INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THESE

pour obtenir le grade de

DOCTEUR DE L'INP GRENOBLE

Spécialité: Micro et Nano Electronique

Préparée au laboratoire **VERIMAG**

dans le cadre de l'Ecole Doctorale

Electronique, **E**lectrotechnique, **A**utomatique & **T**raitement du **S**ignal

présentée et soutenue publiquement

par

Ramzi BEN SALAH

le 12 octobre 2007

SUR L'ANALYSE TEMPORISÉE DE SYSTÈMES DE GRANDE TAILLE

JURY

Mme. Florence Maraninchi	: Président
M. Amir Pnueli	: Rapporteur
M. Ahmed Bouajjani	: Rapporteur
M. Edmund M. Clarke	: Examinateur
M. Klaus Winkelmann	: Examinateur
M. Ziyad Hanna	: Examinateur
M. Oded Maler	: Directeur de thèse
M. Marius Bozga	: Co-encadreur

ON TIMING ANALYSIS OF LARGE SYSTEMS

by

Ramzi BEN SALAH

12 October 2007

"I think the right way, of course, is to say that what we have to look at is the whole structural interconnection of the thing; that all the sciences, and not just the sciences but all the efforts of intellectual kinds."

[Richard Feynman]

Abstract

This thesis is part of an effort to advance the algorithmic analysis of timed systems from a proof-of-concept phase toward industrial usage, by improving the performance of timing verification algorithms. Models and tools based on timed automata can be used for timing and performance analysis of complex systems, hardware and software alike, but the problem of scalability prevents the adaptation of these techniques by industry.

The two major contributions of the thesis are an improved algorithm for symbolic reachability computation which reduces significantly the state exposition, and a compositional divide-and-conquer methodology based on conservative approximation of timed components.

With standard reachability computation, a discrete state reached by different orderings of the same set of transitions, will lead often to different symbolic states, contributing to additional state explosion. We provide an algorithm which avoids this problem, taking advantage of the fact that the union of all the zones of these symbolic states is convex and hence they can be merged into one zone. This algorithm has been implemented and tested experimentally.

The second contribution of this thesis benefits from the modularity of component based systems to build for each component a model which approximates its timed interface behavior. These models could be reused for the modeling of the interface of component reusing this last component. Thus the modeling process is done hierarchically starting from the leaves and abstracting at each level the internal complexity of every component, focusing only on their external behavior which is sufficient for their reuse.

Two abstraction techniques are developed in the thesis, one specialized for systems that have to respond to a finite number of events, and one more general for systems working in arbitrary general environment. Both techniques are based on introducing auxiliary input-related clocks, performing reachability computation and then projecting the timing constraints of the obtained automaton on the auxiliary clocks. As a result we obtain an automaton with fewer clocks whose quantitative semantics is an over-approximation but its qualitative semantics is exact. Further steps of hiding and minimization lead to a small automaton for the component.

Acknowledgments

I was very fortunate to work under the supervision of Oded Maler, which has combined together guidance, advice, encouragement and support. I thank him for his confidence in me, which allowed me to benefit fully from this experience. I learned a lot from working with him, from his optimism and his positive attitude.

I also had an enormous pleasure to work with Marius Bozga. I thank him for the time he has devoted to our fruitful discussions. I would like to use this opportunity express my great respect for his professional and human qualities.

I want to thank the members of the jury: Prof. Florence Maranninchi, of the National Polytechnic Institute of Grenoble for chairing the jury, Prof. Amir Pnueli of New York University and Prof. Ahmed Bouajjani of Paris 7 University for the interest they have given to this work; Prof. Edmund Clarke of Carnegie Mellon University of Pittsburgh, and Mr. Klaus Winkelmann of OneSpin Solution in Munich/Germany, for the honor they gave me by participating in my jury.

In Verimag I have had the opportunity to grow and interact with some of the most talented and accomplished researchers in the field. In particular I had the privilege to interact with Paul Caspi, Eugene Asarin, Stavros Tripakis and Sergio Yovine. Thanks for their advices.

I also want to thank Goran Frehse, Dejan Nickovic and Viktor Schuppan for the effort they have invested reading and commenting my thesis. Thanks for the rest of the members of Verimag who were able to create the ideal atmosphere and working conditions.

Thanks to Ananda, Olivier and Hung for these evenings of exotic cuisines. Thanks to all my friends inside and outside the laboratory that made my stay in Grenoble a real human experience.

Finally, a big thank for my family, who despite the distance, were able to find a way to support me. But thinking of my family, my brothers and sisters, I cannot forget other people whose friendship is the better gift that these years gave me; Sometimes words cannot convey what we want to express.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is part of an effort to make formal verification of *timed* models of hardware and software systems applicable to industrial size problems. This effort is inspired by the developments that took place in the last 30 years in the domain of *algorithmic verification* of discrete untimed models. Hence we open this chapter with a very quick introduction to formal verification and its role in the design process, followed by the major motivations for extending it to timed models and a high-level survey of the development of verification tools for timed automata.

1.1 Formal Verification

Hardware and software systems are inevitably growing in scale and functionality leading to very complex systems. In such systems the likelihood of subtle errors is increasing substantially. To avoid undetected bugs which can be extremely expensive[1] to repair, verification and validation procedures have taken a central place in the design flow of contemporary systems.

The most prevalent approach to design verification is simulation. Within this approach one analyzes the response of a *model* of the system to a series of stimuli, that is, possible scenarios of input events. Although this process can detect bugs, it cannot guarantee correctness because the number of such scenarios is prohibitively large, if not infinite. Much of the work in this area is concerned with finding a representative sample of the input stimuli which "covers", in some sense, all the interesting cases. The correct functioning of the system in the presence of these stimuli can increase our confidence in its overall correctness but the number of such test cases can be extremely large. In many application domains verification has become the major bottleneck in design flow and may represent up to 80% of the overall design cost. With marketing pressures demanding shorter and shorter design cycles, for larger and more complex

[1]Especially in hardware if the bugs are discovered after fabrication.

products, it has been recognized that traditional verification practices, which are mostly test based, are not sufficient and complementary methods are needed to meet the design validation challenge. Formal verification suggests an alternative approach to tackle these problems.

Formal methods refer to mathematically rigorous techniques to establish design correctness. The correctness of a system (design, implementation) is defined relative to its *specification* which reflects our expectations from the observable behavior of the system. Mathematically speaking, a formal specification should unambiguously distinguish correct and incorrect system behaviors. Formal logic is considered as the universal formalism for formalization. However, since we are interested in the *dynamic* behavior of systems, *temporal logics* which allow us to express more naturally the succession of events in time, turned out, following the suggestion of Pnueli [Pnu77, Pnu81], to be a more natural and popular specification formalism [MP91, EF06].

The other ingredient of the verification process (formal and informal verification alike) is a mathematical *model* of the system dynamics. Such a model should be, in terms of observables, at least as detailed as the specification so that the behaviors it generates can be checked against it. In addition, it should be sufficiently detailed to include the features that are essential for correctness. A model not satisfying this requirements will be at the risk of generating spurious counterexamples, namely, exhibiting incorrect behaviors which are impossible in a more detailed model of the system. On the other hand, the model should be as abstract as possible to make verification tractable. To give some examples, functional correctness of Boolean gate-level models of circuits can be done while abstracting away from electrical properties of transistors, while correctness of communication protocols is performed on models that focus on the *control* structure and ignore the contents of the *data* which is transmitted.

The mathematical models used in both software and hardware verification are essentially *discrete transition systems (automata)*. In software such models can be derived from the actual code by abstracting the values of variables and complex data structures. In hardware, such models are often based on models already used by CAD tools for simulation and synthesis. Such models exist at various levels of abstractions ranging from silicon and transistor models, via Boolean networks up to high-level functional models. Formal verification of hardware is traditionally done at the RTL (register transfer level) and gate levels which are modeled naturally as finite-state automata. Recently attempts have been made to extend formal verification toward higher levels (SystemC) as well as to lower levels that involve electricity and time delays. As we shall see later, this work is part of the latter effort.

Once we have the specifications and an appropriate model of the system, we can start the verification process whose aim is to provide a proof that the system satisfies the specification, that is, all the behaviors it may exhibit are correct. Two basic proof methodologies can be used to establish this fact, one is *deductive*, that is, using *theorem proving* and one is *algorithmic*, best known as *model-checking*.

The deductive approach consists of using a set of *axioms* and *inference rules* to prove that the system satisfies its specification. Such proofs are typically long and tedious and theorem proving tools may help a lot in the book keeping associated with the process. However, there is no way to replace the human in the loop for suggesting auxiliary theorems and proof strategies. Hence the process is only partially automated and requires highly-skilled professionals with a lot of patience and understanding of the design. A deductive framework for linear-time temporal logic is presented in [MP95b].

The algorithmic approach, the source of inspiration for this thesis, uses graph algorithms to explore all the paths in a *finite-state* model of a system. The behaviors that correspond to these paths are checked against the specifications. The idea of model checking for temporal properties was introduced in the seminal papers of Clarke and Emerson [CE81], and Queille and Sifakis [QS82] for the branching-time temporal logic CTL and was later extended to the linear-time LTL in [LP85]. In automata-theoretic terms this process can be viewed as checking containment between two ω-regular languages, one generated by the system and the other derived from the specification [VW86]. Other approaches to verification used automata also as a specification formalism [HK90, Kur94]. There are many variants of model checking problems that differ in the choice of specification formalisms, the conformance relation used to compare implementation and specification (trace inclusion, refinement ordering, observational equivalence [CPS93]), etc. More on the topic can be found in books such as [CGP99, BBF$^+$01].

Unlike theorem proving, model checking is, in principle, fully automatic: the user provides a high-level representation of the model and the specifications to be checked, and the model checker can decide correctness without further intervention. If it turns out that the specification is not satisfied by the model, a *counterexample* is produced. Counterexamples represent, perhaps, the most valuable outcome of model checking as they may capture subtle errors in the design that could pass unnoticed by non-exhaustive simulation.

The main limitation of model-checking is that, in general, it requires exploring the complete state graph of the model, a graph whose size increases exponentially with the number of system components. This problem, known as the *state explosion problem*, has always been considered as a major barrier which, for a long time, has restricted the applicability of this methodology to verify rudimentary sized systems.

Dealing with the state explosion is still one of the major research concerns in algorithmic verification. Several breakthroughs took place over the years, including *symbolic* model checking [BCM$^+$90, McM92b] which, by representing sets of states via Boolean formulae, provided for the analysis of huge transition graphs which could not be explicitly enumerated. Another performance improvement, known as *bounded model checking* [BCC$^+$03], takes advantage of the

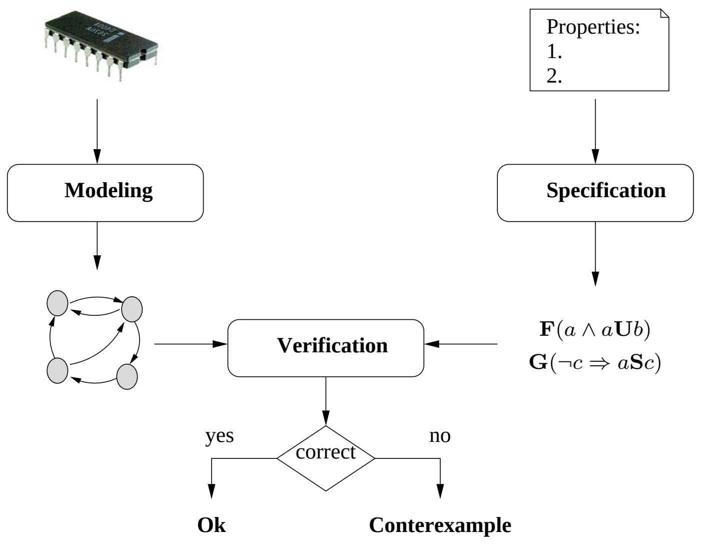

Properties:
1.
2.
Modeling
Specification
Verification
$\mathbf{F}(a \wedge a\mathbf{U}b)$
$\mathbf{G}(\neg c \Rightarrow a\mathbf{S}c)$
yes
correct
no
Ok
Conterexample

Figure 1.1: Model Checking.

capabilities of modern SAT solvers to check whether all system behaviors up to certain length are correct. Although these techniques cannot change the inherent computational hardness of the verification problem, they succeeded in increasing the size of practically verifiable systems by several orders of magnitude and moved formal verification from theory to practice.

Beyond the contribution of these and other techniques, the only scalable way to cope with state explosion is to use the same principles used by the designers of complex systems, that is *compositional* (modular, hierarchical) reasoning. The underlying observation is that a designer of a component of a complex system does not maintain in his head the huge global transition graph associated with the whole system. An individual component is designed to work properly based on a small abstract model of its environment (the rest of the system) which focuses on those aspects of the other parts of the system which are relevant for the interaction with the component. Compositional proof methodologies such as *assume-guarantee reasoning* are subject to ongoing research, and like in theorem proving, a major issue here is the *automatic* generation of abstract models of components. Much of this thesis will be concerned with automatic abstraction of timed systems.

1.2 Timed Systems

The level of abstraction of discrete transition systems is very useful for proving functional correctness but it is not sufficiently detailed for certain purposes. We will illustrate this claim using two classes of applications, one from hardware and one from software.

