative timing is of the same complexity as for untimed systems. Thus Definition 2.5 can be extended to LzTSs in a straightforward manner.

Figure 2.5 depicts the relationships among the major notions around transition systems, and their timed and lazy counterparts.

2.5 Petri nets

Petri nets (PNs) were initially proposed in [Pet62] as a graphical and mathematical formalism for describing information processing systems, characterized as being concurrent, asynchronous, distributed, parallel, non deterministic and/or stochastic. Since their introduction, Petri nets have been used in a wide range of areas such as communication networks, computer architecture, distributed systems, manufacturing, digital circuit synthesis and verification, etc.

Even though Petri nets constitute a powerful model, they consist of a few objects, relations and behavior rules. Namely, a Petri net consists of:

- The net structure, a bipartite directed graph containing places, transitions and arcs, that represent the static nature of the model.

- The net marking, which represents a distributed state of the model, in which a place can be marked by several tokens.

- The execution rules, which represent the dynamic evolution of the state of the model, i.e. describe how the tokens evolve through the places and transitions.
Places can be seen as the state variables of the model. Transitions can be seen as the events that transform the state of the model. Arcs determine which are the necessary conditions for an event to occur and the values of the variables once an event has occurred.

A PN may also have additional information in its structure or in the markings, leading to different classes of nets, such as High-level Petri nets [Jen92] or Coloured Petri nets [Mau92]. In the sequel, we restrict ourselves to the use of Place/Transition Petri nets, in which the tokens do not carry any particular information. For a good introduction to the different classes of Petri nets, their properties and usage, the reader is referred to [Pet81], [Rei85] and [Mur89], for example.

The remaining of this section introduces the notions related to Petri nets necessary for the work presented in the subsequent chapters.

**Definition 2.10 (Petri net)**

A Petri net (PN) is a quadruple $N = (P, T, F, M_0)$, where $P$ is a finite set of places, $T$ is a finite set of transitions, $F : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ is the flow function, and $M_0 : P \rightarrow \mathbb{N}$ is the initial marking (state) of the Petri net.

If the flow function is a relation on $P \cup T$, i.e. a mapping $F : (P \times T) \cup (T \times P) \rightarrow \{0, 1\}$, then the PN is called ordinary. In the sequel we will assume all the PNs to be ordinary, and will often talk about the flow relation $F$ instead of the flow function.

The pre-set and post-set of a node $x \in P \cup T$ are denoted by $^*x = \{y \mid (y, x) \in F\}$ and $^x = \{y \mid (x, y) \in F\}$, respectively. Informally, the pre-set of a transition (place) corresponds to its input places (transitions), whereas its post-set corresponds to its output places (transitions).
When Petri nets are graphically represented, places are drawn as circles, transitions are drawn as boxes (or bars), the flow relation is represented by directed arcs, and tokens appear as dots circumscribed into the places.

If restrictions are imposed on the structure of the net, several subclasses of PNs can be defined [Mur89]: state machines, in which each transition has exactly one input place and one output place; marked graphs, in which each place has exactly one input transition and one output transition; free-choice Petri nets, in which if \((p, t) \in F\) then \(\cdot t \times p \cdot \subseteq F\) for every place \(p\); etc.

**Example 2.3** Figure 2.6 depicts a Petri net, consisting of seven places, \(P = \{p_1, \ldots, p_7\}\) and seven transitions, \(T = \{t_1, \ldots, t_7\}\). The initial marking is indicated by the token in place \(p_1\). When a place has only one predecessor and only one successor transition, it is often replaced by a simple arc between both transitions. According to this, in the PN of Figure 2.6, places \(p_2, p_3, p_4\) and \(p_5\) could be omitted.

This Petri net has only one choice place, \(p_1\), which has two successor transitions, \(t_1\) and \(t_2\). Since \(p_1\) is the only predecessor of \(t_1\) and \(t_2\), \(p_1\) is a free-choice place and so it is the Petri net. \(\blacksquare\)

The structure of a Petri net defines the rules that determine its dynamic behavior. That is, what are the conditions that make a transition to become enabled and what happens when such transition actually fires.

**Definition 2.11 (Enabling and Firing)**

Given a Petri net \(N = (P, T, F, M_0)\):

(a) A marking is a function \(M : P \to \mathbb{N}\), which can be represented by a vector in \(\mathbb{N}^{\#P}\), such that a place \(p\) is said to be marked at \(M\) by \(M(p)\) tokens if \(M(p) > 0\).

(b) A transition \(t \in T\) is enabled in marking \(M\), denoted by \(M[t]\), if all places in \(\cdot t\) are marked by at least one token. That is \(\forall p \in \cdot t, M(p) > 0\). We denote by \([t]\) the set of all markings where transition \(t\) is enabled.

(c) When a transition \(t \in T\) is enabled in marking \(M\), it can fire reaching a new marking \(M'\), denoted by \(M[t]M'\), by removing a token from each place in \(\cdot t\) and adding a token to each place in \(t^*\). Formally:

\[
\forall p \in P, \quad M'(p) = \begin{cases} 
M(p) - 1 & \text{if } p \in \cdot t \setminus t^* \\
M(p) + 1 & \text{if } p \in t^* \setminus t \\
M(p) & \text{otherwise}
\end{cases}
\]

\(\blacksquare\)

With the rules provided by the previous definition, the complete state space determined by the PN can be defined as follows:
**Definition 2.12 (Reachability)**

Given a Petri net $N = \langle P, T, F, M_0 \rangle$:

(a) Marking $M'$ is reachable from marking $M$ if there is a firing sequence of transitions $\sigma = t_1t_2 \ldots \in T^*$ that transforms $M$ into $M'$, i.e. $M(\sigma)M'$. The reachability set of $N$, denoted by $[M_0]$, contains all the markings reachable from $M_0$.

(b) The reachability graph (RG) of $N$ contains all the reachable markings and all the possible firing sequences of $N$. It is a directed graph $RG = ([M_0], E)$ where $E \subseteq [M_0] \times T \times [M_0]$ is the set of arcs such that $(M, t, M') \in E \iff M[t]M'$. A formal definition of $RG$ in terms of transition systems is given in Section 2.2.

\[ 2.12 \]

**Example 2.3 (cont.)** Figure 2.7 (a) depicts the reachability graph corresponding to the Petri net of Figure 2.6, consisting of eight markings. For example, marking $M_1 = \{p_2, p_3\}$ indicates that places $p_2$ and $p_3$ contain one token each, whereas the rest of the places are empty.

\[ 2.3 \]

In marking $M_1$ of the previous example, transitions $t_3$ and $t_4$ are both enabled simultaneously and can fire in any order without disabling each other. Thus, it can be expected that they can potentially fire together at the same instant, as it is shown by the arc between $M_1$ and $M_7$ if Figure 2.7 (b). The semantics of PNs that allow this type of step-transitions is called a true concurrency semantics, see the RG in Figure 2.7 (b). Conversely, the semantics that only allows a single transition to fire at a time is called interleaving semantics, see the RG in Figure 2.7 (a). Although true concurrency is more general than interleaving, it is often much more difficult to deal with it. In the sequel, only interleaving semantics is considered since it is general enough for our purposes. In fact, Definition 2.12 for reachability of PNs is already built upon the interleaving semantics.

Transition systems and Petri nets are linked through the notion of reachability graph. In fact the reachability graph of a PN is a transition system. Thus, given a PN, $N = \langle P, T, F, M_0 \rangle$, its reachability graph is a transition system, $RG(N) = \langle S, \Sigma, T, s_0 \rangle$, in which the set of states of the TS corresponds to the reachability set of the PN ($S = [M_0]$), the events of the TS correspond to the transitions of the PN, and a transition $(M_i, t, M_j)$ exists in the TS if and only if $M_i[t]M_j$ in the PN. For example, it can be easily seen that the TS depicted in Figure 2.1 is isomorphic to the reachability graph depicted in Figure 2.7 (a) derived from the PN of Figure 2.6.

Even though the construction of a TS equivalent to a PN is a straightforward task, the opposite is not that simple in general. The interested reader is referred to [NRT92] and [CKLY98] for more details on the problem.
A PN is called live iff every transition can be infinitely enabled through some feasible sequence of firings from any marking in \([M_0]\). A PN is called safe if no marking in \([M_0]\) assigns more than one token to any place. Safe PNs are widely used in many applications since they have simple analysis algorithms and simple semantics [EN94]. Without loss of generality for our purposes, in the sequel we assume all the PNs to be safe. For example, the Petri net of Figure 2.6 is safe.

A static characterization of the dynamic behavior of the PN was defined by [Chu87] in terms of the so-called temporal relations. The relations define which pairs of transitions are causally related or are concurrent. Conflict relations are also defined, i.e. if the firing of one transition prevents the firing of another transition which was already enabled. These relations are often defined using the reachability graph of the PN, however they can be computed efficiently from the structure of the net [KE96]. Since the reachability graph of a PN is actually a transition system and these relations are also naturally defined for them, we omit the definition of the temporal relation at this point and refer the reader to Definition 2.3.

### 2.5.1 Labeled Petri nets

A labeled Petri net is a PN augmented with a labeling function which puts in correspondence every transition of the net with a symbol (called label) of an alphabet. If no two transitions have the same label (unique labeling), then each transition in the net can be uniquely identified by its label. Formally:
**Definition 2.13 (Labeled Petri Net)**

A labeled Petri net is a triple \( LPN = (N, \Sigma, \Lambda) \), where \( N \) is a Petri net, \( \Sigma \) is a finite alphabet, and \( \Lambda : T \rightarrow \Sigma \cup \{ \epsilon \} \) is a labeling function. The special symbol \( \epsilon \) is used to label those transitions without a particular meaning. Such “silent” transitions are often called dummy or sequencing transitions.

Figure 2.8 depicts a labeled PN with \( \Sigma = \{ a+, a-, b+, b-, c+, c- \} \).

Signal Transition Graphs (STGs), which are a particular class of PNs, are used in Chapter 5 to model asynchronous circuits. STGs were introduced independently in [RY85] and [Chu87] as a formalism for modeling the behavior of asynchronous circuits and their environment. In short, an STG is a labeled Petri net interpreted such that transitions describe value changes at the signals of a circuit. A signal transition can be represented by \( a+ \) (or \( a- \)) for the transition of signal \( a \) from 0 to 1 (or from 1 to 0), while \( a+ \) is a generic name for either a rising or a falling transition of signal \( a \). Formally:

**Definition 2.14 (Signal Transition Graph)**

A signal transition graph (STG) is a labeled Petri net \( LPN = (N, \Sigma, \Lambda) \), where \( \Sigma = \Sigma_T \cup \Sigma_O \cup \Sigma_I \) is a set of signal names formed by the union of three non-intersecting subsets of input, output and internal signals, and \( \Lambda : T \rightarrow \Sigma \times \{ +, - \} \) is the labeling function.

The PN in Figure 2.8 can be interpreted as an STG that specifies the behavior of a circuit with, for example, two input signals \( (\Sigma_T = \{ a, b \}) \) and an output signal \( (\Sigma_O = \{ c \}) \).
2.6 Conclusions

This chapter has presented the fundamentals of the formal models for concurrent systems that will be used in the subsequent chapters.

Transition systems (TS) are introduced as a state-based formalism for untimed systems. A TS is a mathematically simple formalism, however it allows to deal with a wide variety of other formalisms and to reason about them using a common formalism. As an example, the relation between PNs and TSs has been outlined. Fundamental notions for our later developments, such as the notion of enabling interval are presented.

Time can be incorporated on top of TSs in different forms. Timed transition systems (TTS) constitute the most common alternative, which associate time intervals to the events of the system. Therefore time is incorporated as an explicit real-valued exact magnitude. This fact causes that the state explosion problem becomes even less tractable than in the case of TSs. Another fundamental notion, that of timing-consistent run, is presented.

Conversely, lazy transition systems (LzTS) incorporate time using the relative timing paradigm, thus abstracting exact timing away and dealing only with partial orders of the events in the timed domain. LzTSs are the model underlying the development of the formal verification algorithms for timed systems in Chapter 4 and the related.

Petri nets provide a graphical and mathematical formalism for describing concurrent systems in a very intuitive way at the level of events. Apart from the net structure and its graphical representation, some basic properties have been briefly introduced. We have chosen PNs as the model for systems in which time is not a relevant magnitude. In particular, STGs will be used in Chapter 5 to model speed-independent asynchronous circuits.
VERIFICATION OF TIMED SYSTEMS

Time has no divisions to mark its passage, there is never a thunderstorm or blare of trumpets to announce the beginning of a new month or year.

—Thomas Mann - The Magic Mountain, 1924

Time is such a simple, almost primitive idea. It is just a means of material differentiation, a way of uniting us all; for in our external, material lives we value the synchronized efforts of individual people.

—Andrei Tarkovsky - Time Within Time: The Diaries, 1989

Summary

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

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

The concept of timed temporal logic is introduced as an appropriate mechanism for the specification of timing-related properties. Several alternatives are briefly commented.

Since most verification methods rely on the analysis of the timed state space of the system, the reachability problem on timed systems is analyzed.

Finally, a brief review is provided of the approaches that use Petri nets for the timing analysis and the verification of timed systems.
3.1 Introduction

Three major ingredients are required to accomplish the verification task successfully:

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

Several modeling formalisms have been already introduced in Chapter 2, namely those used for the research presented in this work. Among the formalisms used by other researchers, timed automata [AD94] deserve particular attention since they are the model of choice in many verification methodologies.

Regarding the specification formalisms, there exists a wide spectrum which can be roughly divided into two categories: logic-based and automata-based approaches.

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

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

The following sections review the major approaches used for the modeling, the specification and the analysis of timed systems. First, several alternatives for the representation of quantitative timing information are reviewed. Next, timed automata are presented as the most commonly used model for timed systems, whereas timed temporal logic is introduced as a specification formalism used to state properties in which a quantitative notion of time is required. Then, some strategies for the representation of the system’s timed state space are analyzed. Finally, some approaches that use Petri nets for the verification of timed systems are reviewed.
3.2 Quantitative timing information

Most of the early works in formal verification were only focused at verifying the functional properties of systems (see Section 1.2.1). In those works, time was present into the models (mostly finite automata) and into the specifications (mostly temporal logic), only as a qualitative notion. In such cases, properties only assert, for example, that a certain condition is always true or that an expected response of the system eventually occurs. While qualitative modeling of time allows the efficient verification of certain properties, it is not satisfactory for verifying the correctness of systems that depend crucially on timing: combinatorial circuits must meet some given clock requirements, embedded controllers must respond to interrupts within some time interval, etc. As a more precise example, consider the statement “trigger the alarm upon detection of an intruder” referred to a security system. This temporal sequencing carries no quantitative information on the delay allowable between the detection and the alarm action. Hence, it is not possible to directly model the triggering of the alarm “less that 5 seconds after detecting the intruder”.

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

Formalisms based on the discrete-time notion (e.g. [AK83, JM86, EMSS90, BMPY97]) map time onto the integer domain. This approach is appropriate to describe the behavior of synchronous systems where all components are driven by a common global clock. However, in order to model asynchronous systems they require to discretize time by choosing some fixed time quantum, so that the separation of two events in the timed domain is always a multiple of such quantum. The main advantage of this approach is that the timing analysis and timed state exploration techniques are generally simpler than their counterparts for continuous-time. However, the main drawback is that determining the time quantum a priori may not be easy and therefore may compromise the accuracy of the resulting model. In this sense, in [BS91] it is shown that the reachability problem for asynchronous circuits with bounded delays cannot be solved correctly using the discrete-time model. Also, the choice of a sufficiently small time quantum to model the system accurately enough, may blow up the timed state space so that verification becomes a no longer feasible task. Figure 3.1 (a) illustrates the concept of discrete-time, where events can only occur at instants multiple of the fixed time quantum.

The fictitious-clock approach introduces a special tick event into the model (e.g. [AH89, Bur89, Ost90, HLP90]). Thus, time is understood as a global state variable that ranges over the domain of natural numbers, and is incremented with every tick event. Generally, this paradigm allows arbitrarily many events of any process between two successive tick
events. The timing delay between two events is measured by counting the number of ticks between them. When it is required that there be \( k \) ticks between two events, it can only be inferred that the actual delay between them is at least \( k - 1 \) time units and at most \( k + 1 \) time units. Therefore, it is impossible to determine precisely some typical and simple requirements on the delays between events, e.g. “the delay between the detection and the alarm equals 2 seconds”. In general, the models based in the fictitious-clock approach require a somewhat cumbersome encoding mechanism to measure time intervals. This reduces the readability of the model and makes modifying the model a tricky and prone to errors task. As a result, ensuring that the model obtained is a good characterization of the actual system is often very difficult. Finally, notice that the discrete-time approach can be seen as a special case of the fictitious-clock approach where the events occur only in lock-step with the ticks. Figure 3.1 (b) illustrates the fictitious-clock approach. Although events may occur at any time, the precise occurrence instant can only be approximated to be in between of two tick events.

The third approach for the modeling of real-time behavior, models time more realistically as a continuous magnitude. Some examples of the use of the so-called continuous-time or dense-time can be found in [DiI89b, Koy90, ACD90, HMP92a, HNSY92, YSSC93, RM94, LPY95, SB97]. These approaches associate a non-negative real value to each event of the system, and therefore to each reachable state. Continuous-time differs from the
other time models because the exact bounds on the actual delays between the events can be expressed. Moreover, the use of continuous-time allows a more precise modeling of analog or asynchronous systems, as well as systems that operate at different clock frequencies. Figure 3.1 (c) illustrates the continuous-time notion.

Since this approach does not rely on the use of a discretization constant, one possible drawback of using the reals as time domain is the added complexity. However, it has been shown that with appropriate techniques (e.g. [AIKY92, HMP92b, ABH+97, TKY+98]), the analysis of continuous-time models does not increase in complexity, if compared to the discrete-time counterparts. The main idea behind such techniques consists in breaking the infinite continuous timed state space into equivalence classes, such that all states in the same class lead to the same behavior and can be analyzed together.

3.3 Timed automata

With the wide adoption of the continuous-time paradigm, the timed automata framework, pioneered by [Di89b, ACD90], has become one of the most popular choices to incorporate quantitative time into the system’s models. Several timing verification tools use this formalism as their basis: COSPA[N [AK95], KRONOS [Yov97], UPAA[L [BLL+95], MOCHA [AHM+98], among others.

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

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

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

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

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

Notice that if \( X \) contains \( k \) clocks, then each clock constraint delimits a convex region in a \( k \)-dimensional Euclidean space. This observation provides a way for representing the timed state space of a timed automaton (see Section 3.5).

The formal definition of a timed automaton follows:

**Definition 3.1 (Timed automata) [AD94]**

A timed automaton is a 6-tuple \( A = (\Sigma, S, S_0, X, I, T) \) such that: \( \Sigma \) is a finite alphabet; \( S \) is a finite set of locations (states); \( S_0 \subseteq S \) is a set of initial locations; \( X \) is a set of clocks; \( I : S \rightarrow \Phi(X) \) is the location invariant; and \( T \subseteq S \times \Sigma \times \Phi(X) \times 2^X \times S \) is a set of transitions.

The 5-tuple \( \langle s, a, \varphi, \lambda, s' \rangle \in T \) is a transition from location \( s \) to location \( s' \) corresponding to the action labeled as \( a \). The clock constraint \( \varphi \) specifies when the transition is enabled, and \( \lambda \subseteq X \) is the set of clocks that are reset when the transition is taken.

**Example 3.1** Consider the timed automaton in Figure 3.2.

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

The invariant \( x < 1 \) associated to locations \( s_1 \) and \( s_2 \), ensures that the c-labeled switch from location \( s_2 \) to \( s_3 \) happens within time 1 of the preceding \( a \). Resetting the other independent clock \( y \) together with the b-labeled switch from \( s_1 \) to \( s_2 \), and checking its value on the d-labeled switch from \( s_3 \) to \( s_0 \) ensures that the delay between \( b \) and the following \( d \) is always greater that 2.

Notice that locations \( s_0 \) and \( s_3 \) have no invariant constraint. This means that the system can spend an arbitrary amount of time in such locations. As a consequence, there is no guarantee that the a-labeled switch from \( s_0 \), or the d-labeled switch from \( s_3 \) are taken at some time instant.
3.3 Timed Automata

The semantics of a timed automaton $A$ is defined by associating a transition system, $\mathcal{T}(A)$, to it. At any time, the configuration or global state of the system modeled by the timed automaton is given by a location, $s$, of the automaton and a clock interpretation, $v$, that assigns a real value to each clock. Thus, a configuration is a pair $(s, v)$ where $s \in S$ and $v : X \rightarrow \mathbb{R}^+$. The set of initial configurations is given by the set $\{(s, v) \mid s \in S_0 \land \forall x \in X [v(x) = 0]\}$, i.e. the set of initial locations in which all the clocks are set to 0.

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

- **Delay transition**: which lets a time delay $\delta \in \mathbb{R}$ to elapse, i.e. increasing the value of all clocks by $\delta$. Then, the system moves from configuration $(s, v)$ to configuration $(s, v')$, written as $(s, v) \xrightarrow{\delta} (s, v')$, where $\forall x \in X v'(x) = v(x) + \delta$.

- **Action transition**: which executes an actual transition $(s, a, \varphi, \lambda, s') \in \mathcal{T}$ of the automaton. This is written as $(s, v) \xrightarrow{a} (s', v')$, such that $v$ satisfies the guard $\varphi$ and $v' = v[\lambda := 0]$.

Thus, the timed state space of a timed automaton can be seen as an infinite transition system $\mathcal{T}(A) = (Q, \Sigma \cup \mathbb{R}, R, Q_0)$, where: $Q$ and $Q_0$ are the set of configurations and the initial configurations, respectively; the original alphabet $\Sigma$ is augmented with the real numbers to include the delay transitions; and $R$ is the transition relation obtained by combining the delay and the action transitions.

**Example 3.1 (cont.)** Let the timed automaton in Figure 3.2 be called $A$.

The state-space of $\mathcal{T}(A)$ is given by $Q \subseteq \{s_0, s_1, s_2, s_3\} \times \mathbb{R}^2$. A sequence of possible transitions is, for example:

$$(s_0, 0, 0) \xrightarrow{1.2} (s_0, 1.2, 1.2) \xrightarrow{a} (s_1, 0, 1.2) \xrightarrow{0.7} (s_1, 0.7, 1.9) \xrightarrow{b} (s_2, 0.7, 0) \xrightarrow{0.1} (s_2, 0.8, 0.1) \xrightarrow{0.1} \cdots$$

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

Solving the reachability problem for a timed automaton is a nontrivial task since the number of potential configurations is infinite. In order to solve the task, finite representations of the infinite state space are required (see Section 3.5). However, even using such representations, the state explosion problem often limits the practical applicability of the algorithms and tools that rely on the reachability set. More precisely, the problem is PSPACE-hard [AD94]. Moreover, in [CY91] it was proved that both sources of complexity, the number of clocks and the magnitude of the constraints yield to PSPACE-hardness independently of each other.
3.4 Timed specifications

Given the model of a timed system, the next step is to state and then verify properties of such system. Some of the properties are just temporal e.g. "when the gate is opened the alarm is always triggered". Other properties involve quantitative delay information, e.g. "the alarm is triggered if the gate keeps opened more than 30 seconds". For the first type of properties, temporal logic may be a good choice. For the second type of properties it is more suitable to use a timed logic, which is an extension of a temporal logic with primitives expressing conditions on the duration of events.

3.4.1 Temporal logic

Temporal logic is a form of logic specifically tailored to state and reason about the notion of order in time, using a simple and clear notation. Time is represented as an implicit magnitude by means of constructs that mimic the time adverbs of natural language (e.g. "always", "until", etc.). The remaining of this section assumes the reader has some familiarity with temporal logic. A good survey about the theoretical foundations behind temporal logic can be found in [Eme90].

Each type of temporal logic offers its own temporal operators which can deal with time according to two basic paradigms. A linear time model assumes that for each time instance there exists exactly one successor time point. This model is particularly well suited for physical time. The resulting linear temporal logic (LTL) was originally pioneered by [Pmu77, Ols82]. Conversely, a branching time model allows several successors of each time instance. This model is appropriate to capture computations, where different execution traces can be selected at a certain step of the ongoing calculation. Hence, time is modeled by tree-like structures where the different possible successor computation paths are chosen non-deterministically. The so-called computational tree logic (CTL) [BAPM81, EC82], follows the branching time paradigm.

In order to establish a comparison between both paradigms, the more expressive logic CTL* [EH86] is briefly introduced first. Despite of the boolean propositions used as the atomic formulas of the logic, CTL* contains both LTL and CTL, thus provides operators for linear and branching time.

Linear time operators make statements about a single computation path (a sequence of states) which starts in the actual state. Thus, the $\Box$ (always) operator indicates that a formula must hold for all successor states on the path; the the $\Diamond$ (sometimes) operator indicates that a formula must hold in some successor state (without telling which one) on the path; the $X$ (next) operator indicates that a formula must hold in the immediate successor state on the path; and the $U$ (until) operator which combines two formulas, where the first one must hold along the path until the second formula becomes true.
The previous operators deal with a single execution path from a given state. Branching time provides two quantifiers over sets of executions which allow to express formulas about the many possible executions starting from a given state. Thus, the quantifiers $A$ and $E$ indicate respectively that, for all paths out of the current state a given formula holds, and that there exists at least one path where the formula holds. It is important not to confuse $A$ and $G$ : the formula $A\phi$ states that all the possible executions from the current state satisfy $\phi$, whereas $G\phi$ indicates that $\phi$ holds at every state of a particular execution being considered.

The aforementioned constructs are summarized in the following grammar for $\text{CTL}^*$:

$$\phi, \psi ::= P_1 \mid P_2 \mid \ldots \quad \text{(atomic propositions)}$$
$$\mid \neg \phi \mid \phi \land \psi \mid \phi \Rightarrow \psi \mid \ldots \quad \text{(boolean operators)}$$
$$\mid F\phi \mid G\phi \mid X\phi \mid \phi U \psi \quad \text{(temporal operators)}$$
$$\mid A\phi \mid E\phi \quad \text{(path quantifiers)}$$

**Example 3.2** Let us consider some properties related to the automaton in Figure 3.3 and how they can be expressed using temporal logic. The automaton consists of three states and three atomic propositions simply called $pa$, $pb$ and $pc$.

In any execution of the automaton, either the proposition $pa$ holds infinitely often or the automaton ultimately remains forever in state $s_2$ where $pc$ holds. This can be expressed with the formula: $GF \, pa \lor FG \, pc$.

Notice that there is one execution from $s_0$ which does not satisfy the formula $pb \, U \, pc$, i.e. that execution in which states $s_0$ and $s_1$ alternate forever.

Notice also that all executions out of $s_0$ visit state $s_1$. Since in one step from $s_1$ a state satisfying $pc$ is reachable, any execution out of $s_0$ satisfies the formula $FEX \, pc$. Observe that the $E$ quantifier is important in this formula, since the execution in which $s_0$ and $s_1$ alternate does not satisfy $FX \, pc$. Thanks to the $E$ quantifier the executions in which $s_2$ follows $s_1$ are also covered.

Although the origins of $\text{LTL}$ and $\text{CTL}$ differ, both can be seen as subsets of the more expressive logic $\text{CTL}^*$. $\text{LTL}$ is obtained from $\text{CTL}^*$ by subtracting the $A$ and $E$ path quantifiers. Thus, a formula in $\text{LTL}$ cannot cover the possible alternative executions.
which split at every state. Similarly, CTL is the subset of CTL\(^1\) in which the use of a
temporal operator must be under the immediate scope of a path quantifier. The basic
valid combinations are: AF, EF, AG, EG, AX, EX, A\(\cup\) and E\(\cup\).

From a syntactical point of view, there are LTL formulas that cannot be expressed in
CTL, and vice versa. Moreover, there are CTL\(^1\) formulas which cannot be expressed in
neither of both. The analysis of the differences between LTL and CTL from a semantical
point of view requires a more detailed study of the types of properties that can be expressed
with each (see [BBF\(^+\)01] for more details).

Reachability properties state that some particular situation can be reached by the sys-
tem. CTL to models reachability properties in a natural way by means of the EF
construct. Thus, EF\(\phi\) can be read as “there exists a path, from the current state, along which
some state satisfies \(\phi\)” . In order to state a reachability property from all reachable states,
the AG and EF constructs must be nested, e.g. AG(EF\(\phi\)) . Conversely, and since LTL
implicitly quantifies for all executions of the system, only the negation of reachability can
be expressed in LTL. That is, “something is never reachable”, e.g. G(\(\neg\phi\)) . However, this
type of property is often seen as a safety property.

Safety properties express that, under certain conditions, something never occurs. Safety
can be expressed naturally in both LTL and CTL, by means of the expressions G\(\phi\) and
AG\(\phi\), respectively.

Liveness properties express that, under certain conditions, something will ultimately
occur. Despite of the discussion of whether liveness properties are useful in practice (see
[BBF\(^+\)01]) it is not easy to formally capture such notion. Two types of liveness properties
are often distinguished: simple liveness or progress, and repetitive liveness or fairness.
Progress is generally easier to formalize, as in the following typical example. The property
“any request will ultimately be satisfied” is expressed as AG(req ⇒ AFsat) in CTL and
as G(req ⇒ Fsat) in LTL. Regarding fairness, in [EH86] it is shown that in contrast to
LTL fairness properties cannot be expressed in CTL.

Deadlock-freeness is a special property relevant in systems which are supposed to operate
indefinitely. Although deadlock-freeness is often seen as a safety property (“something
undesirable will never happen”), theoretically it is not clear if it is actually a liveness
property. Anyway, deadlock-freeness can be expressed in CTL as AG EX true, i.e.
“whatever the state reached is (AG), there exists an immediate successor state (EX true)”.

Finally, remark that both linear and branching time have their strengths and weak-
nesses. Thus the resulting logics, LTL and CTL (and their derivatives) are better suited
for a particular subset of properties and also for a particular class of systems. Moreover,
the verification methodology is often tailored to a specific logic thus gaining in aspects
like efficiency, for example. See [Kro99, BBF\(^+\)01] for more details about this discussion.
3.4 Timed specifications

3.4.2 Timed temporal logic

In order to state and verify timing properties, the simplest way is to express them in terms of the reachability (or non-reachability) of some sets of configurations of the automaton. For more complicated properties observer automata can be used. For example, given a property $\phi$ and a timed automaton $A$, a new automaton $A_{\phi}$ is built and synchronized with $A$. Then, verifying $\phi$ is reduced to testing reachability of some particular states in the resulting composed automaton $A_{\phi}$.

Another possibility is to use a timed temporal logic, which consists in extending with timing constraints the operators of temporal logic. The timing constraints are often expressed in terms of one-sided inequalities or time intervals, and may take integer, rational or real values. Thus, for example the formula $\text{EF}_{<5} \phi$ indicates that “there exists a state satisfying $\phi$ along some execution within 5 time units”. Similarly, the formula $\text{F}_{[3,10]} \phi$ indicates that “some state in which $\phi$ holds is actually reachable, after a minimum of 5 time units, and never later than 10 time units”. Also, the formula $\phi U_{<2} \psi$ states that proposition $\phi$ holds until proposition $\psi$ becomes true, and that $\psi$ will become true within two time units.

To provide an example, the grammar of the timed version of CTL (TCTL) [Koy90] is the following:

$$
\begin{align*}
\phi, \psi & ::= P_1 \mid P_2 \mid \ldots \quad \text{(atomic propositions)} \\
& \mid \neg \phi \mid \phi \land \psi \mid \phi \Rightarrow \psi \mid \ldots \quad \text{(boolean operators)} \\
& \mid \text{EF}_{(\sim k)} \phi \mid \text{EG}_{(\sim k)} \phi \mid \text{E} \phi U_{(\sim k)} \psi \quad \text{(temporal operators)} \\
& \mid \text{AF}_{(\sim k)} \phi \mid \text{AG}_{(\sim k)} \phi \mid A \phi U_{(\sim k)} \psi
\end{align*}
$$

where $\sim$ is a comparison operator from the set $\{<, \leq, =, \geq, >\}$ and $k$ is a rational number.

A wide range of timed temporal logics have been developed along the years by extending in several ways the original LTL and CTL. Corresponding verification algorithms have been also developed, mostly under the paradigm of model-checking.

As derivatives of LTL, the following remarkable examples can be cited [Hen98]: TPTL [AH89] and MTL [Koy90] whose formulas can be verified in exponential time if discrete-time is assumed, but are undecidable if continuous-time is chosen; MITL [AFH91] which can be verified in exponential time regardless of the time paradigm; and ECL [HRS98] which can be always verified in polynomial time. For discrete-time, all these logics are equally expressive. For continuous-time, TPTL is more expressive than MITL, which in turn is more expressive than MTL and ECL.

As derivatives of CTL, RTCTL (real-time CTL) [EMSS90] and TCTL (timed CTL) [Koy90, ACD93] are the most common representatives. Both use the continuous-time domain, are very similar syntactically and semantically, and their verification has PSPACE-complexity. TCTL, for example is supported by the verification tool KRONOS [Yov97].
3.5 Verification of timed systems

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

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

3.5.1 Clock regions

The main idea to overcome the aforementioned complexity is the use of clock regions [ACD90] which we introduce intuitively as follows. Consider two configurations \((s,v)\) and \((s,v')\) of a timed automaton, where the clock valuations \(v\) and \(v'\) are very close. Assume, for example, that there is a single clock \(x\) and that \(v(x) = 1.2347\) and \(v'(x) = 1.235.\) Given a certain notion of closeness for configurations (see below), the automata will behave in roughly the same way from either of both configurations, and hence the same properties will be satisfied.

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

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

- \(\forall x \in X,\) either \([v_1(x)] = [v_2(x)]\) or, \([v_1(x)] > c_x\) and \([v_2(x)] > c_x\)

- \(\forall x, y \in X\) with \(v_1(x) \leq c_x\) and \(v_1(y) \leq c_y, \) \(fr(v_1(x)) \leq fr(v_1(y))\) iff \(fr(v_2(x)) \leq fr(v_2(y)).\)

- \(\forall x \in X\) with \(v_1(x) \leq c_x, \) \(fr(v_1(x)) = 0\) iff \(fr(v_2(x)) = 0.\)
Then, a clock region is an equivalence class of clock valuations induced by $\equiv$. Each region can be characterized by the finite set of constraints it satisfies. For example, given a clock valuation $v(x) = 0.5$ and $v(y) = 0.8$ for clock $x$ and $y$, every clock valuation in the clock region for $v$, denoted $[v]$, satisfies the constraint $0 < x < y < 1$. The following example illustrates these ideas more intuitively.

**Example 3.3** Figure 3.4 [Kro99] shows the set of possible clock regions for two clocks $x$ and $y$. Only constraints of the form $x \sim k$ with $k \in \{0,1,2\}$, and $y \sim k$ with $k \in \{0,1\}$ have been considered. Recall that $\sim \in \{<,\leq,=,\geq,\geq\}$.

The example contains 28 clock regions. Some of them correspond to corner points, like $r_0$, characterized by the constraint $[x = y = 0]$. Other regions are open surfaces in the plane, like $r_9$ characterized by $[0 < y < x < 1]$, or $r_{27}$ characterized by $[x > 2 \land y > 1]$. Finally, the other regions are open segments, like $r_7$, characterized by $[0 < x = y < 1]$.

The system starts in $r_0$ and as time passes, the clocks increase their values simultaneously. Thus, $r_7$ is visited next, then $r_4$, etc. If instead of letting time to elapse, a transition that resets some of the clocks is performed, a region on the axes is reached. For example, the reset of clock $y$ while in $r_7$ leads to $r_8$.