Delays in Circuits: Suppose we realize a synchronous sequential machine using gates and latches. The discrete abstraction of the system as an automaton is valid if the time it takes for the gates to compute the next-state function is always smaller than the period of the clock. To see whether this relation between period and maximal delay holds or to compute the maximal frequency in which the circuit can work, we need to refine the gate model from an *instantaneous* Boolean function into a model where the output reacts to a change in its input within some propagation *delay*. In industrial practice, timing analysis and functional verification are often done separately: first, the maximal frequency is determined by approximating the maximal delay of the circuit by the sum of delays along the longest path from input to output, regardless of the logic. This process is called *static timing analysis* to distinguish it from simulation which considered "dynamic". After that, functional verification is pursued on the untimed model. While this separate treatment of logic and timing is the only practical way to cope with the complexity of large circuits such as processors, it is clear that the full picture on the systems behavior is obtained by combining the logic and timing into one model. For example, it might be the case that due to the logical structure of the circuit, the longest path is a "false path" and cannot be exercised by any input scenario. A more detailed model which interleaves discrete changes with propagation of delays will be more faithful to the dynamics of the circuit. Such models can also be useful for analyzing asynchronous circuit that operate without a central clock and to which static timing analysis is difficult to apply.

Performance of Real-time Software: The terms *real-time* or *embedded* software are often used to denote systems that interact with the external physical world, for example control of airplanes, cars and other complex systems. In such interactions, the importance of timely response cannot be underestimated, and the system may become useless, or even dangerous, if it produces the right result in the wrong time. Less safety critical, but economically no less important are systems that process and transfer streams of data of various types (communication, music, video). In such systems the so-called *quality of service* can make the difference between a useful and a useless system.

Consider a *scheduler* which receives requests for some common resources from different *clients* and allocates the resources according to some rule. Using classical verification with automata, one can prove that the scheduler does not commit errors such as allocating the same resource simultaneously to two or more clients (mutual exclusion) or entering a state where no client can progress (deadlock avoidance). There are, however other properties of interest which are not captured by the discrete model. Suppose the clients are real-time programs that need to complete their execution within some *deadline*. In order to check whether a given scheduler guarantees that every client will be serviced within its respective deadline, we need to introduce into the model *quantitative* timing information such as the expected execution time of each type of client, the constraints on the time between successive arrivals of clients, etc. As in the case of circuits, incorporating this kind of information amounts to refining the way we model the act of executing a piece of code. In the traditional modeling with automata, such an act could be seen as a *instantaneous action*, while in a timed model it is split into a sequence consisting of: 1) the discrete event of starting the process; 2) passage of time during the execution; and 3) the discrete event of termination (see Figure 1.2). Only on the basis of this quantitative information one can check whether such a system meets its deadlines or infer other time-related performance measures such as throughput and latency.

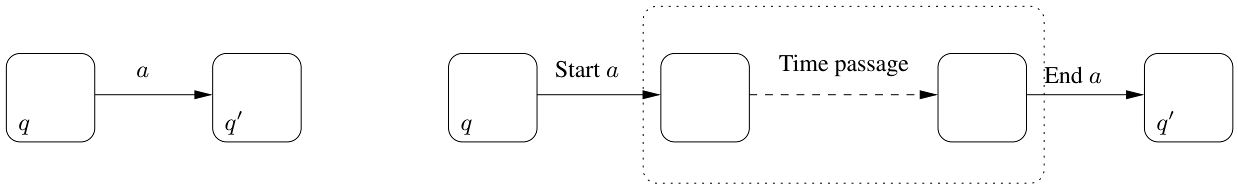


Figure 1.2: Refining a discrete transition into a process that takes time.

It is worth observing that like in static timing analysis, early work on real-time scheduling was based on separation of concerns. In the classical Liu and Layland model [LL73], the tasks are assumed to be unrelated to each other (except for requiring the same processor) and to arrive with a fixed period. Hence their schedulability could be computed cheaply according to an analytic condition on their deadlines, periods and execution times. However, it is commonly recognized that such techniques are not satisfactory for modern and complex systems that admit sporadic arrivals, more complex inter-dependence among tasks, combination of hard and soft constraints and multi-processors.

These two application domains show the need for models that combine *discrete events* with *time passage*, models that can capture the competition between parallel processes and the subtle interaction between logic and timing. Such models augment the discrete state variables of a system with additional variables that measure the elapse of time between events. Using these variables one can express the *duration* of processes by measuring the evolution of such variables. Various approaches have been proposed for extending transition systems with quantitative timing information, for example timed variants of Petri nets [Ram74, MF76, BD91, Rok94], timed discrete-event systems [BW94, CL06] and Max-Plus systems [BCOQ93]. The most fruitful approach to these problems came from the verification community which at the end of the 80s started exploring the extension of verification methodology toward real time systems, for example, [dBHdRR92].

Most of the current literature and tools in algorithmic timed verification is based on the *timed automaton* model, an automaton augmented with real-valued clock variables, first introduced formally in the seminal paper of Alur and Dill [AD90, AD94]. Other models proposed around the same time are the *timed transition systems* of Henzinger, Manna and Pnueli [HMP92] and a precursor of timed automata proposed by Dill [Dil89] based on counters, clocks that go down to zero. The results of [AD90] showed that the basic verification-related questions such as language and ω-language emptiness, are decidable in this model despite its infinite state space. The complexity of these problems was shown to be PSPACE-complete, at least as hard as untimed verification. It is interesting to note that the finite-quotient property underlying this decidability result has been discovered much earlier by Berthomieu and Menasche in the context of timed Petri nets [BM83] but did not have a similar impact at that time.

Since the publication of [AD90] several research groups worked on various theoretical aspects of timed automata and on building progressively more efficient verification tools based on this model. The decidability result of [AD90] was based on transforming the timed automaton into a finite quotient, also known as the *region graph*, on which discrete verification algorithms could be applied. This construction turned out to be impractical and further efforts focused on finding more efficient ways to perform verification. We mention some of these efforts without attempting to be exhaustive or chronological.

The tool KRONOS [DOTY95, Yov97] has been developed at Verimag under the supervision of Sifakis, starting with the theses of Yovine [Yov93], Olivero [Oli94] and Daws [Daw98]. It was initially based on the results reported at [HNSY94] for model-checking timed automata against TCTL formulae using a backward reachability algorithm, manipulating sets of clock values called *zones* which are defined by conjunctions of difference constraints among pairs of clock variables. Zones can be represented by *difference-bound matrices* (DBM), as has actually been suggested in [Dil89]. Around the same time a real time extension of the SPIN model checker [Hol97] has been developed by Tripakis using a zone-based forward reachability algorithm [TC96]. The tool OPENKRONOS has been developed in the thesis of Tripakis [Tri98] using on-the-fly forward reachability. Later, these and other ideas have been integrated into the IF toolbox [BSGS04].

The tool UPPAAL [LPY97] has been developed in collaboration between Yi from Uppsala and Larsen from Aalborg, starting with [YPD94, LPY95, BLL$^+$95] and being the topic of the thesis of Pettersson [Pet99]. Over the years, UPPAAL has been subject to continuous large investments in many aspects that make a tool usable, including a graphical user interface which is popular among end users, and many algorithmic and software engineering improvements, see for example [ABB$^+$01, BBD$^+$02] and the thesis of Behrmann [Beh03]. The verification engine of UPPAAL, like that of IF is based today on on-the-fly forward reachability on zones, to be described in the next section.

While we can say that the verification of timed systems has passed the proof-of-concept phase, there is still much to be done in fighting the state explosion problem, aggravated by the presence of the additional state variables, the clocks. Despite all the improvements made over the years, one may observe that a performance breakthrough, similar to symbolic model checking in untimed verification, has not yet taken place in the timed domain, and one cannot point out an application area where timed automata has proliferated into the daily practice. This thesis is part of an effort to change this situation, using the IF toolset [BSGS04] as an implementation medium.

1.3 Thesis Framework

The goal of this thesis is to develop methods that will allow us, eventually, to analyze systems modeled by timed automata with dozens and even hundreds of components, each with its own state variables and clocks. Since our major challenge is algorithmic we have chosen to focus on large *synthetic* examples rather than on industrial case studies. Most of our work uses networks of Boolean gates with delays according to the model proposed in [MP95a] due to the facility of generating large examples within this model. This is not meant to imply that circuit models at the gate level are the most appropriate application domain. As the reader will see, the approach developed in this thesis applies to a wide class of systems, including embedded software, where the future applications of these techniques are more likely to be found.

The major contributions of the thesis are the following:

1. A new reachability algorithm which exploits the fact that the union of all zones reached by different permutations of a set of local actions is convex. This result removes a particular contribution to the state explosion associated with timed automata, which is due to the fact that different interleavings of the same actions lead to different zones.

2. Development of a divide-and-conquer methodology for large networks of timed components, consisting of analyzing subsystems in isolation and then performing automatic abstraction on their timed automaton models, to yield small complexity approximate models. Such models can replace the more detailed models in a compositional reasoning framework.

3. Development of the major ingredient of this methodology, an algorithm for automatic abstraction of timed components which over-approximates their timed input-output behavior. The abstraction procedure involves some novel ideas for timed automata such as the use of *dynamic clocks* which are associated with those input events to which the system has not yet fully responded.

4. Implementation of all these ideas into a tool chain, based on IF. This includes translation from high-level circuit descriptions to timed automata, extension of the timed automaton model and verification algorithms to deal with dynamic input clocks which are created and discarded according to the propagation of their corresponding event in the circuit, and procedures for abstraction and minimization.

The rest of the thesis is organized as follows: Chapter 2 is a a quick introduction to timed automata and their verification. In Chapter 3 we present the new reachability algorithm that uses the convexity result for interleaving; Chapter 4 discusses circuit timing analysis and presents the timed circuit model used in the thesis and shows how we model such circuits using timed automata. In Chapter 5 we present the first ideas underlying our abstraction technique, starting with closed systems, that is, systems whose inputs change at most once at the beginning. The more general abstraction technique for open reactive systems is presented in detail in Chapter 6 together with some preliminary experimental results. Chapter 7 gives a relatively-short description of the implementation effort associated with these developments. Conclusion and suggestions for further research close this thesis.

Part I

Timed Automata and Interleaving

Chapter 2

Timed Automata

This chapter gives an introduction to the basics of timed automata to be used in the rest of the dissertation. Section 2.1 introduces timed automata informally through an example. Section 2.2 defines formally the syntax and the semantics of timed automata. The infinite transition system that a timed automaton specifies must be reduced, via some equivalence relation, into a finite graph in order to be analyzed. Section 2.3.1 focuses on the commonly-used verification approach which constructs a graph called the *reachability graph* or *simulation graph* which captures all the possible qualitative behaviors that the automaton may exhibit. Such a graph can be converted back into a timed automaton with special features described in Section 2.3.2.

Timed automata are introduced in this chapter as in the original papers, that is, with a "flattened" state space admitting no product structure. This presentation style is useful for introducing zone-based reachability techniques, but the reader should keep in mind that we want to treat large *products* of timed automata in which states are encoded using explicit discrete variables. We also adhere in this chapter to the "classical" *event-based* definitions where input-output symbols are associated with *transitions*, but when we move in subsequent chapters to circuits, we will use a state-based (or signal-based) semantics and associate input-output symbols with states.

2.1 Timed Automata through an Example

In this section we give the basic intuitions concerning timed automata using an example of a timed system. This example will serve in the sequel to illustrate certain definitions and transformations.

Description of the timed system In many systems, excessive heat, for instance, is often considered to be a pathological sign and keeping the system operating in such conditions may cause severe problems that could be impossible to fix. To prevent such a serious damage, a component is often added to the system, to stop it if the symptoms endure. Between detection and shutdown the user is given a chance to resolve the problem and avoid the system outage. An informal description of the prevention component is given by the following operating rules:

- The component is in an *idle* state as long as the system functions properly.
- Once an anomaly is detected, a warning light is turned on to give the user a first indication.
- In addition to the warning light, the anomaly detection triggers a saving process for all the current system applications. This urgent operation is achieved within a delay d_{save} (1 minute). Meanwhile the user cannot start any repair task.
- After d_{save} time from the anomaly detection the user can intervene and try to repair the source of the problem.
- Within a delay d_{alarm} (3 minutes), if the problem has not been fixed, an additional acoustic alarm is turned on.
- The user can deactivate this alarm anytime and return to the state where only the warning light is on, and then proceed to reparation.
- However if this reparation is not achieved within a delay d_{alarm} (3 minutes) from the alarm deactivation, then the acoustic alarm will be turned on again. Further deactivating of the alarm by the user is still possible.
- If the problem has not been fixed within a total delay of $d_{anomaly}$ (8 minutes) from detection then the system will be forced to stop immediately.
- On the other hand, if successful repair is claimed by the user then the prevention system will wait for d_{ensure} (5 minutes) before returning to its idle state. However, if the problem reappears within this delay, then the anomaly is considered as serious and the system will be shut down immediately, without further attempts to repair it.

System Modeling Looking at these operating rules, we can notice that the system has several hard timing constraints. Timed automata are one of the natural formalisms for modeling this kind of systems. In fact, the timed automaton given in Figure 2.1 constitutes a formal model of the previously-described prevention system.

This automaton is presented as a *discrete structure* of five nodes, called *discrete states* or *locations*. Discrete states are supposed to capture all information about the current status of the

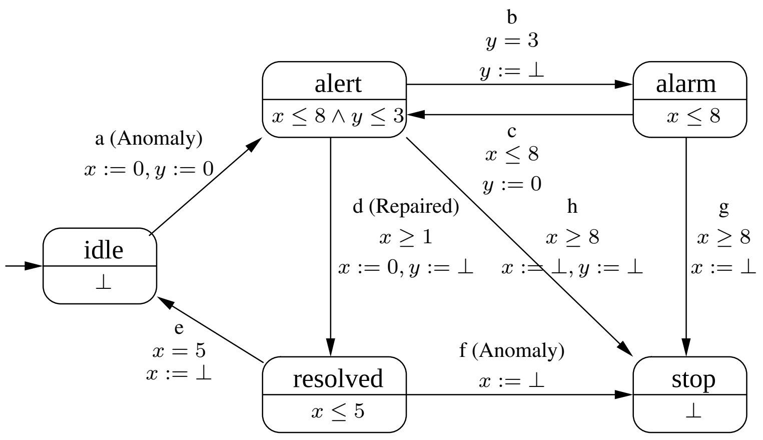


Figure 2.1: A timed automaton modeling the prevention system.

system, *except for timing information.* In the example above we have five possible states: an *idle* state which represents the normal operation mode of the controlled system; state *alert* which models the first detection of the anomaly, which means also that the warning light is turned on; state *alarm* representing the state of the system when the alarm is activated; state *resolved* which stands for the system waiting after a repair has been attempted and state *stop* modeling the situation when no successful repair has been done in time and the system is shut down by the prevention component.

The edges of this discrete structure represent *events* or *transitions* which change the discrete state of the system. For example, the anomaly detection event will make the system change its state from *idle* to *alert.* Then, a repair attempt will make it switch from state *alert* to state *resolved.* Events are considered to be *atomic* and *instantaneous.*

Notice that time progress, which is not expressed explicitly in this structure, takes place *inside* the *discrete states.* Actually, time passage is recorded using positive real variables called *clocks.* All the active clocks of the system increase synchronously at the same rate. These clocks can be set to zero, or deactivated[1] when a transition is taken. To model the prevention system, two clock variables x and y are used. The transition from *idle* to *alert* as well as the transition from *alarm* to *alert* are transitions that perform clock resetting.

Clocks constraints are used to restrict the behavior of the automaton by forcing it to leave a state or forbidding it from taking a certain transition. There are two different styles for expressing these constraints. In the first style that we describe below, there is a separate treatment of constraints associated with states and transitions. In the other style, advocated by [SY96, BST97], both types of information are associated with the *transitions.*[2]

[1]This concept of clock deactivation will be explained in the sequel.

[2]Note that by transition we mean here a transition of one component of a system which will correspond to a

Constraints on states are expressed using staying conditions called *invariants* [HNSY94] which are timing conditions related to each discrete state. The automaton may stay in the state (while the active clocks are progressing) as long as the state invariant holds, otherwise it has to *leave* the state via one of the enabled transitions.

The second type of timing constraints are the transition guards. A transition can be taken only if its guard constraint is satisfied. For instance, in the given example, the system can stay at the *alert* state as long as the invariant $(x \leq 8) \wedge (y \leq 3)$ is satisfied. From this state, the transition labeled by d (anomaly repair) can be taken only after the value of clock x exceeds the threshold $d_{save} = 1$, which expresses the inability to repair before application saving. Notice that there is no urgency in taking this transition and the user may as well decide not to take it at all even though its guard $x \geq d_{save}$ is satisfied. But in any case, the automaton should leave state *alert* before clock x exceeds $d_{anomaly}$ or clock y exceeds d_{alarm}. Hence transition d could be taken at *any time* in which the value of clock x is in the time interval $[d_{save}, d_{anomaly}] = [1, 8]$. The ability to take a transition anywhere within an interval of time introduces a *dense non-determinism* which is a very useful modeling feature when we have uncertain information about process durations. The upper bound of the interval is urgent and the system *must* leave the "alert" state when $x = 8$. If at that time no transition is enabled then the system will be in a *blocking state*. We will avoid such blocking situations in our modeling approach.

The last notion that we present via this example is that of *clock activity*, first introduced by Yovine and Daws [DY92] in order to reduce the number of clocks and the complexity of timed verification. A clock is said to be active at some discrete state when its value is relevant for the *future evolution* of the system, for example, when the clock appears in the state invariant or in a guard of a transition outgoing from the state. In the automaton of Figure 2.1 the value of clock y is not relevant at state *resolved* because the only references to its value are in state *alert* and its outgoing transition, and all the paths from *resolved* to *alert* reset this clock without testing its value. Hence we can conclude that clock y is inactive in state *resolved.*

The work of [DY92] had two major parts. The first was the (approximate) detection of such clock inactivity by performing a syntactic data-flow analysis of the automaton. Then this information was used to reduce the dimensionality of the clock space, by manipulating in each state polyhedra whose dimension is equal to the number of clocks active in this state. In our modeling framework we express clock deactivation *explicitly* by assigning a clock value to $\perp$ and, as we shall see, in the class of automata that we use for modeling, clock activity tracking is self-evident and we can benefit from the dimensionality reduction without performing the analysis.

family of transitions in the global automaton after composition.

2.2 Preliminary Definitions

The major novelty in timed automata is the connection between the discrete dynamics and the timing constraints. To express such relations we need to be able to speak of clock constraints and resets and of sets of clock valuations encountered during the verification process. We use $\mathbb{Z}$ and $\mathbb{R}$ to denote, respectively the integer and real numbers, while $\mathbb{N}$ and $\mathbb{R}_+$ will stand for their respective non-negative restrictions. Throughout this thesis we will use $\mathbb{R}_+$ as the time domain on which clock variables range, but most of the definitions will hold also for $\mathbb{N}$. We use $\mathbb{R}_\perp$ to denote $\mathbb{R}_+ \cup \{\perp\}$ were $\perp$ is a special symbol meaning "inactive" or "irrelevant". We extend the addition operation to $\mathbb{R}_\perp$ by letting $\perp + d = \perp$.

2.2.1 Clocks, Time Constraints and Zones

Clocks and Valuations

Let $C = \{c_1, ..., c_n\}$ be a finite set of variables called *clocks*, each ranging over $\mathbb{R}_\perp$. A *clock valuation* is a function $v : C \to \mathbb{R}_\perp$ assigning to each clock $c \in C$ its value $v(c)$. The set of possible valuations of C is then $\mathbb{R}_\perp^n$. A clock c is said to be *active* in valuation v iff $v(c) \neq \perp$, otherwise it is *inactive*. Due to the distinction between active and inactive clocks, we will have to deal with clock valuations of varying dimensionality. Given some $C' \subseteq C$, we use $\mathcal{V}_{C'}$ to denote valuations in which $v(c) \neq \perp$ iff $c \in C'$. Elements of $\mathcal{V}_{C'}$ are then non-negative real vectors (points) of dimension $|C'| \leq n$. The projection operation $v' = v/_{C'}$ maps elements of $\mathcal{V}_C$ to elements of $\mathcal{V}_{C'}$ by letting $v'(c) = v(c)$ if $c \in C'$ and $v'(c) = \perp$ otherwise.

In timed automata, clock valuations change due to two types of activities: *time progress* which happens inside a discrete state and *clock assignments* which take place during discrete transitions:

Time progress: Let d be a non-negative real. We say that clock valuation v' is the result of applying *d-time-progress* to clock valuation v, denoted by $v' = v + d$, if for every clock c, $v'(c) = v(c) + d$. Note that by the definition of addition on $\mathbb{R}_\perp$, all the clocks inactive in v do not change their value while all the other clocks advance uniformly.

Clocks assignments: A clock assignment is a function $\gamma : \mathbb{R}_\perp^n \to \mathbb{R}_\perp^n$ indicating a transformation of clock values which occurs during a transition. The type of assignments that we allow in the definition of timed automata is restricted to compositions of one or more of the following basic assignments:

- $c_i := 0$ (resetting to zero)
- $c_i := \bot$ (deactivation of a clock)
- $c_i := c_j$ (clock copying)

We denote by $v' = \gamma(v)$ the fact that v' is the result of applying assignment γ to clock valuation v and use $\gamma_1 \circ \gamma_2$ for composition of assignments. The set of all assignments obtained by arbitrary compositions of basic assignments is denoted by Γ_C. We use the shorthand $r_{C'}$ to denote resetting all the clocks in some $C' \subseteq C$ and $kill_{C'}$ to denote the deactivation of these clocks. We define the restriction of some $\gamma \in \Gamma_C$ to a set of clocks $C' \subseteq C$ as the assignment $\gamma/_{C'} \in \Gamma_{C'}$ obtained from γ by removing all basic assignments that mention clocks outside C'.

Clocks Constraints

Clocks constraints are used to express the influence of clock values on the discrete dynamics (invariants and transition guards). We restrict ourselves to a family of constraints that we denote by Ψ_C, defined by the following grammar:

$$\psi ::= true \,|\, c_i \prec k \,|\, c_i - c_j \prec k \,|\, \psi \wedge \psi$$

where $c_i, c_j \in C$, $k \in \mathbb{N}$ and $\prec \in \{<, \leq, =, \geq, >\}$. The constraints of the form $x \prec k$ and $x - y \prec k$ are called *atomic*.

We define the *restriction* of some $\psi \in \Psi_C$ to a set of clocks $C' \subseteq C$, denoted by $\psi/_{C'}$, as the constraints syntactically obtained by adding to the ψ all the minimal constraints of the form $c \leq d$ and $c - c' \leq d$ which are implied by the constraints in ψ, and then removing all the constraints that mention a clock outside of C'.

Timed Zones

Clock constraints define subsets of the set of clock valuations, those that satisfy them. These are subsets of $\mathbb{R}_+^m$ where $m \leq n$ is the number of active clocks in the state where they are evaluated. We will always assume that constraints evaluated in a state mention only variables that are active in that state. We use $v \models \psi$ to denote the fact that valuation v *satisfies* a clock constraint ψ, and $[\![\psi]\!]$ to denote all satisfying valuations.

Every constraint $\psi \in \Psi_C$ is a conjunction of *atomic constraints*. Knowing that the set of valuations satisfying an atomic constraint defines a *half-space*, every constraint $\psi \in \Psi_C$ will define a *convex polyhedron* which is the intersection of those half-spaces. We use, in addition to $[\![\psi]\!]$, the notation Z_ψ for this polyhedron and call it the *timed zone* associated with ψ. The set of all the zones defined on C will be denoted $\mathcal{Z}_C$. Since the half-spaces are either orthogonal ($c_i \prec k$) or diagonal ($c_i - c_j \prec k$) with integer k, the vertices of these polyhedra are integer points and there is a finite number of zones in any bounded subset of $\mathbb{R}^n$.

Remark: It is a well-known fact in the theory of timed automata that there is a constant l, the largest constant appearing in the clock constraints of an automaton, beyond which further changes in the value of clock c do not matter for any constraint of the form $c \prec k$ because substituting any $l' > l$ instead of l for c will preserve the truth value of the constraint. Hence, for all practical purposes, we can see clock valuations as ranging over $[0, l]$, implying a finite number of zones. However, as shown in [Bou02], this fact should be exploited with care, otherwise, subtle bugs related to difference constraints can be manifested. We do not get into the details of this technical issue in this thesis to avoid making the definitions and notations heavier. The automata that we use in this thesis do not use difference constraints in their definition.

In the following we will define some useful operations on zones that will be used throughout this document. Let $C' \subseteq C$ be a set of clocks, and let $Z_1, Z_2 \in \mathcal{Z}_C$ be two timed zones defined on C, then:

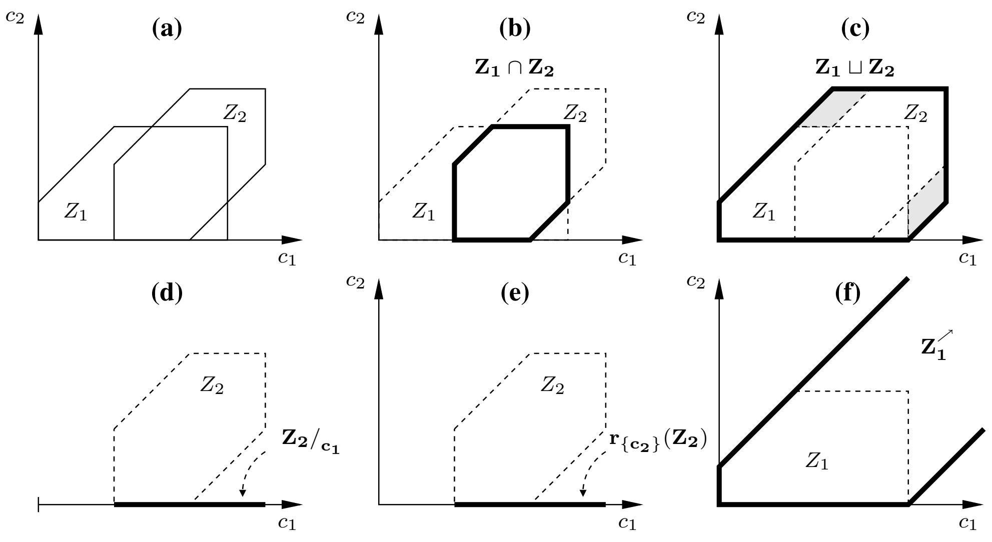


Figure 2.2: Operations on timed zones.

$Z_1 \cap Z_1$ is the *intersection* of two zones Z_1 and Z_2, which is a convex zone, see Figure 2.2-(b).

$Z_1 \sqcup Z_2$ is what we call the *timed convex hull* of the two zones Z_1 and Z_2 defined as:

$$Z_1 \sqcup Z_2 = \min\{Z \in \mathcal{Z}_C \,|\, (Z_1 \subseteq Z) \wedge (Z_2 \subseteq Z)\},$$

that is the smallest (in terms of inclusion) timed zone containing both Z_1 and Z_2, see Figure 2.2-(c). Since zones are not closed under union, $Z_1 \sqcup Z_2$ can be used as an over-approximation of $Z_1 \cup Z_2$.

$Z^{\nearrow}$ is what we call the *forward projection* of Z, that is, all clock valuations that can result by applying time progress to elements of Z:

$$Z^{\nearrow} = \{v \in \mathcal{V}_C \mid \exists d \geq 0,\ v - d \in Z\},$$

see Figure 2.2-(f).