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

The equivalence relation \( \equiv \) over the clock valuations can be extended over the set of possible configurations of the timed automaton. Thus, two configurations are equivalent, i.e. \( (s_1, v_1) \equiv (s_2, v_2) \) iff \( s_1 = s_2 \) and \( v_1 \equiv v_2 \). The resulting equivalence classes of configurations of a timed automaton \( A \), are captured by the so-called region automaton [AD94], denoted by \( \mathcal{R}(A) \). A state in \( \mathcal{R}(A) \) is of the form \( (s, \alpha) \) where \( s \) is a location of \( A \) and \( \alpha \) is a clock region.

The interpretation is that whenever the configuration in \( A \) is \( (s, v) \), the state of \( \mathcal{R}(A) \) is \( (s, [v]) \). Thus, the initial states of \( \mathcal{R}(A) \) are of the form \( (s_0, [v_0]) \) where \( s_0 \in S_0 \) and \( \forall x \in X \, v_0(x) = 0 \). Also, there is an edge \( (s, \alpha) \xrightarrow{a}(s', \alpha') \) in \( \mathcal{R}(A) \) iff \( (s, v) \xrightarrow{a}(s', v') \) in \( A \) for some \( v \in \alpha \) and \( v' \in \alpha' \).

**Example 3.4** Consider the timed automaton and its corresponding region automaton shown in Figure 3.5. Only the regions reachable from the initial region \( (s_0, [x = y = 0]) \) are shown. Notice that the timing constraints cause that the switch from \( s_2 \) to \( s_3 \) is never taken. The only reachable region for location \( s_2 \) satisfies the clock constraint \( [1 = y < x] \). This region has no outgoing edges because, in order for event \( c \) to happen, the constraint \( [x < 1] \) must hold, and that is not possible. \( \blacksquare \) 3.4

3.5.3 Zone automata

Region automata can be easily simplified by collapsing groups of regions into convex geometric regions or clock zones [BD91, ACD+92, AD94]. For example, in the region automaton of Figure 3.5 (b), there are three regions for location \( s_1 \) with associated clock regions \( [y = 0 < x < 1] \, [y = 0, x = 1] \) and \( [y = 0, x > 1] \). These regions could be collapsed to obtain the union \( [y = 0 < x] \), for example. More precisely, a clock zone is formed by a conjunction of clock constraints each of which puts a lower or upper bound on a clock or a difference of two clocks. Given a timed automaton \( A \), its zone automaton \( Z(A) \) can be obtained in a similar way as for the region automaton.

**Example 3.5** Figure 3.6 depicts the zone automaton for the timed automaton in Figure 3.5. Notice that unlike the region automaton, in the zone automaton each vertex has at most one successor per symbol. Also, the number of vertexes of \( Z(A) \) is less than those in \( \mathcal{R}(A) \). \( \blacksquare \) 3.5

Theoretically, in the worst case, the number of zones is exponential with respect to the number of regions, therefore the zone automaton may be exponentially bigger than the region automaton. However, in most practical cases, the zone automaton has less reachable vertexes and provides an improvement in performance. The reason is that, while the number of clock regions depends on the magnitudes of the constants used by the clock constraints, the number of zone regions is relatively insensitive to such fact.
3.5 Verification of timed systems

Figure 3.5 Timed automaton (a) and corresponding region automaton (b).

3.5.4 Difference-bound matrices

Clock zones can be efficiently represented by sets of linear inequalities using difference-bound matrices (DBM) [Dil89b]. Suppose a timed automaton has \( k \) clocks, \( x_1, \ldots, x_k \). Then a clock zone can be represented by a \( (k+1) \times (k+1) \) matrix \( D \). The entry \( D_{ii} \) gives an upper bound of the clock \( x_i \), whereas the entry \( D_{0i} \) gives a lower bound of the clock. For every pair \( i, j \) the entry \( D_{ij} \) gives an upper bound on the difference of clocks \( x_i \) and \( x_j \). To distinguish between a strict and a non-strict bound and allow the absence of a bound, the so-called bounds-domain \( \mathbb{D} \) for the entries of the matrix is
Figure 3.6 Zone automaton for the region automaton in Figure 3.5 (b).

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

**Example 3.6** Consider the clock zone defined by the following constraints:

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

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

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

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

### 3.5.5 Discussion

Although many techniques have been devised to alleviate the state-explosion problem, e.g. partial orders [YSSC93, RM94] or approximations [HPR97], super-exponential improvements in the resulting representations and algorithms are unlikely. This fact is specially true for timed systems. In consequence, other high-level techniques (e.g. abstraction, compositional reasoning, induction, etc.) appear as the more promising ones for future developments.

Nevertheless, several tools exist for the verification of timed systems. The real-time extension of cospan [AK96] allows the analysis of timing constraints using both region
or zone automata. The state space exploration can be performed either by an on-the-fly explicit enumeration or by a BDD-based symbolic approach. With a similar approach, KRONOS [Yov97] supports model-checking of the branching real-time temporal logic TCTL, and has interfaces to a variety of process-algebraic notations. In [BMPY97], an experimental extension to KRONOS is presented, which relies on a canonical representation of discretized sets of clocks configurations using BDDs. The method takes advantage of the symbolic representation and allows to deal with systems that cannot be treated with state-of-the-art DBM-based tools. UPAAL [BLL+95] allows the verification of safety and liveness properties on networks of communicating automata. The check relies on an on-the-fly reachability analysis of the zone automaton. Moreover, compositional reasoning techniques are used to reduce the search space [LPY95].

3.6 Petri net-based methods

Time has been incorporated to Petri nets in several ways. In timed PNs [Ram74] a finite fire duration is associated to each transition of the net. Thus, the firing rule is modified such that transitions must fire once they are enabled but the actual firing has a given duration. Time PNs [MF76] generalize this model by associating a time interval (delay bounds) inside which the transition can fire once it has been enabled. This model is more general than timed PNs and hence has been much more widely used. In contrast to these models, orbital nets [Rok93] associate a pair of delay bounds to the places of the net. The notion of age of a token is defined to capture the time elapsed since a token was put in a place. Then, transitions become enabled only when all its predecessor places are marked and the age of all the tokens belongs to the corresponding time interval. In general, PNs with time associated to places can be easily modeled by PNs with time associated to transitions, whereas the reverse is more complicated [SY96].

PNs augmented with timing information have been extensively used for the verification of timed systems. Two main areas of research can be distinguished: timing analysis, i.e. the computation of the separation time between the occurrence of events; and techniques to alleviate the state explosion problem.

Regarding timing analysis, [MD92] presents a polynomial algorithm for the computation of the minimum and maximum separation time between events in acyclic graphs. Although this work does not refer to PNs it is the precursor of many later works. For example, in [MM93] a polynomial algorithm is presented that estimates the minimum and maximum time differences between events in a cyclic free-choice net. The algorithm unfolds the net into an infinite acyclic graph and examines two finite acyclic sub-graphs to determine the time-separation bounds. The limitation to free-choice nets is partially overcome by the work in [HB94, Hul95]. It provides a way to compute a single exact time separation between two events in a cyclic PN with more general types of choice.
A number of approaches have been provided to alleviate the state explosion problem in timed systems. Most of them [YSSC93, KT94, SY96, VdJL96, BJKY98] rely on the use of partial order techniques derived from the original work on unfoldings of PNs [McM92]. Although these techniques allow significant improvements for highly concurrent systems, their major drawback is that they still require a time region per every sequence leading to each reachable state. To solve this problem, the works in [Rok93, RM94, MRM99] and the related verification tool ORBITS, take a different approach. They reduce the number of time regions per state by using POSETs of events rather than linear sequences, to construct the geometric regions. In turn, they can only handle a class of systems in which the firing time of an event only depends on a single predecessor event. The work started in ORBITS has been extended to deal with a wider class of systems [BM97] and improved with more efficient representations of the timed state space [BMH01]. The resulting techniques have been incorporated to the verification tool ATACs [BMH99].

On a completely different approach, recently [KBS02] have proposed a verification method for timed systems that uses the relative timing paradigm to avoid the computation of the exact timed state space. However, they restrict to a class of systems with only certain types of causality relation between the events.

3.7 Conclusions

The chapter has reviewed the most relevant approaches for the modeling, specification and verification of timed systems: timed automata, timed temporal logic and timed model-checking. Also, relevant approaches to the timing analysis and verification based on the use of Petri nets have been briefly summarized.

A number of verification tools have resulted from all these approaches, however their practical applicability is often restricted to systems with a small state space, or with a particular structure that fits well with a given verification approach. Although efficient methods for the representation of the state space have been devised, the underlying problem is still the state-explosion, which is exacerbated when timing information comes into play.

The verification methodology we propose in the next chapter uses timed transition systems as the underlying formalism to model timed systems under the continuous-time paradigm. Instead of computing the exact timed state space, the relative timing paradigm is used to abstract exact time information from the representation. Hence, LzTΣs are used, which represent the ordering relations between events in the timed domain, by explicitly distinguishing between their enabling and their actual firing conditions. The approach is applicable to systems modeled by timed transition systems without restrictions. For example, no requirement is imposed about the causality relations between events or about the types of choice allowed.
VERIFICATION WITH RELATIVE TIMING

When you are courting a nice girl an hour seems like a second. When you sit on a red-hot cinder a second seems like an hour. That’s relativity.
—Albert Einstein - Quoted in the News Chronicle, 1949

Summary

This chapter presents the theoretical aspects of our relative timing-based verification approach for timed systems. Most of the material was already published in [PCKP00].

First, two small examples conduct a review of the notion of relative timing and an outline of the overall verification strategy.

Next, the different theoretical aspects of the verification approach are introduced. A trace semantics is defined to unify the reasoning with the different computational models used by the verification. The main notion presented is that of enabling compatibility, which makes possible that the timing analysis over the set of events in a trace, can be also applied over a set of traces which share the same enabling orderings. Event structures are then introduced as a model that represents succinctly a set of enabling-compatible traces, and for which efficient timing analysis algorithms exist. The enabling-compatible product of transition systems is then presented as a way to refine the untimed state space of a system with a set of relative timing constraints.

Finally, all these ideas are combined together in a fully automated iterative verification methodology. Relevant aspects such as the correctness and the convergence of the approach are discussed.
4.1 Introduction

The verification of concurrent systems typically suffers from the well known state explosion problem. In systems with a finite number of states, the problem is often alleviated by using symbolic techniques to represent the reachable states. This is also combined with partial order techniques or abstractions that reduce the complexity of the models. However, when time is an essential dimension in the verification problem, complexity is drastically affected. Since the correctness of the system depends on the actual values of event delays and not only on its functional behavior, the verification becomes unmanageable even for moderate-size systems. More precisely, computing the reachability space of a timed system is proved to be a PSPACE-complete problem [AD94], and demonstrated to be highly complex in several practical contexts. Although several techniques have been devised to alleviate such complexity (see Chapter 3), the size of the untimed state space is still the major bottleneck for the analysis of highly concurrent systems.

This chapter describes a novel approach that extends the applicability of the conventional methods based on symbolic reachability analysis, to the verification of safety properties in timed systems. The approach is based on two fundamental facts:

- The observation that the set of runs of a transition system can be covered by a set of event structures [NPW81]. This reduces the verification problem to 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] allows to represent the timed domain of a system in an efficient way. When considering precise delay bounds in timed systems, the complexity blow-up often makes the analysis an intractable problem, even for small systems. Instead, relative timing considers the effect of delays in a system in terms of relative ordering of events (e.g. a happens before b).

The verification approach can be briefly summarized as follows. Rather than calculating the exact timed state space, an off-line timing analysis is performed on a set of event structures that covers the runs leading to system failures. Several timing analysis algorithms have been provided for acyclic graphs, including exact and approximated methods. In our case, the timing analysis is efficiently performed by using McMillan and Dill's algorithm [MD92], which is the precursor of most latter algorithms. The resulting timing constraints are incorporated to the system in the form of relative timing information along a series of iterative refinements of the original untimed state space. If some of the runs leading to failure situations cannot be proved to be timing-inconsistent, then the system is incorrect and the failure run is a counterexample.
4.1 Introduction

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 use of event structures for timing analysis was also proposed in [KBO02]. However, no algorithm was presented that can handle a general class of transition systems for verification. Moreover, the approach presented here, not only verifies the correctness of the system with respect to a set of given safety properties, but also provides as back-annotation a set of timing constraints sufficient to prove such correctness. This information is crucial in frameworks in which synthesis and verification are iteratively invoked to design systems that must meet functional and non-functional constraints.

We want to remark that the application of the method for the verification of untimed systems does not involve any additional overhead with respect to the conventional symbolic methods (e.g. [BCM92]).

4.1.1 Relative Timing

So far we have talked about the idea of relative timing but no illustrative example has been provided that can help to understand some of its benefits, specially in areas other than the verification of timed systems. In this section we reproduce partially an example from [CKK02] where relative timing is used to improve the synthesis of asynchronous control circuits. The synthesis process takes relative timing information into account thus allowing the generation of smaller and faster circuits.

Example 4.1 Consider the asynchronous circuit in Figure 4.1 (a). The delays of the gates are represented by intervals of the form \([d, D]\), which indicate that the output of the gate driving a given signal \(x\) will change \(\delta(x)\) time units after the gate became enabled, with \(d \leq \delta(x) \leq D\). That is, the firing time is bounded by the given delay interval.

After the occurrence of a rising transition of signal \(y\), the behavior represented by the STG of Figure 4.1 (b) is enabled to happen. The rising transition of signal \(b\) appears to be concurrent with the rising transition of signals \(c\) and \(d\). The corresponding underlying TS is depicted in Figure 4.1 (c).
If the actual delays of the gates driving these signals are considered, it is easy to realize that \( b^+ \) will always happen before \( d^+ \). Clearly, the earliest time for \( d^+ \) to occur is 5 time units after \( a^- \), whereas the latest time for \( b^+ \) to occur is only 3 time units after \( a^- \). This observation can be translated into the fact that state \( s_5 \) of the untimed TS will never be reached (see the resulting LzTS in Figure 4.1 (d)).

Provided that \( b^+ \) will always happen before \( d^+ \), the causality relation \( b^+ \rightarrow e^- \) is always guaranteed by the actual delays and the causality relation \( d^+ \rightarrow e^- \). Thus, a potential optimization of the circuit may consider the relative timing constraint between \( b^+ \) and \( d^+ \), and ignore the explicit causality relation \( b^+ \rightarrow e^- \), which leads to the optimized circuit of Figure 4.1 (e).

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

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

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

4.2 Overview

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

This work develops a formal approach to verify that a system with certain timing constraints satisfies a given safety property $P$. The system is modeled by means of a timed transition system, $A$, composed by an underlying transition system, $A^{-}$, and two functions, $\delta$ and $\delta''$, which associate minimal and maximal delays, respectively, to each event of the system. A given sequence of events of a TTS is said to be timing-consistent if it is possible to assign increasing time values to all the events such that their firing times are within the allowed bounds. The modeling formalism of timed transition systems was introduced in Section 2.3.

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

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

The state space of the (timed) event structure is composed with the untimed abstraction of the system $A_0$, in such a way that at least the failure run $\rho$ is removed and no timing-consistent run is removed. A series of successive approximations $A_i$ of $A$ are constructed iteratively, with containment $\mathcal{L}(A) \subseteq \mathcal{L}(A_i)$ and monotonic convergence, $\mathcal{L}(A_{i+1}) \subseteq \mathcal{L}(A_i)$. At every step $\mathcal{L}(A_i) \subseteq \mathcal{L}(P)$ is checked. Verification stops successfully if the inclusion holds, or fails if a counterexample run is found. For a discussion on the convergence of the method refer to Section 4.6.6.

Iterative approaches for the verification of real-time systems have been also presented in [AK95, BSV95]. The major novelty of the approach presented in this thesis is the use of event structures to perform efficient off-line timing analysis, and to incorporate the resulting timing information in the form of relative timing constraints.

**Example 4.2** Figure 4.2 depicts the TTS modeling a simple timed system. Figures 4.2 (a) and (b) show respectively, the underlying (untimed) TS and the delay intervals of events $a$, $b$, $c$ and $g$. The delay interval for the rest of events is assumed to be unbounded, i.e. $[0, \infty)$. Figure 4.2 (e) depicts the state space of the system, when the delays are taken into account the shadowed states are not reachable. A crucial observation is that all runs in the TS of Figure 4.2 (a) that start and end at state $s_0$ can be covered by the two event structures depicted in Figures 4.2 (c) and (d): black states are covered by the event structure (c), white states are covered by the event structure (d) and grey states are covered by both event structures. Thanks to this fact the later verification process can be carried out with just a couple of refinements.

Assume that the property to be verified indicates that event $g$ must always precede event $d$ in any possible run after having visited state $s_0$. The property holds in the timed state space since no state where $d$ can fire before $g$ is reachable. Conversely, the property does not hold in the untimed state space, for example in state $s_{10}$ where $d$ can fire before $g$.

The analysis starts by generating a run that leads to the failure situation, for example a run from $s_0$ to $s_{10}$ followed by the firing of $d$ before $g$ can be generated (Figure 4.3 (a)). Next, an event structure that captures the causality relations of the events in the run is derived (Figure 4.3 (b)). Notice that, in the event structure, $c$ is only triggered by $a$ but not triggered by $b$, as one may expect by looking at the transition system. This is due to the fact that the event structure only contains those causality relations derived from the run. In the failure run under analysis, $c$ is not enabled in $s_1$ and is enabled after having fired $a$ from $s_1$. Thus $a$ triggers $c$, while $b$ is concurrent to it.
Figure 4.2  Example of verification with relative timing: (a,b) TTS and delay intervals. (c,d) Event structures covering the runs starting from $s_0$. (e) Timed state space (shaded states are unreachable).

By timing analysis over the event structure, we find that b and g always precede c. These timing relations are shown as the dotted arcs incorporated to the event structure in Figure 4.3 (c). Such timing analysis is only valid for the causal relations expressed in the event structure, but it is not valid, for example, in the case when b triggers c. Figure 4.3 (d) depicts the state space of the timed event structure, where the shadowed states are not reachable due to the timing relations. Namely, event c is prevented to fire in some states, where its firing would be inconsistent with the timing analysis.

Finally, all this information is incorporated into the system (Figure 4.3 (e)) by composing the original system and the event structure. An event structure being derived from a
particular run gives only partial behaviors of the original system. When the behaviors of the system and the event structure mismatch, the special symbol \( \perp \) is used. Some states in the composed system are split into two instances depending on whether they are reached by runs matching (enabling compatible) the event structure or not (see states \( s_5, s_6, s_{11} \) and \( s_{13} \)). Figure 4.3 (e) shows the resulting system. Notice that the set of runs is smaller than that of the original system, but larger than that of the actual state space when the delays are considered (Figure 4.2 (e)), and that only timing-inconsistent runs have been removed.
This completes the first refinement of the untimed state space taken as starting point. This step has removed some of the failure runs but not all of them. For example, if the new state \((s_{10}, \perp)\) is reached, \(d\) can fire before \(g\) and this contradicts the property under verification. Figure 4.4 summarizes one more refinement. In the resulting system all the failure runs have been removed, which proves that the system satisfies the property. Although it is not generally true, in this case the final state space contains exactly the same runs as the actual state space shown in Figure 4.2 (e).
Several objects and notions have been mentioned along the previous example, such as event structures, enabling compatibility, etc. These and other notions, as well as the theoretical aspects of the verification with relative timing are presented in detail in the following sections.

4.3 Trace semantics

As we have discussed in the previous section, the verification problem is posed in terms of the language generated by the system under verification and the language of all behaviors satisfying a given property. The verification process involves lazy transition systems and event structures (see Section 4.4) as major models. The process consists in an iterative incremental refinement of the system under verification with the timing information derived from the event structures.

A common semantics that unifies the models involved in the verification process can be defined in terms of traces. Based on traces, we will derive several notions that formalize our refinement approach for verification. This flow, illustrated in Figure 4.5, covers the contents of Sections 4.3 and 4.4.

4.3.1 Traces and languages

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

Definition 4.1 (Trace)

Let \( \Sigma \) be an alphabet of events. A trace \( \theta = E_1 \xrightarrow{e_1} E_2 \xrightarrow{e_2} \cdots \) is a sequence such that \( \forall i \geq 1 : E_i \subseteq \Sigma \) and \( e_i \in E_i \), where \( E_i \) denotes the set of events enabled when \( e_i \) fires.

Henceforth, and for the sake of simplicity, all events in a trace will be assumed to be distinct. This assumption can always be enforced by renaming different occurrences of the same event. This renaming does not affect the validity of the theory presented.

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

Definition 4.2 (Traces in lazy transition systems)

Each run \( \rho = s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \cdots \) of a \( LzTS \) defines a trace \( \theta_\rho = E_1 \xrightarrow{e_1} E_2 \xrightarrow{e_2} \cdots \) where \( E_i \) is the set of events enabled at \( s_i \), i.e. \( E_i = \mathcal{E}(s_i) \).

Notions defined over the runs of a transition system can be naturally extended for their traces counterparts. Specially relevant for the verification problem are the notion of
enabling interval of an event along a run (see Definition 2.4), and the notion of timing-consistent run (see Definition 2.7).

Figure 4.6 (a) depicts a run, taken from the TTS in Figure 4.2, and its corresponding trace. A state of the run is substituted in the trace by the set of events enabled at such state in the transition system. The enabling intervals of the events in the trace are depicted as vertical lines in Figure 4.6 (b).

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

**Definition 4.3 (Languages)**

The language \( L(A) \) of a LzTS \( A \) is the set of traces defined by all runs of \( A \). The language \( L(A) \) of a TTS \( A = (A^-, \delta, \delta^u) \) is the set of traces defined by all timing-consistent runs of \( A^- \).

**Lemma 4.1 (Language Inclusion)**

Let \( A = (A^-, \delta, \delta^u) \) be a TTS. Then, its language is a subset of that of its underlying TS, i.e. \( L(A) \subseteq L(A^-) \).

The proof of the lemma directly follows from Definition 2.7 and Definition 4.3.

### 4.3.2 Trace-based verification

In order to solve the verification problem for safety properties, the language of the TTS that models the system under verification must be computed or, at least, conserva-
tively estimated. According to Definition 4.3 this requires a mechanism to check whether a trace of the underlying TS is timing-consistent or not (see Definition 2.7). Checking the timing-consistency of a trace can be formulated quite simply by means of a set of expressions that bound the firing times of the events in the trace according to their delays.

The firing time of event $e_i$, denoted by $ft(e_i)$, is bounded according to the expression:

$$ft(e_j) + \delta'(e_i) \leq ft(e_i) \leq ft(e_j) + \delta''(e_i)$$  \hspace{1cm} (4.1)

where $e_j$ is the event that triggers $e_i$ in the trace. Event enabled in the initial state of the trace have no trigger event, therefore its firing time is only bounded by its delays.

On the other hand, there are events in the trace which are disabled by the firing of another event. For example in the right of Figure 4.7, event $e_k$ is enabled in the trace but its actual firing is prevented by the firing of another (disabler) event $e_i$. The disabling of $e_k$ must occur before the maximum delay since $e_k$ was enabled, has elapsed. Otherwise $e_k$ should have fired yet. Conversely, the disabling can occur as soon as $e_k$ is enabled, no matter if its minimum delay has already elapsed or not. Therefore, the firing time of the disabler event $e_i$ is bounded according to the following expression:

$$ft(e_j) \leq ft(e_i) \leq ft(e_j) + \delta'(e_k)$$  \hspace{1cm} (4.2)

where $e_j$ is the event that triggers $e_k$ in the trace. Since for the disabling to occur in the trace (firing of $e_i$), the disabled event $e_k$ must be already enabled by the firing of
4.3 Trace semantics

\[ e_i \notin E_j \]

\[ e_k \in E_i \]

\[ e_i \in E_{j+1} \]

\[ e_k \notin E_{i+1} \]

\[ \forall 1 \leq i : ft(e_i) \leq ft(e_{i+1}) \]  \hspace{1cm} (4.3)

Figure 4.7 Enabling and disabling in a trace: (a) event \( e_j \) enables event \( e_i \), and (b) event \( e_k \) is disabled by the firing of event \( e_i \) (the disabler).

\( e_j \), the firing of \( e_j \) always happens before that of \( e_i \). Hence, the inequality in the left of expression (4.2) is actually redundant.

Finally, the order in which the events fire along the trace provides additional information for the timing analysis, i.e., time must monotonically increase as long as new events fire. Assuming that the events that fire in the trace are numbered according to their firing order, the following expression must hold:

\[ \forall 1 \leq i : ft(e_i) \leq ft(e_{i+1}) \]  \hspace{1cm} (4.3)

The conjunction of expressions (4.1), (4.2) and (4.3) determine the timing-consistency of the trace. If a solution can be found that assigns firing times to the events of the trace according to the set of inequalities, the trace is timing-consistent. Otherwise, the trace is not timing-consistent and therefore it does not belong to the language generated by the system. Checking the timing-consistency of a trace using the formulation provided by expressions (4.1), (4.2) and (4.3) can be easily performed using linear programming.

**Example 4.2 (cont.)** Figure 4.6 (c) shows the set of constraints of the linear programming model to check the timing-consistency of the trace of Figure 4.6 (b). Since no disabling situation appears in the trace, only expressions (4.1) and (4.3) apply.

In this case, the problem has no solution if the delays shown in Figure 4.2 (b) are considered. Therefore, the trace is not timing-consistent and does not belong to the language of the system. Notice that this result is coherent with that obtained in Example 4.2. The trace was removed from the LzTS obtained after the first refinement of the verification approach (see Figure 4.3).

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

The impossibility of this trace-based method for verification is obvious. The number of
traces between two states of the system may be extremely large or even infinite if cycles
are allowed. Moreover, the number of failure states would suffer from the state explosion
problem as well. Therefore, some strategy to alleviate this complexity is required.

4.3.3 Enabling compatibility

This section provides a fundamental result that helps addressing the complexity problem
exposed by the trace-based verification method outlined above. In short, Theorem 4.1
estates that the results obtained from the timing analysis over a given trace can be applied
to all those traces that have the same causality relations. The theorem is based upon the
notion of enabling-compatibility, that characterizes the relation between traces in which
events are enabled (disabled) by the same triggers (disablers), and events fire in the same
order. Since the time at which an event fires or is disabled only depends on the instant
it became enabled plus certain delays within the given bounds, the timing analysis for a
trace is also applicable to all the traces that are enabling-compatible.

The notion of trace, as it is given by Definition 4.1, does not explicitly distinguish
between those events in a trace that fire after being enabled for some time, and those
events that are disabled by the firing of another event in the trace. However, the disabling
phenomenon is relevant for the timing analysis over a set of traces and must be properly
modeled. In the following definition, that complements Definition 4.1, each element of the
trace keeps track of which events are enabled, which event fires at each step, and which
events are disabled due to such firing.

**Definition 4.4 (Disabling in a Trace)**

Let \( \theta = E_1 \cdot e_1 E_2 \cdot e_2 \cdots \) be a trace. The set \( D_i \subset E_i, \ i \geq 1 \) is the set of events
disabled by the firing of event \( e_i \) in \( \theta \), defined by \( D_i = \{ d \in E_i \mid d \neq e_i \land d \not\in E_{i+1}\} \).
The set of all the events disabled along trace \( \theta \) is the set \( D(\theta) = \bigcup_{i \geq 1} D_i \). We
denote by \( e_i \text{ dis d} \) the fact that event \( e_i \) disables event \( d \). Event \( e_i \) is the
disabler of \( d \) in \( \theta \).

**Example 4.3** The circuit in Figure 4.8 (a) reacts to changes at the input signal \( a \) by
producing some changes at the output signals \( d \), \( e \) and \( f \). In a particular run, after firing
\( a+ \), the AND gate driving signal \( d \) is enabled to rise since inputs \( b \) and \( e \) are both
high. However, a negative transition of \( e \) disables the gate switch. This situation can be
observed in the corresponding untimed TS of Figure 4.8 (b). Transition \( d+ \) is enabled
in state $s_7$ after $b+$. However, when $e-$ occurs and state $s_{12}$ is reached, $d+$ cannot happen anymore. Thus, event $d+$ is enabled for some time and then becomes disabled.

The trace in Figure 4.8 (c) illustrates the disabling phenomenon. Given $E_5 = \{d+, e-\}$ corresponding to state $s_{11}$ and the firing of $e-$, $E_6 = \{a-\}$ ($s_{12}$) is reached where $d+$ is no longer enabled. Thus we have $D_5 = \{d+\} \subseteq E_5$ and $e-$ **disables** $d+$.

Figure 4.8 (d) shows a linear programming model to check the timing-consistency of the trace. According to Definition 2.7 for timing-consistency and the formulation of the problem in the previous section, the inequality $ft(e-) \leq ft(b+) + \delta^n(d+)$ indicates that $e-$ must disable $d+$ before its maximum possible firing time has elapsed. This constraint to expression (4.2) from Section 4.3.2.

---

*Figure 4.8* (a) Circuit with a potential disabling at gate $d$. (b) Portion of the timed state space (shadowed states are unreachable). (c) A run and the corresponding trace illustrating the enabling intervals and the disabling of event $d+$ due to the firing of $e-$. (d) Timing analysis of the trace.
Figure 4.9 Enabling-compatible (left) and non-enabling-compatible mappings (center and right).

With all the above, the following definition introduces the cornerstone notion of the verification strategy presented in this thesis.

**Definition 4.5 (Enabling-compatible trace mapping)**

Let $\theta = \cdots \rightarrow E_0 \xrightarrow{e_1} E_1 \xrightarrow{e_2} E_2 \cdots \xrightarrow{e_n} E_{n+1} \rightarrow \cdots$ be a trace over the alphabet of events $\Sigma$ and let $\theta' = E'_1 \xrightarrow{e'_1} \cdots \xrightarrow{e'_{m+1}} E'_{m+1}$ be a trace over the alphabet $\Sigma' \subseteq \Sigma$. Let $\theta_i = E_1 \xrightarrow{e_1} E_2 \cdots \xrightarrow{e_i} E_{i+1}$ be a fragment of $\theta$.

An enabling-compatible mapping of $\theta_i$ onto $\theta'$ is a function $\text{map} : \{E_1, \ldots, E_{n+1}\} \mapsto \{E'_1, \ldots, E'_{m+1}\}$ such that:

a) $\text{map}(E_i) = E'_i$ \hspace{1cm} (initialization)

b) $\forall 1 \leq i \leq n$, $\text{map}(E_i) = E_i \cap \Sigma'$ \hspace{1cm} (projection)

c) $\forall 1 \leq i \leq n$, $(\text{map}(E_i) = \text{map}(E_{i+1}) \cap e_i \not\in \Sigma') \lor (\text{map}(E_i) = E'_i \land \text{map}(E_{i+1}) = E'_{i+1} \land e_i = e'_i)$ \hspace{1cm} (firing) 4.5

The mapping of $\theta$ onto $\theta'$ is a function that preserves the enabledness of the events in $\Sigma'$. Initially, the events enabled in $E'_i$ must also be enabled in $E_i$ (initialization condition). Next, the events of $\Sigma'$ enabled along $\theta$ and $\theta'$ must be the same (projection condition). Moreover, $\theta$ may fire events that are not relevant to $\theta'$ (when $\text{map}(E_i) = \text{map}(E_{i+1})$ in the firing condition). The second part of the firing condition captures implicitly the disabling of events produced by the firing of $e_i$ in $\theta$ and the firing of $e'_j$ in $\theta'$, that is if an event is disabled in one trace it must be disabled also in the other trace. Since the firing
time of an event only depends on its enabling time and its delay (see Section 2.3), this notion will allow us to apply the timing analysis of $\theta'$ to $\theta$ in the fragment $\theta_1$.

Figure 4.9 shows three examples of trace mapping of the shadowed fragment. The mapping at the left, with $\Sigma' = \{a, c, d, g\}$, is enabling-compatible. The mapping at the center, with $\Sigma' = \{b, c, d\}$, is not enabling-compatible since it violates the projection condition when taking $E_i = \{b, c, g\}$ and $map(E_i) = \{b\}$. Clearly, $a$ enables $c$ in $\theta$, whereas $c$ is enabled by $b$ in $\theta'$. The mapping at the right, with $\Sigma' = \{a, b, c, g\}$, is not enabling-compatible since it violates the projection condition when taking $E_i = \{c, g\}$ and $map(E_i) = \{g\}$. The firing of $b$ in $\theta'$ disables $c$ whereas this does not happen in $\theta$.

The following theorem is the main theoretical result of the verification approach presented in this thesis. The theorem relies on the notion of enabling-compatibility between traces. Since the time at which events fire or are disabled only depends on the enabling instant plus certain delay, the timing analysis on a trace is also applicable to all the traces that share the same enabling, disabling and firing order. That is the timing analysis applies to the set of enabling-compatible traces.

**Theorem 4.1 (Enabling-compatibility and Timing-consistency)**

Let $\theta$, $\theta'$ and $\theta_1$ be traces with the same conditions as in Definition 4.5. Let $map$ be an enabling-compatible mapping from $\theta_1$ onto $\theta'$. Let $\delta^l$ and $\delta^u$ be two functions that assign arbitrary min/max delays to the events in $\Sigma'$ and 0 and $\infty$ delays to the events in $\Sigma \setminus \Sigma'$, respectively. Then, $\theta$ is timing-consistent $\iff \theta'$ is timing-consistent.

**Proof:**

Given that events not in $\Sigma'$ have delays in the interval $[0, \infty)$, no attention must be paid to their timing-consistency. So the proof can be concentrated on the events in $\Sigma'$ that appear in $\theta_1$ and $\theta'$.

$\Rightarrow$

Let $\tau_1 \leq \cdots \leq \tau_n$ be the time stamps assigned to $E_1, \ldots, E_{n+1}$ that make $\theta$ timing-consistent. The same time stamps can be assigned to $\theta'$ as follows.

Let $j$ be the smallest index such that $map(E_j) = E_i'$. Then we assign the time stamp $t_j$ to $E_i$. Under this assignment we have that for any $e_k \in \Sigma'$, the time stamp assigned to $FirstEnabled(\theta', E_j, e_k)$ is the same as the one assigned to $FirstEnabled(\theta, map(E_j), e_k)$. This is ensured by Definition 4.5, that enforces the set of enabled events in $\Sigma'$ to be the same in $E_j$ and $map(E_j)$. Then, since

- the disabling of an event $e \in \Sigma \cap \Sigma'$ must be due to the firing of another event $e_k \in \Sigma \cap \Sigma'$ and such events are enabled and fire at the same time in both sides,
- the disabling of $e$ must also occur at the same time stamp in $\theta$ and $\theta'$.

Now, by Definition 2.7 of timing-consistency, it immediately follows that the assignment of time stamps also makes $\theta'$ timing-consistent.
Given a set of consistent time stamps assigned to \( E'_1, \ldots, E'_{m+1} \), they can also be assigned to \( E_1, \ldots, E_{n+1} \) by the function \( \text{map}^{-1} \). Timing-consistency immediately follows by using a reasoning similar to the previous case. \( \blacksquare \) 4.1

The previous theorem states that the timing analysis of a trace can be reduced to the timing analysis of those events that are causally related (events in \( \Sigma' \)). Therefore, the events that are concurrent with all the events of \( \Sigma' \) can be abstracted out. Hence, the timing analysis for one trace can be applied to all those traces that have the same causality relations among the events in \( \Sigma' \).

We want to remark that the notion of enabling-compatibility as well as the previous theorem have been updated with respect to those in [PCKP00] in order to properly accommodate the disabling notion into the theory.