$Z/_{C'}$ is the projection of a zone Z on a clock subset $C' \subseteq C$:

$$Z/_{C'} = \{v/_{C'} \mid v \in Z\},$$

see Figure 2.2-(d). This operation is related to clocks deactivation.

$\gamma(Z)$ is the result of applying the clock assignment function γ to all the elements of Z:

$$\gamma(Z) = \{\gamma(v) \mid v \in Z\}.$$

As we did previously we will distinguish two particular instances of this operation:

$r_{C'}(Z)$ is the *resetting* of the clocks within the set $C' \subseteq C$, see Figure 2.2-(e).

$kill_{C'}(Z)$ is the *deactivation* (or "*killing*") of clocks of the set C'. Note that $kill_{C'}(Z) = Z/_{C \setminus C'}$.

It is important to note the difference between $r_{C'}(Z)$ and $kill_{C'}(Z)$. Applying $kill_{C'}$ we may reduce the dimensionality of the *space* on which the zone is defined. Applying $r_{C'}$ we may reduce the dimensionality of the *set* but *not* of the space on which it is defined[3], and after some time progress the set will regain its reduced dimensions. It is not hard to see that zones are closed under all the above operations. Moreover, all these operations can be computed efficiently on a DBM representation of the zones. More details can be found, for example, in [Yov93].

2.2.2 Timed Automata Syntax and Semantics

DEFINITION 2.1 (Timed Automaton) *A timed automaton is a tuple* $\mathcal{A} = (Q, q_0, C, \Sigma, I, \Delta)$ *where:*

[3] In fact, we may increase the dimension if we reset a clock which was previously inactive.

Q	*is a finite set of discrete states,*
$q_0 \in Q$	*is the initial state*
C	*is a finite set of clocks,*
Σ	*is a finite set of labels,*
$I \in Q \to \Psi_C$	*is a function associating a staying condition (invariant) with every state* q*. The automaton is permitted to stay at* q *only as long as the clock constraint* $I(q)$ *is satisfied.*

$\Delta \subseteq Q \times \Psi_C \times \Sigma \times \Gamma_C \times Q$

is the transition relation consisting of elements of the form $e = (q, g, a, \gamma, q')$ *where:*

$q, q' \in Q$	*are, respectively, the source and the target of the transition,*
$g \in \Psi_C$	*is an enabling condition called the transition guard. It restricts the execution of the transition to clock valuations that satisfy it.*
$a \in \Sigma$	*is the transition label,*
$\gamma \in \Gamma_C$	*is a clock assignment function which takes place during a transition.*

We assume, without loss of generality, that from every state q there is at most one transition labeled by a for every $a \in \Sigma$.

Example The automaton of Figure 2.1 is defined as:

Q : $\{idle, alert, alarm, resolved, stop\}$,

q_0 : $\{idle\}$,

C : $\{x, y\}$,

Σ : $\{a, b, c, d, e, f, g, h\}$,

I : $\{idle \mapsto true,$
$alert \mapsto x \leq 8 \wedge y \leq 3,$
$alarm \mapsto x \leq 8,$
$resolved \mapsto x \leq 5,$
$stop \mapsto true\}$

$$
\begin{aligned}
\Delta: \quad & \{\big(idle, true, a, r_{\{x,y\}}, alert\big),\\
& \big(alert, y = 3, b, kill_{\{y\}}, alarm\big),\\
& \big(alarm, x \leq 8, c, r_{\{y\}}, alert\big),\\
& \big(alert, x \geq 1, d, r_{\{x\}} \circ kill_{\{y\}}, resolved\big),\\
& \big(alert, x \geq 8, h, kill_{\{x,y\}}, stop\big),\\
& \big(resolved, x = 5, e, kill_{\{x\}}, idle\big),\\
& \big(resolved, true, f, kill_{\{x\}}, stop\big),\\
& \big(alarm, x \geq 8, g, kill_{\{x\}}, stop\big)\}.
\end{aligned}
$$

Let us remark that it is sometimes more convenient to refer to guards, invariants and sets of clock valuations as syntactic objects (constraints) and sometimes as semantic geometric objects (zones). In order not to introduce too many notations we will use the same symbols I and g for both, so that we can write $v \models I$ as well as $v \in I$ or $g \wedge \psi$ as well as $g \cap Z$.

Parallel Composition of Timed Automata

A timed automaton is often considered to be an element in a network of components running in parallel and communicating with each other. The global behavior of such a network is captured by the global timed automaton, called the *product*. There are many variations of composition depending mainly on the interaction mechanisms through which the automata influence each other. At this point we use a definition based on a *distributed alphabet* [DR95] where each component $\mathcal{A}^i$ has its alphabet Σ^i. The alphabets of the components may have non-empty intersections and any global transition labeled by a must involve a local a-transition in *every* automaton $\mathcal{A}^i$ such that $a \in \Sigma^i$. Independent local transitions (transitions with different labels) enabled at the same global state can be executed in *any* order (interleaving).

DEFINITION 2.2 (Parallel Composition of Timed Automata)
Let $\mathcal{N} = \{\mathcal{A}^i = (Q^i, q_0^i, C^i, \Sigma^i, I^i, \Delta^i) \mid i \in \{1,..,n\}\}$ be a network of timed automata. We assume the sets of clocks of each pair of automata to be disjoint and denote by $\mathcal{J}(a)$ the indices i such that $a \in \Sigma^i$. The composition of these automata, denoted by $\mathcal{A}^1 \parallel .. \parallel \mathcal{A}^n$ is a timed automaton $\mathcal{A} = (Q, q_0, C, \Sigma, I, \Delta)$ where:

$Q = Q_1 \times \ldots \times Q_n$ *is the set of global discrete states of the form $q = (q^1, \ldots, q^n)$,*

$q_0 = (q_0^1, \ldots, q_0^n)$ *is the initial state,*

$C = \bigcup_{i=1}^n C_i$ *is the global set of clocks,*

$\Sigma = \bigcup_{i=1}^n \Sigma^i$ *is the global alphabet,*

I *is the global state invariant* $I(q) = \bigwedge_{i \in \{1..n\}} I^i(q^i)$,

Δ *is the global transition relation consisting of tuples of the form* $((q^1, \ldots, q^n), g, a, \gamma, (q'^1, \ldots, q'^n))$ *such that*

- *for every* $i \notin \mathcal{J}(a)$, $q'^i = q^i$,
- *for every* $i \in \mathcal{J}(a)$, $(q^i, g^i, a, \gamma^i, q'^i) \in \Delta^i$,
- $g = \bigcap_{i \in \mathcal{J}(a)} g^i$,
- $\gamma = \circ_{i \in \mathcal{J}(a)} \gamma^i$

Notice that since each assignment γ^i operates on a distinct set of clocks, the composition of assignments is commutative. An example of a timed automata product is given in Figure 2.3.

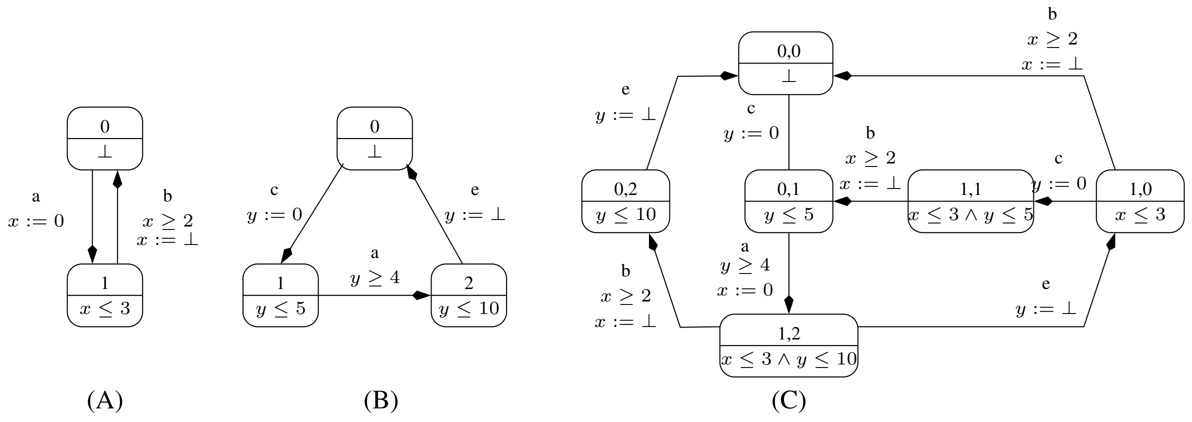


Figure 2.3: A timed automata product: $C = A \parallel B$.

Semantics of Timed Automata

Timed automata define infinite transition systems whose states are *configurations* of the form (q, v) consisting of a discrete state q and a clock valuation v. The initial configuration is $s_0 = (q_0, \perp)$ with all clocks inactive and the transitions are either discrete transitions of the automaton or time-passage transitions. This is formalized by the notion of a step.

DEFINITION 2.3 (Steps) *A step of a timed automaton $\mathcal{A}$ is one of the following:*

- *A* discrete step*:* $(q, v) \xrightarrow{a} (q', v')$, *for some transition* $(q, g, a, \gamma, q') \in \Delta$ *such that* $v \models g$ *and* $v' = \gamma(v)$,
- *A* time step*:* $(q, v) \xrightarrow{d} (q, v + d)$ *for some* $d \in \mathbb{R}_+$ *such that* $v + d$ *satisfies* $I(q)$.

Note that the concatenation of two time steps is a time step:

$$(q, v) \xrightarrow{d_1} (q, v + d_1) \xrightarrow{d_2} (q, v + d_1 + d_2) \equiv (q, v) \xrightarrow{d_1+d_2} (q, v + d_1 + d_2).$$

Conversely, due to the dense nature of the real numbers, a time step can be split into any number of smaller time steps.

A *compound step* is a discrete step followed by a time step (possibly of zero duration):

$$(q, v) \xrightarrow{a,d} (q', v' + d) \equiv (q, v) \xrightarrow{a} (q', v') \xrightarrow{d} (q', v' + d)$$

A *run* of the automaton $\mathcal{A}$ starting from a configuration (q, v) is a finite sequence of compound steps.[4] In the following we use the notation $(q, v) \xrightarrow{\xi} (q', v')$ for runs.

These definitions apply to products as well. Note that a global time step in a global state $q = (q^1, \ldots, q^n)$ is just a local (and uniform) time step for each component $\mathcal{A}^i$. The global invariant requires that all local invariants hold at $v + d$. On the other hand a global discrete step labeled by a is a local discrete step for all components $\mathcal{A}^i$ such that $a \in \Sigma^i$.

Example The following is a run of the automaton presented in Figure 2.1. It starts at $idle$, where it can stay indefinitely. After 16 minutes it moves to state $alert$ while detecting an anomaly and reset clocks x and y, stays at $alert$ for 3 minutes without any intervention from the user, then moves to $alarm$ and so on until it reaches $stop$ and stays there for 14 minutes until the end of the run.

$$(idle, (\bot, \bot)) \xrightarrow{16} (idle, (\bot, \bot)) \xrightarrow{a} (alert, (0, 0)) \xrightarrow{3} (alert, (3, 3))$$

$$\xrightarrow{b} (alarm, (3, \bot)) \xrightarrow{5} (alarm, (8, \bot)) \xrightarrow{g} (stop, (\bot, \bot)) \xrightarrow{14} (stop, (\bot, \bot))$$

2.3 Symbolic Reachability

Having defined timed automata and their semantics, we would like to verify them, that is, to see what the possible runs of a given automaton are, and whether they satisfy a given property. However, since the state space is infinite, simple enumeration of the possible runs is impossible. Fortunately, these runs can be grouped into equivalence classes, each consisting of runs having the same *qualitative* form in the sense that their sequences of discrete steps are identical. The

[4] The first step can be a pure time step.

enumeration of possible runs is done in a *set-based* fashion, computing in one step all the successors of a set of configurations by an arbitrary passage of time and by transitions, as will be explained in the sequel.

The original decidability proof for verification of timed automata [AD90] was based on partitioning the state space into a finite number of equivalence classes called *regions*. Regions are the "atomic" zones from which all other zones can be constructed. Two configurations (q, v) and (q, v') are region-equivalent if for every transition guard g, $v \models g$ iff $v' \models g$, and if by letting time pass they reach the same region. Hence for every sequence of regions visited by a run from (q, v) there is a run from (q, v') visiting the same sequence. The timed automaton can thus be reduced to a finite automaton whose states are regions with discrete transitions and special transitions that correspond to the passage of time. Region equivalence is guaranteed to capture all the qualitative behaviors of *any* automaton, but its force is also its weakness because the large number of regions renders this approach impractical.

Consequently, existing verification tools [DOTY95, BLL$^+$95, BSGS04] use more efficient verification methods based on coarser equivalence relations that depend on the structure of the *particular* automaton to be verified. The most popular approach today is on-the-fly forward search based on zones. This approach has the following advantages: 1) It does not explore the parts of the state space which are not reachable from the initial state; 2) It does not refine zones beyond what is necessary and will typically result in a number of generated zones much smaller than the number of regions; 3) It uses an efficient data structure, the DBM, to store and manipulate zones.

2.3.1 From Timed Automata to Reachability Graphs

The principle of symbolic reachability computation for discrete systems is to take a representation of a *set* P of states reachable after k steps, and compute from it the set $succ(P)$ of its successors by all transitions, that is, the set

$$succ(P) = \bigcup_{q \in P} \bigcup_{a \in \Sigma} \{succ^a(q)\}.$$

The application of this idea to timed automata is more subtle. Successors by different transitions are treated *separately* and *enumeratively*, while the symbolic treatment is reserved for time passage and clock valuations. In other words, the basic element in the computation is an object consisting of *one* discrete state and a *set* of clock valuations. To this object one applies time passage of arbitrary duration and transitions as formalized below.