### 4.4 Event structures

This section presents the basic theory on causal event structures (CES). A CES describes all the possible sequential and concurrent executions of a set of events. Thus, it allows to capture a set of enabling-compatible traces under a single mathematical object. Event structures are the only object for which we perform timing analysis, which is rather simple and efficient because CESs are acyclic directed graphs. The timing constraints derived from such analysis apply to the whole set of enabling-compatible traces covered by the CES.

The usual notion of CES is not able to model the disabling of events, which is a relevant phenomenon for our verification approach. A class of event structures with conflict relations was proposed in [NPW81]. However, such relations are symmetric and correspond to mutual disabling of competing events. A more general relation is required for our purposes that models the asymmetry of event disabling, such as it appears in digital circuits, for example. The main reason for considering asymmetric conflict relations is because, in our verification approach, event structures are derived from traces and a particular trace can only capture a single branch of a conflict relation. Consider the portion of TS in Figure 4.10 (a). It contains a symmetrical conflict since \( x \) and \( y \) mutually disable each other. Two important facts are observed in the trace of Figure 4.10 (b) extracted from such TS:

- Only the disabling of \( y \) due to the firing of \( x \) is captured by the trace.
- Since \( y \) is disabled, it can no longer fire along the trace and therefore it cannot enable other events.

The first fact leads to the need for considering an asymmetric conflict relation. The second fact imposes the restriction that a disabled event cannot have causal successors in the event.
structure. With these two ideas in mind an asymmetric conflict relation, denoted by $\triangleright$, is added to the notion of causal event structure used in [PCKP00], such that the disabling of events along a trace is properly handled.

**Definition 4.6 (Causal Event Structure)**

A causal event structure (CES), $CS = (\Sigma, \triangleright, \triangleleft)$, is a finite set $\Sigma$ of events, a precedence relation $\triangleright \subseteq \Sigma \times \Sigma$ (irreflexive, antisymmetric and transitive) called the causality relation, and a conflict relation $\triangleleft \subseteq \Sigma \times \Sigma$ (irreflexive and antisymmetric).

$\triangleright$ is inherited via $\triangleleft$ in the sense of: $\forall e_1, e_2, e_3 \in \Sigma \mid e_1 \triangleright e_2 \land e_1 \triangleleft e_3 \Rightarrow e_3 \triangleright e_2$.

Moreover $\triangleleft$ and $\triangleright$ satisfy the following two properties:

- $\triangleleft \cap \triangleright = \emptyset$ and
- $e_1 \triangleright e_2 \Rightarrow \exists e_3 \in \Sigma : e_2 \triangleleft e_3$.

That is, the causality and the disabling relations are disjoint, and disabled events cannot be causal predecessors of other events in the CES.

Notice that this definition excludes symmetric conflicts, i.e. mutual disabling between events, by the anti-symmetry of $\triangleright$. This fact does not constitute a limitation of the model but an intended feature that fits in our purposes.

Given a CES $CS = (\Sigma, \triangleleft, \triangleright)$ and two events $e, e' \in \Sigma$, the disabling relation between $e$ and $e'$ is defined as follows:

$$e \triangleright_{\mu} e' \defeq e \triangleright e' \land \forall e_1, e'_1 \in \Sigma : [e_1 \triangleleft e \land e'_1 \triangleleft e' \land e_1 \triangleright e'_1 \Rightarrow e_1 = e \land e'_1 = e']$$

$\triangleright_{\mu}$ identifies the minimal elements (under $\triangleleft$) of the $\triangleright$ relation. The $\triangleright$ relation identifies pairs of events which are inconsistent due to the disabling of some of the predecessors, and propagates to causally-related events generating other conflicts.
A CES can then be depicted as a Hasse diagram, by showing the transitivity-irredundant
\( \prec \) relations in form of solid arcs and the \( \nabla^+_\mu \) relations as dashed arcs. Figure 4.10 (b)
shows a CES that contains the disabling relation \( \nabla^+_\mu y \). Other examples of causal event
structures without disabling relations can be seen in Figure 4.3 (b) and Figure 4.4 (b).

Given a CES \( CS = \langle \Sigma, \prec, \nabla \rangle \) and a set of events \( X \subseteq \Sigma \), the following sets can be
defined [RE88]:

\[
(\rightarrow X)_{\prec} \overset{\text{def}}{=} \{ e_1 \in \Sigma \mid \exists e_2 \in X : e_1 \prec e_2 \}
\]

\[
(X_{\rightarrow})_{\prec} \overset{\text{def}}{=} \{ e_1 \in \Sigma \mid \exists e_2 \in X : e_2 \prec e_1 \}
\]

\[
(\nearrow X)_{\prec} \overset{\text{def}}{=} \{ e_1 \in X \mid \exists e_2 \in X : e_2 \prec e_1 \}
\]

\[
(X^*)_{\prec} \overset{\text{def}}{=} \{ e_1 \in X \mid \exists e_2 \in X : e_1 \prec e_2 \}
\]

\[
(\downarrow X)_{\prec} \overset{\text{def}}{=} \{ e_1 \in \Sigma \mid \exists e_2 \in X : e_1 \preceq e_2 \} = (\rightarrow X)_{\prec} \cup X
\]

When it is clear from the context, we will just write \( \rightarrow X \), \( X_{\rightarrow} \), \( \nearrow X \), \( X^* \) and \( \downarrow X \). Intuitively, \( \rightarrow X \) is the part of \( CS \) before \( X \), \( X_{\rightarrow} \) is the part of \( CS \) after \( X \),
\( \nearrow X \) are the root (with no predecessors) events of \( CS \), and \( X^* \) are the sink (with no
successors) events of \( CS \). Finally, \( \downarrow X \) is called the left-closure of \( X \).

**Definition 4.7 (Words and prefixes)**

A word of the events of \( CS = \langle \Sigma, \prec, \nabla \rangle \) is a sequence \( \omega = e_1 \cdots e_n \in \Sigma^n \ (n \leq |\Sigma|) \),
such that all the events are distinct and \( \forall 1 \leq i, j \leq n : \)

- \( e_i \prec e_j \Rightarrow i < j \) (events are ordered in \( \omega \) according to \( \prec \), and
- \( e_i \nabla^+_\mu e_j \Rightarrow e_j \not\in \omega \) (disabled events do not appear in \( \omega \)).

Given a word \( \omega = e_1 \cdots e_i e_{i+1} \cdots e_n \), the \( i \)-th prefix of \( \omega \) is denoted by \( \omega_i = e_1 \cdots e_i \).
The empty prefix is denoted by \( \omega_0 \). \( \text{4.7} \)

Notice that an event \( e_j \) for which a disabling relation \( e_i \nabla^+_\mu e_j \) exists in the CES cannot
fire along a word \( \omega \). However, \( e_j \) will become enabled somewhere in \( \omega \) as long as its
predecessors by \( \prec \) fire in \( \omega \). Also, \( e_j \) will be disabled when its predecessors by \( \nabla \) fire in \( \omega \). This notions are formally captured by the following definition:

**Definition 4.8 (Events enabled/disabled by a prefix)**

Let \( CS = \langle \Sigma, \prec, \nabla \rangle \) be a CES and let \( \omega \) be a word of \( CS \). The set of events
enabled by a prefix \( \omega_i \) is defined as \( E(\omega_i) = \{ e_k \not\in \omega_i \mid \forall e_j \in \Sigma : e_j \prec e_k \Rightarrow e_j \in \omega_i \wedge e_j \nabla^+_\mu e_k \Rightarrow e_j \not\in \omega_i \} \). Similarly, the set of events disabled by a prefix \( \omega_i \) is
defined as \( D(\omega_i) = \{ e_k \not\in \omega_i \mid \exists e_j \in \Sigma : e_j \nabla^+_\mu e_k \Rightarrow e_j \in \omega_i \} \).

That is, an event \( e_k \) is enabled by a prefix \( \omega_i \) if all the causal predecessor events (by \( \prec \))
are in \( \omega_i \), none of its disablers are in \( \omega_i \) and \( e_k \) is not in \( \omega_i \). Also, an event is disabled by
\( \omega_i \) if it was enabled and later disabled along \( \omega_i \) but it never fired. \( E(\omega_i) \) contains all the events still enabled by the sequence of firings in \( \omega_i \), while \( D(\omega_i) \) accumulates all the events disabled along \( \omega_i \). In the sequel we will denote by \( D \subseteq \Sigma \) the set of all events that are disabled along some word of a CES, i.e. \( D = \{ e_i \in \Sigma \mid \exists e_j \in \Sigma : e_j \triangleright e_i \} \).

The notion of word in a CES is similar to the general notion of trace. This fact is expressed by the following definition:

**Definition 4.9 (Traces generated by words)**

Let \( CS = \langle \Sigma, \prec, \triangleright \rangle \) be a CES and let \( \omega = e_1 e_2 \cdots e_n \) be a word of \( CS \). The trace generated by \( \omega \) is defined as: \( \theta_\omega = E(\omega_0) \xrightarrow{e_1} E(\omega_1) \xrightarrow{e_2} \cdots \xrightarrow{e_{n-1}} E(\omega_{n-1}) \xrightarrow{e_n} \emptyset \).

**4.9**

According to this definition, events disabled in the CES can appear only in the sets of enabled events in each state of \( \theta_\omega \), but never in the transitions between states.

**Definition 4.10 (Causal event structure generated by a trace)**

Let \( \theta = E_1 \xrightarrow{e_1} E_2 \xrightarrow{e_2} \cdots \xrightarrow{e_{n-1}} E_n \xrightarrow{e_n} E_{n+1} \) be a finite trace with \( D(\theta) \) as the set of disabled events along it. The causal event structure \( CS_\theta = \langle \Sigma, \prec, \triangleright \rangle \) generated from \( \theta \) is defined as follows:

- \( \Sigma = \bigcup_{i=1}^{n} E_i \).

Not only the firing events are included but all those enabled along the trace, including the disabled ones.

- \( e_i \prec e_j \iff i < j \land \{ \exists E_k \in \theta : \{ e_i, e_j \} \subseteq E_k \} \land e_i \notin D(\theta) \).

The last condition emphasizes the fact that disabled events can not be causal predecessors of other event since they do not fire along the trace.

- \( e_i \triangleright e_j \iff \begin{cases} e_i \text{ dis } e_j \text{ in } \theta, \text{ or } \\ e_k \triangleright e_j \land e_k \prec e_i \end{cases} \)

\( \triangleright \) captures the non-symmetric conflicts along the trace, that is \( e_i \) may disable \( e_j \) but this has nothing to do with \( e_j \) disabling \( e_i \). Both relations can never appear together in the same trace. \( \triangleright \) corresponds exactly to the dis relation of the trace.

**4.10**

**Example 4.3 (cont.)** Figure 4.11 (a) depicts a trace extracted from the TTS in Figure 4.8. The trace contains the disabling of event \( d+ \). Figure 4.11 (b) shows the CES derived from the trace according to Definition 4.10. The disabling relation between \( e- \) and \( d+ \) is represented by the dashed arc.
The following sequences of events are words of the CES: \(a+b+c-f-e-\), \(a+f-c-e-b+\), \(a+c-b+e-f-\), etc.

Consider \(\omega = a+b+c-f-e-\), then \(\omega_1 = a+\) is the first prefix of \(\omega\), \(\omega_2 = a+b+\) is the second prefix of \(\omega\), etc. The set of events enabled and disabled by \(\omega_2\) are \(E(\omega_2) = \{c-,d+,f-\}\) and \(D(\omega_2) = \emptyset\). Similarly, the set of events enabled and disabled by \(\omega_5\) are \(E(\omega_5) = \emptyset\) and \(D(\omega_5) = \{d+\}\). Finally, according to Definition 4.9, \(\omega\) generates the trace of Figure 4.11 (a).

Despite of the previous example, Figure 4.10, Figure 4.3 and Figure 4.4 show other examples of CESs derived from a given trace.

We use CESs derived from failure traces to perform timing analysis. Hence, relative timing relations among the events in the CES can be found that help to prove the timing-consistency or timing-inconsistency of a failure trace. Moreover, thanks to Theorem 4.1, the timing relations derived from the analysis also apply to the set of traces enabling-compatible with the failure trace.

### 4.4.1 Timing analysis on event structures

CESs with timing assumptions can be derived from traces with events annotated with minimum and maximum delay bounds (see Definition 4.10). These assumptions are captured by the notion of maximal separation time between the events of a CES. The maximal separation time of two events \(e_1\) and \(e_2\) is computed as the maximum difference be-
between their firing times, provided any possible assignment of delays to the events in the CES. That is, \( Sep_{max}(e_1, e_2) = \max \{ f_t(e_1) - f_t(e_2) \mid \text{for any delay assignment} \} \), where \( f_t \) denotes the firing time of an event.

In [MD92] several algorithms for the timing analysis on acyclic graphs where presented. Those algorithms included: a polynomial algorithm for the timing analysis with \( max \) constraints only; an exponential, but feasible in practice, algorithm for the case with \( max \) and linear constraints; and a branch and bound approach for the general case including \( min/max \) and linear constraints. The information obtained from these algorithms can be used to analyze whether two concurrent events are actually ordered in the timed domain. That is, \( e_1 \) precedes \( e_2 \) in the timed domain if \( Sep_{max}(e_1, e_2) < 0 \). Appendix A provides details on the timing analysis algorithm of [MD92] for \( max \) constraints.

The verification approach presented in [PCKP00] used the algorithm of [MD92] with only \( max \) constraints to perform timing analysis on CES derived from traces without disabling relations. However, such algorithm is not sufficient when the disabling of events is involved. The following example illustrates why.

**Example 4.3 (cont.)** Recall the circuit of Figure 4.8 (a) where a falling transition of gate \( e \) disables the rising transition of gate \( d \). Assume also, that a situation in which if “\( f^- \) occurs before \( e^- \) once \( a^+ \) has happened” is considered a failure by the designer of the circuit. A trace that captures such failure and also includes the aforementioned disabling situation is shown in Figure 4.11 (a). Provided the delay bounds of the events shown in Figure 4.8 (a), it can be proved that the trace is not timing-consistent. Notice that if \( e^- \) disables \( d^+ \), then \( e^- \) must fire before the maximum possible firing time of \( d^+ \) has elapsed (see Definition 2.7). That is, the latest firing time of \( e^- \) must be, some amount of time before 6 time units, after \( a^+ \) fired. This means that \( e^- \) will always fire before \( f^- \), whose minimum delay is also 6 time units and is also triggered by the reference event \( a^- \). This fact can be better analyzed by looking at the CES in Figure 4.11 (b), in which events have been annotated with their respective delay intervals.

According to the discussion in the previous example, the timing analysis on the CES should provide a relative timing relation showing the fact that \( e^- \) always fires before \( f^- \). However, the \( max \)-only algorithm cannot handle the disabling situation and no such timing relation can be obtained. This indicates that \( e^- \) and \( f^- \) can occur concurrently in the timed domain. Which, in turn, implies that the failure trace is possible even when the delays are considered. Therefore the circuit is faulty. This leads to contradiction since the failure trace of the example is not timing-consistent.

The source of the contradiction resides in the fact that the disabling relation between \( e^- \) and \( d^+ \) cannot be expressed in the \( max \)-only algorithm for timing analysis, which can only handle the causal relations among the events. Moreover, such disabling relation is
relevant for the timing analysis of this case. If the disabling is not considered, the relative timing relation between \( e^- \) and \( f^- \) necessary to prove the timing-inconsistency of the failure trace, cannot be found.

Disabling relations are incorporated to the timing analysis of a CES in the form of linear constraints. Such constraints express the fact that if event \( e_i \) disables event \( e_k \), then \( e_i \) fires before the maximum delay of \( e_k \) has elapsed, since it was enabled by its trigger \( e_j \). Otherwise \( e_k \) should have fired before the disabling could take place. More formally, given a CES \( CS = \langle \Sigma, \prec, \succ \rangle \), such that \( e_i \preceq e_k \) and \( e_j \prec e_k \) \((e_i, e_j, e_k \in \Sigma)\) a linear constraint of the form \( f(t(e_i)) \leq f(t(e_j)) + \delta^\mu(e_k) \) is added to the timing analysis. Then the timing analysis algorithm for max and linear constraints in [McM92] can be used.

In the previous example, a linear constraint corresponding to the disabling of \( d+ \) by \( e^- \) is imposed to the timing analysis, such that \( e^- \) must fire before the delay of \( d+ \) elapses. That is \( f(t(e^-)) \leq f(t(b+)) + \delta^\mu(d+) \). Under this condition, which reflects what happens in the trace, we have that \( Sep_{\text{max}}(e^-, f^-) < 0 \). As we expected, this means that \( e^- \) precedes \( f^- \) in the timed domain.

The maximum separation times computed by the timing analysis algorithm can be incorporated to the CES in the form of relative timing constraints between pairs of events. Thus, \( e_1 \) precedes \( e_2 \) in the timed domain if \( Sep_{\text{max}}(e_1, e_2) < 0 \). We refer to these new relations as lazy relations, expressing the fact that \( e_2 \) is lazy to fire until \( e_1 \) has fired. This notion of laziness is similar to that for lazy transition systems (see Section 2.4).

**Definition 4.11 (Lazy CES generated by a trace)**

Let \( \theta = E_1 \xrightarrow{e_1} \cdots E_n \xrightarrow{e_n} E_{n+1} \) be a trace of a TTS \( A = \langle A^-, \delta^\mu, \delta^\nu \rangle \), and let \( CS_\theta = \langle \Sigma, \prec, \succ \rangle \). The triple \( LCS = \langle \Sigma, \prec', \succ' \rangle \) is called a lazy causal event structure (LzCES), where \( \prec' = \prec \cup T \), and \( T \subseteq \Sigma \times \Sigma \) is a set of lazy relations such that \( T = \{(e_i, e_j) \in \Sigma \times \Sigma \mid e_i \not\prec e_j \land e_j \not\prec e_i \land Sep_{\text{max}}(e_i, e_j) < 0\} \).

The LzCES obtained after the timing analysis on the CES of Example 4.3 is shown in Figure 4.11 (c). The lazy relations corresponding to the relative timing information are depicted as dotted lines. Despite of the previous example, Figure 4.3 and Figure 4.4 show other examples of LzCESs derived from a given trace.

The delays assigned to the events in the CES for timing analysis play a crucial role in the context of our verification approach. Failure traces and/or CESs may come from a variety of sources: a designer, who’s knowledge about the possible sources of failures in a system can be useful to guide the verification; a CES derived not from a whole trace that starts from the initial state, but from a portion of trace that ensures a localized failure analysis that involves less events; etc. In these cases, the prehistory of the enabledness of some events involved in the analysis may be unknown to the verification algorithm. As
a consequence, given a CES $CS = \langle \Sigma, \prec, \triangleright \rangle$, the minimum delay bound for all the root events $((c^* \Sigma)_\omega)$ i.e., those for which no causal predecessor exists, is conservatively set to 0. That is, mimicking an infinitely early enabling time. With this strategy, timing analysis is still exact in case the CES has only one root event, since the relative firing order of all other events does not depend on the enabling time of their common predecessor.

A $L_zCES$, obtained by the techniques described in previous sections, partially specifies the behavior of the system under verification and incorporates a set of relative timing constraints. Such timing constraints are mapped back to the system under verification by means of composing appropriately the system and the $L_zCES$.

4.5 Enabling-compatible product

This section describes how to refine the set of traces produced by a $L_zTS$ by considering the timing constraints coming from event delay bounds. The timing constraints are derived by a timing analysis on a CES corresponding to an eligible trace of a $L_zTS$ in the untimed domain. The refinement is performed through the parallel composition of a $L_zTS$ and a $L_zCES$. Defining such composition requires both descriptions to be represented in a uniform way. To satisfy this requirement we first introduce a state-based representation for CESs.

4.5.1 State-based representation of a CES

An underlying transition system can be obtained from a CES. This process relies on the notion of configuration, which plays the role of global state of the CES.

**Definition 4.12 (Configuration)**

Let $CS = \langle \Sigma, \prec, \triangleright \rangle$ be a CES. $C \subseteq \Sigma$ is a configuration iff:

- $C$ is left-closed, i.e. $\forall e_i \in C$ all predecessors of $e_i$ by $\prec$ are in $C$, and
- disabled events do not belong to $C$, i.e. $e_i \in C \Rightarrow \exists e_j \in \Sigma : e_j \triangleright\triangleright e_i$.

Notice that both $\emptyset$ and the set of not disabled events $\Sigma \setminus \mathcal{D}$ are trivial configurations.

Event $e \in \Sigma$ is enabled in configuration $C$ iff $\vdash \{e\} \subseteq C$ and $\forall e_j \in \Sigma : e_j \triangleright e_i : e_j \notin C$. We denote by $\mathcal{E}(C)$ the set of all enabled events in configuration $C$.

Configuration $C$ precisely identifies a state of a CES, as the set of events occurred so far, such that if $e \in C$ all its causal predecessors must be also in $C$.

Every prefix $\omega_i$ of a word $\omega$ in a CES is left-closed and disabled events do not fire along it (see Definition 4.7). Thus every prefix $\omega_i$ defines a configuration which is reached by firing the events from $\omega_i$. Consideration of all possible words of a CES and their prefixes gives the set of reachable configurations, $C$, where the initial configuration due to the empty prefix $\omega_0$ is denoted by $\top$. The set of reachable configurations together
with the partial order induced by the strict set inclusion $\subset$, defines the \textit{graph of reachable configurations}.

**Definition 4.13 (Graph of reachable configurations)**

Let $CS = (\Sigma, \prec, \triangleright)$ be a CES, and $C$ be the set of reachable configurations of $CS$. The graph of reachable configurations (GRC) of $CS$ is a Hasse diagram over $C$ and the partial order $\subset$ interpreted in set-theoretical sense. \[\text{4.13}\]

For the general case of a LzCES, $LCS = (\Sigma, \prec', \triangleright)$, the graph of reachable configurations can be modeled by a LzTS $G = \langle C, \Sigma, T, \top, \text{EnR} \rangle$ where: there is one state per configuration; $C_1 \rightarrow e \rightarrow C_2 \in T$ iff $C_2$ is reached by firing $e \in \Sigma$ from $C_1$; the initial state corresponds to the initial configuration $\top$; and $\text{EnR}(e) = \{C \in C \mid e \in \mathcal{E}(C)\}$.

**Example 4.3 (cont.)** Figure 4.12 (a) depicts the resulting graph of reachable configurations for the CES in Figure 4.11 (b). In this graph every arc $(C_1, e, C_2)$ is attributed by an event $e$ which expands configuration $C_1$ into $C_2$ (the firing event). The shadowed configurations are those unreachable due to the laziness of $f-$ relative to $b+$ and $e-$ as imposed by the lazy arcs of the LzCES of Figure 4.11 (c). Thus, we have that $\text{EnR}(f+) = \{a+, {a+, b+}, {a+, c-}, {a+, b+, c-}\}$ and $\text{FR}(f+) = \{a+, b+, c-, e-\}$. \[\text{4.3}\]

\[\text{Figure 4.12} \quad (a) \text{ Graph of reachable configurations for the LzCES of Figure 4.11 (c).} \quad (b) \text{ Corresponding graph of reachable enablings. Shadowed configurations are not reachable due to the laziness of event f-.} \]
The following theorem shows that a configuration in a CES is uniquely defined by the set of events enabled in it. The result applies in general to a LzCES since its set of reachable configurations is a subset of that of the original CES from which the LzCES was derived.

**Theorem 4.2 (Configurations and Enablings)**

Any pair of configurations $C_1$ and $C_2$ ($C_1 \neq C_2$) of a CES $CS = \langle \Sigma, \prec, \triangleright \rangle$ has different sets $\mathcal{E}(C_1)$ and $\mathcal{E}(C_2)$ of enabled events, i.e. $C_1 \neq C_2 \Rightarrow \mathcal{E}(C_1) \neq \mathcal{E}(C_2)$.

**Proof:**

By contradiction. Suppose that $\mathcal{E}(C_1) = \mathcal{E}(C_2)$. Two cases arise:

- $C_1 \subseteq C_2$ The configurations are ordered (similarly for $C_2 \subseteq C_1$).

Then, any sequence of firings $\sigma$, from $C_1$ to $C_2$ must contain at least one event $e \in \mathcal{E}(C_1)$. If $\mathcal{E}(C_1) = \mathcal{E}(C_2)$, we have that $e$ fires in $C_1$ but is again enabled in $C_2$. Therefore, $e \prec e$, which is a contradiction.

- $C_1 \nsubseteq C_2 \land C_2 \nsubseteq C_1$ The configurations are not ordered.

Let us consider the nearest predecessor configuration $C_3$ such that $C_3 \subset C_1$ and $C_3 \subset C_2$, and there is no configuration $C_4$ such that $C_4 \subset C_1$, $C_4 \subset C_2$, and $C_3 \subset C_4$.

Let $e$ be the first event firing in any feasible sequence of firings $\sigma_1$ from $C_3$ to $C_1$. Clearly, $e$ does not appear in any feasible sequence of firings $\sigma_2$ from $C_3$ to $C_2$, otherwise $C_1$ and $C_2$ would be reachable from $C_4$ such that $C_3 \xrightarrow{e} C_4$. Therefore, $e$ should be still enabled in $C_2$. Since we assumed that $\mathcal{E}(C_1) = \mathcal{E}(C_2)$ we have that $e$ is again enabled in $C_1$ and therefore, $e \prec e$, which is a contradiction. $\blacksquare$

In the sequel we will indistinctly use configurations or their enablings to characterize the states of a CES. Based on this one-to-one correspondence instead of a graph of reachable configurations one could consider an isomorphic graph of reachable enablings (GRE). Figure 4.12 (b) shows the GRE corresponding to the GRC of Figure 4.12 (a).

### 4.5.2 Refining the reachability space by timing constraints

At this moment we have two objects at hand: a lazy TS $A$, and another lazy TS $G$ obtained from an event structure $CS_\theta$. $CS_\theta$ is derived from a particular trace $\theta$ of $A$ (actually by an appropriate suffix), thus giving only a partial specification of the behavior of $A$. $CS_\theta$ is refined through timing analysis yielding the lazy TS $G$.

Refining the behavior of $A$ by the timing constraints incorporated in $G$ can be done by calculating the enabling-compatible product of $G$ and $A$, which is a particular case of transition system product under the restrictions of making synchronization by the same transitions and the same enabling conditions.

For sake of simplicity, and before introducing the rules of the enabling-compatible product below, we will add the special configuration $\perp$ to $G$. $\perp$ denotes the fact that the
product is not synchronizing, \textit{i.e.} there is no enabling-compatibility with the state space of the CES and therefore, timing analysis does not apply for the involved traces.

Given the system \( A = \langle S, \Sigma_A, T_A, s_0, \text{En} \rangle \) and the state space of the LzCES containing the relative timing constraints \( G = \langle \Sigma, T, \Sigma_G, T_G, \text{En}_G \rangle \), with \( \Sigma_G \subseteq \Sigma_A \), the enabling-compatible product of \( A \) and \( G \) is a new LzTS \( \langle S', \Sigma_A, T', s'_0, \text{En}_R \rangle \) where:

- \( S' \subseteq S \times (C \cup \bot) \),
- \( s'_0 = (s_0, \top) \) if \( \mathcal{E}(\top) \subseteq \mathcal{E}(s_0) \), and \( s'_0 = (s_0, \bot) \) otherwise, and
- \( \forall e \in \Sigma_A, \text{En}_R'(e) = \{(s, c) \in S' \mid s \in \text{En}_R(e)\} \).

The transition relation \( T' \) is defined by the rules below. The rules are implied by the conditions of Definition 4.5 on enabling-compatibility of traces. The fact that \( (s, c) \in S' \) denotes that \( s \) and \( c \) have been reached by prefixes that are enabling-compatible, and that \( \text{map}(\mathcal{E}(s)) = \mathcal{E}(c) \). Given a state of the product \( (s, c) \) with \( C \neq \bot \), we will say that the state is in the timed domain, indicating that the timing analysis performed on \( C S_0 \) can be applied to \( s \).

The rules that define the enabling-compatible product are as follows:

**Transitions entering the timed domain**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s, \bot) \xrightarrow{\epsilon} (s', \top))</td>
<td>enter ( \equiv s \mathrel{\xrightarrow{\epsilon}} s' \in T_A \land \mathcal{E}(\top) \subseteq \mathcal{E}(s') \cap \Sigma_G )</td>
</tr>
</tbody>
</table>

These transitions are fired when the events enabled in \( \top \) are also enabled in \( s' \). Thus, timing analysis can start being applied from \( (s', \top) \).

**Staying inside the timed domain**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s, c) \xrightarrow{\epsilon} (s', c))</td>
<td>inside1 ( \equiv s \xrightarrow{\epsilon} s' \in T_A \land \mathcal{E}(s) \cap \Sigma_G = \mathcal{E}(s') \cap \Sigma_G )</td>
</tr>
<tr>
<td>((s, c) \xrightarrow{\epsilon} (s', c'))</td>
<td>inside2 ( \equiv s \xrightarrow{\epsilon} s' \in T_A \land c \xrightarrow{\epsilon} c' \in T_G \land \mathcal{E}(s') \cap \Sigma_G = \mathcal{E}(c') )</td>
</tr>
</tbody>
</table>

\( \text{inside1} \) corresponds to the condition in which \( \epsilon \) does not synchronize with \( G \). Here the enablings of configuration \( C \) must be preserved, \textit{i.e.} the firing of \( \epsilon \) cannot disable or enable events in \( \Sigma_G \).

For \( \text{inside2} \), both \( A \) and \( G \) make a synchronized move which might affect the events from \( \Sigma_G \) in exactly the same way: if \( a \in \Sigma_G \) becomes enabled in \( A \) due to this move, it should also become enabled in \( G \), and vice versa.

**Exiting or staying outside the timed domain**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s, c) \xrightarrow{\epsilon} (s', \bot))</td>
<td>exit ( \equiv s \xrightarrow{\epsilon} s' \in T_A \land \neg(\text{enter} \lor \text{inside1} \lor \text{inside2}) )</td>
</tr>
</tbody>
</table>
4.6 Verification methodology

The different elements of the verification methodology have been introduced along the

EXAMPLE 4.3 (CONT). Figure 4.13 summarizes the refinement of the state space of

of the circuit of the running Example 4.3. The figure shows the circuit (a) and a

of the circuit are shown in (c) and (d), respectively. The LTS corresponding to

the circuit is shown in (e). Finally, (f) shows a portion of the resulting

LTS after performing the enabling-compatible product of (a) and (b). States
correspond to those where the enabling-compatible product is

not satisfied, and thus are not out of the product. Notice that all traces where event d+ is

load out of the product, since they are not enabling-compatible with the LCES (d), where
event d+ is disabled.

Despite of the previous example, Figure 4.3 and Figure 4.4 show other examples of

enabling-compatible product.

It can be shown that, in the enabling-compatible product, only the traces which are not timing-consistent with respect to the timing constraints coming from the timing analysis on the event structure. All other traces are not changed, thus guaranteeing the conservativeness of the approach.

LTS which are enabling-compatible with the event structure are pruned. This refinement

exhausts the traces which are not timing-consistent with respect to the timing constraints.

prove that some of the states in the system are unreachable. In particular, at least the
Figure 4.13  (a) Circuit with a potential disabling at gate d.  (b) Portion of the untimed state space and (c) trace extracted from it.  (d) LzCES induced by the trace (c) and delays.  (e) LzTS of the corresponding GRC and (f) resulting LzTS after the enabling-compatible product of (b) and (e).
failure trace found in the initial step is removed. The process is repeated until no violation of the property \( P \) exists or a timing-consistent failure trace is found.

Along the series of refinements each LzCES is reported. At the end of the process, the resulting set of LzCESs constitute a set of sufficient relative timing constraints that prove the correctness of the system. They can also be used as valuable back-annotation information to help the designer improve his/her knowledge of the parts of the system which are critical for its correct operation.

### 4.6.1 Iterative refinement

Figure 4.15 shows the timed verification algorithm, where \( A \) is the TTS that models the system under verification, and \( P \) is a safety property. In general, a set of safety properties can be handled simultaneously with similar computational effort.

First, a LzTS \( A' \) is obtained corresponding to the underlying TS of \( A \). The enabling information in \( A' \) coincides with that of firing, since no refinement with the timing information has been carried out yet.

The function \texttt{untimed\_verification} checks whether a trace violating the property \( P \) is present in \( A' \). If such a trace exists, a finite prefix, \( \theta \), demonstrating the wrong behavior is returned. This prefix is checked for timing-inconsistency by building and analyzing the corresponding causal event structure (see function \texttt{build\_event\_structure} in Figure 4.16). If no CES can disprove the feasibility of the trace \( \theta \) the verification returns \( \theta \) as an
function timed_verification ( A = \langle S_A, \Sigma_A, T_A, s_0, A, \delta, \delta^u \rangle, P )

A' = \langle S_A, \Sigma_A, T_A, s_0, A, \delta^1, \delta^u \rangle

repeat

\theta := untimed_verification ( A', P )

if (empty \theta) return (success)

LCS := build_event_structure ( A', \theta, \delta^1, \delta^u )

if (empty LCS) return (FAIL, \theta)

A'' := compose ( A', LCS )

A' := A''

end repeat

end function

Figure 4.15 Main algorithm of the relative timing-based verification approach.

function build_event_structure ( A' = \langle S, \Sigma, T, s_0, EnR \rangle, \theta, \delta^1, \delta^u )

\theta'' := shortest_suffix ( \theta )

repeat

\theta'' := add_predecessor ( \theta'', \theta )

CS := build_event_structure ( A', \theta'' )

if (timing_consistent ( CS, \delta^1, \delta^u ) )

L := compute_lazy_arcs ( CS, \delta^1, \delta^u )

LCS := add_lazy_arcs ( CS, L )

return ( LCS )

end if

while ( \theta'' \neq \theta )

return ( empty CS )

end function

Figure 4.16 Algorithm for the derivation of a LzCES from a trace.

example of violation of P. Otherwise the system is refined through the composition with the LzCES LCS. LCS contains a set of relative timing constraints that apply over a set of enabling-compatible traces, including \theta.

The timed_verification algorithm does not depend on any particular implementation of the untimed_verification function. We have implemented, however, an approach based on efficient symbolic model checking techniques [BCM+92]. Basically, we explore A' looking for failure states where P is violated. Then, a backward traversal is performed to generate
4.6 Verification methodology

Figure 4.17 Generation of the sufficient shortest suffix of a trace. Three steps are needed to obtain a LzCES that proves the timing-inconsistency of the trace.

a trace, leading from the initial state to the failure one, reproducing the discrepancy with $P$. A fast simulation-guided traversal technique [PP03] has been also implemented. With this technique, the cost of the search for property violations is drastically reduced, and the failure trace is incrementally built during the process. Moreover, significant savings in CPU time and memory requirements are achieved.

4.6.2 Off-line timing analysis of failures

The function `build_event_structure` (see Figure 4.16) builds the shortest suffix $\theta''$ of the trace $\theta$ generated by the function `untimed_verification`. $\theta''$ is built such that the timing analysis shows a timing-inconsistency with the delays $\delta'$ and $\delta''$ imposed by $A$. A causal event structure $CS$ is constructed by using the causal relations of the events in $\theta''$ (see Section 4.4).

Function `timing_consistent` performs timing analysis over $CS$. It implements the algorithm described in [MD92] for timing analysis over an acyclic graph of events with min/max and linear constraints (see Section 4.4.1).