A *symbolic state* of a timed automaton $\mathcal{A} = (Q, q_0, C, \Sigma, I, \Delta)$ is a pair (q, Z) where $q \in Q$ is a discrete state and Z is a zone. Symbolic states are closed under the following operations:

- The *time successor* of a symbolic state (q, Z) is the symbolic state (q, Z') where Z' is the set of clock valuations reachable from Z by letting time progress without violating the staying condition $I(q)$:

  $$post^t(q, Z) = \{(q, v + d) \,|\, (v \in Z) \wedge (d \geq 0) \wedge ((v + d) \models I(q))\} = (q, (Z^{\nearrow} \cap I(q)))$$

  We say that (q, Z) is *time-closed* if $(q, Z) = post^t(q, Z)$.

- Let $(q, g, a, \gamma, q') \in \Delta$ be a transition. The *a-transition successor* of a symbolic state (q, Z) is the set of configurations reached by taking this transition. Only clock valuations of Z that satisfy the guard g are concerned with this transition. This clock valuations will be transformed according to the assignment function γ while taking this transition:

  $$post^a(q, Z) = \{(q', v') \mid \exists v \in Z,\, v \models g \wedge v' = \gamma(v)\} = (q', (\gamma(Z \cap g)))$$

- The *a-successor* of a symbolic state (q, Z) is the set of configurations reached from (q, Z) by an a-transition followed by passage of time:

  $$succ^a(q, Z) = post^t(post^a(q, Z)) = (q', (\gamma(Z \cap g))^{\nearrow} \cap I(q'))$$

To have a better feel of the way these operations are used in a set-based computation of all the runs of a timed automaton, let us look closely at the relation between a symbolic state and its successor.

Proposition 2.1 *Let $(q', Z') = succ^a(q, Z)$ for a transition (q, g, a, γ, q'). A configuration (q', v') belongs to (q', Z') if and only if it is the endpoint of a compound step*

$$(q, v) \xrightarrow{a,d} (q', v')$$

for some $(q, v) \in (q, Z)$ and some $d \geq 0$.[5]

Hence computing (q', Z') amounts to computing "in parallel" the first segment of an uncountable number of runs, all starting from (q, Z) by making an a-transition followed by arbitrary passage of time.

[5]Note, however, that not all elements of (q, Z) are the start points of such compound steps because for some points, the time successors will not intersect g.

Equipped with these operators under which the set of symbolic states is *closed*, we can now introduce the *forward reachability* algorithm for timed automata which computes this way all the runs of $\mathcal{A}$. We present the breadth-first version of the algorithm but other exploration orders are possible. The algorithm terminates because there are finitely many zones in any bounded subset of $\mathbb{R}^n_\perp$.

Algorithm 2.1 Forward reachability algorithm (breadth first).

```
$Explored := \emptyset$
$New := \emptyset$
$Waiting := \{(q_0, \perp)\}$
while $(Waiting \neq \emptyset)$ do
   for each $(q, Z) \in Waiting$ do
      for each $(q, g, a, \gamma, q') \in \Delta$ do
          $New := New \cup succ^a(q, Z)$
      $Explored := Explored \cup (q, Z)$
   $Waiting := New \backslash Explored$
   $New := \emptyset$
return $(Explored)$
```

As a byproduct, this algorithm produces the *reachability* (or *simulation*) *graph* which can be viewed as the finite state automaton defined below.

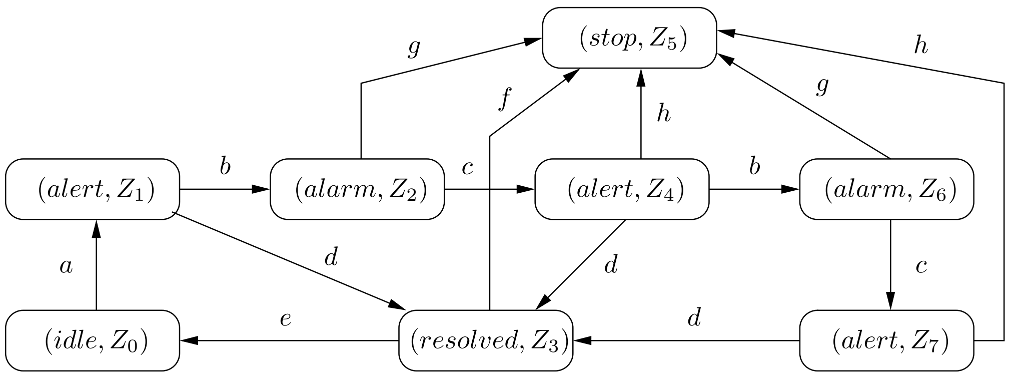
Definition 2.4 (Reachability Graph) *The reachability graph associated with a timed automaton $\mathcal{A} = (Q, q_0, C, \Sigma, I, \Delta)$ is a finite automaton $\mathcal{G} = (S, s_0, \Sigma, \delta)$ such that S is the smallest set of symbolic states containing the initial state $s_0 = (q_0, \perp)$ and closed under $\{succ^a\}_{a\in\Sigma}$. The transition relation δ consists of all triples of the form $((q, Z), a, (q', Z'))$ such that $(q', Z') = succ^a(q, Z)$.*

The fundamental property of the reachability graph is the following: for every sequence over Σ that this automaton may generate, there is a run of the timed automaton $\mathcal{A}$ generating the same sequence of events. In other words, the timed automaton $\mathcal{A}$ and the automaton $\mathcal{G}$ based on its reachability graph generate the same subsets of Σ^* or Σ^ω. Hence, if the properties we are interested in are purely qualitative, that is, concerned with the *order* of events, not with the *distance* between them, and if they have no alternating path quantifiers,[6] applying classical model checking algorithms to $\mathcal{G}$, we will obtain results which are valid for $\mathcal{A}$ as well. For branching-time properties, there is a finer equivalence relation, the time-abstract bisimulation [TY01], whose induced finite-state quotient can be used for verification (and synthesis).

Example of Reachability Graph The reachability graph of the timed automaton presented as an example in Figure 2.1 is depicted in Figure 2.4. Looking at the example we see some

[6]For example, properties expressed in LTL or in the existential or universal fragments of CTL.

important features of the reachability graph. The first is that some discrete states of the timed automaton are "split" into several copies, each with a different zone. For example the *alert* state has three copies depending on the number of times (zero, one, or two) the alarm was set and then disabled by the user. The reason these states are considered as different is due to the different values of x which determine the number of times the alarm can be triggered before the condition $x \geq 8$ will impose shutdown. Another important aspect of the reachability graph is that it eliminates states and transitions which are impossible according to timing constraints, for example, the transition labeled b is not possible in $(alert, Z_7)$. The effect of this reduction is not very visible in this example where all states are reachable, but will be more significant when we treat large *products* of timed automata where many global states may turn out to be unreachable due to competition among parallel processes.



$Z_0 : \bot$	$Z_4 : 3 \leq x \leq 8 \wedge 0 \leq y \leq 3 \wedge 3 \leq x - y \leq 8$
$Z_1 : 0 \leq x = y \leq 3$	$Z_5 : \bot$
$Z_2 : 3 \leq x \leq 8$	$Z_6 : 6 \leq x \leq 8$
$Z_3 : 0 \leq x \leq 5$	$Z_7 : 6 \leq x \leq 8 \wedge 0 \leq y \leq 2 \wedge 6 \leq x - y \leq 8$

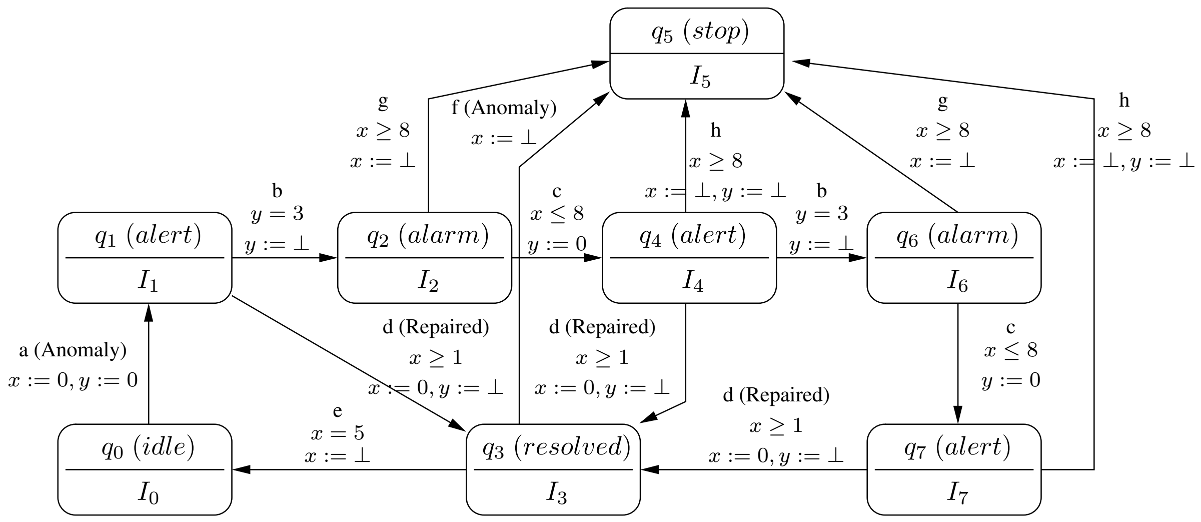
Figure 2.4: The reachability graph of the timed automaton of Figure 2.1.

2.3.2 From Reachability Graphs to Interpreted Timed Automata

Although the reachability graph, viewed as an untimed automaton, suffices for untimed verification, it is more useful for our purposes to see it as a timed automaton $\mathcal{A}^r$, semantically equivalent to the timed automaton $\mathcal{A}$ from which it was derived.

DEFINITION 2.5 (Interpreted Timed Automaton) *Let $\mathcal{A} = (Q, q_0, C, \Sigma, I, \Delta)$ be a timed automaton and let $\mathcal{G} = (S, s_0, \Sigma, \delta)$ be its corresponding reachability graph. The interpreted timed automaton for $\mathcal{A}$ is $\mathcal{A}^r = (S, s_0, C, \Sigma, I^r, \Delta^r)$ such that $I^r(q, Z) = Z$ and for every transition $((q, Z), a, (q', Z')) \in \delta$, corresponding to a transition $(q, g, a, \gamma, q') \in \Delta$ we define a transition $((q, Z), g \cap Z, a, \gamma, (q', Z')) \in \Delta^r$.*

In other words, we restrict the invariant and transition guards of each symbolic state (q, Z) to those clock valuations against which they are to be evaluated, namely those in Z. Clearly, the semantics of $\mathcal{A}$ and of $\mathcal{A}^r$ are equivalent. The interpreted timed automaton inherits the properties of the simulation graph in the sense that, unlike $\mathcal{A}$, all the paths in its transition graph are paths which are indeed realizable when timing constraints are taken into account. Thus, even if we later relax the timing constraints in $\mathcal{A}^r$ and introduce more behaviors, we do not introduce any new *qualitative* behavior.[7] This is certainly not true of $\mathcal{A}$: if we remove the timing constraints from it, we will typically add more behaviors and ignore the timing constraints altogether. Relaxation of the timing constraints in $\mathcal{A}^r$, after having used them to eliminate behaviors, is the crucial ingredient of our abstraction techniques to be described after. The interpreted timed automaton derived from the timed automaton of Figure 2.1 and its reachability graph of Figure 2.4 is shown in Figure 2.5.



$I_0 : \perp$

$I_1 : 0 \leq x = y \leq 3$

$I_2 : 3 \leq x \leq 8$

$I_3 : 0 \leq x \leq 5$

$I_4 : 3 \leq x \leq 8 \wedge 0 \leq y \leq 3 \wedge 3 \leq x - y \leq 8$

$I_5 : \perp$

$I_6 : 6 \leq x \leq 8$

$I_7 : 6 \leq x \leq 8 \wedge 0 \leq y \leq 2 \wedge 6 \leq x - y \leq 8$

Figure 2.5: Interpreted timed automaton of the timed automaton of Figure 2.1.

Complexity As mentioned earlier, in the worst case the size of the reachability graph can be as large as that of the region graph but in practice it can be much smaller. To compute the worst-case number of zones, assume we have n interacting timed automata, each having m discrete states and one clock, ranging over $[0, d]$. The number of discrete states in the product can be up to m^n. The number of zones in $[0, d]^n$ can be up to $d^n n!$ (assuming that all clock constraints use

[7]In fact, $\mathcal{G}$ can be seen as an extreme form of relaxation where all guards and invariant are replaced by $true$.

non-strict inequalities). Hence the number of symbolic states in the reachability graph can be up to $m^n d^n n!$ and each of them takes $O(n^2)$ space. As one can see, this is not an easy problem.

Chapter 3

On Interleaving in Timed Automata

In this chapter we present a major improvement to the symbolic reachability algorithm for timed automata which reduces significantly the number of generated zones.

3.1 Introduction

When analyzing large asynchronous products of automata, one faces the following phenomenon. Except for synchronized actions, each automaton may evolve locally, taking its own transitions independently of the others and ordering between transitions of different automata is not constrained. Interleaving semantics enters the picture when we want to give semantics to the product automaton in terms of *global* runs. According to this approach, the set of global runs is considered to be the set of all possible ways to interleave (or shuffle) these independent actions in a manner consistent with the local orders implied by each automaton. To illustrate the idea consider the two automata of Figure 3.1. The first automaton takes local transition a before the synchronized transition c, while the second automaton takes local transition b before c. The corresponding local behaviors are $a \cdot c$ and $b \cdot c$ while a global behavior would be a sequence consisting of a and b in any order followed by c, an element of the language $(a \cdot b + b \cdot a) \cdot c$. Speaking in order-theoretic terms, and letting t_a, t_b and t_c denote, respectively, the times when the transitions take place, a global run should satisfy $t_a < t_c$ and $t_b < t_c$ while the ordering between t_a and t_b is not fixed[1]. The part of the global automaton between 00 and 11, which specifies different interleavings of independent actions is called a *diamond*. Needless to say, the size of such diamonds (and the number of global runs) grows exponentially with the number of components. This issue has been studied extensively both from a purely semantic perspective

[1]This is why techniques that deal with the explosion caused by interleaving are often called *partial-order techniques*.

and from a practical point of view [God96, PPH97, DR95]. A typical question in this domain would be to find conditions under which it will be sufficient to look at some *representatives* of the set of these global runs in order to satisfy a property. We are not dealing here with these issues but rather with the *additional* overhead that interleaving introduces into the verification of timed automata. We will demonstrate the problem in Section 3.2 and then prove a simple convexity result in Section 3.3 that will allow us to define in Section 3.4 an improvement to the symbolic reachability algorithm (Algorithm 2.1) which eliminates this type of explosion. This algorithm has been incorporated into the IF toolbox and the experimental results are reported in Section 3.5.

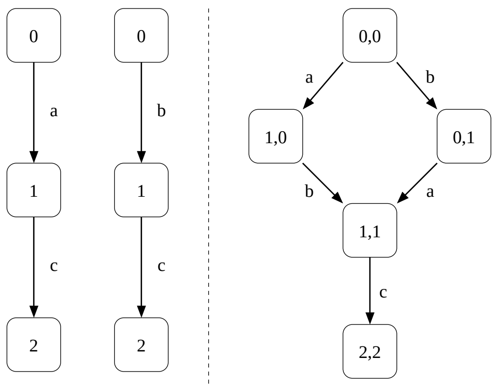


Figure 3.1: Composing two automata with independent actions a and b and a common action c.

3.2 Zone Explosion due to Interleaving

Consider the product $A \parallel B$ of the two timed automata of Figure 3.2-(a). Each automaton may choose to make its own independent transition which resets its own clock. In the untimed model, the two possible interleavings, $a \cdot b$ and $b \cdot a$ converge to the same state $(1, 1)$ as in Figure 3.2-(b). However this is not the case for the simulation graph produced by the symbolic reachability algorithm for timed automata, as shown in Figure 3.2-(c). The algorithm will generate two symbolic runs depending on whether a occurred before b or vice versa:

$$\xi_{ab} : ((0,0), \bot) \xrightarrow{a} ((1,0), Z_a) \xrightarrow{b} ((1,1), Z_{ab})$$
$$\xi_{ba} : ((0,0), \bot) \xrightarrow{b} ((0,1), Z_b) \xrightarrow{a} ((1,1), Z_{ba})$$

with

$$Z_a = \{x \,|\, x \leq 5\} \qquad Z_b = \{y \,|\, y \leq 3\}$$

$$Z_{ab} = \left\{ (x,y) \,|\, \begin{array}{r} x \leq 5 \\ y \leq 3 \\ x \leq y \end{array} \right\} \quad Z_{ba} = \left\{ (x,y) \,|\, \begin{array}{r} x \leq 5 \\ y \leq 3 \\ y \leq x \end{array} \right\}$$

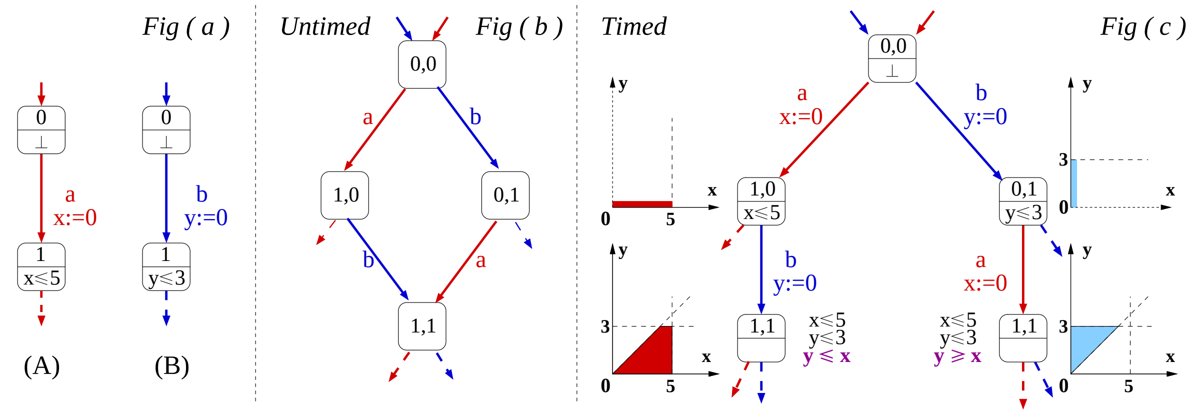


Figure 3.2: (a) Two timed automata; (b) their discrete diamond; (c) results of reachability analysis.

As we see, unlike the case of untimed systems, the two paths do not commute as they lead to two *different zones* Z_{ab} and Z_{ba}. If we look closer we see that the constraints defining these zones differ in one inequality: in Z_{ab} we have $x \leq y$ because transition a which resets x occurred before transition b that resets y, and likewise Z_{ba} has the constraint $y \leq x$. In general if we have n such components, the standard reachability algorithm for timed automata will generate up to $n!$ zones, each corresponding to a different order of these actions. To continue the computation, this algorithm will have to generate the successors of each of these zones *separately* by intersecting each of them with the *same* guards of transitions outgoing from the same discrete state. It is not hard to see that continuing this way, the verification of very simple timed automata that do not even interact, will lead to very quick explosion in time and, mostly, in space.

However if we look closer at Z_{ab} and Z_{ba}, either as a set of inequality constraints or as geometrical objects (Figure 3.3), we observe that their union is the *convex* zone

$$Z = \left\{ (x,y) \,|\, \begin{array}{r} x \leq 5 \\ y \leq 3 \end{array} \right\}$$

obtained by removing the constraints on the difference between x and y. In other words, Z represents the set of all $(1,1)$-configurations which are reachable by taking a and b, in *any order*. Since Z is a convex zone, to which we can apply intersection, we can think of the following optimization. For every guard g of a transition outgoing from $(1,1)$, instead of computing both $Z_{ab} \cap g$ and $Z_{ba} \cap g$ in order to find transition successors, we can merge them into $Z = Z_{ab} \cup Z_{ba}$ (see Figure 3.3) and compute $Z \cap g$ instead. Due to distributivity of union and intersection we get the same set of configurations

$$Z \cap g = (Z_{ab} \cup Z_{ba}) \cap g = (Z_{ab} \cap g) \cup (Z_{ba} \cap g),$$

but represented as a *single zone* instead of two. In the next section we show that this is not accidental and that the union of *all* zones reachable by different interleavings of a fixed set of actions is convex and such zones can be progressively merged to eliminate the type of explosion associated with interleaving.

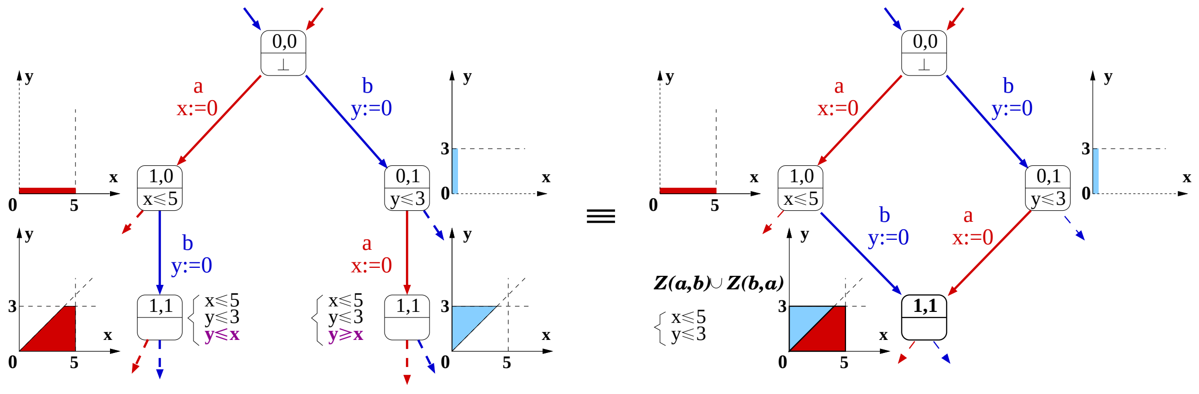


Figure 3.3: Semantics preserving reduction of the simulation graph by merging two zones whose union is convex.

3.3 Convexity Result

We first introduce some definitions and notations concerning the states and runs of a product

$$\mathcal{A} = \mathcal{A}^1 \parallel \mathcal{A}^2 \parallel ... \parallel \mathcal{A}^n$$

of n timed automata. Global configurations of the product are of the form $s = (q, v)$ where $q = (q^1, ..., q^n)$ and $v = (v^1, ..., v^n)$ where each (q^i, v^i) is a local configuration of automaton $\mathcal{A}^i$ consisting of its discrete state q^i and the valuation v^i of its clocks. Given a run ξ of $\mathcal{A}$, we let the local run ξ^i be the *projection* of ξ on automaton $\mathcal{A}^i$. This projection is done in three stages. First we project the configurations appearing in ξ on the states and clocks of $\mathcal{A}^i$. Then we "hide" transitions in which $\mathcal{A}^i$ does not participate, that is, we replace every transition label $a' \notin \Sigma^i$, by the "silent" symbol ε. Finally, we collapse together time steps which are separated by silent transitions. This is done by applying, successively, the following transformation:

$$(q,v) \xrightarrow{d,a} (q',v') \xrightarrow{d',\varepsilon} (q'',v'') \quad \Longrightarrow \quad (q,v) \xrightarrow{a,d+d'} (q'',v'').$$

Example: Consider again the product $A \parallel B$ of Figure3.2-(a) and a global run ξ consisting of five steps. The discrete steps are an a-transition, taken by A and a b-transition, taken by B. The other three steps are time steps, which are common to the two automata. Global states are represented as tuples of the form $((q_A, q_B), (v_x, v_y))$:

$$\xi : ((0,0),(0,0)) \xrightarrow{6} ((0,0),(6,6)) \xrightarrow{a} ((1,0),(0,6)) \xrightarrow{3} ((1,0),(3,9)) \xrightarrow{b} ((1,1),(3,0)) \xrightarrow{1.3} ((1,1),(4.3,1.3))$$

The projection of ξ on automaton B works as follows:

1. Projection on the states and clock valuations of B:

$$\xi_1 : ((0),(0)) \xrightarrow{6} ((0),(6)) \xrightarrow{a} ((0),(6)) \xrightarrow{3} ((0),(9)) \xrightarrow{b} ((1),(0)) \xrightarrow{1.3} ((1),(1.3))$$

2. Hiding the transition a in which B does not participate:

$$\xi_2 : (0,0) \underbrace{\xrightarrow{6} (0,6) \xrightarrow{\varepsilon} (0,6) \xrightarrow{3}} (0,9) \xrightarrow{b} (1,0) \xrightarrow{1.3} (1,1.3)$$

3. Combining time steps separated by hidden transitions which gives the projected run:

$$\xi^B : (0,0) \xrightarrow{9} (0,9) \xrightarrow{b} (1,0) \xrightarrow{1.3} (1,1.3)$$

As explained in the previous chapter, symbolic reachability computation groups together runs that are *qualitatively* equivalent, a notion that we formalize below.

DEFINITION 3.1 (Qualitative equivalence) *Two runs ξ and ξ' are qualitatively equivalent, denoted by $\xi \approx \xi'$, if they go through the same sequence of discrete transitions and may differ only on timing. The class of runs qualitatively equivalent to ξ is denoted $[\xi]$.*

An equivalence class can thus be seen as a sequence of events that we denote as $[\xi] = a_1 \cdot a_2 \cdots a_m$.