If the timing analysis shows that the trace is not timing-consistent, function `compute_lazy_arc` extracts a set of relative timing constraints from $CS$, i.e. a set of additional orderings between the events of $\theta''$ imposed by the delay bounds. These new constraints are added to the initial $CS$ in the form of lazy arcs by the function `add_lazy_arc`. The resulting lazy event structure $LCS$ models only those orderings of the events of $\theta''$ which are timing-consistent with the delays imposed by $A$. 
**Example 4.4** Recall the example developed in Section 4.2. Consider the TS of Figure 4.2 (a) and the delay bounds specified in Figure 4.2 (b). Recall also that, in this example, the property being verified states that event $g$ must always fire before event $d$. Thus, the following trace $\{x\xrightarrow{a} \{a, b\} \xrightarrow{a} \{b, c, g\} \xrightarrow{b} \{b, g\} \xrightarrow{d} \{g\} \xrightarrow{g} \{y\}$ illustrates a violation of the property.

The shortest possible suffix is given by the trace $\{d, g\} \xrightarrow{d} \{g\} \xrightarrow{g} \{y\}$, from which the simple event structure of Figure 4.17 (a) is derived. Clearly, the timing analysis cannot be exact since $g$ was already enabled in the pre-history of the trace. Thus, the lower delay bound of $g$ is conservatively set to 0. The timing analysis can conclude nothing about the occurrence order of events $d$ and $g$, since both can fire concurrently. The algorithm continues by moving one step backwards along the trace and repeats the same process again, building the corresponding CES incrementally. Figure 4.17 depicts the three attempts needed to find the shortest sufficient suffix of the original failure trace.

According to the causality relations extracted from the suffix of Figure 4.17 (c), timing analysis concludes that events $b$ and $g$ occur before event $c$ (and consequently before $d$). This relative timing constraints are depicted by the dotted arcs in the corresponding lazy event structure. The derived timing relations demonstrate the infeasibility of the given failure trace in the timed domain.

We have illustrated the process of deriving a sufficient LzCES that proves the timing-inconsistency of a trace, by considering its shorter suffix. This, however, does not guarantee the maximum effectiveness of the later refinement of the state space of the system, with the timing constraints in the LzCES. In some cases, using the shortest prefix of the trace results in better pruning of the set of failure traces. Similarly, the removal from the LzCES of timing-unrelated concurrent events, or the addition of causal predecessors that improve the knowledge of the enabling prehistory of the events, may affect the quality of the LzCES obtained.

A set of trade-offs must be considered, which correlate aspects such as: how many events are added to the CES such that the timing analysis can still be carried out effectively; how readable will be the resulting LzCES so that it can be useful for back-annotation; how effective in removing failure traces will be the later enabling-compatible product with the system, etc.

### 4.6.3 Incorporation of relative timing constraints

Finally, we develop the composition algorithm (the `compose` function) that implements the enabling-compatible product (see Section 4.5) between $A'$ and the LzCES LCS. The result is a new LzTS $A''$ in which all traces contradicting the timing orderings of the events in LCS have been removed from $A'$. Therefore $L(A'') \subseteq L(A')$. The resulting system $A''$ is a new LzTS where:
4.6 Verification Methodology

- The state space may be split in two parts: one following the enabling orders (enabling-compatible) of the events in LCS, and the other one where the enabling are not followed. The former corresponds to the state subspace where the constraints imposed by LCS apply (the timed subspace). In the latter, LCS does not apply (the untimed subspace).

- In the timed subspace, some events are prevented to fire when they are enabled. More precisely, the composition with LCS allows only those firing orderings which are consistent with the timing analysis.

4.6.4 Back-annotation

A nice feature of the verification approach is its back-annotation capability. Namely, the LzCESes used to represent the timing constraints applied along the series of iterative refinements of the state space of the system under verification, are reported at each iteration of the process.

Given the causality relations modeled by a LzCES and the delays of the events, each LzCES contains a set of additional ordering relations between the events in the timed domain. They provide a set of sufficient conditions for the system under verification to be correct. Moreover, the relative timing nature of such ordering relations and the fact that a CES often contains a small set of events, make the information contained in the LzCES rather easy to interpret.

As a result, the verification approach, not only verifies the correctness of a timed system with respect to a set of safety properties. In case the system does not satisfy the properties, a timed trace showing the sequence of events that lead to a failure and their firing times according to the delays of the system, is provided as counterexample to prove the system malfunction. Otherwise, if the system is correct, a set of timing constraints that prove such correctness is provided in the form of LzCESs.

All this back-annotation information may result crucial in design frameworks where synthesis and verification are invoked iteratively, for the design of systems that must meet functional and non-functional constraints.

4.6.5 Correctness

The correctness of the timed verification algorithm is guaranteed by the following facts:

- The language of the TTS being verified is a subset of the language of the initial untimed abstraction, i.e. its underlying TS. This condition is proved by Lemma 4.1.

- Conservativeness: the compose function does not remove any trace which is timing-consistent with the delays $\delta^l$ and $\delta^u$ of the verified TTS. This is guaranteed by the composition rules of the enabling-compatible product (see Section 4.5).
- Convergence: for a particular class of systems the verification requires only few refinements to converge (more details in Section 4.6.6). For the general class of systems a pre-defined upper bound on the number of refinements can be imposed. Although this could produce false negatives during verification, it is in full correspondence to the conservative nature of the suggested verification approach. However, in most practical cases, those systems where the upper bound on the number of refinements is required, are systems which untimed state space is indeed too big to be handled by conventional symbolic techniques.

4.6.6 Convergence

Each composition step of the original LzTS $A'$ with the lazy event structure $LCS$ implicitly performs an unfolding of $A'$ separating traces that are enabling-compatible with $LCS$ and those which are not.

The convergence of the refinement procedure for the class of Marked Graphs is guaranteed by the known results on termination of separation times analysis in a finite number of unfolding iterations [HB94]. Nevertheless the upper bound on the number of iterations could be quite high (depends on the ratio of critical and sub-critical cycles). This is an inherent limitation of exact separation analysis and, for practical applications, it is better to work with pre-established separation bounds and do not unfold beyond those bounds. Although it gives only conservative verification, an acceptance of pre-defined upper bounds seems to be a reasonable option because the largest class of systems for which the separation times analysis could be performed exactly are free and unique choice systems [HB94]. Beyond them the calculation of separation times is inherently conservative.

However there is an important practical class of systems for which the refinement procedure is especially simple and is exact for few unfolding iterations. The characterization of this class is done in terms of the so-called nodal states.

**Definition 4.14 (Nodal state)**

Let $A = \langle S, \Sigma, T, s_0 \rangle$ be a TS. A state $s \in S$ is called nodal if $\forall s' \in S$, $s \xrightarrow{e'} s' \in T$, $e \in \mathcal{E}(s) \Rightarrow e \notin \mathcal{E}(s')$.

Definition 4.14 points that all direct predecessors of a nodal state are synchronized in that state, i.e. at the moment when a system arrives to a nodal state all concurrent activities have been finished. Figure 4.18 illustrates the concept by showing two portions of a transition system. State $s$ in the left portion is a nodal state since all the events enabled in $s$ (c and d) are not enabled in any of the predecessor states of $s$. Conversely, in the right portion state $s$ is not nodal since event $e$ is not newly enabled in $s$. Other examples of nodal states can be found, for example, in the TS of Figure 4.2, where states...
\(s_0, s_1 \text{ and } s_{13}\) are nodal. This TS has no conflicting events (no choice) and therefore each of the nodal states is a "global synchronizer" because it breaks all the TS cycles.

Nodal states are natural points from which the timing analysis is convenient to start. Any event enabled somewhere in a path to a nodal state must fire before reaching this state and, hence, timing analysis from a nodal state does not depend on the prehistory of the process behavior. We will call a TS in which every trace passes through at least one nodal state as strongly synchronized. Note that the requirement of breaking traces by a set of nodal states is essential here because it is easy to construct an example of TS with choices, in which different branches of a choice would have different nodal states and none of them could serve as a "global synchronizer" for the whole TS.

In a strongly synchronized TS, given a failure trace \(\theta\) with an "improper" ordering of the pair of events \(a\) and \(c\), checking the timing-consistency by \(a\) and \(c\) might be reduced to consideration of the suffix \(\theta_1\) starting from the nodal state closest to the enabling of events \(a\) and \(c\).

By \(\theta_1\) one can construct the corresponding CES to check whether \(a\) and \(c\) might occur in the order they have in \(\theta\). However in case of cyclic behavior, \(\theta\) might continue in such a way that the first \(n\) occurrences of events \(a\) and \(c\) satisfy the checked properties while their \(n+1\) occurrences have an "improper" ordering. The nice feature of strongly synchronized TSs is that timing analysis made for trace \(\theta\) can be equally applied for "later" occurrences of \(a\) and \(c\) because the analysis, started at a nodal state, does not depend on the enabledness prehistory of the events. Therefore timing-inconsistency of \(\theta\) implies also timing-inconsistency for any cyclic unfolding of \(\theta\), from which it immediately follows the exactness and convergence of the suggested procedure for verification.

The practical significance of the class of strongly synchronized TS could be shown by analyzing the known set of asynchronous circuits benchmarks (see Chapter 5): more than 80% of the specifications are strongly synchronized. Beyond the class of strongly synchronized TSs our verification procedure would be conservative in general. Still in many cases it might require just few iterations in unfolding the TS to reach the exact
separation analysis. For example, [AH99] shows the fast convergence of separation times analysis for pipelined specifications, which are inherently not strongly synchronized.

Finally remark that no formal study has been carried out about the convergence of the verification method in the absence of nodal states. Nevertheless, our intuition indicates that the method should generally converge after a bounded number of iterations that guarantee a precise-enough timing analysis. Similar results have already been obtained in the context of marked graphs, where a bounded number of unfoldings suffice to compute the cycle times of a system [NK94]. A detailed formal study on the topic is left for future work.

4.7 Conclusions

This chapter has presented a novel verification methodology for safety properties in timed systems. The methodology combines relative timing with conventional methods based on symbolic reachability analysis. Two fundamental facts are at the basis of the approach: the set of traces of a transition system can be covered by a set of event structures, and the use of relative timing allows to represent the timed domain of a system in an efficient way.

Rather than calculating the exact timed state space, the verification approach performs an off-line timing analysis on a set of event structures that covers the traces leading to failure states. 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.

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

The key features of the verification approach can be summarized by:

- Relative timing allows to avoid the computation of the exact timed state space of the system. Instead, the timed behavior of events is captured by means of partial orders that represent simple facts, such as if an event happens before another.

- The timing analysis is performed locally for a set of failure traces that are covered by an event structure. Therefore, only a subset of the events is involved and the timing analysis can be carried out efficiently.
Because of the iterative nature of the 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.

The verification not only proves or disproves the correctness of the system with respect to a set of safety properties. In case the system is correct the algorithm provides the set of relative timing relations used for the proof, which can be used as valuable back-annotation information. In case the system is incorrect, a counterexample failure trace is provided.

Several issues remain open for future developments of the proposed verification approach. Among others:

- Although BDDs are a good data structure for the representation of symbolic boolean information, they often suffer from a memory blow-up during the intermediate computations, thus limiting the applicability of certain algorithms. Therefore, it would be desirable to experiment with other data structures which provide similar benefits than BDDs and allow better manipulation of bigger sets of states.

- Similarly, in order to reduce the memory requirements during the verification of big systems, partial order techniques [GW91, Pd96, VdJL96, ABH+97, BJLY98] could be combined with symbolic methods for state space representation and exploration.

- Incorporate symbolic algorithms for timing analysis (e.g. [AH99]), such that actual delay values are not required for verification. Instead, the verification can be tuned to discover the appropriate delays that make a system correct for a given property.

- CESs can model only conjunctive causality relations. However, the causality relations in a TS can be more general, involving disjunctive causality or combinations of both. As a consequence, our approach may need several refinements in order to cover the different causality relations among a set of events. Therefore, it would be desirable to allow the CESs to incorporate other types of causality relations. This would require to review the notions of enabling-compatibility, the way timing analysis is carried out in a CES, the enabling-compatible product, etc.

- Another interesting feature to enrich the verification approach would be the possibility to quantify the effectiveness of an enabling-compatible product before actually performing it. This would allow to choose the best LzCESs at each iteration, so that the biggest number of failure traces are pruned, or the least possible state splitting is produced, etc.

- The back-annotation produced by the tool consists of a set of LzCESs that contain the relative timing constraints used along the verification process. Some of those
constraints may appear several times in different iterations, thus being redundant. Therefore, it would be desirable to have a mechanism to summarize the set of timing constraints and provide them in a more readable form to the user of the tool.
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.


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 taif [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 bllf 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 	exttt{tsif} format used in 	extsc{transyt}.

### 5.1.1 Representation of LzTSSs with boolean algebras

In order to provide an efficient symbolic representation of LzTSSs, 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 	extsc{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 \( f : \mathbb{B}^n \rightarrow \mathbb{B} \) (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 \( v_i \):

\[
\begin{align*}
    f_{v_i} &= f_{v_i=1} = f(v_1, \ldots, v_{i-1}, 1, v_{i+1}, \ldots, v_n), \\
    f_{\overline{v_i}} &= f_{v_i=0} = f(v_1, \ldots, v_{i-1}, 0, v_{i+1}, \ldots, v_n).
\end{align*}
\]
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:

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

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:

\[ \exists v_i f = f_{v_i} + f_{\overline{v}} \quad \text{and} \quad \forall v_i f = f_{v_i} \cdot f_{\overline{v}} \]

In order to illustrate these concepts, let us consider the function: \( f(a, b, c) = bc + a \overline{b}c + \overline{a}c \). The positive and negative cofactors with respect to variable \( a \) are: \( f_a = bc + \overline{b}c \) and \( f_{\overline{a}} = c \). The abstractions with respect to variable \( a \) are: \( \exists_a f = f_a + f_{\overline{a}} = b + c \) 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 \( \text{LzTSes} \)**

Let \( A = \langle S, \Sigma, T, s_0, \text{EnR} \rangle \) be a \( \text{LzTS} \). 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 \( s \in S \) can be represented by means of an encoding function \( Q : S \to \mathbb{B}^n \), such that \( n \geq \lfloor \log_2 |S| \rfloor \). That is, given the set of boolean variables \( \mathcal{V} = \{v_1, \ldots, v_n\} \), each state \( s \in S \) 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 \( \chi_P^Q : \mathbb{B}^n \to \mathbb{B} \) that evaluates to 1 for those vertexes of \( \mathbb{B}^n \) that correspond to states in the set \( P \), encoded using \( Q \). Whenever the encoding is understood, we simply write \( \chi_P \). 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 \( \mathsf{LzTS} \ A = \langle S, \Sigma, T, s_0, \text{EnR} \rangle \). 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 \( \mathsf{LzTS} \) can be extended to contain \( \mathcal{V} \) and \( \mathcal{V'} \), i.e. \( A = \langle \mathcal{V}, \mathcal{V'}, \Sigma, T, s_0, \text{EnR} \rangle \). Now, given an event \( e \in \Sigma \) we can represent its enabling region, its firing region and its transitions relation, by means of the following characteristic functions:

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

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

- \( \text{TR}(e) : \mathbb{B}^{2n} \to \mathbb{B} \) such that \( \text{TR}(e) = 1 \) for all the relations \( (s_1, s_2) \) such that there is a transition of event \( e \), \( s_1 \xrightarrow{e} s_2 \in T \). 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 \( \text{EF}(e) \) and \( \text{FF}(e) \), respectively. Also, when the sets of variables in a transition relation are interchanged we will write \( \text{TR}(e)^{-1} \).

Figure 5.1 (a) shows a simple \( \mathsf{LzTS} \). 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 \( \mathsf{LzTS} \) but encoded using an arbitrary binary encoding of the states. Thus, given a set of states \( P = \{s_0, s_1, s_2\} \), its characteristic function will be: \( X_P = \overline{v_1} \overline{v_2} + \overline{v_1} v_1 \overline{v_2} + \overline{v_1} v_1 v_2 = \overline{v_1} + \overline{v_2} \). Similarly, \( \text{EF}(c) = v_2 \overline{v_1} \) and \( \text{FF}(c) = \overline{v_1} v_2 \overline{v_3} \).

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 \( \text{TR}(b) = \overline{v_1} v_2 \overline{v_3} v'_1 v'_2 v'_3 + v_1 \overline{v_2} \overline{v_3} v'_1 v'_2 \overline{v_3} \).
Finding the set of states, \( P_2 \in S \), that can be reached after the firing of a given event, \( e \in \Sigma \), from any of the states in another set, \( P_1 \in S \), is reduced to compute:

\[
X'_2 = \exists_{v_1, \ldots, v_{|P_1|}} (TR(e) \cdot X_{P_1})
\]

Thus, in the example, in order to compute the states reached after firing event \( b \) from state \( s_0 \) we just have: \( \exists_{v_1, v_2, v_3} (TR(b) \cdot X_{\{s_0\}}) = \exists_{v_1, v_2, v_3} (v_1 v_2 v_3 \overline{v}_1 \overline{v}_2 \overline{v}_3) = \overline{v}_1 \overline{v}_2 \overline{v}_3 \).

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 \textit{tsif} (see \cite{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 \textit{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 \textit{logic relations}.

**Flat systems**

The basic elements required for the description of a flat transition system (\textit{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_ofVariables> ;
```

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.
An event declaration consists of five elements: the \texttt{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 (\texttt{EQN} keyword) can be specified to declare the transition relation of the event (\texttt{TR} keyword), failure conditions (\texttt{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 \texttt{END} keyword. Namely:

\begin{verbatim}
EVENT <event_name> <label_name>
 <list_of_equations>
END
\end{verbatim}

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

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

Finally, remark that the basic TS model is augmented in \texttt{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 rendezvous).
- 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:

\[
\{ \text{DELAY: [MAX=\langle\text{max\_delay}\rangle; MIN=\langle\text{min\_delay}\rangle; TYP=\langle\text{typ\_delay}\rangle;]} \}
\]

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
5.2 An example with forward unfolding

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
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 $t$, 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 ($ip_1$, ..., $ip_n$) and

\(^1\)The read pn command of TRANSYT reads PNs and STGs specified using the astg format of PETRIFY [CKK+97].
5.2 An example with forward unfolding

output \((o_1, \ldots, o_m)\) places are shown. Hence, for transition \(t\), the following transition relation will be specified:

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

where \(NS(x)\) 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:

\[
TR(a) = p_0 \cdot \overline{p_1} \cdot \overline{p_3} \cdot \overline{NS(p_0)} \cdot NS(p_1) \cdot NS(p_3)
\]

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: \(p_0 \cdot \overline{p_1} \cdot \overline{p_3} \cdot \overline{NS(p_0)} \cdot NS(p_1) \cdot NS(p_3)\).

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 \(p_3\) 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.
Chapter 5: Experimental results

$ transyt

TRANSYT 1.6.3 (compiled mon may  6 16:17:42 CEST 2002 on linux) running at minkar
By E.Pastor (emric@ac.upc.es) and M.A.Penya (marcos@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 $s_0$ $a$ $s_2$ $b$ $s_4$ $c$ $s_6$ $d$ $e$ $s_0$ 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
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 LzT S corresponding to the GRC of the LzCES, and (d) the resulting LzT S after the enabling-compatible product of (c) with the LzT S 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 5.5 Yoneda’s example: first (left) and second (right) refinements.
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 s3 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 s2 and s3. However, states s2 and s3 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, these 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 combine 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 $s''_2$ 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 disables 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 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.

\footnote{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.
5.2 An example with forward unfolding

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

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

nts:: Starting verification iteration 5.
ts:: Searching a failure trace
nts:: 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 866 MHz 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, Di89a, 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
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 \(\varepsilon\) 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 5.8](image-url)
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 (ramrsdbuf+) 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 (ramdsbuf–) 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 PN 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:

\[ TR(\text{busack}+) = \overline{p_5} \cdot \overline{p_6} \cdot \overline{p_7} \cdot \text{busack} \cdot NS(p_5) \cdot NS(p_6) \cdot NS(p_7) \cdot NS(\text{busack}) \]
Figure 5.10  (a) STG specification of the sbuf-read-ctl controller and (b) TRANSYT input file.
5.3 Verification of Complex-Gate Decompositions in Speed-Independent Circuits

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 `buf-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_bufreadc1` and `y1_bufreadc1` 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 a 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:

\[ TR(busreq+) = (y0_{bufreadc1} \cdot busreq + ackread \cdot y1_{bufreadc1}) \cdot \overline{busreq} \cdot NS(busreq) \]

and

\[ TR(busreq-) = (y0_{bufreadc1} + \overline{busreq}) \cdot (ackread + y1_{bufreadc1}) \cdot busreq \cdot \overline{NS(busreq)} \]

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 (`buf-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
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 TRANSYF, 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
### 5.3 Verification of Complex-Gate Decompositions in Speed-Independent Circuits

```c
# H's Post Office shuf-read-ctl circuit implementation
# shuf-read-ctl_net INTERLEAVED

INPUT_VARS req ackreqd bsmack;
OUTPUT_VARS ack ramndshuf busreq yl_shufreadctl y0_shufreadctl;
INTERNAL_VARS F G H I J;

INPUT_LABELS req ackreqd bsmack;
OUTPUT_LABELS ack ramndshuf busreq yl_shufreadctl y0_shufreadctl;
INTERNAL_LABELS F G H I J;

# Circuit behavior
EVENT rise ack
    NDEF TR SH(G) x' (bsmack' y0_shufreadctl');
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

EVENT fall ack
    NDEF TR SH(G) x ack (bsmack' y0_shufreadctl');
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

EVENT rise ramndshuf
    NDEF TR SH(G'mandshuf) ramndshuf' (yl_shufreadctl' + bsmack);
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

EVENT fall ramndshuf
    NDEF TR SH(G'mandshuf) ramndshuf (yl_shufreadctl' + bsmack);
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

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

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

EVENT rise yl_shufreadctl
    NDEF TR SH(yl_shufreadctl) yl_shufreadctl' (I' + B');
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

EVENT fall yl_shufreadctl
    NDEF TR SH(yl_shufreadctl) yl_shufreadctl (I' + B');
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

EVENT rise y0_shufreadctl
    NDEF TR SH(y0_shufreadctl) y0_shufreadctl' (J' + bsmack);
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

EVENT fall y0_shufreadctl
    NDEF TR SH(y0_shufreadctl) y0_shufreadctl (J' + bsmack);
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

# Internal gates
EVENT rise F
    NDEF TR SH(F') F' (y0_shufreadctl' + bsmack');
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

EVENT fall F
    NDEF TR SH(F') F (y0_shufreadctl' + bsmack');
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

EVENT rise G
    NDEF TR SH(G) G' (ackreqd' + yl_shufreadctl');
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
END

EVENT fall G
    NDEF TR SH(G) G (ackreqd' + yl_shufreadctl');
    {DELAY: [MIN= 2.7; MAX= 3.3;]}
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 \( x \) (\( \text{IMP}.x \) in the circuit and \( \text{ENV}.x \) in the environment) to cause an input-output conformance failure in a given state can be posed as:

\[
\text{Fail}(x) = FF(\text{IMP}.x) \cdot \overline{FF(\text{ENV}.x)}
\]

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

Failure conditions for input-output conformance can be automatically computed by \textsc{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 \( y \):

\[
\text{Fail}(x) = TR(x) \cdot FF(x) \cdot \bigcup_y \left( EF(y) \cdot \overline{EF'(y)} \right)
\]

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 **uverify** 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 (**uverify** 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 other 
commands of the verification session.

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

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

ts > uverify -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_sbufreacctl' 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_sbufreacctl'. 
ts::: Condition #0 defined (on states) 
ts::: with equation: 
ts:::  y1_sbufreacctl IMP.H IMP.I (busreq' + ramdbsuf' + ack + busack' + ackread' + req') 
  + y1_sbufreacctl' (IMP.I' + IMP.H') (y0_sbufreacctl + busreq + ramdbsuf + 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
Table 5.1  Failure situations in sbuf-read-ctl along the verification.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Failure type</th>
<th>Initial</th>
<th>It.1</th>
<th>It.2</th>
<th>It.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Induces non-persistency to busreq</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>I</td>
<td>Induces non-persistency to y1_sbufreadctl</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>J</td>
<td>Induces non-persistency to y0_sbufreadctl</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>req</td>
<td>Induces non-persistency to J</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>busack</td>
<td>Induces non-persistency to I</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>y0_sbufreadctl</td>
<td>Induces non-persistency to F</td>
<td>6</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>busreq</td>
<td>Non-conformance</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>y1_sbufreadctl</td>
<td>Non-conformance</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>y0_sbufreadctl</td>
<td>Non-conformance</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>-</td>
</tr>
</tbody>
</table>

Persistence 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 than 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
5.3 Verification of Complex-Gate Decompositions in Speed-Independent Circuits

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 LzTSSs 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 L− causes the disabling of y1_sbufreadctl−. Therefore, the state in which L− 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 L 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 L. Clearly, the failure would not exist if L− occurred before H+. In this sense, the timing analysis derived from the causal relations extracted from the trace reveals that when L− is concurrently enabled with ramrdbuf+ (which triggers the input ackread+), L− 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 L induced by busack and the non-conformance of y1_sbufreadctl have been also removed (see It. 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 It. 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.
5.3 Verification of complex-gate decompositions in speed-independent circuits

<table>
<thead>
<tr>
<th>Circuit</th>
<th>$\Sigma$</th>
<th>$S$</th>
<th>$G$</th>
<th>$S_u$</th>
<th>$S_f$</th>
<th>$TC$</th>
<th>$C$</th>
<th>$CPU$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbuf-read-ctl</td>
<td>8(5)</td>
<td>19</td>
<td>10</td>
<td>74</td>
<td>16</td>
<td>3</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>rcv-setup</td>
<td>5(2)</td>
<td>14</td>
<td>6</td>
<td>78</td>
<td>34</td>
<td>2</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>alloc-outbound</td>
<td>9(5)</td>
<td>21</td>
<td>11</td>
<td>82</td>
<td>20</td>
<td>4</td>
<td>Y</td>
<td>3</td>
</tr>
<tr>
<td>ebergen</td>
<td>5(3)</td>
<td>18</td>
<td>9</td>
<td>83</td>
<td>22</td>
<td>1</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>mp-forward-pkt</td>
<td>8(5)</td>
<td>22</td>
<td>8</td>
<td>186</td>
<td>70</td>
<td>8</td>
<td>Y</td>
<td>5</td>
</tr>
<tr>
<td>dff</td>
<td>3(1)</td>
<td>14</td>
<td>6</td>
<td>255</td>
<td>164</td>
<td>6</td>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>half</td>
<td>4(2)</td>
<td>14</td>
<td>7</td>
<td>227</td>
<td>133</td>
<td>1</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>chul33</td>
<td>7(4)</td>
<td>24</td>
<td>9</td>
<td>288</td>
<td>204</td>
<td>2</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>converta</td>
<td>5(3)</td>
<td>18</td>
<td>12</td>
<td>408</td>
<td>244</td>
<td>10</td>
<td>N</td>
<td>12</td>
</tr>
<tr>
<td>nowick</td>
<td>6(3)</td>
<td>20</td>
<td>10</td>
<td>510</td>
<td>292</td>
<td>4</td>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>chul50</td>
<td>6(3)</td>
<td>26</td>
<td>8</td>
<td>520</td>
<td>339</td>
<td>3</td>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>sbuf-send-ctl</td>
<td>8(5)</td>
<td>27</td>
<td>13</td>
<td>1592</td>
<td>1081</td>
<td>18</td>
<td>N</td>
<td>54</td>
</tr>
<tr>
<td>vme</td>
<td>6(3)</td>
<td>24</td>
<td>12</td>
<td>1736</td>
<td>1460</td>
<td>21</td>
<td>Y</td>
<td>30</td>
</tr>
<tr>
<td>rpdfsf</td>
<td>5(1)</td>
<td>22</td>
<td>8</td>
<td>2612</td>
<td>1841</td>
<td>2</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>tsend-bm</td>
<td>9(4)</td>
<td>40</td>
<td>12</td>
<td>3880</td>
<td>2999</td>
<td>3</td>
<td>N</td>
<td>46</td>
</tr>
<tr>
<td>sbuf-send-pkt2</td>
<td>9(5)</td>
<td>28</td>
<td>13</td>
<td>4544</td>
<td>4044</td>
<td>19</td>
<td>Y</td>
<td>103</td>
</tr>
<tr>
<td>sbuf-ram-write</td>
<td>12(7)</td>
<td>64</td>
<td>15</td>
<td>14016</td>
<td>12362</td>
<td>34</td>
<td>N</td>
<td>415</td>
</tr>
<tr>
<td>ram-read-sbuf</td>
<td>11(6)</td>
<td>39</td>
<td>16</td>
<td>19328</td>
<td>17488</td>
<td>36</td>
<td>Y</td>
<td>550</td>
</tr>
<tr>
<td>mr1</td>
<td>9(5)</td>
<td>190</td>
<td>16</td>
<td>21076</td>
<td>11574</td>
<td>29</td>
<td>Y</td>
<td>317</td>
</tr>
<tr>
<td>mr0</td>
<td>11(6)</td>
<td>302</td>
<td>20</td>
<td>727304</td>
<td>642291</td>
<td>2</td>
<td>N</td>
<td>48</td>
</tr>
<tr>
<td>trimos-send</td>
<td>9(6)</td>
<td>336</td>
<td>24</td>
<td>2.1 $10^6$</td>
<td>1.8 $10^6$</td>
<td>1</td>
<td>N</td>
<td>127</td>
</tr>
<tr>
<td>mmu</td>
<td>8(4)</td>
<td>174</td>
<td>22</td>
<td>5.6 $10^6$</td>
<td>5.2 $10^6$</td>
<td>3</td>
<td>N</td>
<td>480</td>
</tr>
</tbody>
</table>

*Table 5.2* Experimental results for the verification of asynchronous circuits.

The 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 $\Sigma$ 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 $S_n$ 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 mri). 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 tuning 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 Ultraspare 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 and 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 $a < b$ indicates that $a$ will always fire before $b$. In terms of separation times between events, $a < b$ is given when $\text{Sep}_{\text{peak}}(a, b) < 0$, i.e. the upper bound on the difference between the firing times of $a$ and $b$ is negative.

In a LzTS, $a < b$ 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 $a < d$ causes the removal of the arc $s_1 \xrightarrow{d} s_0$. As a consequence state $s_0$ is unreachable and event $d$ becomes lazy since $\text{FR}(d) = \{s_2, s_3, s_4\}$ and $\text{EnR}(d) = \text{FR}(d) \cup \{s_1\}$ (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.
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: \( \forall e_i, e_j \in E_s, |Sep_{\text{max}}(e_i, e_j)| < \delta(a) \). 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 \( |Sep_{\text{max}}(d, a) < \delta(b)| \) (from the simultaneity assumption) and \( |Sep_{\text{max}}(a, b) < 0| \) (from the causality between \( a \) and \( b \)), implies that \( |Sep_{\text{max}}(d, b) < 0| \), i.e. \( d \) must fire before \( b \).

- Additionally, \( \text{EnR}(b) \) 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. \( \delta(a) < \delta(b) \). 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 \( \text{EnR}(c) \) 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 \( \text{EnR}(c) \) and \( \text{Fr}(c) \) 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.
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-
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. $d_1$ should ensure that $\text{LDS}+/2 > \text{D}+/2$ and $\text{LDS}− > \text{D}−/1$ hold, whereas $d_2$ should ensure that $\text{DTACK}+/1 > \text{D}+/1$ and $\text{DTACK}+/2 > \text{D}−/2$ 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].
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↓DTACK, in order to represent the two internal nodes before the delay element \( d_1 \) and after the delay element \( d_2 \), respectively. Thus, both B↓LDS and A↓DTACK 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↓DTACK 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
5.4 VERIFICATION OF RELATIVE TIMING ASSUMPTIONS

# VME bus controller: implementation with relative timing assumptions
TN vme_set INTERLEAVED

INPUT VAMS dev ls ldtack;
OUTPUT VAMS dev ldtack ds;
INTERNAL VAMS p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10;
INTERNAL VAMS p11 p12 p13 p14;

INPUT LABELS dev ldtack;
OUTPUT LABELS ldtack dev;

# End cycle
EVENT d+ d
EGT TR p13 NS(p13)’ p0' NS(p0) ds' NS(ds)'; END

EVENT ls+1 d
EGT TR p14 NS(p14)’ p0 NS(p0)’ p1’ NS(p1)’ ds’ NS(ds)’; END

EVENT ldtack+1 ldtack
EGT TR p11 NS(p1)’ p0’ NS(p0)’ ldtack’ NS(ldtack)’; END

EVENT d+1 d
EGT TR p2 NS(p2)’ p3’ NS(p3)’ d’ NS(d)’; END

# Write cycle
EVENT d+ d
EGT TR p13 NS(p13)’ p6’ NS(p6) ds’ NS(ds)’; END

EVENT d+2 d
EGT TR p6 NS(p6)’ p7’ NS(p7)’ d’ NS(d)’; END

EVENT ls+2 d
EGT TR p14 NS(p14)’ p7 NS(p7)’ p8’ NS(p8)’ ds’ NS(ds)’; END

EVENT ldtack+2 ldtack
EGT TR p0 NS(p0)’ p9’ NS(p9)’ ldtack’ NS(ldtack)’; END

EVENT d+2 d
EGT TR p0 NS(p0)’ p10’ NS(p10)’ d’ NS(d)’; END

EVENT d+2 d
EGT TR p10 NS(p10)’ p11’ NS(p11)’ ldtack’ NS(ldtack)’; END

EVENT d+ d
EGT TR p2 NS(p2)’ p3’ NS(p3)’ p11’ NS(p11)’ dsw NS(ds)’; END

# Return to zero
EVENT ls+ d
EGT TR p2 NS(p2)’ p12’ NS(p12)’ ds’ NS(ds)’; END

EVENT ldtack+ d
EGT TR p13 NS(p13)’ p3’ NS(p3)’ ldtack NS(ldtack)’; END

EVENT ldtack+ ldtack
EGT TR p14 NS(p14)’ p12’ NS(p12)’ ldtack NS(ldtack)’; END

# Initial state
EVENT STATE COMPLETIVE
p0’ p1’ p2’ p3’ p4’ p5’ p6’ p7’ p8’ p9’ p10’;
p11’ p12’ p13’ p14’ d+ dsw’ ldtack’ ldtack’ ds’ d’;

Figure 5.18 TRANSYT input files for the specification (left) and circuit (right) of the VME bus controller.
falling transition of \( A_{\text{DTACK}} \) 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 \( a < b \) for two concurrent events \( a \) and \( b \), denoted by \( a < b \), 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:

\[
Fail(a < b) = EF(a) \cdot FF(b)
\]

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 \( a = b@c \), 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:

\[
Fail(a = b@c) = \left[ EF(a) \cdot EF(b) + EF(a) \cdot SF(b) + SF(a) \cdot EF(b) \right] \cdot FF(c)
\]

where \( SF(a) \) characterizes the set of states where \( a \) has already fired and it is not enabled. More precisely, \( SF(a) = \exists_{N \in \mathcal{S}} TR^{-1}(a) \) 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 \( a > b \), 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:

\[
Fail(a > b) = EF(b) \cdot FF(a)
\]

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

In general, given an early enabling timing assumption such as \( a > b_1 \ldots > b_n \), the failure condition would be such as:

\[
Fail(a > b_1 \ldots > b_n) = \left( \bigcup_{i=1}^{n} EF(b_i) \right) \cdot FF(a)
\]

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 (\texttt{vme}) is read first followed by the circuit implementation model (\texttt{vme.net}). The corresponding input files \texttt{vme.g.ts} and \texttt{vme.blif.delays.ts} are shown in Figure 5.18.