Example: Consider three global runs ξ_1, ξ_2, ξ_3, of $A \parallel B$ of Figure 3.2-(a):

$$\xi_1 : ((0,0),(0,0)) \xrightarrow{6} ((0,0),(6,6)) \xrightarrow{\mathbf{a}} ((1,0),(0,6)) \xrightarrow{3} ((1,0),(3,9)) \xrightarrow{\mathbf{b}} ((1,1),(3,0)) \xrightarrow{1.3} ((1,1),(4.3,1.3))$$

$$\xi_2 : ((0,0),(0,0)) \xrightarrow{4} ((0,0),(4,4)) \xrightarrow{\mathbf{a}} ((1,0),(0,4)) \xrightarrow{1} ((1,0),(1,5)) \xrightarrow{\mathbf{b}} ((1,1),(1,0)) \xrightarrow{2} ((1,1),(3,2))$$
$$\xi_3 : ((0,0),(0,0)) \xrightarrow{2} ((0,0),(2,2)) \xrightarrow{\mathbf{b}} ((0,1),(2,0)) \xrightarrow{2} ((0,1),(4,2)) \xrightarrow{\mathbf{a}} ((1,1),(4,0)) \xrightarrow{0.7} ((1,1),(4.7,0.7))$$

For these runs we have $\xi_1 \approx \xi_2$ as $[\xi_1] = [\xi_2] = a \cdot b$ but $\xi_3 \not\approx \xi_1$ because $[\xi_3] = b \cdot a$. Let us now look at the local projections of these runs on A and B:

$$\begin{array}{ll}
\xi_1^A : (0,0) \xrightarrow{6} (0,6) \xrightarrow{\mathbf{a}} (1,0) \xrightarrow{4.3} (1,4.3) & \xi_1^B : (0,0) \xrightarrow{9} (0,9) \xrightarrow{\mathbf{b}} (1,0) \xrightarrow{1.3} (1,1.3) \\
\xi_2^A : (0,0) \xrightarrow{4} (0,4) \xrightarrow{\mathbf{a}} (1,0) \xrightarrow{3} (1,3) & \xi_2^B : (0,0) \xrightarrow{5} (0,5) \xrightarrow{\mathbf{b}} (1,0) \xrightarrow{2} (1,2) \\
\xi_3^A : (0,0) \xrightarrow{4} (0,4) \xrightarrow{\mathbf{a}} (1,4) \xrightarrow{0.7} (1,4.7) & \xi_3^B : (0,0) \xrightarrow{2} (0,2) \xrightarrow{\mathbf{b}} (1,0) \xrightarrow{2.7} (1,0.7)
\end{array}$$

These projections are qualitatively equivalent:

$$\xi_1^A \approx \xi_2^A \approx \xi_3^A \text{ and } \xi_1^B \approx \xi_2^B \approx \xi_3^B$$

because in each of them, the automaton is taking the same transition:

$$[\xi_1^A] = [\xi_2^A] = [\xi_3^A] = a \text{ and } [\xi_1^B] = [\xi_2^B] = [\xi_3^B] = b.$$

This leads to the following definition of *local equivalence*.

DEFINITION 3.2 (Local equivalence) *Let ξ and ξ' two global runs of $\mathcal{A} = \mathcal{A}^1 \parallel \mathcal{A}^2 \parallel ... \parallel \mathcal{A}^n$, and let ξ^i and ξ'^i be their local projections on $\mathcal{A}^i$. We say that ξ and ξ' are locally equivalent, denoted by $\xi \sim \xi'$, iff all their local projections are qualitatively equivalent: $\bigwedge_{i=1}^n (\xi^i \approx \xi'^i)$. The class of runs locally equivalent to ξ is denoted $\langle\xi\rangle$.*

We will write

$$\langle\xi\rangle = a_1^1 \cdot a_2^1 \cdots a_{m_1}^1 \parallel a_1^2 \cdot a_2^2 \cdots a_{m_2}^2 \parallel \ldots \parallel a_1^n \cdot a_2^n \cdots a_{m_n}^n$$

to express the fact that $\langle\xi\rangle$ consists of all the runs whose projection on every component $\mathcal{A}^i$ belong to the same qualitative equivalence class $a_1^i \cdot a_2^i \cdots a_{m_i}^i$. For our example all the three runs are locally equivalent, $\xi_1 \sim \xi_2 \sim \xi_3$, with their equivalence class being

$$\langle\xi_1\rangle = \langle\xi_2\rangle = \langle\xi_3\rangle = a \parallel b.$$

Clearly qualitative equivalence is stronger than local one, $\xi \approx \xi' \Rightarrow \xi \sim \xi'$, because it requires both local equivalence *and* the same interleaving order. Another obvious observations is that two locally equivalent runs that start from the same discrete state, end up in the same discrete state.

We can now state the main result of this chapter:

THEOREM 3.1 (Convexity) *Let Z be a time zone and let q and q' be two global states of $\mathcal{A}$. Let ξ be a run starting at q and ending at q'. Then the set*

$$Z' = R_{Z,\langle\xi\rangle} \equiv \bigcup_{\xi' \in \langle\xi\rangle} \left\{ v' \,|\, \exists v \in Z,\, (q,v) \xrightarrow{\xi'} (q',v') \right\}$$

is convex.

The proof is given via a characterization of the reachable clock valuations by a quantified formula consisting of conjunctions of atomic clock constraints on the values of clocks and some auxiliary time-stamp variables. Since convex sets are closed under projection the result will follow. For economy of notation we assume that ξ is such that each automaton $\mathcal{A}^i$ makes exactly k steps. The restriction of $\mathcal{A}^i$ to the states and transitions involved in ξ is of the form depicted in Figure 3.4, that is, $\mathcal{A}^i$ starts from q_0^i with invariant I_0^i, then takes a transition, guarded by g_1^i, labeled by a_1^i and involving reset r_1^i, to state q_1^i, etc.

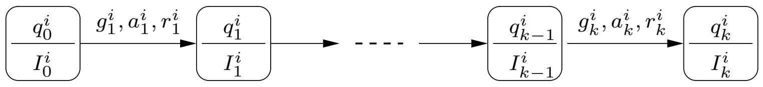


Figure 3.4: The part of $\mathcal{A}^i$ which participates in ξ^i.

As a first step we extend the description of local runs to include the *time stamps* of the transitions:

$$\xi^i :\ (q_0^i, v_0^i, t_0^i) \to (q_1^i, v_1^i, t_1^i) \to \cdots \to (q_{k-1}^i, v_{k-1}^i, t_{k-1}^i) \to (q_k^i, v_k^i, t_k^i).$$

Each t_j^i variable denotes the *absolute time* at which the corresponding transition has been taken. Every global run in $\langle\xi\rangle$ is *uniquely determined* by the values t_j^i and v_j^i for $i = 1..n$ and $j = 0..k+1$ and can be seen as a point in some space of appropriate dimension.

All the runs satisfy the natural local ordering among time stamps of transitions of the same automaton:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=0}^{k-1} (t_j^i \le t_{j+1}^i).$$

Those runs that are $\approx$-equivalent agree also on the linear ordering of all time stamps which characterize the particular interleaving (shuffle) of the local runs.

We can now proceed, progressively, to the logical characterization of the set of reachable configurations. We will use the following auxiliary notations and abbreviations:

- Global states after step j: $\mathbf{q}_j = (q_j^1, \dots q_j^n)$
- Global clock valuations after step j: $\mathbf{v}_j = (v_j^1, \dots v_j^n)$
- The set of local valuations appearing in a local run ξ^i: $\mathbf{v}^i = \{v_0^i, \dots, v_k^i\}$,
- The set of local time stamps appearing in a local run ξ^i: $\mathbf{t}^i = \{t_0^i, \dots, t_k^i\}$
- The set of all valuations: $\mathbf{v} = \bigcup_i \mathbf{v}_i$
- The set of all time stamps: $\mathbf{t} = \bigcup_i \mathbf{t}^i$.

The family of predicates $\{\Phi_j^i\}$ characterizes the clock values and time stamps in a valid step j of automaton $\mathcal{A}^i$. The predicate $\Phi_j^i(v_{j-1}^i, t_{j-1}^i v_j^i, t_j^i)$ holds if the j^{th} transition is taken at t_{j-1}^i and then time elapses until t_j^i. This is nothing but a recapitulation of the definition of a compound step, namely that the transition guard is satisfied, the reset takes place and the subsequent time passage does not violate the staying condition of q_j^i:

$$\Phi_j^i(v_{j-1}^i, t_{j-1}^i v_j^i, t_j^i) = \left(\begin{array}{l} v_{j-1}^i \models g_j^i \wedge \\ \exists v\ v = r_j^i(v_{j-1}^i) \wedge \\ t_{j-1}^i \leq t_j^i \wedge \\ v_j^i = v + t_j^i - t_{j-1}^i \wedge \\ v_j^i \models I_j^i \end{array} \right)$$

It is not hard to see that this condition defines a convex zone. Note that this definition is invariant under a shift of global time, in other words, $\Phi_j^i(v, t, v', t')$ is equivalent to $\Phi_j^i(v, t+d, v', t'+d)$ for every d. We can now define what constitutes a *valid run* of $\mathcal{A}^i$ *in isolation*, without taking into account synchronization constraints. We keep this definition shift-invariant as well and do not yet insist on the initial zone which is defined globally:

$$\Phi^i(t^i, v^i) = \bigwedge_{j=1}^{k-1} \Phi_j^i(v_{j-1}^i, t_{j-1}^i v_j^i, t_j^i)$$

The predicate which defines what constitutes a valid *global* run is a conjunction of the conditions for local runs with additional conditions that take care of all the synchronization aspects, including the fact that all runs start and terminate simultaneously. For every $a \in \Sigma$ let

$S_a = \{(i,j) \mid a^i_j = a\}$ be the set of steps that synchronize on a. To force all a-transitions to take place at *the same time* we define the predicate

$$\Psi_a(\mathbf{t}) = \bigwedge_{(i,j),(i',j') \in S_a} t^i_j = t^{i'}_{j'}.$$

The condition for a valid global run starting at Z_0 is then:

$$\Phi(\mathbf{t}, \mathbf{v}) = \left(\begin{array}{ll} t^1_0 = t^2_0 = \ldots = t^n_0 & \wedge \\ v_0 \in Z_0 & \wedge \\ \bigwedge_{i=1}^n \Phi^i(v^i, t^i) & \wedge \\ \bigwedge_{a \in \Sigma} \Psi_a(t) & \wedge \\ t^1_{k+1} = t^2_{k+1} = \ldots = t^n_{k+1} & \end{array} \right)$$

Note that the first and last conditions can be viewed as synchronization conditions for two additional fictitious transitions *start* and *end* in which all automata participate. This set is a convex subset of the space consisting of all valuations and time stamps in the run, and so is its projection on the set of valuations after the last step, which is the reachable set:

$$R_{Z,\langle\xi\rangle}(\mathbf{v}_k) \equiv \exists \mathbf{t} \exists \mathbf{v}_1, \ldots, \mathbf{v}_{k-1} \Phi(\mathbf{t}, \mathbf{v}_1, \ldots \mathbf{v}_{k-1}, \mathbf{v}_k).$$

$\square$

Note that an alternative (but longer) way to obtain the result would be to define $\Phi(\mathbf{t}, \mathbf{v})$ separately for each class of $\approx$, which will involve a specific linear order over all $\mathbf{t}$, and then the union over all these classes will eliminate these order constraints.

Let us mention that the result extends naturally to arbitrary "linear" hybrid automata with convex guards and invariants.

3.4 Application to Reachability Computation

We will now modify the standard reachability computation algorithm for timed automata (Algorithm 2.1) to take advantage of this result. The idea is to generate symbolic states in a *breadth-first* manner and at each level merge those reached by locally-equivalent runs, that is, by the same set of compound steps. Note the breadth-first exploration is suitable here because it groups together symbolic states reached by the same number of discrete transitions, and only such states can potentially be equivalent and hence be merged.

Let us note that this merging can be done progressively and we need not wait until the whole diamond is generated. Suppose we have three automata and three independent transitions a, b and c. We first compute Z_a, Z_b and Z_c. From there we compute their successors Z_{ab}, Z_{ac}, Z_{ba}, Z_{bc}, Z_{ca} and Z_{cb}. At this stage we can merge three pairs of zones:

$$Z_{a||b} = Z_{ab} \cup Z_{ba} \qquad Z_{a||c} = Z_{ac} \cup Z_{ca} \qquad Z_{b||c} = Z_{bc} \cup Z_{cb},$$

compute their successors and then merge them all into one zone

$$Z_{a||b||c} = Z_{(a||b)c} \cup Z_{(a||c)b} \cup Z_{(b||c)a}.$$

To identify zones that can be merged we need to decorate symbolic states with (partially ordered) path information.

DEFINITION 3.3 (Shuffle Expression) *A shuffle expression over a distributed alphabet* $\Sigma = \Sigma^1 \cup \ldots, \cup \Sigma^n$ *is*

$$\alpha = \alpha^1 \parallel \ldots \parallel \alpha^n$$

with $\alpha^i \in (\Sigma^i)^*$. *Concatenation of a shuffle expression and a symbol* a *is defined as*

$$(\alpha^1 \parallel \ldots \parallel \alpha^n) \cdot a = (\beta^1 \parallel \ldots \parallel \beta^n)$$

where

$$\beta^i = \begin{cases} \alpha^i & \textit{if } a \notin \Sigma^i \\ \alpha^i \cdot a & \text{otherwise.} \end{cases}$$

Algorithm 3.1 performs breadth-first exploration and stores symbolic states together with their shuffle expressions. At each level it scans the newly-generated zones stored in *New* and performs the *Merge* operation which replaces every set of symbolic states of the form

$$\{(q, Z_1, \alpha), \ldots, (q, Z_m, \alpha)\}$$

by a single state (q, Z, α) where Z is the convex hull of all these zones. From Theorem 3.1 it follows that Z is exactly the union of the zones. Note that the path labels of a zone need not be kept after its successors have been computed. This also guarantees termination due to the finite number of zones.

Let us illustrate Theorem 3.1 and Algorithm 3.1 through an example. Consider the product of the two timed automata of Figure 3.5. The standard reachability algorithm will generate for these automata the 19-state simulation graph depicted in Figure 3.6. If we merge the zones

Algorithm 3.1 Breath-first forward reachability with merging.

```
$Explored := \emptyset$
$New := \emptyset$
$Waiting := \{(q_0, \perp, \varepsilon \parallel .. \parallel \varepsilon)\}$
while $(Waiting \neq \emptyset)$ do
    for each $(q, Z, \alpha) \in Waiting$ do
        for each $(q, g, a, \gamma, q') \in \Delta$ do
            $New := New \cup (succ^a(q, Z), \alpha \cdot a)$
        $Explored := Explored \cup (q, Z)$
    $New := Merge(New)$
    $Waiting := New \backslash Explored$
    $New := \emptyset$
return $(Explored)$
```

reachable by locally-equivalent runs, we obtain the reduced graph of Figure 3.7 which has only 9 states. Figure 3.8 shows how the reduced simulation graph is generated by Algorithm 3.1 as an alternating sequence of successor computation and zone merging.