Then, the closed system for verification is built using the \texttt{uverif} command. The resulting system is called \texttt{CM[vme][vme.net]}, as the (C)losing of the (M)irrored specification \texttt{vme} and the circuit implementation \texttt{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.
Next, the closed system is flattened to produce a new single monolithic system called \texttt{vme\_flat}. In \texttt{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 \texttt{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 \texttt{add\_fail} command. The failure conditions are derived as discussed
### Table 5.3  Failure situations in the VME bus controller along the verification.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Failure type</th>
<th>Initial</th>
<th>It.1</th>
<th>It.2</th>
<th>It.3</th>
<th>It.4</th>
<th>It.5</th>
<th>It.6</th>
<th>It.7</th>
<th>It.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSw</td>
<td>LDTACK- &lt; DSw+</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dsr</td>
<td>LDTACK- &lt; Dsr+</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LDS</td>
<td>LDS+ /2 &gt; D+ /2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DTACK</td>
<td>DTACK+ /2 &gt; D- /2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LDS</td>
<td>LDS- &gt; D- /1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DTACK</td>
<td>DTACK+ /1 &gt; D+ /1</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>B_LDS</td>
<td>Ind. non-persistence to DTACK</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A_LDTACK</td>
<td>Ind. non-persistence to DTACK</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>LDTACK</td>
<td>Ind. non-persistence to D</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>DTACK</td>
<td>Non-conformance</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>10</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>LDS</td>
<td>Non-conformance</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td>Non-conformance</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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 LzCSEs are shown in Figure 5.19 and Figure 5.20.

```plaintext
ts > tverif -HTL -WriteTrace1 -WriteTES1 -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. O+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. O+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 \( \text{LDS}+/2 > \text{D}+/2 \). 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\(_1\)LDS+, B\(_1\)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\(_1\)DTACK, 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 \( \text{DTACK++1} > \text{D++1} \) 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 \( \text{DTACK++2} > \text{D--2} \), 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+.  

---

**Note**: The above text is a continuation of the previous page, discussing verification results and the analysis of certain traces. The specific details and further context are provided in the document itself.
In the fourth and sixth iterations, the violations of the difference timing assumptions $\text{LDTACK}^- < \text{DSr}^+$ (see Figure 5.19 (a4)) and $\text{LDTACK}^- < \text{DSw}^+$ (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 LD− > 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⊥DTACK. In particular, the seventh iteration deals with the potential hazard due to the disabling of DTACK+ 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⊥DTACK 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⊥DTACK 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 866 MHz 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 TRANSY. 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
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-
5.5 Conclusions

Appropriate 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 transyst. 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.
COMPOSITIONAL VERIFICATION

Composition is notation of distortion of what composers think they have heard before. Masterpieces are marvelous misquotations.
—Ned Rorem - The Paris Diary of Ned Rorem Braziller, 1966

Summary

This chapter presents a case study for the verification of a complex timed system. Most of the contents was already published in [PCSP02].

The system consists of an \( n \)-stage pulse-driven IPCMOS pipeline and the verification is carried out for any value of \( n \) greater than 0. Each stage is described by a circuit at transistor level and delay information is provided for each transistor. The correctness of the circuit strongly depends on the timed behavior of its components and the environment in which the circuit operates. The combination of the level of detail at which the circuit is described, together with its dependency on timing parameters, makes it attractive for verification.

To verify the system, three techniques have been combined: (1) the relative timing-based verification approach described in Chapter 4, (2) assume-guarantee reasoning to verify untimed abstractions of timed components, and (3) mathematical induction to verify pipelines of any length.

In first place, the IPCMOS architecture is described at high level for better comprehension of its overall behavior. Also, the different modules that form the architecture are described at low level in order to identify critical parts of the circuitry. High-level techniques for verification are introduced which are required in order to tackle the complexity of the verification. Then, the overall strategy for the verification of IPCMOS pipelines is developed. Finally, details on the verification of the circuitry that implements a single stage are provided.
6.1 Introduction

Chip performance, power consumption, noise reduction and clock synchronization are becoming critical challenges as microprocessor performance moves into GHz regimes. Interlocked pipelined CMOS (IPCMOS) circuits [SRC+00], provide an asynchronous clocking technique that can help to tackle these challenges. Measured results on an experimental chip that implemented a 64-bit floating point multiplier using this architecture, demonstrated robust operation at 3.3GHz under typical conditions and 4.5GHz under best-case conditions in a 0.18µm 1.5V CMOS technology. The circuit showed also robust operation with large variations in power supply voltage, operating temperature, threshold voltage, and transistor channel length.

The general concepts of interlocking, pipelining and asynchronous self-timing are not new and have been proposed in a variety of forms since [Sei80] and [Sut89]. However, the techniques used in those approaches are too slow, specially for blocks which receive data from many separate sources. In contrast, the performance achieved by IPCMOS circuits is due to their pulse-based communication mechanism. The protocol decouples the communication of a block with its predecessors and its successors, thus achieving a high degree of concurrency.

Although a single IPCMOS block that operates in a pipeline can be implemented using only 32 transistors, the concurrency achieved in the overall system leads to the state explosion problem even for a few interlocked blocks.

The correct operation of the system highly depends on its timing parameters, hence the complexity of the analysis is drastically affected by the time dimension. High-level techniques for verification have been typically used to handle the complexity of a system. For example, abstractions [Me88] tackle the complexity by hiding those implementation details that are irrelevant for the verification of a given property.

In this chapter, the verification a complex timed system such as an IPCMOS pipelines is carried out by combining three techniques:

1. The relative timing-based iterative verification approach presented in Chapter 4 is used as the basic verification engine to perform all the required correctness proofs.

2. The assume-guarantee paradigm [Pnu84] is used to perform a hierarchical verification on large systems by means of abstractions.

3. Finally, mathematical induction is used to prove the correctness of infinite-state systems, such as an n-stage IPCMOS pipeline for $n > 0$.

The rest of the chapter is organized as follows. Section 6.2 presents the details of the IPCMOS architecture. Transistor-level descriptions of the circuits that implement the interlocked modules are provided. Section 6.3 introduces some background on techniques
used for compositional verification, such as abstraction, assume-guarantee and induction. The correctness of IPCMOS pipelines regardless of their length is demonstrated in Section 6.4. The proof includes the verification of the circuitry that implements a single stage. Details of this step are provided in Section 6.5.

### 6.2 The IPCMOS architecture

Figure 6.1 depicts the general block-level interlocking scheme of the IPCMOS architecture. In the figure, block D is interlocked with blocks A, B, C, E and F. In the forward direction, dedicated VALID signals emulate the worst-case performance path through each driving block (A, B and C), and thus determine when data can be latched within block D. In the reverse direction, ACK signals indicate when the data has been received by the subsequent blocks (E and F) and that new data may be processed within block D. In this interlocked approach, local clocks are generated only when there is an operation to perform.

#### 6.2.1 IPCMOS pipelines

A single IPCMOS control block is relatively small and can be used to build meshes, pipelines and other scalable architectures. A linear version of the IPCMOS architecture is depicted in Figure 6.2. The system implements a pipeline composed of IPCMOS control blocks and latches that isolate the logic between stages. When input data for a stage is signaled to be available (VALID signal), the IPCMOS control block generates a local clock signal to latch the data (CLKE signal). Data receipt is confirmed to the sender and as soon as the data is processed (ACK signal). This eliminates the need of global clock distribution.
and at the same time contributes to the power consumption reduction by clocking data only when it is available at the inputs.

The IPCMOS block communicates with other blocks via request signals (VALID), acknowledgment signals (ACK) and produces a local data clock signal (CLKE). VALID indicates data availability to the receiver(s), while ACK acknowledges to the sender(s) that data has been received. Generally IPCMOS blocks can be fed multiple ACK and VALID signals to enable safely processing data from multiple sources and feeding the result to multiple destinations. In a pipeline, only one VALID signal and only one ACK signal is sent/received by each stage.

IPCMOS circuits are pulse-driven or edge sensitive. Their operation is illustrated by means of the 2-stage pipeline in Figure 6.3 (a), where the structure of the control blocks is detailed. The figure also shows a waveform that illustrates how two data items propagate through the pipeline. Initially the pipeline is empty: all VALID and CLKE signals are high, whereas all CLR and ACK signals are low. As soon as negative pulses are received at the VALID input of a stage, a positive pulse is generated at the ACK to acknowledge the data receipt. Input data is clocked by a negative pulse on the CLKE signal, produced by the strobe module. After some delay designed to match the worst case computation time of the logic attached to the stage, a negative pulse is generated by the valid module at the VALID output line indicating the data availability to the receiver. From this point on, the block waits for positive pulses to be received from the data consumer at the ACK input. The pulses are recorded by the reset module. When the acknowledgment is received, a positive pulse at CLR is produced. This indicates the strobe and valid modules that the stage is ready to acknowledge new incoming data and to pass new processed data to the receiver, respectively. Meanwhile VALID input pulses indicating the new input data availability are also recorded. Hence, new data receipt at every stage is interlocked.
Figure 6.3  Detailed 2-stage IPCMOS pipeline (a) and waveform of its behavior (b). Communication at the extremes of the pipeline is pulse-based (thick lines) but the stages communicate through handshakes.

with the acknowledgment of the data by the following stages. For correct operation, the only restriction the IPCMOS modules pose on the environment is the pulse length of the incoming VALID and ACK signals.

Even though a pulse-driven environment is accepted by each pipeline stage, the internal communication between adjacent stages is performed in a partially handshaked protocol between the positive edges of the pulses (see Figure 6.4). These additional causality relations enable to abstract the behavior of such components when interacting among them, in such a way that internal timing information can be neglected. As we will see later on, this phenomenon considerably simplifies the verification of the pipeline.
The dashed boxes in the waveform of Figure 6.3 (b) show the signal edges affected by the handshaking mechanism. The dashed arrows show the restricting causal dependencies that must not exist in the environment (IN, OUT) but take place in the IPCMOS stages. In the diagram we also show that all stages in a sequence cannot be filled with data at the same time, but "bubbles" (empty stages) are needed to propagate data in one direction and the acknowledgment in the other. The causal dependencies demonstrating this fact are highlighted in the diagram by means of dotted arrows.

The following sections provide details on the circuit implementations of each IPCMOS block, and the behavior of the environment modules IN and OUT.

### 6.2.2 Strobe circuit

The general structure of the *strobe* circuit is depicted in Figure 6.5 (a). The *strobe switch* circuit is shown in detail in the left of the figure. Figures 6.5 (b) and (c) illustrate the behavior of the *switch* and the *strobe* circuits by means of waveforms. Notice that when building an IPCMOS pipeline, the *strobe* circuit contains only one switch block. Thus, the waveform in Figure 6.5 (c) shows a single Vint signal.

The *strobe* circuit is responsible for accumulating the negative pulses from the VALID input lines. It is also responsible for generating the clock strobe CLK that latches the input data as soon as it is available from all sources (all VALID input pulses are received). The clock strobe cannot be issued unless all the data receivers have acknowledged the previous processed data as indicated by CLKR.

Every *strobe switch* module is responsible for storing a single negative pulse from the corresponding VALID input line and for lowering the Vint signal afterwards.

The strobe circuit weak transistor pulls up the X signal as soon as all Vint signals are low (all VALID pulses are received), and the CLKR positive pulse has discharged Rint indicating the acknowledgment receipt from all the receivers. Finally, the switches are reset by the positive pulse sent through the ACK output line.

The p-type transistor that charges X is weak with respect to the n-type transistors that discharge it. Thus, in case Vint, is the last signal to arrive, charging X does not produce a short-circuit. Signal Rint is also important since it prevents X to be discharged in case the acknowledge from the previous cycle has not been received yet.
Figure 6.5  (a) The strobe circuit in detail with the strobe switch highlighted in the left. Waveforms describing the behavior of (b) the switch and (c) the strobe circuits.

No specific technology library is used, hence the delay bounds for each stack of transistors is set to be in the range of $[1,2]$ delay units. Other appropriate delay ranges are used in case of weak transistors or fan-out considerations (see Figure 6.5 (a)). Thus, the delay of charging node $X$ in the strobe circuit is set to be in the range $[5,6]$, whereas the discharge delay is set to be in the range $[2,3]$. Also, signals ACK and CLKE have delays in the range $[2,3]$ and $[3,4]$ respectively, due to fan-out considerations.

6.2.3  Reset circuit

The general structure of the reset circuit is depicted in Figure 6.6 (a). The reset switch circuit is shown in detail in the left of the figure. Figures 6.6 (b) and (c) illustrate the behavior of the switch and the reset circuits by means of waveforms. Notice that when
building an IPCMOS pipeline, the reset circuit contains only one switch block. Thus, the waveform in Figure 6.6 (c) shows a single Aint signal.

The switch circuit is aimed at detecting positive pulses of the input ACK lines coming from the successor blocks. Having received pulses from all the ACK lines it produces a positive pulse on CLKR for resetting the valid and the strobe circuits.

The detailed behavior of the reset circuit is as follows. Upon the receipt of a pulse from all ACK lines, all the Aint signals will be low, what allows for CLKR to be charged. Similarly to the strobe circuit, the p-type transistor that charges CLKR is weak with respect to the n-type transistors that discharge it. Thus, in case Aintn is the last signal to arrive, charging CLKR does not produce a short-circuit. Finally, after rising CLKR, CLKRN resets the Aint signals, which in turn discharge CLKR.

Similarly to the strobe circuit, the delay ranges of the reset circuit are set taking into account the weakness of the transistors (for signal CLKR) and fan-out considerations (for signal CLKRN). Signal Aint in the switch has a double fall delay corresponding to the two stacked n-type transistors. The delay intervals are annotated in Figure 6.6 (a).
6.2 The IPCMOS Architecture

Figure 6.7 (a) The valid circuit and (b) waveform describing its behavior.

Figure 6.8 STGs modeling the pulse-based behavior of the IN and OUT modules.

6.2.4 Valid circuit

The details of the valid circuit are depicted in Figure 6.7.

The valid circuit incorporates a delay element matching the worst-case delay of the logic associated with the stage, so that it lowers the VALID output signal only after that delay is elapsed since latching the new data by the CLKE low pulse. On the other hand, the CLKR signal is fed to the valid circuit to ensure that the VALID output signal is raised again only after the data transfer has been acknowledged by the successor stage. Until that happens the VALID output signal is kept low. On the other hand, the delay associated to the changes in signal W is in the range [1, 2].

6.2.5 The environment modules

Figure 6.8 depicts the communication protocol implemented by the IN (left) and the OUT (right) parts of the environment, in the form of Signal Transition Graphs. The underlined transitions represent the behavior of the circuit stage signals. The IN and OUT modules operate in a pulse-based manner. This fact is highlighted by the thick lines that connect the environment modules with the pipeline.
The $OUT$ module acknowledges the data available at the output of the pipeline by rising the $ACK$ line. Once this happens, both the last stage of the pipeline and the $OUT$ module reset the respective $VALID$ and $ACK$ lines independently. A restriction must be imposed to $OUT$ to avoid early resetting of $ACK$. That is, if $ACK^-$ arrives too fast after $ACK^+$, the falling edge of $ACK$ may not be properly recorded by the reset switch circuit of the last stage of the pipeline. Therefore a minimum width is required to the positive pulse of $ACK$.

The $IN$ module notifies new data availability at the input of the first stage of the pipeline by lowering the $VALID$ line. The stage acknowledges the incoming data by rising the $ACK$ line. The reset of both lines is carried out independently and no new data can be issued by $IN$ until the first stage has acknowledged the previous data portion. Also a restriction must be imposed to $IN$ to avoid early resetting of $VALID$. That is, if $VALID^+$ arrives too fast after $VALID^-$, the falling edge of $VALID$ may not be properly recorded by the strobe switch circuit of the first stage of the pipeline. Therefore a minimum width is required to the negative pulse of $VALID$.

The aforementioned conditions on the width of the pulses produced by the environment modules, will be checked later on during the verification process.

6.2.6 About complexity

The complexity of an IPCMOS control block depends on the number of data suppliers and data receivers attached to it. Looking at the different circuits that implement the control block, the number of transistors that form it can be computed as: $N_{\text{transistors}} = 21 + 7 \times N_{\text{inputs}} + 4 \times N_{\text{outputs}}$. Thus, a single stage of a linear pipeline contains 32 transistors. Notice that data keepers are not taken into account because they do not introduce additional complexity to our model.

Although the number of transistors of a single stage is small, when a stage is composed to build a pipeline that interacts with the $IN$ and $OUT$ modules, the state explosion problem appears rather soon. Thus, for example, a simple 1-stage pipeline has $212040$ untimed states, a 2-stage pipeline has about $2.5E + 9$ untimed states, a 3-stage pipeline has about $3.0E + 13$ untimed states, etc. In fact the state space of a 4-stage pipeline could not be computed in a 866MHz Pentium-III computer with 1GB of RAM running Linux, due to memory overflow. To our knowledge, no verification approach for timed systems can handle such amount of untimed states unless higher-level techniques for verification are used.

6.3 Compositional verification

In order to overcome the complexity of the verification, symbolic representations of the state space of the systems are commonly used. However, such representations do not
suffice when complex realistic systems are analyzed. Several techniques have been proposed which, combined with symbolic representations, allow the analysis of larger systems. Those of interest to our purposes are: abstraction, assume-guarantee reasoning and induction.

**Abstraction**

The abstraction mechanism \cite{Mel88, CGL92, TAKB96, DGG97} allows to reduce the size of the state space by removing details irrelevant for proving a given property. When performing abstraction, information about the exact behavior of the system is lost, therefore the truth of some properties cannot be determined by looking only at the abstracted system.

It is important that the verification methodology does not lead to false positive results. That is, if a given property holds in the abstraction a mechanism is required to show that the property actually holds in the non-abstracted system. A verification methodology with this property is said to be conservative. Notice that, in general, nothing can be concluded about what happens in the actual system if the property does not hold in the abstraction. This, will depend on the level of detail hidden by the abstraction procedure.

**Assume-guarantee**

The assume-guarantee paradigm \cite{Pnu84, CLM89, Lon93, GL94, HRS98} exploits the modular structure of systems. It reasons about the correctness of the overall system by checking only the local properties of the components. Unfortunately, a component is designed to operate only in the environment of that system, thus it is unlikely to satisfy any interesting property unless analyzed together with such environment. However, such analysis would lead again to the state explosion problem.

The assume-guarantee technique tackles this intimate relation among the components of a system. In the example of Figure 6.9 (a), since the behavior of $X$ depends on the behavior of $Y$, the correctness of $X$ can be proved only if certain assumptions are satisfied.
by $Y$. Then, one must guarantee that $Y$ actually meets such assumptions. A similar reasoning can be done in the side of $Y$. By combining appropriately the assumed and guaranteed properties, it is possible to establish the correctness of the entire system, without building the global state space. Moreover, once a guarantee is proved it can be used as an assumption for a later stage in the verification process. To prevent from erroneous conclusions circularity must be avoided in the reasoning chain. Finally, the assumptions often come in the form of abstractions such that both techniques are combined.

Figure 6.9 depicts the verification scheme for the $X \parallel Y$ system, using assume-guarantee reasoning with abstractions: in (b) the local properties of $X$ are verified assuming $Y'$ is a valid abstraction of $Y$; in (c) the local properties of $Y$ are verified assuming $X'$ is a valid abstraction of $X$; in (d) $Y'$ is checked to be a valid abstraction of $Y$ in order to guarantee (b); and in (e) $X'$ is checked to be a valid abstraction of $X$ in order to guarantee (c). Each guarantee verification checks for language containment of the implementation $X$ ($Y$) with respect to the abstraction $X'$ ($Y'$). In our framework, these proofs can be performed by checking that any output produced by the implementation can also be produced by the abstraction under the same input stimuli.

**Induction**

*Induction* is used to prove properties on systems composed of a number of similar components, organized in some inductively definable structure like a pipeline, a matrix, etc. These techniques rely on the concept of invariant [BSV94, ES96] or the so-called behavioral fixed point [VK98], to reason about the behavior of systems with any number of components.

Another possibility is to use mathematical induction over the size of the system, by successively increasing the number of components that form it. This process can be carried out manually, or can be automated using automatic theorem provers [LG95].

### 6.3.1 Framework

Several formal frameworks have been presented in literature [GL94, McM97, HRS98] that support correct assume-guarantee reasoning with abstractions. These frameworks often rely on a preorder relation $\preceq$ between processes, a composition operator $\parallel$ for processes and a logic to specify properties. The preorder relation $X \preceq X'$ denotes that the abstraction $X'$ captures more behaviors than $X$, i.e. $X$ refines or implements $X'$. The composition operator must be monotonic with respect to the preorder, i.e. $X \preceq X' \land Y \preceq Y' \Rightarrow (X \parallel Y) \preceq (X' \parallel Y')$. And the properties must be preserved through the preorder, i.e. $X \preceq X' \land X'$ satisfies property $P \Rightarrow X$ satisfies property $P$.

Since we verify safety properties, the only condition we have to enforce for an abstraction to be appropriate, is that its state space is a superset of that of the original system. That
is, each state of the original system corresponds to a state in the abstraction. Therefore, the observable properties are preserved through the abstractions and false positive results cannot be produced. This, together with the relative timing-based verification approach presented in Chapter 4, provides a sound framework for performing assume-guarantee verification with abstractions.

6.4 Verification of IPCMOS pipelines

This section shows how abstraction, assume-guarantee and induction can be combined with our strategy for verification with relative timing, in order to prove the correctness of an IPCMOS pipeline regardless of its length.

6.4.1 Verification strategy

The correct operation of an IPCMOS pipeline initially empty, is given by the following informal specification \( S \):

"Every data item entered to the pipeline is acknowledged once and only once at every stage."

We verify the correctness of the IPCMOS control circuit, i.e. the data path is assumed to be correct. Even though the previous property involves a liveness and a safeness condition, both can be modeled as safety conditions during the calculation of the state space. They can be modeled by means of a deadlock-freeness invariant in the control circuitry of the pipeline, such that the control deadlocks when either some data is not acknowledged or some data cannot move to the next stage.

Additionally, specific conditions about the correct behavior of CMOS circuits must be also ensured. These conditions are described in Section 6.5.1.

In particular, all properties required in this case study have been modeled with very simple temporal expressions that require at most the analysis of 1-step transitions. Therefore, it is not necessary a powerful engine to verify branching time or linear time logic for such task.

Due to the pulse-driven nature of the architecture, its correctness strongly depends on the delay margins associated to the components of the control stage.

The goal of the verification is to check whether an IPCMOS pipeline with any number of stages behaves correctly according to \( S \). That is to check:

\[
IN \parallel I_1 \parallel \cdots \parallel I_n \parallel OUT \leq S
\]  

for any value of \( n > 0 \), where \( I_i \) are identical instances of the circuit implementation \( I \) of a stage. The environment is formed by the data sender \( IN \) and the data receiver \( OUT \) described in Section 6.2.5. \( IN \) and \( OUT \) are indeed part of the specification in the sense that \( S \) also specifies the interface behavior of the pipeline, and \( IN \) and \( OUT \) can be obtained by simply mirroring such behavior.
The verification of (6.1) becomes exponentially more costly as \( n \) increases, specially because the communication protocol in both ends of a stage is highly decoupled. Thus, if the verification is carried out using the level of detail provided by \( I \), in practice \( n \) cannot go beyond 2 stages. That is, although the number of untimed states can be computed for pipelines with several stages (see Section 6.2.6), when the time dimension is added for verification, the complexity is drastically affected. As a consequence, in order to overcome such complexity, the verification of longer pipelines must be carried out using abstractions.

\( IN \) and \( OUT \) operate according to the pulse-based protocol and so does the left side of a stage, whereas the right side of a stage operates according to a two-phase handshake protocol (see Section 6.2). Therefore, the communications between stages \( I_2 \) and \( I_{n-1} \) inside the pipeline use the handshake scheme (thin arrows in Figure 6.10), whereas the pulse-based behavior only appears at the extremes of the pipeline (thick arrows in Figure 6.10).

We propose abstractions that hide the pulse-based behavior in a way that all the timing restrictions related to the correctness of such protocol are also encapsulated inside the abstractions (see \( A_{in} \) and \( A_{out} \) in Figure 6.10). Therefore, since \( A_{in} \) and \( A_{out} \) communicate through the handshake protocol, the abstractions can be untimed and assume-guarantee reasoning can be used. Hence, we pose the verification of (6.1) in terms of:

\[
A_{in} \parallel A_{out} \leq S
\]

(6.2)

We proceed as follows:

- Build abstractions \( A_{in} \) and \( A_{out} \) for the components shown in Figure 6.10. \( IN \) and \( OUT \) communicate by pulses respectively with \( I_1 \) and \( I_n \) inside the abstractions. On the other hand, the rest of the stages communicate by handshakes.
- Prove (6.2) assuming that \( A_{in} \) and \( A_{out} \) are correct abstractions of the respective parts of the pipeline.
- Guarantee the soundness of the abstractions. Discharge the assumptions by proving that \( A_{in} \) and \( A_{out} \) are indeed correct abstractions of \( IN \parallel I \) and \( I \parallel OUT \), respec-
Moreover, prove that $A_{in}$ is also a good abstraction of $A_{in} \parallel I$, i.e. $A_{in}$ is a behavioral fixed point that abstracts the sender $IN$ and a chain of $n$ stages.

- Finally, prove the correctness of a pipeline formed by a single fully detailed implementation ($I$) of a stage. The verification checks if $I$ is a correct CMOS circuit and satisfies $S$ in the given environment, that is $IN \parallel I \parallel OUT \leq S$.

The first three items are covered by the remaining of this section, whereas Section 6.5 shows in detail the use of the relative timing-based approach described in Chapter 4 to perform the proof in the last item.

6.4.2 Abstractions

The models $A_{in}$ and $A_{out}$ must describe the observable behavior of the abstracted parts of the pipeline at a higher level (see Figure 6.10). That is, $A_{in}$ and $A_{out}$ hide the internal communications inside the abstracted blocks. The two-phase handshake protocol in the communication interface of the abstractions is modeled by the fact that the rising edge of ACK to acknowledge a data portion is always interlocked within the falling edge and the next rising edge of VALID. The models for the environment ($IN$ and $OUT$) used for the verification were already shown in Figure 6.8. Figure 6.11 depicts the models for the abstractions $A_{in}$ and $A_{out}$, represented by Signal Transition Graphs. The underlined transitions represent inputs in their respective models.

$A_{in}$: abstraction of $IN-I_1 - \cdots - I_{n-1}$.

$A_{in}$ hides the pulse-based communication between $IN$ and the first stage of the pipeline. $A_{in}$ signals the data availability at the input of the next stage of the pipeline by lowering the output VALID line. VALID is not raised again until the pipeline acknowledges the receipt of the data by rising the ACK line. The two-phase handshake protocol is completed by resetting the VALID line independently of the resetting of the ACK line by the pipeline.

$A_{out}$: abstraction of $I-OUT$.

$A_{out}$ hides the pulse-based communication between the last stage of the pipeline and the $OUT$ module. $A_{out}$ samples the data available at the end of the pipeline signaled by the
low value of VALID, and acknowledges it by producing a positive ACK pulse. The resetting of the ACK and VALID lines to their initial state is done independently by the abstraction and the pipeline, respectively.

6.4.3 Assume-guarantee verification

The verification methodology described in Chapter 4 is used in this section to perform the experiments of the assume-guarantee strategy.

We want to prove that the abstract system built from $A_{in}$ and $A_{out}$ is a good abstraction of an IPCMOS pipeline, i.e.:

$$IN \parallel I_1 \parallel \cdots \parallel I_n \parallel OUT \leq A_{in} \parallel A_{out}$$

We use assume-guarantee reasoning in five steps in order to carry out the proof using the abstract models described above. Steps 3 and 4 are the ones that use induction to prove the correctness of an $n$-stage pipeline, for $n \geq 2$.

The second, third and forth verification steps are graphically depicted in Figure 6.12. The symbol $\leq$ models a component that checks that any event produced by the refinement is also produced by the abstraction (i.e. the language produced by the refinement is included in the language produced by the abstraction).

1. **Assume:** We must prove that the system formed by the abstractions meets the specification of the IPCMOS pipeline, that is: $A_{in} \parallel A_{out} \leq S$. Provided the models in Figure 6.11 this verification step is straightforward and completes successfully in less than a second of CPU time.

2. **Guarantee correctness of $A_{out}$:** We prove the correctness of $A_{out}$ with respect to the system formed by the implementation of a stage of the pipeline $I$ and the $OUT$ module, when $I$ communicates with the rest of the pipeline using the handshake protocol. That is: $A_{in} \parallel I \parallel OUT \leq A_{in} \parallel A_{out}$. For this, the system shown in Figure 6.12 (a) is built. The verification consists in checking the language containment of $I \parallel OUT$ with respect to $A_{out}$, which is reduced to checking that any output produced by $I \parallel OUT$ can also be produced by $A_{out}$ at the same time instant. In this case, the only relevant output is signal ACK.

3. **Guarantee correctness of $A_{in}$ with one stage:** We prove the correctness of $A_{in}$ with respect to the system formed by the pulse-based $IN$ module and the implementation of a stage of the pipeline $I$, when $I$ communicates with the next stage in the pipeline using the handshake protocol. That is: $IN \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$. For this analysis, the system shown in Figure 6.12 (b) is built. The verification consists in checking that whenever $I$ is ready to change the value of VALID, $A_{in}$ is also ready for that, thus guaranteeing language containment of $IN \parallel I$ with respect to $A_{in}$.
4. **Guarantee $A_{in}$ is a behavioral fixed point**: The previous proof only guarantees the correctness of $A_{in}$ as an abstraction of $IN$ and a single stage. However, that result serves as the induction hypothesis to prove that $A_{in}$ is a correct abstraction of $IN \parallel I \parallel \cdots \parallel I_{n-1}$, for any $n \geq 2$, as shown in Figure 6.10. Namely, $A_{in} \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$. For this, the system shown in Figure 6.12 (c) is built and the verification is done similarly to the previous proofs, but now checking signal VALID.

$A_{in}$ is an abstraction of the IN module and a chain of IPCMOS stages. Thus, $A_{in}$ is said to be a behavioral fixed point [VK98], since no matter how large $n$ is, $A_{in}$ can be used as a correct abstraction. This, together with the previous proofs, demonstrate the correctness of an $n$-stage pipeline for $n \geq 2$.

5. **Guarantee correctness of a 1-stage pipeline**: The previous proofs demonstrate the correctness of IPCMOS pipelines with 2 or more stages. It is still needed to prove the correctness of a pipeline with a single stage, that is $IN \parallel I \parallel OUT \leq S$. This step is necessary to consider the case in which a stage is interacting with a pulse-driven environment at both sides. This is the step in which more timing constraints are required to guarantee a correct behavior of the components. Despite of its complexity, since this step also requires the refinement of the model at the level of transistors, we describe it in detail in Section 6.5.
<table>
<thead>
<tr>
<th>Proof</th>
<th>System</th>
<th>CPU time</th>
<th>Refinements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $A_{in} \parallel A_{out} \leq S$</td>
<td>Ain --- Aout</td>
<td>&lt; 1 sec.</td>
<td>-</td>
</tr>
<tr>
<td>2. $A_{in} \parallel I \parallel OUT \leq A_{in} \parallel A_{out}$</td>
<td>Ain --- Aout</td>
<td>28 min.</td>
<td>7</td>
</tr>
<tr>
<td>3. $IN \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$</td>
<td>Ain --- I --- Aout</td>
<td>9 min.</td>
<td>3</td>
</tr>
<tr>
<td>4. $A_{in} \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$</td>
<td>Ain --- I --- Aout</td>
<td>10 min.</td>
<td>3</td>
</tr>
<tr>
<td>5. $IN \parallel I \parallel OUT \leq S$</td>
<td>IN --- I --- OUT</td>
<td>35 min.</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 6.1 Summary of the results for the 5 steps of the verification.

Table 6.1 summarizes the results of the five verification steps, using the \textsc{transyt} tool implementing the relative timing-based verification approach described in Chapter 4. The first column shows the formula of the proof corresponding to each step of the verification strategy. The second columns depicts graphically the system built for each proof. The CPU times, indicated in the third column, have been rounded to minutes and correspond to executions in a 866 MHz Pentium-III computer with 1GB of RAM running Linux. The number of refinements, indicated in the last column, correspond to the number of iterations needed by \textsc{transyt} to successively incorporate the timing constraints that help pruning the failure traces from the state space of the corresponding models.

In the first experiment, no refinement is required since the verification only consists in computing the untimed state space of the abstractions involved and realizing that no violation of the specification arises. Notice also, that although experiments 2, 3 and 4 require a few refinements the CPU times are comparatively high with respect to that of experiment 5. This is due to the complexity of the required models (see Figure 6.12) and the resulting BDD explosion when doing reachability analysis. Finally, the last experiment requires a lot of refinements that correspond to all the constraints related to the timing-dependent pulse-based communication at both sides of the stage.
6.5 Verification of a stage

This section addresses the verification of the circuit implementation of an IPCMOS pipeline stage \((I)\), in an environment formed by a data sender \(IN\) and a data receiver \(OUT\). We will show the modeling mechanisms to describe the transistor-level circuits.

6.5.1 Modeling CMOS circuits

The behavior of the circuit is modeled by a TTS with a set of events that update the value of variables and therefore modify the state of the system. In particular we use a boolean variable to model each circuit node and several events that model the rising and falling transitions of the node value. For every event a transition relation is defined, including an enabling condition and a delay interval \([\delta, \delta^\prime]\) specifying the delay bounds of the signal switch once it becomes enabled.

A node in the circuit may be driven by stacks of pull-up and pull-down transistors, and possibly pass-transistors. Each stack is modeled by an event that sets the proper value to the variable (one for pull-up, zero for pull-down and copies the value of another variable for a pass-transistor). Delay intervals can be computed using the technology parameters (if they are available) and the fan-out conditions for each signal. The broader the delay intervals for which the circuit is proved to be correct, the more general is the verification, i.e. the more robust is the circuit.

As an example, consider the transition relations for signal \(Y\) in the strobe switch circuit (see Figure 6.5). \(Y^+\), takes place if \(Y\) is low and it is pulled up by the \(p\)-type transistor controlled by \(Z\). Thus, the enabling condition is given by \(En(Y^+) = \neg Y \land \neg Z\). Similarly, the enabling condition for \(Y^-\) is given by \(En(Y^-) = Y \land \text{ACK}\) because \(Y\) can only be pulled down by the \(n\)-type transistor controlled by the ACK signal.

We have carried out the verification process with no specific technology library in mind. Hence the delay bounds for each stack of transistors is assumed to be in the range of \([1,2]\) abstract delay units. Other appropriate delay ranges are used in case of weak transistors or fan-out considerations. We want to remark that since our verification approach uses relative timing, the particular values of the delay bounds are only relevant to compute the relative differences between the accumulated delays of the critical paths related to a failure situations.

Tables 6.2, 6.3, 6.4, 6.5 and 6.6 summarize the models of each of the sub-circuits that compose a general IPCMOS control block. Thus, for example, the models of the strobe and the reset circuits consider multiple \(Vin\) and \(Ain\) input signals, respectively. Notice also that more than one event can be specified for the same signal switch if it is produced from different sources (e.g. the \(Vin^+\) event in Table 6.3). This allows potentially for a very detailed model in which an event can have different delays depending on what caused it. This is in direct correspondence with what happens in an actual circuit.
### Table 6.2 Model of the strobe circuit.

<table>
<thead>
<tr>
<th>Enabling condition</th>
<th>Event</th>
<th>Delay</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg X \land \neg Rint \land \bigwedge_i \neg Vint_i )</td>
<td>X+</td>
<td>[5,6]</td>
<td>weak ( p )-type transistor</td>
</tr>
<tr>
<td>( X \land (Rint \lor \bigvee_i Vint_i) )</td>
<td>X−</td>
<td>[2,3]</td>
<td>( n )-type transistor</td>
</tr>
<tr>
<td>( \neg K \land \neg X )</td>
<td>K+</td>
<td>[1,2]</td>
<td>inverter</td>
</tr>
<tr>
<td>( K \land X )</td>
<td>K−</td>
<td>[1,2]</td>
<td>inverter</td>
</tr>
<tr>
<td>( \neg ACK \land \neg K )</td>
<td>ACK+</td>
<td>[2,3]</td>
<td>inverter</td>
</tr>
<tr>
<td>( ACK \land K )</td>
<td>ACK−</td>
<td>[2,3]</td>
<td>inverter</td>
</tr>
<tr>
<td>( \neg CLKE \land \neg ACK )</td>
<td>CLKE+</td>
<td>[3,4]</td>
<td>inverter</td>
</tr>
<tr>
<td>( CLKE \land ACK )</td>
<td>CLKE−</td>
<td>[3,4]</td>
<td>inverter</td>
</tr>
<tr>
<td>( \neg Rint \land \neg CLKE )</td>
<td>Rint+</td>
<td>[1,2]</td>
<td>( p )-type transistor</td>
</tr>
<tr>
<td>( Rint \land CLKR )</td>
<td>Rint−</td>
<td>[1,2]</td>
<td>( n )-type transistor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Failure condition</th>
<th>Comment</th>
<th>Initial state</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg CLKE \land CLKR )</td>
<td>Short-circuit at Rint</td>
<td>( \neg X \land K \land \neg ACK \land CLKE \land \neg Rint )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enabling condition</th>
<th>Event</th>
<th>Delay</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg Rint \land \neg VALID )</td>
<td>Z+</td>
<td>[1,2]</td>
<td>inverter</td>
</tr>
<tr>
<td>( Z \land VALID )</td>
<td>Z−</td>
<td>[1,2]</td>
<td>inverter</td>
</tr>
<tr>
<td>( \neg Y \land \neg Z )</td>
<td>Y+</td>
<td>[1,2]</td>
<td>( p )-type transistor</td>
</tr>
<tr>
<td>( Y \land ACK )</td>
<td>Y−</td>
<td>[1,2]</td>
<td>( n )-type transistor</td>
</tr>
<tr>
<td>( \neg Vint \land Y \land VALID )</td>
<td>Vint+</td>
<td>[1,2]</td>
<td>( p )-type transistor</td>
</tr>
<tr>
<td>( \neg Vint \land \neg CLKE )</td>
<td>Vint+</td>
<td>[1,2]</td>
<td>( p )-type transistor</td>
</tr>
<tr>
<td>( Vint \land Y \land \neg VALID )</td>
<td>Vint−</td>
<td>[1,2]</td>
<td>( n )-type transistor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Failure condition</th>
<th>Comment</th>
<th>Initial state</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg Z \land ACK )</td>
<td>Short-circuit at Y</td>
<td></td>
</tr>
<tr>
<td>( \neg VALID \land Y \land \neg CLKE )</td>
<td>Short-circuit at Vint</td>
<td>( \neg Z \land Y \land Vint )</td>
</tr>
</tbody>
</table>

### Table 6.3 Model of the strobe switch circuit.
<table>
<thead>
<tr>
<th>Enabling condition</th>
<th>Event</th>
<th>Delay</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬CLKR ∧ ( \bigwedge_i \neg Aint_i )</td>
<td>CLKR+</td>
<td>[5, 6]</td>
<td>weak p-type transistor</td>
</tr>
<tr>
<td>CLKR ∧ (( \bigvee_i Aint_i ))</td>
<td>CLKR−</td>
<td>[2, 3]</td>
<td>n-type transistor</td>
</tr>
<tr>
<td>¬CLKRN ∧ ¬CLKR</td>
<td>CLKRN+</td>
<td>[3, 6]</td>
<td>3 inverters</td>
</tr>
<tr>
<td>CLKRN ∧ CLKR</td>
<td>CLKRN−</td>
<td>[3, 6]</td>
<td>3 inverters</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial state</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬CLKR ∧ CLKRN</td>
</tr>
</tbody>
</table>

**Table 6.4** Model of the reset circuit.

<table>
<thead>
<tr>
<th>Enabling condition</th>
<th>Event</th>
<th>Delay</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬Aint ∧ ¬CLKRN</td>
<td>Aint+</td>
<td>[1, 2]</td>
<td>p-type transistor</td>
</tr>
<tr>
<td>Aint ∧ CLKRN ∧ ACK</td>
<td>Aint−</td>
<td>[2, 4]</td>
<td>2 n-type transistors</td>
</tr>
<tr>
<td>¬failctl ∧ CLKRN ∧ ACK ∧ ¬Aint</td>
<td>failctl+</td>
<td>[0, 0]</td>
<td></td>
</tr>
<tr>
<td>failctl ∧ ¬ACK</td>
<td>failctl−</td>
<td>[0, 0]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Failure condition</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>failctl ∧ ¬CLKRN ∧ NS(CLKRN)</td>
<td>Slow resetting of ACK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aint ∧ failctl</td>
</tr>
</tbody>
</table>

**Table 6.5** Model of the reset switch circuit.

<table>
<thead>
<tr>
<th>Enabling condition</th>
<th>Event</th>
<th>Delay</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬W ∧ ¬CLKE</td>
<td>W+</td>
<td>[1, 2]</td>
<td>p-type transistor</td>
</tr>
<tr>
<td>W ∧ CLKR</td>
<td>W−</td>
<td>[1, 2]</td>
<td>n-type transistor</td>
</tr>
<tr>
<td>¬VALID ∧ ¬W</td>
<td>VALID+</td>
<td>[Delay]</td>
<td>delay(logic) - delay(strobe)</td>
</tr>
<tr>
<td>VALID ∧ W</td>
<td>VALID−</td>
<td>[Delay]</td>
<td>delay(logic) - delay(strobe)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Failure condition</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬CLKE ∧ CLKR</td>
<td>Short-circuit at W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial state</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬W ∧ VALID</td>
</tr>
</tbody>
</table>

**Table 6.6** Model of the valid circuit.
Provided this modeling mechanism, correctness of CMOS circuits can be posed in terms of the following properties:

**Persistency.** The temporal behavior of a gate is described by the *inertial delay* model. In this model, input pulses shorter than the lower delay bound $\delta^l$ are not propagated to the output. Pulses longer than the upper delay bound $\delta^u$ are always propagated. However, propagation of pulses with duration between $\delta^l$ and $\delta^u$ is uncertain and may produce *glitches*, hence signal persistency conditions are imposed. Persistency implies that every transition must fire once it is enabled and cannot be disabled by the firing of another transition.

Consider for example a given event $e$ for which a persistency condition must be ensured. The following invariant condition must be satisfied:

$$EF(e) \cdot EF'(e) \cdot TR \setminus TR(e)$$

That is, it must never happen that if event $e$ is enabled in some state, the firing of another event of the system leads to a state where $e$ is no longer enabled. In the expression, $TR$ corresponds to the transition relation of the system, $TR(e)$ corresponds to the transition relation of event $e$, and $EF$ and $EF'$ are enabling functions expressed using current and next-state variables, respectively.

Notice that this condition for persistency is different to that presented in Section 5.3.6. There, the condition expressed the fact that the firing of an event $x$ could induce non-persistency to any other event $y$ in the system, *i.e.* $x$ disables $y$. Conversely, the invariant presented here states the dual condition.

**Short-circuits.** Custom designs exploit the flexibility of CMOS technology, relaxing the complementarity between the pull-up and pull-down stacks. This introduces a potential source of short-circuits during circuit operation. Although short-circuits can be exploited by considering the pull-up/pull-down relative impedance, generally they are undesirable because they may leave the driven signals undefined, increase the power dissipation, or even cause a circuit damage. Therefore, the complementarity of the pull-up and pull-down conditions for each circuit node must be ensured.

Several potential short-circuits can happen in the *strobe* (signal $\text{Rint}$), the *strobe switch* (signals $Y$ and $\text{Vint}$) and the *valid* (signal $\text{W}$) circuits. See Figures 6.5 and 6.7. Consider for example the potential short-circuits in the *strobe* circuit of Figure 6.5. They are identified by the following invariants:

1. $\neg Z \land \text{ACK}$: The $Y$ node is pulled down by the $n$-type transistor controlled by $\text{ACK}$ and pulled up by the $p$-type transistor controlled by $Z$. The short-circuit occurs if both transistors conduct.
2. $\neg\text{VALID} \land Y \land \neg CLKE$ : The VALID line is pulled down by the input and pulled up by the $p$-type transistor controlled by CLKE. The short-circuit occurs if the $n$-type transistor controlled by $Y$ conducts.

Thus, the invariant conditions that must be satisfied during the verification are the negated of the short-circuit conditions. That is, (1) $Z \lor \neg\text{ACK}$ and (2) $\text{VALID} \lor \neg Y \lor \text{CLKE}$.

Failure conditions due to short-circuits in the different modules of an IPCMOS stage are summarized in Tables 6.2, 6.3 and 6.6.

### 6.5.2 Modeling IPCMOS circuits

Despite of the above conditions regarding the correctness of CMOS circuits in general, each circuit of the IPCMOS control block may exhibit its particular set of failure situations.

For example, in the reset switch circuit, a failure condition has been defined that models the situation in which signal $\text{ACK}$ is slow to fall once the falling of $\text{Aint}$ is produced. If $\text{ACK}$ does not reset before $\text{CLKRN}$ rises again, the circuit might produce a second falling edge in $\text{Aint}$, in response to a single rising edge in $\text{ACK}$. This situation cannot be captured with a boolean expression in terms of the current and next state signals of the circuit, since it involves a sequence of firings. Therefore, we have introduced an extra internal boolean variable $\text{failctl}$ in the model. This variable is set to $\text{one}$ when the falling edge in $\text{Aint}$ is produced, and $\text{falls}$ when $\text{ACK}$ falls. In such a way the failure condition can be stated as that situation in which $\text{failctl}$ is high ($\text{Aint}$ fell) and $\text{CLKRN}$ tries to rise. See Table 6.5.

Of special importance in the modeling are the bounds associated to the delay element in the $\text{valid}$ circuit. Recall that such delay element mimics the operation time of the logic attached to the stage. The delay range should be general enough to allow correct operation with whatever logic attached to the stage, however the IPCMOS implementation imposes certain limitations on such possible delay.

Suppose a new data portion is already available at the stage inputs and so is notified by the low value of the incoming $\text{VALID}$ line. On the other hand, the previous data portion is yet processed by the stage but the receiver has still not acknowledged it (outgoing $\text{VALID}$ and incoming $\text{ACK}$ are low). As soon as the acknowledgment arrives, a race is produced between to paths inside the stage. Namely, the sequence $\text{CLKR}+ \rightarrow W- \rightarrow \text{VALID}^+$ that resets the outgoing $\text{VALID}$ signal to its initial high value, and the sequence $\text{CLKR}+ \rightarrow \text{Int}- \rightarrow W+ \rightarrow K- \rightarrow \text{ACK}+ \rightarrow \text{CLKE}- \rightarrow W+ \rightarrow \text{VALID}-$ that notifies the availability of the new data portion to the receiver by lowering the outgoing $\text{VALID}$ signal. In this situation, if the delay element behaves according to the inertial delay model, and the separation time of both sequences to complete is too small for the positive pulse of $\text{VALID}$ to propagate, an undesirable data loss will occur. That is, $\text{VALID}$ might be kept low such that no notification falling edge will be seen by the receiver. This situation would
be detected by the verification during the persistency check on the VALID signal. To avoid this faulty behavior the upper bound of the delay element in the valid circuit must be set in accordance to the difference between the accumulated upper delay bounds of the reset sequence, and the accumulated lower delay bounds of the notification sequence.

### 6.5.3 Verification results

Provided the models above, the verification succeeds proving that the invariants characterizing the circuit correctness conditions always hold. That is, the circuit implementing the IPDPCMOS stage operates correctly in the given IN-OUT environment and shows no short-circuits, no persistency violations and no deadlocks. The verification process finishes in about 35 minutes of CPU time in a 866MHz Pentium-III computer with 1GB of RAM running Linux.

The verification succeeds and also provides back-annotation indicating a set of sufficient timing relations between events that guarantee the correctness of the implementation. These relations are provided as the timed event structures obtained at every iteration of the verification process. This information permits to guess about the delay margins allowable to keep the correctness.

Figure 6.13 shows some of the timing relations obtained during the verification, that guarantee the correctness of the *strobe switch* module (see left side of Figure 6.5). Recall that signals ACK and CLKE are inputs coming from the *strobe* circuit, whereas VALID is an input signal coming from the environment.

For simplicity the chain of events $\text{Vint} - \rightarrow \text{X+} \rightarrow \text{K} - \rightarrow \text{ACK+}$ produced by the *strobe* circuit has been collapsed in figures (a), (b) and (d). Thus yielding an accumulated delay for ACK+ in the range $[8,11]$. Similarly, the chain of events $\text{Vint+} \rightarrow \text{X-} \rightarrow \text{K+} \rightarrow \text{ACK-}$ has been collapsed in figure (e), which yields an accumulated delay for ACK- in the range $[5,10]$. 

![Figure 6.13](image)

*LzCES used to prove correctness of the *strobe switch* circuit: (a) Failure trace and (b) corresponding LzCES. (c)-(e) LzCESs showing other relative timing constraints (dotted arcs) for correctness.*
6.5 VERIFICATION OF A STAGE

Figure 6.13 (a) shows a trace leading to a failure situation in which the early firing of ACK+ causes a short-circuit at node Y. Event structure (b) shows the actual ordering of Z+ and ACK+ in the timed domain proving that trace (a) is not timing consistent. This situation corresponds to the case where a falling edge of VALID occurs, followed by the fall of signal Vint and the rise of ACK to indicate the data receipt. Z+ must be faster than ACK+ to avoid the short-circuit at Y corresponding to invariant (1) of Section 6.5.1.

Event structure in Figure 6.13 (c) corresponds to a situation where after rising signal ACK, the transition Y− turns off the n-type transistor that isolates Vint from VALID. Thus Y falls before the VALID line is pulled up by CLKE−. This ordering is required to guarantee invariant (2) of Section 6.5.1.

Event structure in Figure 6.13 (d) shows a situation where event ACK− is ordered with Z− ensuring invariant (1). Indeed it shows that signal ACK always falls before Z thus avoiding the short-circuit. The delay of VALID+ is set so that the appropriate ordering of the events is guaranteed.

Event structure in Figure 6.13 (e) shows the ordering relation between CLKE+ and VALID−. The delay of VALID− is set to reset CLKE before the falling edge of VALID. This ordering contributes to guaranteeing invariant (2).

Similar event structures containing the timing constraints required to prove the correctness of the different parts of the circuit are generated during the verification. Instead of showing all of them, what would require a lot of space, we summarize some of the relevant conditions required for the circuit correct operation as follows.

Correctness of the strobe switch circuit

- Short-circuit at Y by early ACK+ in response to VALID− from the previous stage. Z+ must happen before ACK+, such that the p-type transistor driving Y opens, and prevents the short-circuit when ACK+ arrives. This requirement is met since it involves the delay of the inverter driving Z against a long chain of gates in both the strobe switch and the strobe circuits.

- Short-circuit at VALID once ACK+ is produced. In this case Y must discharge faster that CLKE coming from the strobe circuit. This requirement is met since it involves the delay of the transistor responsible of discharging Y, against the delay of the inverter driving CLKE, which is also a slow one due to fan-out considerations.

- Short-circuit at Y by slow ACK−. It must be guaranteed that ACK− is faster than Z−, i.e. the strobe circuit resets itself before the VALID line coming from the previous stage resets and charges Y again. This requirement is met by properly setting the lower delay bound of VALID+ in the valid circuit and the INV module.
- Short-circuit at VALID if a new VALID− from the previous stage is produced without allowing the proper resetting in CLKE of the previous cycle of operation. This requirement is met by properly the lower delay bound of VALID− in the valid circuit and the IN module.

Correctness of the strobe circuit
- Short-circuit at Rint due to an early CLKR+ in response to the rise of ACK. This is a problem if the self-resetting fall of ACK and the corresponding rise in CLKE are slower than CLKR. This requirement is met since CLKR+ is at the end of a long chain of events involving the valid circuit of the current stage, and also the strobe circuit of the next stage.
- Short-circuit at Rint due to a late CLKR− of the previous cycle of operation, against a new falling edge in CLKE. Solving this problem guarantees the complete resetting of the strobe circuit. This requirement is met since the new CLKE− comes from a long chain of events involving a new data VALID coming from the previous stage.

Correctness of the reset circuit
- Slow self-resetting of ACK in the strobe circuit against a complete cycle of operation in the reset circuit, right after ACK+ is produced. This requirement is met by balancing properly the delays in both paths, since the path leading to ACK− is just the self-resetting part of the cycle of operation of the strobe circuit, whereas the path in the reset circuit corresponds to a complete set and reset cycle of the involved signals.

Correctness of the valid circuit
- Persistency violation of VALID+ caused by an early firing of W+ after CLKR+ has fired. To meet this requirement the upper delay bound of VALID+ must be set below the minimum accumulated delay of the chain of events leading from CLKR+ to W+, minus the maximum delay of W−, who triggers VALID+ after CLKR+.

Despite of these list of conditions for circuit correctness, recall that the pulse-based environment modules must also satisfy certain conditions. In particular, a restriction is imposed to OUT to avoid early resetting of the ACK line. That is, if ACK− arrives too fast after ACK+, the falling edge of ACK may not be properly recorded by the reset switch circuit of the last stage of the pipeline. Therefore a minimum width is required to the positive pulse of ACK. Similarly, a restriction is imposed to IN to avoid early resetting of the VALID line. That is, if VALID+ arrives too fast after VALID−, the falling edge of VALID may not be properly recorded by the strobe switch circuit of the first stage of the pipeline. Therefore a minimum width is required to the negative pulse of VALID.
Finally, we want to remark the difficulty of our verification approach to check some of the invariants required for correctness. Namely, the condition corresponding to the short-circuit at W inside the valid circuit (see Table 6.6), needed more than ten refinements, each covering different excitation sequences. Similarly happens with the failure condition about the short-circuit at node Y inside the strobe switch circuit. Further analysis is required to clarify the reasons for such computational effort. We believe this effort can be significantly reduced by constructing better event structures which abstract away the complexity due to concurrency-causality combinations.

6.6 Conclusions

The verification of a complex timed system, the IPCMOS architecture, has been tackled in this chapter. The correctness of the system highly depends on the delays of the internal gates of the circuit, and also of the environment.

The verification has been carried out by combining the core verification algorithm presented in Chapter 4, together with the use of assume-guarantee reasoning to perform a hierarchical verification by means of abstractions, 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.

The use of the relative timing-based verification approach has been crucial to prove the correctness of such a complex system. Although some other parametrized systems have been verified in the past (see e.g. [LG95]), this is the first case in which delay information and refinements down to transistor level have been provided.

The abstractions of different components of the system have still been derived manually. Also, the chain of assume-guarantee proofs and the required systems for verification have been built manually. Automatic extraction of timed abstractions and automatic derivation of the subsequent chain of reasoning are important topics for future research in this area.
CONCLUSIONS

That was Ender’s gift to us, to free us from the illusion that any one explanation will ever contain the final answer for all time, for all heavers. There is always, always more to learn.

—Orson Scott Card - Children of the Mind, 1996

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

—Winston Churchill - The End of the Beginning, 1943

Summary

This chapter presents the conclusions of the thesis. In first place, an overview of the generalities of the work is provided by reviewing the material included in each chapter of this document. Next, the contributions of the work to different aspects of the formal verification of complex timed systems are analyzed. The last section describes a number of open issues for future developments and research.
Chapter 7: Conclusions

7.1 Introduction

This thesis presents a new approach for the formal verification of safety properties in timed systems. The correct operation of such systems not only depends on functional properties but also on assumptions about the delays of the components of the system and the response times of the environment in which the system operates. The approach presented is theoretically sound and has been automated into a software CAD/CAV tool.

Formal verification uses deductive methods in order to prove if a system satisfies a given set of well-defined properties. Formal techniques have received increasing attention in recent years, mainly due to two factors: the high complexity of nowadays systems, and the high cost of correcting errors at the end of the design cycle or even once a system is yet in the market. These aspects were reviewed in Chapter 1 as an important motivation for this work.

In order to formally verify a system, a precise and unambiguous model of the system is required. Such model must capture the subset of relevant aspects of the system which are of interest for the verification. Several models have been proposed for timed systems, yielding also to different verification approaches. In this work, transition systems are the model of choice (see Chapter 2). On the other hand, Chapter 3 reviews some of the most relevant approaches in the area of verification of timed systems.

The verification of timed systems poses serious complexity problems. Although efficient techniques have been devised to overcome the complexity issue, well-known symbolic methods cannot be easily applied. Since most methods for verification rely on the computation of the complete state space, the combinatorial state explosion problem becomes exacerbated by the time dimension. As a consequence, the practical applicability of the resulting verification methods is often restricted to small systems, or to systems with particular characteristics that fit well with a given verification method.

The theory that supports the verification approach proposed in this thesis was introduced in Chapter 4. The approach extends the conventional symbolic model checking methods to the verification of timed systems. Timed transition systems are used as the underlying formalism for timed systems under the continuous-time paradigm. Instead of computing the exact timed state space, the relative timing paradigm is used to abstract exact time information from the representation. Hence, lazy transition systems are used, which represent the ordering relations between events in the timed domain by explicitly distinguishing between their enabling and their actual firing conditions. This simple yet powerful model allows the representation of the timed domain of a system using efficient symbolic methods.

The verification approach has been fully implemented in an experimental tool called TRANSYT. The tool can handle hierarchical and distributed modular systems which can inter-operate by a variety of communication mechanisms. TRANSYT proved its function-
ality as well as the validity of the overall verification approach in Chapters 5 and 6. In Chapter 5, a number of timed asynchronous circuits with up to several millions of untimed states were verified with reasonable CPU and memory resources. The experiments covered the verification of complex gate decompositions in quasi-speed-independent asynchronous circuits, and the verification of circuits optimized for area and/or speed using relative timing assumptions. Additionally, in Chapter 6, compositional verification methods were combined with the basic verification approach in order to tackle the size/complexity issues involved in the verification of complex timed systems. Thus, abstractions, assumption reasoning and mathematical induction were used to prove the correctness the IPCMOS architecture.

7.2 Contributions

This thesis proposes a novel approach for the formal verification of timed systems. The thesis contributes to this field of research both by providing an original theoretical framework as well as a software tool that implements it. The verification approach is based on two fundamental facts that we want to remark:

- The observation that the set of traces of a transition system can be covered by a set of partial orders. This fact allows to reduce 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 fact that relative timing allows to represent the timed domain of a system in an efficient way using symbolic methods. When considering precise delay bounds in timed systems, the complexity blow-up often causes verification to become an intractable problem, even for small systems. Instead, relative timing considers the effect of delays in a system in terms of relative ordering of events.

The verification approach can be briefly summarized as follows. Rather than computing the exact timed state space of the system, the approach starts with a simple approximation in which time is not taken into account. If the property under verification is satisfied in the untimed approximation, it will be also satisfied in the timed domain of the system, and the verification concludes. Otherwise, a counterexample trace is built which reproduces a violation of the property. Timing information is then used to try to refute the counterexample. Thus, an efficient off-line timing analysis is performed on an event structure that covers the counterexample trace. If the counterexample persists, the verification concludes. On the contrary, the system is refined with the relative timing information derived from the timing analysis. This process repeats until an actual counterexample is found or the property is proved correct. Therefore, the proposed approach relies on a
series of incremental refinements of the state space of the system, so that the complexity due to the timing information is incorporated only when it is needed.

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. On the contrary, the approach proposed in this thesis is applicable to systems modeled by timed transition systems without restrictions. For example, no requirement is imposed about the causality relations between events or about the types of choice allowed.

We want also to remark that the use of the proposed approach for the verification of untimed systems does not involve any additional overhead with respect to the conventional symbolic methods for such type of systems.

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

**Relative timing.** 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.

**Symbolic representation.** 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.

**Local timing analysis.** 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 partial order. Therefore, only a subset of the events of the system is involved and the timing analysis can be carried out very efficiently.

**Incremental timing information.** 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.

**Iterative refinement.** 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. As a consequence, the approach can be potentially applied to bigger systems or to systems
with more level of detail, than those that can be handled by similar methods for the verification of timed systems.

**Back-annotation.** A key feature of the proposed verification approach is that it not only proves or disproves the correctness of a timed system with respect to a set of properties. If the system is correct, the set of relative timing relations used for the proof are provided. Such 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 it can be used as valuable back-annotation information along the design process. Hence, bridging the gap between design and verification. This feature constitutes another differential aspect of our verification approach when compared to other equivalent verification methods.

**Automated.** The verification approach has been fully implemented into the experimental 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 millions of untimed states. TRANSYT is available for public download from http://research.ac.upc.es/VLSI/transyt/transyt.html.

**Compositional methods.** Compositional verification provides promising techniques to tackle the complexity in the verification of large and complex systems. In this thesis, compositional methods have been combined with the relative timing-based 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 (IPCMOS). The use of the relative timing-based verification approach has been crucial to prove the correctness of such a complex system. Although some other parametrized systems have been verified in the past, this is the first case in which delay information and refinements down to transistor level of an actual industrial system have been provided.

Finally say that the size/complexity of the systems that can be formally verified is still far from the industry-desired goals. Nevertheless, this thesis has shown that with the proposed methods, relevant systems can be successfully verified.

### 7.3 Future research

Several issues remain open for future developments of the proposed verification approach. Some of them are related to improvements of the current implementation as well as possible developments to enrich the features of the approach. Whereas other relate to new theoretical challenges.

**BDD blow-up.** Although BDDs are a good data structure for the representation of symbolic boolean information, they often suffer from a memory blow-up during the inter-
mediate computations, thus limiting the applicability of certain algorithms. Therefore, it would be desirable to experiment with other data structures which provide similar benefits than BDDs and allow better manipulation of bigger sets of states.

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. 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 BDD representation of 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.

**Partial orders.** In a similar vein than the previous topic, in order to reduce the memory requirements during the verification of big systems, partial order techniques could be combined with symbolic methods for state space representation and exploration. Partial orders have proved their efficiency for that purpose in several contexts [GW91, Pe96, VdJL96, ABH+97, B-JLY98].

**Symbolic timing analysis.** It would be interesting to incorporate symbolic algorithms for timing analysis (e.g. [AH99]), such that actual delay values are not required for verification. Instead, the verification can be tuned to discover the appropriate delays that make a system correct for a given property. This would open the proposed verification approach to new fields of application close to the design tasks.

**Disjunctive causality relations.** Currently, CESs can model only conjunctive causality relations. However, the causality relations in a TS can be more general, involving disjunctive causality or combinations of both. As a consequence, our approach may need several refinements in order to cover with various CESs the different causality relations among a set of events. Therefore, it would be desirable to allow the CESs to incorporate other types of causality relations, so that less iterations of the main verification algorithm would be required. The inclusion of disjunctive causality relations in CESs would require to review the notions of enabling-compatibility, the way timing analysis is carried out in a CES, the enabling-compatible product, etc. Moreover, the resulting CESs might result complicated for back-annotation purposes, and trade-offs might be required about the amount of information allowed in a single CES.

**Enabling-compatible product.** Another interesting feature to enrich the verification approach would be the possibility to quantify the effectiveness of an enabling-compatible
product before actually performing it. This would allow to choose the best LzCESs at each iteration, so that the biggest number of failure traces are pruned, the least possible state splitting is produced, etc.

**Back-annotation.** The back-annotation information currently produced by the tool consists of a set of LzCESs that contain the relative timing constraints used along the verification process. Some of those constraints may appear several times in different iterations, thus being redundant. Therefore, it would be desirable to have a mechanism to summarize the set of timing constraints and provide them in a more readable form to the user of the tool.

**Convergence.** No formal study has been carried out about the convergence of the proposed verification approach in the absence of nodal states. Our intuition indicates that the method should generally converge after a bounded number of iterations that guarantee a precise-enough timing analysis. Similar results have been already obtained in the context of marked graphs, where a bounded number of unfoldings suffice to compute the cycle times of a system (see [NK94]). Such detailed study on the topic is left for future work.

**Hierarchical verification.** All the presented experiments have been performed without any specific of optimization for the different types of 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 [RCP95].

**Automatic abstractions.** The abstractions of different components of a system in the compositional approach of Chapter 6 have been derived manually. Also, the chain of assume-guarantee proofs and the required systems for verification have been built manually. Automatic extraction of timed abstractions and automatic derivation of subsequent chain of reasoning constitute important topics for future research in this area.

**Applications.** Thanks to the rather theoretical nature of the proposed verification approach, its potential applicability covers a wider range of systems than those presented in the thesis, such as: custom transistor-level circuits that exploit the technology limits for performance, complex digital structures where synchronization is a crucial issue (*e.g.* dynamic MOS), asynchronous and GALS-type systems, real-time systems, etc. Therefore, it would be rather challenging to expose the proposed methods to such complex systems.

**Beyond control-dominated systems.** In the field of digital circuits, data-path circuitry is fairly easy to design correctly and become reusable once it is designed. On the contrary, the correct design of custom control circuitry can be a very difficult task. Since
verification of the former type of systems can be carried out much more efficiently with theorem provers such as HOL [GM93], for example, our approach for verification concentrates in the latter type of systems. An interesting approach might be to combine both approaches for the verification of systems composed of control and data-path units, applying each approach to solve problems in their respective area of expertise. Some examples in this direction have recently appeared (e.g. [KN02]).
TIMING ANALYSIS

Time is such a simple, almost primitive idea. It is just a means of material differentiation, a way of uniting us all; for in our external, material lives we value the synchronized efforts of individual people.

—Andrei Tarkovsky - Time Within Time: The Diaries, 1989

Summary

This appendix analyzes the problem of timing analysis as the computation of the time separation between events of a system. The previous work on the topic is reviewed. Finally, an algorithm for timing analysis on acyclic graphs is described in detail.
A.1 Introduction

Determining the time separation between events is a fundamental problem in the analysis, synthesis and optimization of timed concurrent systems. For example, if the bounds on the separation in time of two events can be computed, such information can be used in a number of ways: to simplify combinational and sequential logic by extracting temporal don’t care information (see Example 4.1); to verify that a system meets specified timing constraints; to identify an remove hazards from asynchronous circuits; etc.

The maximal separation time of two events $e_1$ and $e_2$ is computed as the maximum difference between their firing times, provided any possible assignment of delays to the events of the system. That is, $\text{Sep}_{\text{max}}(e_1, e_2) = \max \{ ft(e_1) - ft(e_2) \mid \text{for any delay assignment} \}$, where $ft$ denotes the firing time of an event.

In order to compute the maximal separation time between two events it is required, among other things, to determine how the synchronizations between concurrent executions affect the temporal behavior of the system. Thus, the efficiency of the timing analysis depends on the expressiveness power of the model. The simplest model (vertex 0) in the classification of Figure A.1 has neither concurrency nor conditional behavior. Computing the maximal separation time between two events only requires the sum of the delays on the path between both events. If the delays are expressed as ranges (vertex 1) the computation must consider the upper delay bounds. Similarly, the timing analysis is straightforward for models with only conditional behavior but no concurrency (vertices 1 and 3). On the contrary, the analysis for models that only include concurrency (vertices 4 and 6) is non-trivial even for the case where all delays are given as fixed values. The most general
models considered in Figure A.1 combine concurrency and conditional behavior (vertices 5 and 7). For these models, computing the maximal separation time between events is a PSPACE-hard problem, even in the case with fixed delays.

According the previous discussion, timing analysis techniques often tend to restrict the classes of models that can be analyzed, in favor of developing efficient algorithms. This follows the opposite direction than in timing verification, where most approaches try to cover the widest possible class of systems.

Verification of interfaces is a difficult area of system design because of the interactions of components that must meet specific timing requirements. Among others, the works by McMillan and Dill [MD92], Vanbekbergen [VGM92] and Walkup [Wal95] have addressed this topic by considering systems whose ranges of delays are determined from the implementation. Then, the timing analysis problem consists in computing the separation in time of specific events and ensure that they fall within the given bounds. On the other hand, Amon [AH99] has taken a different approach where the delays are manipulated symbolically leading to a set of inequalities that must be satisfied for any valid system implementation. In all these cases, however, the model of the system is restricted to particularly simple classes. That is, [MD92, VGM92] can only handle acyclic graphs, whereas [Wal95] only supports a limited form of interprocess communication.

When a system event depends on several incoming events, there are several possible timing semantics that can be defined. For example, that the event occurs as soon as one of the incoming events occurs (i.e. a minimum constraint). Or that the event waits for the last of the incoming events to occur (i.e. a maximum constraint). In [MD92] it was showed that the maximum separation problem in an acyclic graph with both minimum and maximum constraints is NPcomplete. The work in [Wal95] develops an algorithm for analyzing systems of maximum constraints and upper bound constraints, but only for acyclic graphs. Also, in [Gun93] it is shown that cyclic systems of minimum and maximum constraints exhibit periodic behavior, and methods for determining the cycle period are developed.

Regarding more sophisticated systems, in [MM93] a polynomial algorithm is presented that estimates the minimum and maximum time differences between events in a cyclic free-choice net. The algorithm unfolds the net into an infinite acyclic graph and examines two finite acyclic sub-graphs to determine the time-separation bounds. The limitation to free-choice nets is partially overcome by the work in [HB94, Hul95]. It provides a way to compute a single exact time separation between two events in a cyclic PN with more general types of choice.
A.2 Timing analysis on acyclic graphs

In [MD92] several algorithms for the computation of the minimum and maximum separation time between events on acyclic graphs were presented. Those algorithms included: a polynomial algorithm for the timing analysis with \( max \) constraints only; an exponential, but feasible in practice, algorithm for the case with \( max \) and linear constraints; and a branch and bound approach for the general case including \( min/max \) and linear constraints. The information obtained from these algorithms can be used to analyze whether two concurrent events are actually ordered in the timed domain. That is, \( e_1 \) precedes \( e_2 \) in the timed domain if \( \text{Sep}_{max}(e_1,e_2) < 0 \).

The verification approach presented in this thesis uses the algorithms of [MD92] to perform timing analysis on CES derived from traces. In this section we describe the \( max \)-only algorithm, which is the precursor of most later algorithms for timing analysis. Recall, however that this basic algorithm is only suitable for CES without disabling relations (see Section 4.4.1). In order to cope with disabling, linear constraints must be also taken into consideration for the analysis. The explanation of this latter algorithm is beyond the scope of this document.

The system under analysis is represented as a directed acyclic graph, where the vertices represent events and the edges are annotated with min-max delay intervals. The intervals are of the form \([d,D]\) or of the form \([d,\infty)\), being \( d \) and \( D \) the minimum and the maximum delay bounds, respectively. Then, the timing analysis problem is as follows: given two events \( e_i \) and \( e_j \) determine the strongest bound \( \Delta \) such that:

\[
ft(e_i) - ft(e_j) \leq \Delta
\]

where \( ft \) denotes the firing time of events. Thus, \( \Delta \) is the maximum difference between the firing times, provided any possible assignment of delays to the events of the system. That is, \( \text{Sep}_{max}(e_i,e_j) = \Delta \).

We describe the algorithm developed in [MD92] using a simplified version of the formulation presented in [AHHB93]. The algorithm consists of two simple steps.

First, we compute the so-called \( m \)-values backwards from \( e_j \) for the rest of events \( e_k \) of the graph in the following way:

\[
m(e_k) = \max \{ d(h) \mid \text{all paths } e_k \xrightarrow{h} e_j \}
\]

where \( d(h) \) is the sum of the minimum delay bounds \( (d) \) of the edges on the path \( h \). The \( m \)-values can be computed in linear time by a reverse topological traversal from \( e_j \). If there is no path from \( e_k \) to \( e_j \), denoted by \( e_k \not\sim e_j \), an arbitrary constant value is assigned to \( m(e_k) \). We set \( m(e_k) = 0 \) in these cases.

The next step consists in computing the so-called \( M \)-values. First, the \( M \)-value of the events without predecessors is set to 0. Then, in normal topological order for the rest of
events \( e_k \) of the graph:

\[
M(e_k) = \max \{ \min (M(e_l) + D - m(e_l) + m(e_k), 0) \mid e_l \to e_k \}
\]

If \( e_k \not\sim e_j \) the minimization with 0 is omitted.

Finally the maximum separation between events \( e_i \) and \( e_j \) is obtained as the following difference:

\[
\text{Sep}_{\max}(e_i, e_j) = \Delta = M(e_i) - m(e_j)
\]

**Example A.1** Figure A.2 illustrates the described algorithm by means of a simple example. Given the acyclic directed graph of Figure A.2 (a), \( \text{Sep}_{\max}(g, d) \) is computed.

Figure A.2 (b) depicts the computation of the \( m \)-values by means of a backwards traversal of the graph. The resulting \( m \)-values are annotated at the right of each corresponding event.

Figure A.2 (c) depicts the computation of the \( M \)-values by means of a forward topological traversal of the graph. The resulting \( M \)-values are annotated at the left of each corresponding event.

Thus we have that \( \text{Sep}_{\max}(g, d) = M(g) - m(d) = -2 - 0 = -2 \). This means that event \( g \) will always happen, at most, two time units before event \( d \).
ON THE ENABLING-COMPATIBLE PRODUCT

A creative artist works on his next composition because he was not satisfied with his previous one.


Summary

This appendix provides some implementation details of one of the key parts of the verification approach for timed systems presented in this thesis. That is, the enabling-compatible product, which allows the refinement of the untimed state space of a system by incorporating a set of relative timing constraints in the form of a lazy causal event structure.

In order to be self-contained, the appendix starts with a brief summary of the important notions around the enabling-compatible product. In particular, the rules that precisely define the product are included.

Then, details on the representation of a $\text{LzTS}$ and the state space of a $\text{LzCES}$ with boolean algebras are provided. Part of this material was already presented in Section 5.1.1 but is included here for completeness.

Finally, the different rules that define the enabling-compatible product are implemented by means of symbolic manipulations of the transitions relations of the events involved in the product.
B.1 Enabling-compatible product

This section describes how to refine the set of traces produced by a LzTS by considering the timing constraints coming from event delay bounds. The timing constraints are derived by a timing analysis on a CES corresponding to an eligible trace of a LzTS in the untimed domain. The refinement is performed through the parallel composition of a LzTS and a LzCES. Defining such composition requires both descriptions to be represented in a uniform way. To satisfy this requirement we first introduce a state-based representation for CESs.

B.1.1 State-based representation of a CES

An underlying transition system can be obtained from a CES. This process relies on the notion of configuration, which plays the role of global state of the CES.

**Definition B.1 (Configuration)**

Let \( CS = \langle \Sigma, \prec, \triangleright \rangle \) be a CES. \( C \subseteq \Sigma \) is a configuration iff:

- \( C \) is left-closed, i.e. \( \forall e_i \in C \) all predecessors of \( e_i \) by \( \prec \) are in \( C \), and
- disabled events do not belong to \( C \), i.e. \( e_i \in C \Rightarrow \exists e_j \in \Sigma : e_j \triangleright e_i \).

Notice that both \( \emptyset \) and the set of not disabled events \( \Sigma \setminus \mathcal{D} \) are trivial configurations.

Event \( e \in \Sigma \) is enabled in configuration \( C \) iff \( \neg \{ e \} \subseteq C \) and \( \forall e_j \in \Sigma \mid e_j \triangleright e_i \) \( e_i : e_j \notin C \). We denote by \( E(C) \) the set of all enabled events in configuration \( C \).

Configuration \( C \) precisely identifies a state of a CES, as the set of events occurred so far, such that if \( e \in C \) all its causal predecessors must be also in \( C \).

Every prefix \( \omega_i \) of a word (a topological order of the events) \( \omega \) in a CES is left-closed and disabled events do not fire along it. Thus every prefix \( \omega_i \) defines a configuration which is reached by firing the events from \( \omega_i \). Consideration of all possible words of a CES and their prefixes gives the set of reachable configurations, \( C \), where the initial configuration due to the empty prefix \( \omega_0 \) is denoted by \( \top \). The set of reachable configurations together with the partial order induced by the strict set inclusion \( \subseteq \), defines the graph of reachable configurations.

**Definition B.2 (Graph of reachable configurations)**

Let \( CS = \langle \Sigma, \prec', \triangleright' \rangle \) be a CES, and \( C \) be the set of reachable configurations of \( CS \). The graph of reachable configurations (GRC) of \( CS \) is a Hasse diagram over \( C \) and the partial order \( \subseteq \) interpreted in set-theoretical sense.

For the general case of a LzCES, \( LCS = \langle \Sigma, \prec', \triangleright' \rangle \), the graph of reachable configurations can be modeled by a LzTS \( G = \langle C, \Sigma, T, \top, \text{EnR} \rangle \) where: there is one state per config-
uration: \( C_1 \xrightarrow{e} C_2 \in T \) iff \( C_2 \) is reached by firing \( e \in \Sigma \) from \( C_1 \); the initial state corresponds to the initial configuration \( T \); and \( \text{EnR}(e) = \{ C \mid e \in \mathcal{E}(C) \} \).

### B.1.2 Refining the Reachability Space by Timing Constraints

In order to refine the state space of the system with a set of relative timing constraints, we have two objects at hand: a lazy TS \( A \), and another lazy TS \( G \) obtained from an event structure \( CS_\theta \). \( CS_\theta \) is derived from a particular trace \( \theta \) of \( A \) (actually by an appropriate suffix), thus giving only a partial specification of the behavior of \( A \). \( CS_\theta \) is refined through timing analysis yielding the lazy TS \( G \).

Refining the behavior of \( A \) by the timing constraints incorporated in \( G \) can be done by calculating the enabling-compatible product of \( G \) and \( A \), which is a particular case of transition system product under the restrictions of making synchronization by the same transitions and the same enabling conditions.

For sake of simplicity, and before introducing the rules of the enabling-compatible product below, we will add the special configuration \( \bot \) to \( G \). \( \bot \) denotes the fact that the product is not synchronizing, i.e., there is no enabling-compatibility with the state space of the CES and therefore, timing analysis does not apply for the involved traces.

Given the system \( A = \langle S, \Sigma_A, T_A, s_0, \text{EnR}_A \rangle \) and the state space of the LzCES containing the relative timing constraints \( G = \langle C \cup \bot, \Sigma_G, T_G, \top, \text{EnR}_G \rangle \), with \( \Sigma_G \subseteq \Sigma_A \), the enabling-compatible product of \( A \) and \( G \) is a new LzTS \( \langle S', \Sigma_A, T', s'_0, \text{EnR}' \rangle \) where:

- \( S' \subseteq S \times (C \cup \bot) \),
- \( s'_0 = (s_0, \top) \) if \( \mathcal{E}(\top) \subseteq \mathcal{E}(s_0) \), and \( s'_0 = (s_0, \bot) \) otherwise, and
- \( \forall e \in \Sigma_A, \text{EnR}'(e) = \{ (s, C) \in S' \mid s \in \text{EnR}_A(e) \} \).

**Remark:** The alphabet \( \Sigma_G \) is not properly a subset of \( \Sigma_A \). In fact \( \Sigma_G \) might contain several instances of any event in \( \Sigma_A \). That is, provided an event \( e \in \Sigma_A \), a set \( \{ e/1, e/2, \ldots \} \) of instances of \( e \) might be present in \( \Sigma_G \). This is the case for example when the corresponding LzCES is generated from a trace of \( A \) where several occurrences of event \( e \) appear along the trace. Thus we define \( \Sigma_G \downarrow \Sigma_A \) as the projection of the event occurrences in \( \Sigma_G \) over the actual events in \( \Sigma_A \). For example if \( \Sigma_A = \{ a, b, c \} \) and \( \Sigma_G = \{ a/1, a/2, c/1 \} \), \( \Sigma_G \downarrow \Sigma_A = \{ a, c \} \).

The transition relation \( T' \) is defined by the rules below. The rules are implied by the conditions on the enabling-compatibility of traces. The fact that \( (s, C) \in S' \) denotes that \( s \) and \( C \) have been reached by prefixes that are enabling-compatible, and that \( \text{map}(\mathcal{E}(s)) = \mathcal{E}(C) \). Given a state of the product \( (s, C) \) with \( C \neq \bot \), we will say that the state is in the timed domain, indicating that the timing analysis performed on \( CS_\theta \) can be applied to \( s \).
The rules that define the enabling-compatible product are as follows:

**Transitions entering the timed domain**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s_1, \bot) \xrightarrow{e} (s_2, \top))</td>
<td>(\text{enter} \equiv s_1 \xrightarrow{e} s_2 \in T_A \land E(\top) \downarrow \Sigma_A \subseteq E(s_2) \cap \Sigma_G \downarrow \Sigma_A)</td>
</tr>
</tbody>
</table>

These transitions are fired when the events enabled in \(\top\) are also enabled in \(s_2\). Thus, timing analysis can start being applied from \((s_2, \top)\).

**Staying inside the timed domain**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s_1, C_1) \xrightarrow{e} (s_2, C_1))</td>
<td>(\text{inside1} \equiv s_1 \xrightarrow{e} s_2 \in T_A \land \Sigma_G \downarrow \Sigma_A = E(s_2) \cap \Sigma_G \downarrow \Sigma_A)</td>
</tr>
<tr>
<td>((s_1, C_1) \xrightarrow{e} (s_2, C_2))</td>
<td>(\text{inside2} \equiv s_1 \xrightarrow{e} s_2 \in T_A \land C_1 \xrightarrow{e} C_2 \in T_G \land E(s_2) \cap \Sigma_G \downarrow \Sigma_A = E(C_2) \downarrow \Sigma_A)</td>
</tr>
</tbody>
</table>

Inside1 corresponds to the condition in which \(e\) does not synchronize with \(G\). Here the enablings of configuration \(C_1\) must be preserved, *i.e.* the firing of \(e\) cannot disable or enable events in \(\Sigma_G\).

For inside2, both \(A\) and \(G\) make a synchronized move which might affect the events from \(\Sigma_G\) in exactly the same way: if \(a \in \Sigma_G\) becomes enabled in \(A\) due to this move, it should also become enabled in \(G\), and vice versa.

**Exiting or staying outside the timed domain**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s_1, C_1) \xrightarrow{e} (s_2, \bot))</td>
<td>(\text{exit} \equiv s_1 \xrightarrow{e} s_2 \in T_A \land \neg(\text{enter} \lor \text{inside1} \lor \text{inside2}))</td>
</tr>
</tbody>
</table>

It can be shown that, in the enabling-compatible product, only the traces of the original LzTS which are enabling-compatible with the event structure are refined. This refinement excludes the traces which are not timing-consistent with respect to the timing constraints coming from the timing analysis on the event structure. All other traces are not changed, thus guaranteeing the conservativeness of the approach.

**B.2 Symbolic representation**

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 can be represented using BDDs for efficiency.
B.2.1 Encoding of a LzTS

Given a LzTS $A = \langle S, \Sigma, T, s_0, \text{EnR} \rangle$ the system $\langle 2^S, \cup, \cap, \emptyset, S \rangle$ is the boolean algebra of sets of states of $A$. As a consequence, each state $s \in S$ can be represented by means of an encoding function $Q : S \rightarrow \mathbb{B}^n$, with $n \geq \lceil \log_2(|S|) \rceil$. That is, given the set of boolean variables $\mathcal{V} = \{v_1, \ldots, v_n\}$, each state $s \in S$ 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 $X^Q_P : \mathbb{B}^n \rightarrow \mathbb{B}$ that evaluates to 1 for those vertexes of $\mathbb{B}^n$ that correspond to states in the set $P$, encoded using $Q$. Whenever the encoding is understood, we simply write $X_P$.

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 $R \subseteq P_1 \times P_2$ it is necessary to use two different sets of variables to identify the elements of each set. Current-state variables $v_1, \ldots, v_n$ for $P_1$ and next-state variables $v'_1, \ldots, v'_n$ for $P_2$. Thus, the cartesian product of a relation between $P_1$ and $P_2$ can be simply expressed as the product of the respective characteristic functions.

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 $A = \langle S, \Sigma, T, s_0, \text{EnR} \rangle$. 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. $A = \langle \mathcal{V}, \mathcal{V}', \Sigma, T, s_0, \text{EnR} \rangle$. Now, given an event $e \in \Sigma$ we can represent its enabling region, its firing region and its transitions relation, by means of the following characteristic functions:

- $EF(e) : \mathbb{B}^n \rightarrow \mathbb{B}$ such that $EF(e) = 1$ for all the states (encoded using $\mathcal{V}$) belonging to the enabling region of $e$, i.e. $\text{EnR}(e)$.
- $FF(e) : \mathbb{B}^n \rightarrow \mathbb{B}$ such that $FF(e) = 1$ for all the states (encoded using $\mathcal{V}$) belonging to the firing region of $e$, i.e. $\text{FR}(e)$.
- $TR(e) : \mathbb{B}^{2n} \rightarrow \mathbb{B}$ such that $TR(e) = 1$ for all the relations $(s_1, s_2)$ such that there is a transition of event $e$, i.e. $s_1 \xrightarrow{e} s_2 \in T$. 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 $EF'(e)$ and $FF'(e)$, respectively. Also, when the sets of variables in a transition relation are interchanged we will write $TR'(e)^{-1}$.

B.2.2 Encoding the state space of a LzCES

The space of configurations of a LzCES $\langle \Sigma_G, < \rangle$ derived from a given encoded LzTS $A = \langle \mathcal{V}, \mathcal{V}', S_A, \Sigma_A, T_A, s_{0_A}, \text{EnR}_A \rangle$, form a LzTS $G = \langle C \cup \{\bot\}, \Sigma_G, T_G, \top, \text{EnR}_G \rangle$. The config-
uations in $C$ can be identified in terms of the set of enabled events. Then for each configuration $C$ a characteristic function $\gamma(C) : \mathbb{B}^n \rightarrow \mathbb{B}$ (encoded using $\mathcal{V}$) that identifies it can be built using such enabling information as follows:

$$
\gamma(C) = \prod_{\xi / i \in \Sigma_G} \left\{ \begin{array}{ll}
EF(e) & \text{if } e \in \Sigma_A \land e / i \in \mathcal{E}(C) \\
\neg EF(e) & \text{if } e \in \Sigma_A \land e / i \notin \mathcal{E}(C)
\end{array} \right.
$$

Notice that the enabling information corresponds to that coming from the LzTS of reference form which the LzCES was derived. If $\gamma$ is expressed using the set of next-state variables $\mathcal{V}'$ we will write $\gamma'(C)$.

In some cases configurations in a CES cannot be distinguished by looking only at the enabling information, either because it is incomplete or because it is ambiguous itself. If that happens some extra encoding variables are required that help to disambiguate. For that purpose, let $\mathcal{U} = \{u_1, \ldots, u_m\}$ and $\mathcal{U}' = \{u'_1, \ldots, u'_m\}$ be respectively, the set of current and next-state extra boolean variables used to encode the configurations of the CES. Hence, in general, for each configuration $C$ a characteristic function $\xi(C) : \mathbb{B}^m \rightarrow \mathbb{B}$ (encoded using $\mathcal{U}$) will exist. Again, if $\xi$ is expressed using the set of next state variables $\mathcal{U}'$ we will write $\xi'(C)$.

Finally one more boolean variable called $IN$ will be used to indicate whether the enabling-compatibility is preserved during the composition process. That is if $IN = 1$ in the characteristic function of a state it means that the enabling compatibility is being satisfied in it, while $IN = 0$ will indicate the opposite. A corresponding next state variable $IN'$ will also be used to properly define the new transition relations.

With all the above considerations, the LzTS corresponding to the space of configurations of a LzCES used in the enabling-compatible product will have the following form: $G = \langle \mathcal{V} \cup \mathcal{U} \cup \{IN\}, \mathcal{V}' \cup \mathcal{U}' \cup \{IN'\}, C \cup \{\bot\}, \Sigma_G, T_G, \top, \text{EnR}_G \rangle$.

### B.3 Computation of the new transition relations

For each event of the system being refined by the enabling-compatible product with a CES, its new transition relation is computed as a set of different parts. Each part corresponds to the different situations of the enabling-compatible product outlined above. The new transition relation is therefore computed as the addition of the parts.

#### B.3.1 Transitions entering the timed domain

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(s_1, \bot) \xrightarrow{e} (s_2, \top)$</td>
<td>$e \in \Sigma_A \land \mathcal{E}(s_2) \downarrow \Sigma_A \subseteq \Sigma_{G \downarrow} \subseteq \Sigma_A$</td>
</tr>
</tbody>
</table>

The enter condition for event $e$ is computed as follows:

$$
\text{enter}(e) = \overline{\mathcal{V}} \cdot \text{cond} \cdot T_R(e) \cdot \gamma'(\top)
$$
Then, the corresponding part of the new transition relation is computed as follows:

$$TR_{\text{enter}}(e) = TN \cdot \text{cond} \cdot TR(e) \cdot \gamma'(\top) \cdot \xi'(\top) \cdot IN'$$

In the previous equations, \( \text{cond} \) depends on the way the CES was built and its relation with the original trace, if any. Thus:

$$\text{cond} = \begin{cases} 
1 & \text{if not nodal point or no reference trace} \\
\prod_{e \in E(\top) \downarrow \Sigma_A} EF(e) & \text{if nodal point}
\end{cases}$$

### B.3.2 Staying inside the timed domain: no synchronization

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s_1, C_1) \xrightarrow{e} (s_2, C_2))</td>
<td>inside1 (\equiv s_1 \xrightarrow{e} s_2 \in T_A \land E(s_1) \cap \Sigma_G \downarrow \Sigma_A = E(s_2) \cap \Sigma_G \downarrow \Sigma_A)</td>
</tr>
</tbody>
</table>

The inside1 condition for event \( e \) is computed as follows:

$$\text{inside1}(e) = IN \cdot \sum_{C_1 \in C} \gamma(C_1) \cdot \xi(C_1) \cdot TR(e) \cdot \gamma'(C_1)$$

Then, the corresponding part of the new transition relation is computed as follows:

$$TR_{\text{inside1}}(e) = \left( \sum_{C_1 \in C} \gamma(C_1) \cdot \xi(C_1) \cdot TR(e) \cdot \gamma'(C_1) \cdot \xi'(C_1) \right) \cdot IN \cdot IN'$$

Notice that if \( e \in E(C) \downarrow \Sigma_A \) and \( e \) is self-disabling, the product \( \gamma(C) \cdot TR(e) \cdot \gamma'(C) \) will be 0.

### B.3.3 Staying inside the timed domain: synchronization

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s_1, C_1) \xrightarrow{e} (s_2, C_2))</td>
<td>inside2 (\equiv s_1 \xrightarrow{e} s_2 \in T_A \land C_1 \xrightarrow{e} C_2 \in T_G \land E(s_2) \cap \Sigma_G \downarrow \Sigma_A = E(C_2) \downarrow \Sigma_A)</td>
</tr>
</tbody>
</table>

The inside2 condition for event \( e \) is computed as follows:

$$\text{inside2}(e) = IN \cdot \sum_{C_1 \xrightarrow{e/\xi} C_2 \in T_G} \gamma(C_1) \cdot \xi(C_1) \cdot TR(e) \cdot \gamma'(C_2)$$

Then, the corresponding part of the new transition relation is computed as follows:

$$TR_{\text{inside2}}(e) = \left( \sum_{C_1 \xrightarrow{e/\xi} C_2 \in T_G} \gamma(C_1) \cdot \xi(C_1) \cdot TR(e) \cdot \gamma'(C_2) \cdot \xi'(C_2) \right) \cdot IN \cdot IN'$$
### B.3.4 Transitions re-entering the timed domain

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s, C) \xrightarrow{e} (s_2, \top))</td>
<td>(\text{enter } \equiv s \xrightarrow{e} s_2 \in T_A \land \neg(\text{inside1 } \lor \text{inside2}) \land \land E(\top) \downarrow \Sigma_A \subseteq E(s_2) \cap \Sigma_G \downarrow \Sigma_A)</td>
</tr>
</tbody>
</table>

This case corresponds to re-entering the enabling-compatibility, i.e. being willing to exit but in fact entering again. The reenter condition for event \(e\) is computed as follows:

\[
\text{reenter}(e) = IN \cdot \text{cond} \cdot \left(\neg \text{inside1}(e) + \text{inside2}(e)\right) \cdot TR(e) \cdot \gamma'(\top)
\]

Then, the corresponding part of the new transition relation is computed as follows:

\[
TR_{\text{reenter}}(e) = \text{reenter}(e) \cdot \xi'(\top) \cdot IN'
\]

Where \(\text{cond}\) is the same equation used to specify the enter condition.

### B.3.5 Exiting or staying outside the timed domain

<table>
<thead>
<tr>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s, C) \xrightarrow{e} (s_2, \bot))</td>
<td>other (\equiv s \xrightarrow{e} s_2 \in T_A \land \neg(\text{enter } \lor \text{inside1 } \lor \text{inside2}))</td>
</tr>
</tbody>
</table>

The other condition for event \(e\) is computed as follows:

\[
\text{other}(e) = \left(\text{enter}(e) + \text{inside1}(e) + \text{inside2}(e) + \text{reenter}(e)\right) \cdot TR(e)
\]

Then, the corresponding part of the new transition relation is computed as follows:

\[
TR_{\text{other}}(e) = \text{other}(e) \cdot \xi'(\bot) \cdot IN'
\]

### B.3.6 New transition relation

Finally, the new transition relation for event \(e\) is computed as the addition of the different parts computed above:

\[
TR(e) = TR_{\text{enter}}(e) + TR_{\text{inside1}}(e) + TR_{\text{inside2}}(e) + TR_{\text{reenter}}(e) + TR_{\text{other}}(e)
\]

### B.3.7 Lazy events

Due to the refinement of the state space imposed by the enabling-compatible product, some events become lazy. That is, the firing region becomes a strict subset of the enabling region. As a consequence their firing function must be also updated. Thus, for each \(e \in \Sigma_A\) which becomes lazy inside the composition area due to some relative timing constraint, that is \(FR_G(e/i) \neq \text{En}_G(e/i)\), we have that:

\[
FF(e) = IN \cdot \left(\sum_{C \in FR_G(e/i) \land e \in \Sigma_A} \gamma(C) \cdot \xi(C)\right) \cdot FF(e) + \overline{IN} \cdot FF(e)
\]
B.3.8 Initial state

Finally, the initial state of the resulting LzTS must be also updated. Its encoding must distinguish the fact that the initial state of the system belongs to the enabling-compatibility area determined by the product or not. Thus we have that:

$$\mathcal{X}_0 = \mathcal{X}_{04} \cdot (\gamma(\top) \cdot \xi(\top) \cdot I N + \overline{\gamma(\top)} \cdot \xi(\bot) \cdot \overline{I N})$$
VERIFICATION-RELATED COMMANDS

"The man of science has learned to believe in justification, not by faith, but by verification."

—Thomas H. Huxley - Aphorisms and Reflections, 1907

Summary

TRANSYT provides a number of commands for the analysis of properties in transition systems, and in particular for the formal verification of timed systems. This appendix introduces the commands and their different options, which are related with the work presented in this thesis. For more information about other commands of the TRANSYT tool, refer to [PPb] or the on-screen help of the tool.
C.1 Failure analysis

Several commands are provided by TRANSYT in order to specify and analyze failure conditions for verification. A brief introduction to these commands is given in the following sections.

C.1.1 The add_fail command

Prior to the verification process, a number of properties must be associated to the system under verification. This can be done inside the tsif files that describe the corresponding models for verification. Moreover, the add_fail command allows to specify the failure conditions externally to the tsif files, such that different properties for the same system can be analyzed without actually modifying the system specification itself. This command allows to assign a boolean failure condition to either a label, an event or the entire transition system. Predefined failure conditions can be also assigned. Examples of the use of this command appeared in Chapter 5.

What follows is the on-screen help provided by TRANSYT for the add_fail command:

```
> help add_fail
TS: help add_fail

Add selected failure conditions to the default system.

Failures can be added to three types of objects:
- FAIL: to a transition system specified by <name1>
- LFAIL: to a label specified by <name1>
- EFAIL: to the event <name1> of label <name2>

The equation of the fail condition is specified by <equation>.

Options:
- -d<level> Temporal setting of the verbose level to N.
- -p Adds a persistency fail condition.
- -i Adds properties to internal labels.
- -m Adds a conformance fail condition.
- -d Adds a deadlock fail condition.
```

C.1.2 The check_fails command

The failure conditions can be evaluated in a variety of ways during the reachability analysis of the system under verification with the traverse command. Such evaluation, however, often makes the traversal process quite costly since, for example, each time an event is fired the failure conditions need to be evaluated. Despite of this mechanism, TRANSYT also allows the evaluation of failure conditions once the reachability analysis has been completed. Thus, the system can be first traversed with no failure analysis (traverse command with the -nF flag), and afterwards the check_fails command can be used to evaluate the failure conditions over the whole reachability space.
What follows is the on-screen help provided by TRANSYT for this command. Currently, no particular options exist for the check_fails command.

```
ts > help check_fails
ts:: check_fails [-dbxW]
```

Checks the fail conditions of the default system after it has been traversed.

Options:
- `dbxW` Temporal setting of the verbose level to N.

C.1.3 The `print_fails` command

The command `print_fails` allows to show information about what failure conditions are associated to the different objects of a system. Moreover, if the failure conditions have been analyzed either during the traversal or with the `check_fails` command, the resulting failure states can be also shown in the form of boolean characteristic functions. Examples of the use of this command appeared in Chapter 5.

What follows is the on-screen help provided by TRANSYT for the `print_fails` command.

```
ts > help print_fails
ts:: print_fails [-dbxW][-e][-l][-t][-a][-s] <model_name>
```

Shows the fail information stored for the specified TS.

Options:
- `dbxW` Temporal setting of the verbose level to N.
- `e` Prints information only for the events.
- `l` Prints information only for the labels.
- `t` Prints information only for the TSs.
- `a` Prints information for all fail conditions (by default only those with failure states are shown).
- `s` Prints the failing states.

C.2 Analysis of delay relations

The command `prune_dr` intends to refine the untimed state space of a system by using easy-to-find timing relations between events. Such relations can have nothing to see with the specified failure conditions, but its application can help to refine fake untimed concurrency situations, for example, so that a later verification process can be less costly.

Three options allow to explore different types of timing relations:

- `-npp` If this option is set, the global nodal states of the system are identified and the delays of the events than become enabled in them are analyzed. Thus, given two events $e_i$ and $e_j$ newly enabled in a nodal point, if the upper delay bound of $e_i$ is smaller than the lower delay bound of $e_j$, then event $e_i$ will always fire before $e_j$ in the timed domain. That is, an obvious relative timing relation is found between both events. As a consequence, the firing function of $e_j$ can
be updated accordingly, so that both events become sequential, and the number of untimed states of the system is reduced. The reachability set of the system is not needed at any time.

-pwp Consider every pair of event in a system such that: they are simultaneously enabled in some set of states; they are simultaneously not enabled in some other set of states; and the first situation is reachable from the second one in a forward traversal step. Under these conditions, a local nodal point for such pair of events exists in the system. Therefore, if the upper delay bound of one of them is smaller than the lower delay bound of the other, an obvious relative timing relation is found, that can be used to prune the state space of the system.

If this flag is set, a CES containing both events is built, a trivial timing analysis is performed over it, and the resulting LzCES is finally composed with the original system. Since the process only modifies the transition relations of the affected events, the reachability set of the system is not needed.

Notice that is options also covers the previous one. However, in this case LzCESs are used, hence increasing the CPU cost and possibly causing state splitting.

-gwp This option extends the type of analysis provided by the previous option but for groups of events, not just pairs.

Finally, what follows is the on-screen output of running the help prune_dr command.

ts > help prune_dr

Pruning of untimed concurrency which is actually fake if delays are considered in <<ts name>>.

Options:
-dh2 Temporal setting of the verbose level to N.
-npp Perform pairwise pruning without using ESs, only with the events around nodal points, if any.
-pwp Perform events pairwise pruning based on their delays. This also covers the previous case, but here timed event structures are used anyway, therefore increasing CPU cost and splitting.
-gwp Perform events group-wise pruning based on their delays.

C.3 The uverif Command

In Chapter 5, a verification framework was presented for the verification of a system which must satisfy input-output conformance with respect to a given specification (see Figure 5.8). The (mirrored) specification acts as the environment of the system under verification, thus exercising the inputs of the system and reacting to the resulting outputs. The framework is typical of the verification of untimed systems such as speed-independent
asynchronous circuits. In such case, the verification consists in building the (untimed) reachability space of the resulting closed system and then checking for violations of the given failure conditions. TRANSYT implements this functionality through the uverify command. The command automatically mirrors the specification to build the environment of the system under verification, and builds the final closed system. Automatic failure conditions for input-output conformance, persistency and dead-lock can be also computed if appropriate options are specified (see below).

This command is particularly interesting for our purposes, due to the possibility of just building the closed system and the failure conditions for verification, and leaving the resulting system available for later manipulation. In particular for the verification of the timed behavior with the tverify command.

What follows is the on-screen output of running the help uverify command. A brief description is provided for the different options.

```
ts > help uverify
```
```
ts:: uverify [options] <ts name spec.> <ts name impl.>

  Untimed verification of a system <ts name impl.> versus its specification <ts name spec.>.

  Options:
  -dN       Temporal setting of the verbose level to N.
  -VnotConformance Do not check conformance of the implementation with respect to the given specification.
  -VnotPersistency Do not check persistency in the implementation.
  -Vdeadlock Check also the presence of dead-locks in the closed system.
  -Vclose    Build the closed system and create fails only.
  -Vnotdestroy Do not destroy intermediate TSs after verification.
```

C.4 The tverify command

The tverify command of the TRANSYT tool implements the relative timing-based verification approach for timed systems presented in Chapter 4.

This command has two possible forms. The first one is similar to that of the uverify command. That is, two systems (the system under verification and its specification system) are given at the command line, together with flags to set the automatic construction of failure conditions (input-output conformance, persistency and dead-lock). This form of the tverify command can be superseded by the combination of the uverify command to build the closed system for verification, and then the second form of the tverify command to verify the closed system. Actually, the second form of the command for the verification of properties on a system is the most commonly used. See Chapter 5 for examples.

A number of options to control the execution and the output produced by this command are available. We describe them in the following sections.
C.4.1 Output

In Chapter 5 the on-screen textual output of the `tverif` command was shown in several examples. This output is very concise since it intends just to provide a sort of progress indicator of the verification process.

Despite of the brief on-screen output the execution provides much more detailed information in a sort of log file. Namely, a file with the name of the system being verified and the `.out` extension is generated. The amount of information in the file depends on the verbosity level specified with the `-dbxN` option, where `N` is a decimal digit. Bigger values of `N` imply more verbosity. A detailed description of the contents of the log file would require a lot of space, hence it is not given here. Simply say that, among other debugging information, details about the state where the failure trace stops, which is the event causing the failure, which strategies are used to build the event structure, etc. are provided for each iteration of the verification process.

Regarding the log file, finally say that of the `-html` option is specified, the log information is generated in the form of a browsable HTML file. The file has the name of the system under verification plus the extension `.html`. The file contains hyperlinks to a number of other files for the different objects (traces, CES, etc.) generated at each iteration of the verification process (see below).

The execution of the `tverif` command also creates a directory named with the name of the system under verification plus the extension `.files`. The directory contains, upon demand, a number of files with specific information for each iteration. Namely, the following different types of information can be provided at each iteration:

- The failure trace leading from the initial state of the system up to the state where the failure being verified was detected.

- A CES capturing the causality relations of all the events in the failure trace. Since the failure trace can be very long in some cases, the construction of this CES can also take a significant amount of time. This object is not necessary for the verification process, but can be useful for debugging purposes, for example.

- The suffix of the failure trace necessary to build a CES that provides enough timing information as to prove the non-existence of the failure in the timed domain.

- The resulting LzCES that captures the relative timing constraints that disprove the failure being verified.

- The complete lazy state space of the system. Producing this output may take a considerably amount of time unless the system is really small, say less than a few hundred states. In any case, the later manual analysis of more than a hundred states
might result in a very tedious task. Therefore, the usefulness of this information is often reduced to debugging purposes.

The generation of the files for the different objects can be controlled respectively with the following options: \texttt{-VwriteTraceN}, \texttt{-VwriteESN}, \texttt{-VwriteSuffixN}, \texttt{-VwriteTESN} and \texttt{-VwriteSTD}. Where \texttt{N} is a decimal digit which specifies the format of the output for each object. More precisely, \texttt{N} may take the following values: 0 (default) to produce no output for the object; 1 to produce a DOT (.dot) file for graphical visualization or printing of the object; 2 to produce a text (.txt) file for manual manipulation or interchange of the object with other users; or 3 to produce an XML (.xml) file for inter-operability between applications. The same option can be specified with different values for \texttt{N}, thus producing output files with different formats for the same object. Notice that in case of the \texttt{-VwriteSTD} option, no value for \texttt{N} can be given. If the option is specified, a DOT file is generated for the lazy state space of the system at the given iteration. Thus, the evolution of the incremental refinements can be graphically analyzed, for example.

**Failure trace and suffix**

A failure trace starts from the initial state of the system under verification. The trace reflects the sequence states and firings of events that lead to the failure state, as well as information about the enabled and fireable events at each visited state. The trace ends in a state where the firing of an event either violates a transition failure condition, or lead to a state where an state failure condition is violated. The following explanations correspond...
to the visual aspect of the DOT file for a failure trace (-VwriteTrace1 option) or suffix (-VwriteSuffix1 option).

States are depicted by means of either ovals or boxes. States inside a box correspond to nodal states. If the initial state of the system belongs to the trace or the suffix being depicted, the corresponding box or oval is filled in yellow. States are named following an increasing sequence of numbers. Since the same state can be visited more than once along a trace, between parentheses the occurrence number of the state inside the trace is shown.

Transitions between states are labeled with the name of the event producing the transition, followed by the name of the label the event belongs to, and the occurrence number of the event along the trace in parentheses. Other events enabled at each state are depicted as hanging arcs. If an enabled event is not fireable in the state, a hanging line is just drawn. Different colors and types of lines are used to depict special events:

- Dotted arcs correspond to events that are disabled by the event firing in the given state.
- Blue arcs correspond to events which are in conflict with the event that fires in the given state. That is, the event with the arc in blue fired, it will disable the event that fires currently in the trace.
- Red arcs correspond to events that cause a failure situation in the given state.
- Finally, green arcs correspond to events fireable concurrently with some failure event, and could escape from the failure situation.

Figure C.1 depicts the visualization of the DOT file generated for the failure trace in the third iteration of the verification of the example in Section 5.2. In the figure: states $s_0$ and $s_1$ are nodal; event $f$ is not fireable in states $s_1$ and $s_2$; event $d$ is disabled by the firing of event $b$ in state $s_1$, and the firing of event $d$ in state $s_1$ would disable $b$; finally, the firing of $f$ in state $s_3$ causes a failure situation, which could be escaped if a second occurrence of $d$ or $b$ fired first.

Similar text-based or XML-based descriptions are produced by TRANSYTT for a given failure trace (-VwriteTrace2 and -VwriteTrace3 options, respectively) or a suffix of the failure trace (-VwriteSuffix2 and -VwriteSuffix3 options, respectively). Details on the formats of the resulting files can be found in [PPl].

**CES and LzCES**

For each failure trace, a CES can be generated which captures the causal information between all the events appearing in the trace. Also the portion of the CES corresponding to the chosen suffix of the failure trace, can be generated in the form of a LzCES including the relative timing relations. The following explanations correspond to the visual aspect of the DOT file for a CES (-VwriteES1 option) or a LzCES (-VwriteTES1 option).
Events are depicted as ovals and are labeled with the name of the event, followed by the name of the label the event belongs to, and the occurrence number of the event in the failure trace in parentheses. Events corresponding to failure situations in the trace used to derive the CES are colored red. The root events (i.e. with no predecessor) of the CES are those events enabled at the initial state of the failure trace or suffix used to derive it.

Ares representing causality relations between events are drawn as black arrows. Such arcs are annotated with the delay ranges of the destination event. The delays are used for timing analysis in the corresponding iteration of the verification process. Disabling relations between events are depicted as dashed blue arcs, from the disabler to the disabled event, according to the information extracted from the trace. Finally, the relative timing relations are drawn as dotted black arcs, indicating that the source event fires before the destination event.

If the initial state of the failure trace or suffix used to derive the CES or the LzCES was a nodal state, then the causal arcs leading to the root events start in an imaginary common event which enabled all the root events simultaneously. And the arcs are annotated with the corresponding $[min, max]$ delay intervals. Conversely, the arcs leading to the root events are parallel and are annotated with $[0, max]$ delay intervals, if the initial state of the failure trace or suffix was not nodal.

Figure C.2 depicts the visualization of the DOT file generated for the LzCES in the third iteration of the verification of the example in Section 5.2. The LzCES was built using the suffix starting from state $s_1$ in the failure trace of Figure C.1. In the figure, the arcs leading to the root events start from a common point since the initial state of the failure
trace was nodal, event $b(1)$ disables event $d(1)$; and two relative timing relations from events $b(2)$ and $d(2)$ to event $f$ are depicted.

Similar text-based and XML-based descriptions are produced by TRANSYT for a given CES (-WriteES2 and -WriteES3 options, respectively) or a LzCES (-WriteTES2 and -WriteTES3 options, respectively). Details on the formats of the resulting files can be found in [Ppb].

Lazy state space

With option -WriteSTD a DOT file for the lazy state space of the system under verification can be generated at each iteration of the process. A circle is drawn for each state, which are numbered. The initial state is indicated by a double circle. Also, those states covered by the last refinement are colored green, whereas the remaining failure states are colored yellow. Arcs between states indicate the transitions of the system. The lazy transitions are not depicted.
Figure C.2 shows the visualization of the DOT file generated for the lazy state space after the third refinement of the verification of the example in Section 5.2.

C.4.2 Construction of the failure trace

At the time of writing, only two options allow to control the way the failure traces are built at each iteration of the verification process.

-VfailTraceN This option indicates the way a failure situation from which the trace will be built is searched. If the value of N is set to 1 or 2, it specifies respectively that a BFS or a chained partial traversal must be executed until a failure state is reached. If N is set to 3, a fast symbolic simulation search for failure states is used. Although the BFS-based method (default) is the slowest, it guarantees the construction of the shortest possible failure traces. Conversely, the simulation-based approach is the fastest, but at the cost of often building much longer failure traces. The approach based on a partial traversal with chaining represents an intermediate alternative.

-VextendTraceN This option indicates whether once the failure trace is built, it must be extended to beyond the failure state found, in order to capture other states that also present failure situations. If N is set to 0 (default), no extension is performed. Whereas if N is set to 1 or 2, the trace is extended visiting states (if any) where the same failure situation is given, or visiting any subsequent failure state, respectively. Notice that covering several failure situation may often yield to more complex CESs and LzCESs. Although this may reduce the number of iterations of the verification process, it also complicates the timing analysis, the enabling-compatible product, and the readability of the timing constraints for back-annotation.

C.4.3 Construction of the LzCES

In order to build the LzCES at each iteration of the verification process, by default, the smallest possible suffix of the failure trace is taken such that it contains at least one failure situation. Then, the suffix is extended backwards until the timing analysis on the CES resulting from the suffix proves the non-existence of the failure, or contradicts the firing order of the events in the trace. That is, the shortest suffix is taken that yields a CES from which it can be proved that the trace is not timing-consistent.

Several options allow to control the way the LzCES is derived from an appropriate suffix of the failure trace.

-Acapf This option indicates that the shortest suffix of the failure trace that can be used to build the LzCES must contain all the failure situations present in the trace. Since more failure situations are covered by a single CES, less iterations of
the verification process are potentially required, at the expense of increasing the size of the LzCES at each iteration, and thus compromising its readability for later back-annotation.

- **-Apreconc** Similarly to the previous option, if the option **-Apreconc** is specified, all the events which are concurrent with the root events of the LzCES according to the failure trace, must be included also in the LzCES. In such a way, the later enabling-compatible product will cover an area of the state space with less entry-points, hence producing less state splitting.

- **-AtaComplete** This option forces to take the whole failure trace as the suffix used to build the LzCES. This results interesting in some cases, since a complete view of the causality and timing relations between the events in the trace is captured under a single, although often big, structure. Moreover, the enabling-compatible product with the resulting LzCES often results in a very localized refinement of the state space, since many conditions on the enabledness of events must be satisfied.

- **-AfailGuided** This option forces the use of heuristics which try to reduce the size of the resulting LzCES for refinement. The heuristics focus the analysis on the particular failure situation being analyzed in a given iteration of the verification process, and discard timing relations unrelated to the failure.

- **-AfilterEdges** In a similar vein, is this option is specified the size of the LzCES is tried to be reduced. In this case, the timing relations that do not help to prove the non-existence of the failure situation in all the failure states of the trace, are ignored. The result is that all the events in the final LzCES are related to the failure situation being analyzed. This and the previous option are recommended in order to improve the readability of the resulting LzCESs for later back-annotation.

Despite of the above options, two additional options control the use of nodal points information in order to build the LzCES. If a nodal point is reached during the extension of the suffix of the trace, the delays of the root events in the LzCES can be set to the corresponding $[\text{min}, \text{max}]$ ranges. Otherwise, the conservative $[0, \text{max}]$ delay range must be used. Although it is generally desirable to consider nodal points for that purpose, sometimes the conservative timing analysis yields LzCESs that produce a better refinement of the state space during the enabling-compatible product. By default, only global nodal points are taken into account. In order to consider also the local nodal points, the **-Elnp** option must be specified. The **-Enotgmp** option disables the use of global nodal points information.
C.4.4 Timing analysis

Two algorithms for timing analysis on the CES are implemented in TRANSY: an exact algorithm due to [MD92] and a faster approximate algorithm due to [CDY99]. Since the latter algorithm is conservative, it often provides timing relations which produce less aggressive state space refinements during the enabling-compatible product. Due to this fact, some failure situations which do not exist in the timed domain of the system under verification, cannot be properly refined from the state space. That is, the verification process can result in conservative false negatives. The exact timing analysis algorithm is used by default. In order to select the approximate timing analysis algorithm, the -EapproxTA option must be specified to the tverif command.

Finally, if option -EcheckConc is specified, only timing relations between concurrently firable events in the CES are taken into account. In such a way, the resulting LzCES is simplified so that less redundant back-annotation information is provided. However, the simplification of the LzCES can result in more iterations of the verification process, to include the discarded timing relations in the analysis of other failure situations.

C.4.5 Miscellaneous

Several options allow to control other miscellaneous aspects of the verification process. Namely:

- VtraverseFreqN Although a complete traversal of the state space of the system is not required along the iterations of the verification process, it can be enforced every N iteration with the VtraverseFreqN option, where N is a decimal digit. The complete traversal allows, among other things, a precise tracking of the remaining failure situations in the refined state space. Also, the VwriteSTD option only applies to iterations where the complete state space has been computed. By default, no complete traversal is performed until the last iteration of a succeeding verification process, i.e. N equals 0.

- VreorderFreqN The boolean variables used to represent the system symbolically can be reordered every certain number of iterations in order to improve the size of the BDDs. Despite of the immediate memory savings, the subsequent iterations of the verification process often complete faster thanks to the improvement in the boolean operations. Nevertheless, it must be taken into account that variable reordering is a very costly operation and should not be invoked too often. The value of N in the option VreorderFreqN fixes the frequency of the variable reordering process, where N is a decimal digit. By default no variable reordering is performed, i.e. N equals 0.
-VminimizeFreqN  As it has been shown in Appendix B the implementation of the
enabling-compatible product using BDDs requires the incorporation of several boolean
variables to encode the configurations of the GRC and to distinguish the states where
the timing analysis applies and those states where it does not apply. As a result,
the size of the BDDs that encode the transition relations of the events of the sys-
tem, sometimes grow exaggeratedly. However, as the number of iterations increases,
some of those variables may become redundant, since later refinements may have
pruned the part of the state space where those variables made sense. For this reason,
TRANSYT can try to remove some of the redundant variables and recompute the sim-
plicated equations periodically. The option -VminimizeFreqN indicates the frequency
of such variable removal, where N is a decimal digit (N equals 0 by default).
Remark that at the time of writing, this option is still in experimental use. However,
the preliminary experiments show promising reductions in the sizes of the BDDs and
also in the CPU times.

-VnIterN  This option allows to specify the number of iterations of the verification
approach that must be performed. As usual, N is a decimal digit. This allows an
incremental verification process controlled by the user. If this option is not give, the
verification process proceeds with as many iterations as required in order to prove
the correctness of the system according to the properties under verification, or a
counterexample failure trace that proves its incorrectness is found.

-Vpwp  This option is equivalent to pruning from the state space, the delays relation
between simultaneously enabled pairs of events, before the actual verification process
starts. That is, this option is equivalent to the prune_dr -pwp command.

C.4.6  Summary of the tverif command

What follows is the on-screen output of running the help tverif command. A brief
description is provided for all the aforementioned options.

  ts > help tverif
  ts:: tverif [options] <ts name spec.> <ts name impl.>

    Timed verification of a system <ts name impl.> versus its specification
    <ts name spec.> .

Specific options:
- VnotConformance  Do not check conformance of the implementation with respect
to the given specification.
- VnotPersistency  Do not check persistency in the implementation.
- Vdeadlock        Check the presence of dead-locks in the system.
ts:: tverif [options] <ts name>

Timed verification of a system <ts name>.

Specific options:
-WriteN iter
Perform only N iterations. If N=0 (default) iterate normally.

Common options:
-Verbose
Temporal setting of the verbose level to N.
-NoDoIt
Do not destroy intermediate TSs after verification.
-Verbose
Perform atom's pairwise pruning before starting verification, based on their delays.
-VfailTrace
Indicates the way the failure trace is searched: (1) to perform a partial traversal; (2) to perform a partial traversal using chaining; (3) to perform a fast simulation (bughunt).
-ExtendTraceN
Extend traces following fails. N may take the values: 0 for no extension, 1 for extensions following a single fail, or 2 for multiple fails extensions (default N=0).
-VtraverseFreqN
Force reachability analysis every N iterations (default N=1).
-VreorderFreqN
Force reorder of BBD variables every N iterations (default N=0, i.e. no variable reorder).
-VminimizeFreqN
Force minimization of redundant BBD variables every N iterations (default N=0). N must be multiple of that specified with -VtraverseFreq option.
-WWriteTraceN
Write the failure trace up to the initial state at each iteration. N indicates the output format: 0 for no output, 1 .dot, 2 for .txt, and 3 for .xml.
The different options are cumulative so that several outputs can be produced. Default value is 0.
-WWriteSuffixN
Write the portion of the failure trace used at each iteration. N indicates the output format: 0 for no output and 1 for .dot .outputs can be produced. Default value is 0.
-WWriteESN
Write the complete ES for the full failure trace, at each iteration.
-WWriteTESN
Write the timed ES portion at each iteration. N indicates the output format: 0 for no output, 1 .dot, 2 for .txt and 3 for .xml.
The different options are cumulative so that several outputs can be produced. Default value is 0.
-WWriteSTD
Write the resulting STD after each iteration.

Options applicable when building timed Event Structures from a Trace:
-Acpf
Capture all fail states in the trace when building the ES.
-Precon
Capture ES preconcurrency following the trace.
-AntiComplete
Builds an ES for the complete trace, then perform timing analysis, etc. This may result interesting is some cases, however the fail removal is very local and thus it is not suitable for hard verification processes.
-AfailGuided
Uses the failure transitions as a source for heuristics which try to reduce the size of the ESs and focus more locally around the particular failure being analyzed.
-AfilterTedges
Tries to filter (remove) those timing edges which do not actually remove the fail from all the failure states of the trace. This is intended to reduce the size of the resulting timed ESs such that all the atoms that appear in it, are related to the fail being analyzed.
Options applicable when building Event Structures:

-\texttt{-Elnp} Use local nodal points when possible.
-\texttt{-Enotgsp} Do not use global nodal points information. If this option is selected, \texttt{-Elnp} is selected automatically, otherwise the entry condition of the enabling compatible composition would not work in some cases.
-\texttt{-EenablingsN} Sets the level of detail to take into account in order to put non-enabling information to the GRC created from an ES. \texttt{N=1} means to use the minimum of information, i.e. a fast algorithm but with the need of more extra encoding variables. \texttt{N=2} means to use information from the trace used to build the ES (if any), i.e. a more complex algorithm but with almost no extra encoding. Default value is \texttt{N=2}.

Timing analysis options:

-\texttt{-EapproxTA} Use the approximate algorithm for timing analysis.
-\texttt{-EcheckConc} Indicates whether not to consider only timing arcs between concurrently firable vertexes (if set), but just consider any possible timing arc (if not set). By default all possible timing arcs are considered.
REFERENCES


215


REFERENCES


REFERENCES


[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using


[Huf54] D.A. Huffman. The synthesis of sequential switching circuits. J.Franklin Institute,


LOTOS, A formal description technique based on the temporal ordering of observational
behaviour, ISO 8807, February 1989.


[KB02] H. Kim, P.A. Beerel, and K. Stevens. Relative timing based verification of timed
circuits and systems. In Proceedings International Symposium on Advanced Research in


[KE96] A. Kovalyov and J. Esparza. A polynomial algorithm to compute the concurrency relation
of free-choice signal transition graphs. In Proceedings of the International Workshop on
Discrete Event Systems, pages 1–6, August 1996.

[KN02] R. Kaivola and N. Narasimhan. Formal verification of the Pentium 4 Floating-Point

[Koy90] R. Koymans. Specifying real-time properties with metric temporal logic. In LNCS,

Programs. In Workshop on Linear Time, Branching Time and Partial Order in Logics and


In Proceedings International Symposium on Advanced Research in Asynchronous


REFERENCES


# GLOSSARY OF SYMBOLS

## Transition systems

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle S, \Sigma, T, s_0 \rangle)</td>
<td>2.1</td>
<td>Transition system (TS)</td>
<td>18</td>
</tr>
<tr>
<td>(A)</td>
<td>2.1</td>
<td>Transition system</td>
<td>18</td>
</tr>
<tr>
<td>(S)</td>
<td>2.1</td>
<td>Set of states</td>
<td>18</td>
</tr>
<tr>
<td>(\Sigma)</td>
<td>2.1</td>
<td>Alphabet of events</td>
<td>18</td>
</tr>
<tr>
<td>(T)</td>
<td>2.1</td>
<td>Transition relation</td>
<td>18</td>
</tr>
<tr>
<td>(s)</td>
<td>2.1</td>
<td>State</td>
<td>18</td>
</tr>
<tr>
<td>(s_0)</td>
<td>2.1</td>
<td>Initial state</td>
<td>18</td>
</tr>
<tr>
<td>(e)</td>
<td>2.1</td>
<td>Event</td>
<td>18</td>
</tr>
<tr>
<td>(s \xrightarrow{e} s')</td>
<td>2.1</td>
<td>Transition</td>
<td>18</td>
</tr>
<tr>
<td>(\langle s, e, s' \rangle)</td>
<td>2.1</td>
<td>Transition</td>
<td>18</td>
</tr>
<tr>
<td>(E(s))</td>
<td>2.1</td>
<td>Events enabled at (s)</td>
<td>18</td>
</tr>
<tr>
<td>(FR(e))</td>
<td>2.1</td>
<td>Firing region of (e)</td>
<td>18</td>
</tr>
<tr>
<td>(\rho)</td>
<td>2.2</td>
<td>Run</td>
<td>19</td>
</tr>
<tr>
<td>(s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \ldots)</td>
<td>2.2</td>
<td>Run</td>
<td>19</td>
</tr>
<tr>
<td>(s_i \in \rho)</td>
<td>Notation</td>
<td>State of a run</td>
<td>19</td>
</tr>
<tr>
<td>(s_i \xrightarrow{e_i} s_{i+1} \in \rho)</td>
<td>Notation</td>
<td>Transition of a run</td>
<td>19</td>
</tr>
<tr>
<td>FirstEnabled((\rho, s_i, e))</td>
<td>2.4</td>
<td>Enabling interval of (e) in (\rho)</td>
<td>20</td>
</tr>
<tr>
<td>(T^r)</td>
<td>Notation</td>
<td>Reachability relation, transitive closure of (T)</td>
<td>20</td>
</tr>
<tr>
<td>(s \xrightarrow{\rho} s')</td>
<td>Notation</td>
<td>(s') is reachable by (\rho) from (s)</td>
<td>20</td>
</tr>
<tr>
<td>(s \xrightarrow{s'})</td>
<td>Notation</td>
<td>(s') is reachable from (s)</td>
<td>20</td>
</tr>
<tr>
<td>Reach((s', T))</td>
<td>2.5</td>
<td>Reachable states from (s)</td>
<td>20</td>
</tr>
<tr>
<td>(\langle A^-, \delta', \delta''\rangle)</td>
<td>2.6</td>
<td>Timed transition system (TTS)</td>
<td>22</td>
</tr>
<tr>
<td>(A^-)</td>
<td>2.6</td>
<td>Underlying transition system</td>
<td>22</td>
</tr>
<tr>
<td>(\delta(e))</td>
<td>2.6</td>
<td>Delay of (e)</td>
<td>22</td>
</tr>
<tr>
<td>(\delta'(e))</td>
<td>2.6</td>
<td>Minimum delay bound of (e)</td>
<td>22</td>
</tr>
<tr>
<td>(\delta''(e))</td>
<td>2.6</td>
<td>Maximum delay bound of (e)</td>
<td>22</td>
</tr>
<tr>
<td>([d, D])</td>
<td>2.6</td>
<td>Min-max delay interval</td>
<td>22</td>
</tr>
</tbody>
</table>
### Glossary of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s, \tau))</td>
<td>2.7</td>
<td>Timed state</td>
<td>22</td>
</tr>
<tr>
<td>(\tau)</td>
<td>2.7</td>
<td>Time stamp</td>
<td>22</td>
</tr>
<tr>
<td>(\langle S, \Sigma, T, s_0, EnR\rangle)</td>
<td>2.8</td>
<td>Lazy transition system (LzTS)</td>
<td>26</td>
</tr>
<tr>
<td>EnR(e)</td>
<td>2.8</td>
<td>Enabling region of e</td>
<td>26</td>
</tr>
</tbody>
</table>

### Petri nets

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle P, T, F, M_0\rangle)</td>
<td>2.10</td>
<td>Petri net (PN)</td>
<td>28</td>
</tr>
<tr>
<td>N</td>
<td>2.10</td>
<td>Petri net</td>
<td>28</td>
</tr>
<tr>
<td>P</td>
<td>2.10</td>
<td>Set of places</td>
<td>28</td>
</tr>
<tr>
<td>T</td>
<td>2.10</td>
<td>Set of transitions</td>
<td>28</td>
</tr>
<tr>
<td>F</td>
<td>2.10</td>
<td>Flow function</td>
<td>28</td>
</tr>
<tr>
<td>M</td>
<td>2.10</td>
<td>Marking</td>
<td>28</td>
</tr>
<tr>
<td>M_0</td>
<td>2.10</td>
<td>Initial marking</td>
<td>28</td>
</tr>
<tr>
<td>p</td>
<td>2.10</td>
<td>Place</td>
<td>28</td>
</tr>
<tr>
<td>t</td>
<td>2.10</td>
<td>Transition</td>
<td>28</td>
</tr>
<tr>
<td>((p, t))</td>
<td>Notation</td>
<td>Flow relation</td>
<td>28</td>
</tr>
<tr>
<td>((t, p))</td>
<td>Notation</td>
<td>Flow relation</td>
<td>28</td>
</tr>
<tr>
<td>*x</td>
<td>Notation</td>
<td>Pre-set of a node</td>
<td>28</td>
</tr>
<tr>
<td>x^*</td>
<td>Notation</td>
<td>Post-set of a node</td>
<td>28</td>
</tr>
<tr>
<td>M(p)</td>
<td>2.11</td>
<td>Number of tokens in p at M</td>
<td>29</td>
</tr>
<tr>
<td>M(t)</td>
<td>2.11</td>
<td>t is enabled at M</td>
<td>29</td>
</tr>
<tr>
<td>([t])</td>
<td>2.11</td>
<td>Set of markings where t is enabled</td>
<td>29</td>
</tr>
<tr>
<td>M(t)M'</td>
<td>2.11</td>
<td>M' is reachable by firing t from M</td>
<td>29</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>2.12</td>
<td>Firing sequence</td>
<td>30</td>
</tr>
<tr>
<td>t_1t_2\ldots</td>
<td>2.12</td>
<td>Firing sequence</td>
<td>30</td>
</tr>
<tr>
<td>M(\sigma)M'</td>
<td>2.12</td>
<td>M' is reachable by (\sigma) from M</td>
<td>30</td>
</tr>
<tr>
<td>([M_0])</td>
<td>2.12</td>
<td>Set of markings reachable from (M_0)</td>
<td>30</td>
</tr>
<tr>
<td>(\langle[M_o], E\rangle)</td>
<td>2.12</td>
<td>Reachability graph (RG)</td>
<td>30</td>
</tr>
<tr>
<td>E</td>
<td>2.12</td>
<td>Set of arcs</td>
<td>30</td>
</tr>
<tr>
<td>((M, t, M'))</td>
<td>2.12</td>
<td>Arc</td>
<td>30</td>
</tr>
<tr>
<td>RG(N)</td>
<td>Notation</td>
<td>Reachability graph of (N)</td>
<td>30</td>
</tr>
<tr>
<td>(\langle N, \Sigma, \Lambda\rangle)</td>
<td>2.13</td>
<td>Labeled Petri net</td>
<td>32</td>
</tr>
<tr>
<td>(\Sigma)</td>
<td>2.13</td>
<td>Alphabet of symbols</td>
<td>32</td>
</tr>
<tr>
<td>(\Lambda)</td>
<td>2.13</td>
<td>Labeling function</td>
<td>32</td>
</tr>
<tr>
<td>(\epsilon)</td>
<td>2.13</td>
<td>“Silent” symbol</td>
<td>32</td>
</tr>
<tr>
<td>(\Sigma_I, \Sigma_O, \Sigma_H)</td>
<td>2.14</td>
<td>Alphabet of (input, output, internal) signals</td>
<td>32</td>
</tr>
</tbody>
</table>
### Glossary of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>Notation</td>
<td>Set of clocks</td>
<td>39</td>
</tr>
<tr>
<td>$x$</td>
<td>Notation</td>
<td>Clock</td>
<td>39</td>
</tr>
<tr>
<td>$\Phi(X)$</td>
<td>Notation</td>
<td>Set of clock constraints</td>
<td>39</td>
</tr>
<tr>
<td>$\varphi$</td>
<td>Notation</td>
<td>Clock constraints</td>
<td>39</td>
</tr>
<tr>
<td>$(\Sigma, S, S_o, X, I, T)$</td>
<td>3.1</td>
<td>Timed automata</td>
<td>40</td>
</tr>
<tr>
<td>$A$</td>
<td>3.1</td>
<td>Timed automata</td>
<td>40</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>3.1</td>
<td>Alphabet</td>
<td>40</td>
</tr>
<tr>
<td>$S$</td>
<td>3.1</td>
<td>Set of locations</td>
<td>40</td>
</tr>
<tr>
<td>$S_o$</td>
<td>3.1</td>
<td>Set of initial locations</td>
<td>40</td>
</tr>
<tr>
<td>$I$</td>
<td>3.1</td>
<td>Location invariant</td>
<td>40</td>
</tr>
<tr>
<td>$T$</td>
<td>3.1</td>
<td>Set of transitions</td>
<td>40</td>
</tr>
<tr>
<td>$s$</td>
<td>3.1</td>
<td>Location</td>
<td>40</td>
</tr>
<tr>
<td>$a$</td>
<td>3.1</td>
<td>Symbol</td>
<td>40</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>3.1</td>
<td>Set of clocks reset by a transition</td>
<td>40</td>
</tr>
<tr>
<td>$(s, a, \varphi, \lambda, s')$</td>
<td>3.1</td>
<td>Transition from $s$ to $s'$</td>
<td>40</td>
</tr>
<tr>
<td>$T(A)$</td>
<td>Notation</td>
<td>Transition system associated to $A$</td>
<td>41</td>
</tr>
<tr>
<td>$v(x)$</td>
<td>Notation</td>
<td>Valuation of a clock</td>
<td>41</td>
</tr>
<tr>
<td>$(s, v)$</td>
<td>Notation</td>
<td>Configuration</td>
<td>41</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Notation</td>
<td>Time increment</td>
<td>41</td>
</tr>
<tr>
<td>$(s, v) \rightarrow_{\delta}(s, v')$</td>
<td>Notation</td>
<td>Delay transition</td>
<td>41</td>
</tr>
<tr>
<td>$(s, v) \rightarrow_{a}(s, v')$</td>
<td>Notation</td>
<td>Action transition</td>
<td>41</td>
</tr>
<tr>
<td>$c_x$</td>
<td>Notation</td>
<td>Maximal constant a clock is compared to</td>
<td>46</td>
</tr>
<tr>
<td>$[v(x)]$</td>
<td>Notation</td>
<td>Integral part of a clock valuation</td>
<td>46</td>
</tr>
<tr>
<td>$fr(v(x))$</td>
<td>Notation</td>
<td>Fractional part of a clock valuation</td>
<td>46</td>
</tr>
<tr>
<td>$\cong$</td>
<td>Notation</td>
<td>Equivalence of clock valuations</td>
<td>46</td>
</tr>
<tr>
<td>$R(A)$</td>
<td>Notation</td>
<td>Region automaton of $A$</td>
<td>48</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Notation</td>
<td>Clock region</td>
<td>48</td>
</tr>
<tr>
<td>$(s, \alpha)$</td>
<td>Notation</td>
<td>State</td>
<td>48</td>
</tr>
<tr>
<td>$Z(A)$</td>
<td>Notation</td>
<td>Zone automaton of $A$</td>
<td>48</td>
</tr>
<tr>
<td>$D$</td>
<td>Notation</td>
<td>Difference-bound matrix</td>
<td>49</td>
</tr>
<tr>
<td>$D_{ij}$</td>
<td>Notation</td>
<td>Upper bound on the difference of two clocks</td>
<td>49</td>
</tr>
<tr>
<td>$\mathbb{D}$</td>
<td>Notation</td>
<td>Bounds domain</td>
<td>49</td>
</tr>
</tbody>
</table>
### Trace semantics

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta$</td>
<td>4.1</td>
<td>Trace</td>
<td>62</td>
</tr>
<tr>
<td>$E_1 \xrightarrow{e_1} E_2 \xrightarrow{e_2} \ldots$</td>
<td>4.1</td>
<td>Trace</td>
<td>62</td>
</tr>
<tr>
<td>$E_i \xrightarrow{e_i} E_{i+1}$</td>
<td>4.1</td>
<td>Transition by firing $e_i$</td>
<td>62</td>
</tr>
<tr>
<td>$E_i$</td>
<td>4.1</td>
<td>Set of events enabled when $e_i$ fires</td>
<td>62</td>
</tr>
<tr>
<td>$D_i$</td>
<td>4.4</td>
<td>Set of events disabled when $e_i$ fires</td>
<td>66</td>
</tr>
<tr>
<td>$d$</td>
<td>4.4</td>
<td>Disabled event</td>
<td>66</td>
</tr>
<tr>
<td>$e_i \mathbf{dis} d$</td>
<td>4.4</td>
<td>$d$ is disabled by the firing of $e_i$</td>
<td>66</td>
</tr>
<tr>
<td>$D(\theta)$</td>
<td>4.4</td>
<td>Set of events disabled along $\theta$</td>
<td>66</td>
</tr>
<tr>
<td>$\theta_\rho$</td>
<td>4.2</td>
<td>Trace defined by a run</td>
<td>62</td>
</tr>
<tr>
<td>$\mathcal{L}(A)$</td>
<td>4.3</td>
<td>Language of a transition system</td>
<td>63</td>
</tr>
<tr>
<td>$\theta_i$</td>
<td>4.5</td>
<td>Fragment of a trace</td>
<td>68</td>
</tr>
<tr>
<td>map</td>
<td>4.5</td>
<td>Enabling-compatible mapping</td>
<td>68</td>
</tr>
</tbody>
</table>

### Event structures

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\Sigma, \prec, \triangleright)$</td>
<td>4.6</td>
<td>Causal event structure</td>
<td>71</td>
</tr>
<tr>
<td>$CS$</td>
<td>4.6</td>
<td>Causal event structure</td>
<td>71</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>4.6</td>
<td>Set of events</td>
<td>71</td>
</tr>
<tr>
<td>$\prec$</td>
<td>4.6</td>
<td>Causality relation</td>
<td>71</td>
</tr>
<tr>
<td>$\triangleright$</td>
<td>4.6</td>
<td>Conflict relation</td>
<td>71</td>
</tr>
<tr>
<td>$\triangleright\mu$</td>
<td>Notation</td>
<td>Disabling relation</td>
<td>71</td>
</tr>
<tr>
<td>$X$</td>
<td>Notation</td>
<td>Subset of events</td>
<td>72</td>
</tr>
<tr>
<td>$(\neg X)_{\prec}$</td>
<td>Notation</td>
<td>Events before $X$</td>
<td>72</td>
</tr>
<tr>
<td>$(X^{\rightarrow})_{\prec}$</td>
<td>Notation</td>
<td>Events after $X$</td>
<td>72</td>
</tr>
<tr>
<td>$(\downarrow X)_{\prec}$</td>
<td>Notation</td>
<td>Root events of $X$</td>
<td>72</td>
</tr>
<tr>
<td>$(X^{\downarrow})_{\prec}$</td>
<td>Notation</td>
<td>Sink events of $X$</td>
<td>72</td>
</tr>
<tr>
<td>$(\downarrow X)_{\prec}$</td>
<td>Notation</td>
<td>Left-closure of $X$</td>
<td>72</td>
</tr>
<tr>
<td>$\omega$</td>
<td>4.7</td>
<td>Word</td>
<td>72</td>
</tr>
<tr>
<td>$e_1 \cdots e_n$</td>
<td>4.7</td>
<td>Word</td>
<td>72</td>
</tr>
<tr>
<td>$\omega_i$</td>
<td>4.7</td>
<td>$i$-th prefix of a word</td>
<td>72</td>
</tr>
<tr>
<td>$\omega_0$</td>
<td>4.7</td>
<td>Empty prefix</td>
<td>72</td>
</tr>
<tr>
<td>$\mathcal{E}(\omega_i)$</td>
<td>4.8</td>
<td>Events enabled by a prefix</td>
<td>72</td>
</tr>
<tr>
<td>$\mathcal{D}(\omega_i)$</td>
<td>4.8</td>
<td>Events disabled by a prefix</td>
<td>72</td>
</tr>
<tr>
<td>$\mathcal{D}(\omega_i)$</td>
<td>Notation</td>
<td>Events disabled by a word</td>
<td>73</td>
</tr>
<tr>
<td>$\theta_\omega$</td>
<td>4.9</td>
<td>Trace generated by word $\omega$</td>
<td>73</td>
</tr>
<tr>
<td>$CS$</td>
<td>4.10</td>
<td>Causal event structure generated from trace $\theta$</td>
<td>73</td>
</tr>
</tbody>
</table>
Glossary of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle \Sigma, \prec', b \rangle )</td>
<td>4.11</td>
<td>Lazy causal event structure</td>
<td>76</td>
</tr>
<tr>
<td>( \prec' )</td>
<td>4.11</td>
<td>Set of lazy relations</td>
<td>76</td>
</tr>
<tr>
<td>( C )</td>
<td>4.12</td>
<td>Configuration</td>
<td>77</td>
</tr>
<tr>
<td>( E(C) )</td>
<td>4.12</td>
<td>Set of events enabled in a configuration</td>
<td>77</td>
</tr>
<tr>
<td>( C )</td>
<td>Notation</td>
<td>Set of reachable configurations</td>
<td>77</td>
</tr>
<tr>
<td>( \bot )</td>
<td>Notation</td>
<td>Final configuration</td>
<td>79</td>
</tr>
<tr>
<td>( \langle \Sigma, \prec, v \rangle )</td>
<td>4.13</td>
<td>Graph of reachable configurations</td>
<td>78</td>
</tr>
<tr>
<td>( \text{EnR}(e) )</td>
<td>Notation</td>
<td>Enabling region of ( e )</td>
<td>78</td>
</tr>
<tr>
<td>( \text{FR}(e) )</td>
<td>Notation</td>
<td>Firing region of ( e )</td>
<td>78</td>
</tr>
<tr>
<td>( C_1 \xrightarrow{e} C_2 )</td>
<td>Notation</td>
<td>Transition between configurations</td>
<td>78</td>
</tr>
<tr>
<td>( (C_1, e, C_2) )</td>
<td>Notation</td>
<td>Transition between configurations</td>
<td>78</td>
</tr>
</tbody>
</table>

**Timing analysis**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_{\text{sep}}(e_1, e_2) )</td>
<td>Notation</td>
<td>Maximal separation time of two events</td>
<td>184</td>
</tr>
<tr>
<td>( \text{ft}(e_1) )</td>
<td>Notation</td>
<td>Firing time of an event</td>
<td>184</td>
</tr>
<tr>
<td>( \Delta )</td>
<td>Notation</td>
<td>Strongest bound for ( S_{\text{sep}}(e_1, e_2) )</td>
<td>186</td>
</tr>
<tr>
<td>([d, D])</td>
<td>Notation</td>
<td>Delay interval</td>
<td>186</td>
</tr>
<tr>
<td>( m(e_k) )</td>
<td>Notation</td>
<td>( m )-value of ( e_k )</td>
<td>186</td>
</tr>
<tr>
<td>( M(e_k) )</td>
<td>Notation</td>
<td>( M )-value of ( e_k )</td>
<td>187</td>
</tr>
<tr>
<td>( h )</td>
<td>Notation</td>
<td>Path</td>
<td>186</td>
</tr>
<tr>
<td>( d(h) )</td>
<td>Notation</td>
<td>Sum of minimum delay bounds along a path</td>
<td>186</td>
</tr>
<tr>
<td>( e_k \xrightarrow{h} e_j )</td>
<td>Notation</td>
<td>A path ( h ) exists between two events</td>
<td>186</td>
</tr>
<tr>
<td>( e_k \not\xrightarrow{e_j} )</td>
<td>Notation</td>
<td>No path exists between two events</td>
<td>186</td>
</tr>
<tr>
<td>( e_1 \xrightarrow{[d, D]} e_2 )</td>
<td>Notation</td>
<td>Edge between two events</td>
<td>187</td>
</tr>
</tbody>
</table>

Conventional symbols regarding sets, boolean functions, temporal logics, etc. is also used along the document. However, it is not included here since it corresponds to well-known notation.