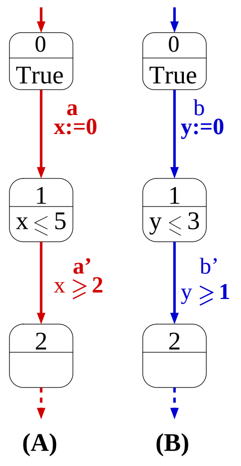


Figure 3.5: A product of two timed automata.

3.5 Experimental Results

We have implemented Algorithm 3.1 inside the IF toolbox. To confirm the complexity reduction empirically we have first tested the algorithm products of chain-like automata. Such automata are notorious for generating state explosion due to interleaving. We have considered two simple families of synthetic benchmarks shown in Figure 3.9. The first consists of parallel compositions of n independent *reset sequences* of length m each. The second class consists of parallel

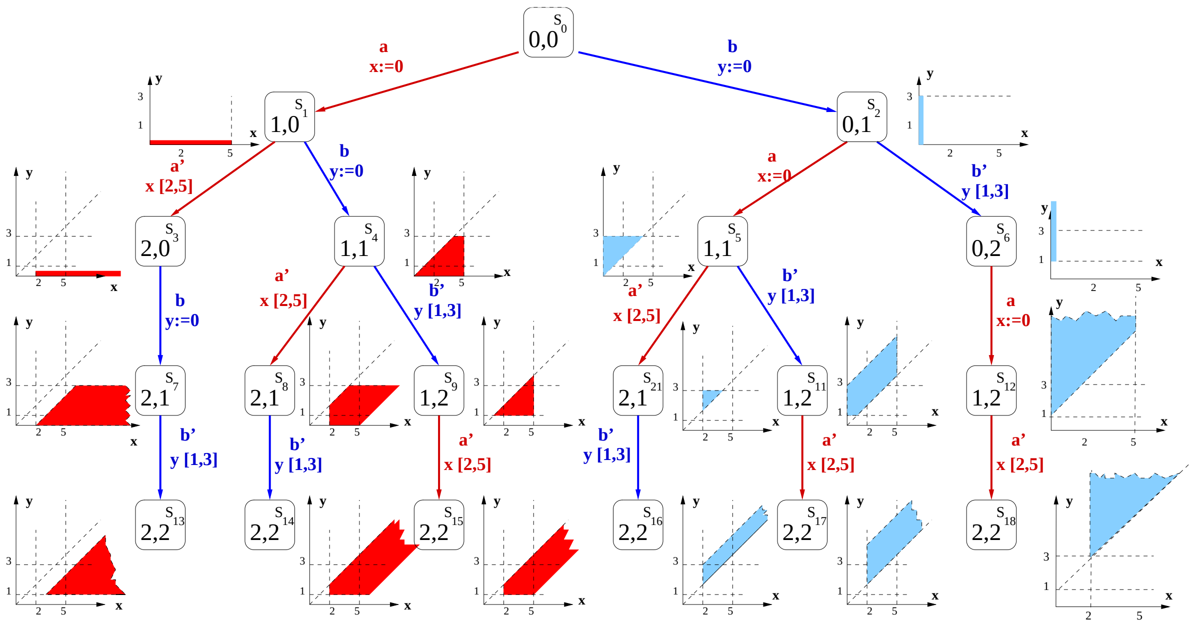


Figure 3.6: The reachability graph for the automata of Figure 3.5 generated by the standard algorithm. Each symbolic state s_i is represented by its discrete state $q_i = (q_i^A, q_i^B)$ and, graphically, by its zone Z_i.

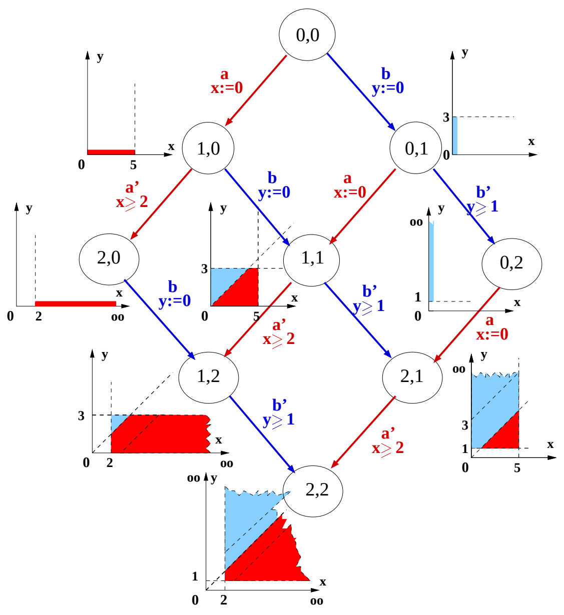


Figure 3.7: A reduced reachability graph for the system of Figure 3.5.

compositions of k independent synchronization chains, each being a parallel composition of n *synchronized sequences* of length m. A synchronized sequence ($\mathcal{A}^{ij}$) alternates between actions that synchronize with the left ($a_{i,j}$) and the right ($a_{i+1,j}$) neighbor while separating them by at least 4 time units.

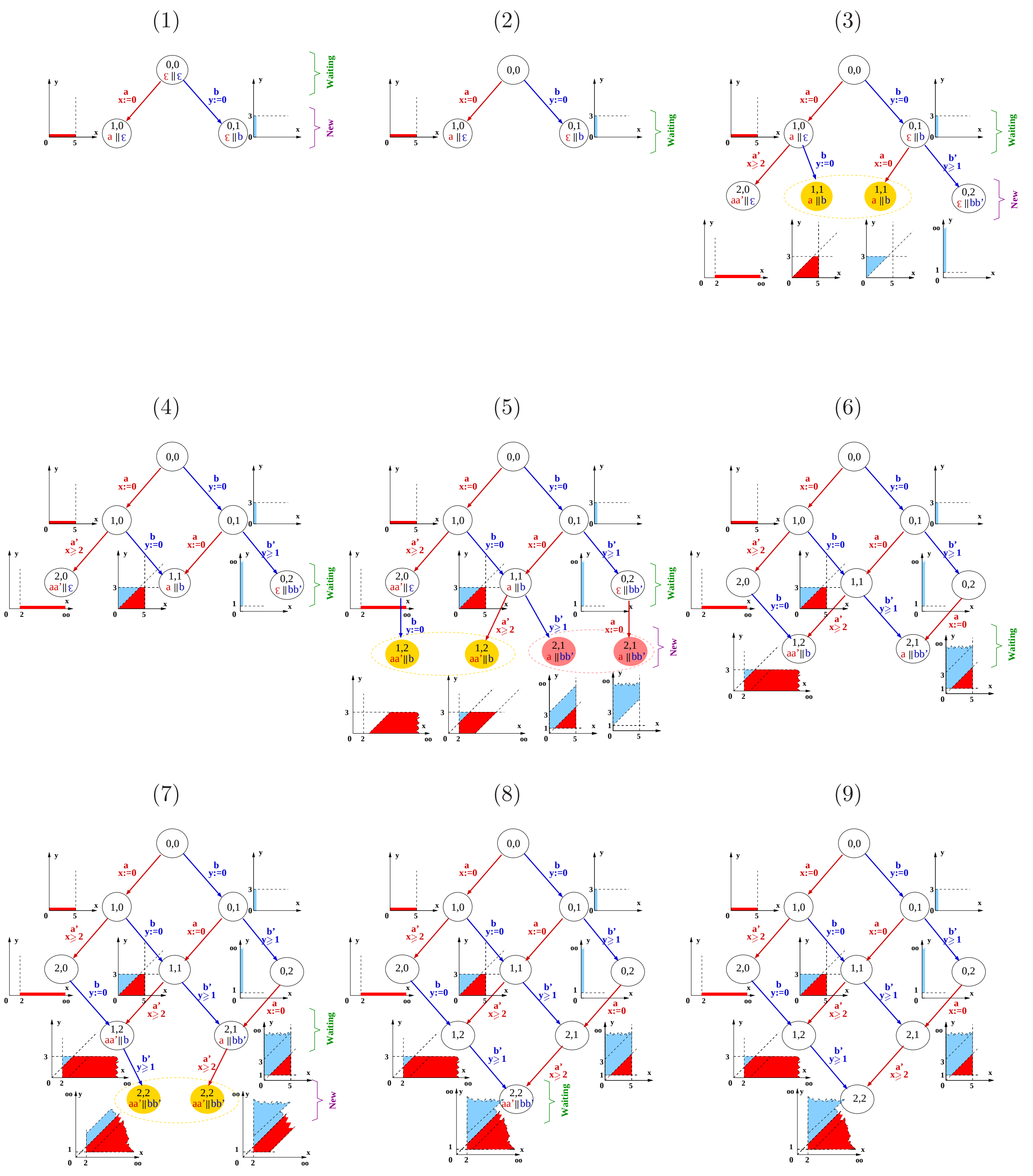


Figure 3.8: The reduced simulation graph of Figure 3.7 as generated on the fly by Algorithm 3.1.

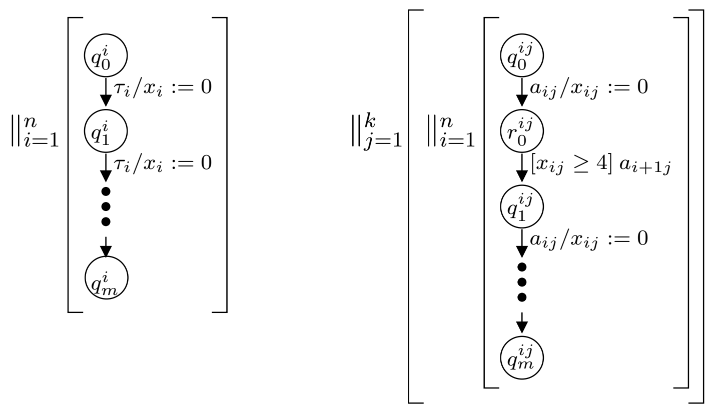


Figure 3.9: The structure of the synthetic benchmarks.

	n=2	n=4	n=6	n=8	n=10
Independent reset sequences					
m=1	5 / 4	65 / 16	1957 / 64	109601 / 256	$\perp$ / 1024
m=2	13 / 9	633 / 81	75973 / 729	$\perp$ / 6561	$\perp$ / 59049
m=3	25 / 16	2713 / 256	732529 / 4096	$\perp$ / 65536	$\perp$ / $\perp$
Synchronization chains $k = 1$					
m=1	4 / 4	6 / 6	8 / 8	10 / 10	12 / 12
m=2	8 / 8	37 / 17	236 / 30	1600 / 47	10949 / 68
m=3	12 / 12	86 / 32	1441 / 72	30841 / 140	660615 / 244
Synchronization chains $k = 3$					
m=1	2012 / 64	812375 / 216	$\perp$ / 512	$\perp$ / 1000	$\perp$ / 1728
m=2	97142 / 512	$\perp$ / 4913	$\perp$ / 27000	$\perp$ / 103823	$\perp$ / 314432
m=3	745197 / 1728	$\perp$ / 32768	$\perp$ / 373248	$\perp$ / $\perp$	$\perp$ / $\perp$

Table 3.1: Experimental results on the synthetic acyclic benchmarks.

The experimental results obtained for the two benchmarks for different values of n, m and k are summarized in Table 3.1. Each entry in the table is of the form B/C where B is the number of symbolic states encountered in an ordinary breadth-first exploration, while C is the number of states explored by Algorithm 3.1. We limit ourselves to instances with less than 10^6 symbolic states, and use the $\perp$ symbol to denote the fact that this limit has been reached. Let us note that we achieve an exponential reduction both for the interleaving of *independent* actions (reset sequences) and for strongly-synchronized actions (a single synchronization chain with $k = 1$). The reduction is clearly much more impressive in the synchronized case, where reductions based on partial order or symmetry are not directly applicable.

We have implemented Algorithm 3.1 into the IF toolset [BSGS04] and tested its performance on several publicly-available benchmarks. Table 3.2 compares the performance of the new al-

Size	**Kronos**	**Uppaal**	**Uppaal-A**	**IF**	**IF-U**
2	-/-	-/0.01s	-/0.00s	29/0.003s	18/0.002s
3	-/-	-/0.03s	-/0.01s	165/0.01s	53/0.01s
4	752/-	-/0.23s	-/0.06s	1099/0.07s	164/0.03s
5	3552/-	-/5.09s	-/0.29s	8453/1.07s	527/0.04s
6	16320/-	-/310.97s	-/1.34s	74939/21.06s	1726/0.20s
7	73620/-	-/51598.17s	-/5.89s	762429/595.75s	5693/1.75s
8	$\perp/\perp$	$\perp/\perp$	-/25.83s	$\perp/\perp$	18792/5.73s
9	$\perp/\perp$	$\perp/\perp$	-/113.53s	$\perp/\perp$	61883/28.42s
10	$\perp/\perp$	$\perp/\perp$	-/498.88s	$\perp/\perp$	202994/367.76s
11	$\perp/\perp$	$\perp/\perp$	-/2525.31s	$\perp/\perp$	662873/4489.23s

Table 3.2: Results on the Fisher protocol benchmark. The Uppaal-A column corresponds to results obtained using the convex-hull approximation, while the IF-U column represents our new algorithm. Table entries represent the number of symbolic states and computation time. The symbol "-" means " not reported" (or "irrelevant" for the case of computation time on older computers) and $\perp$ means "too big".

gorithm on the Fisher mutual-exclusion protocol benchmark with other reported results. We compare with old Kronos results reported in [Tri98], Uppaal results reported in [Upp] and results obtained with IF without using the new algorithm. It is interesting to note that although our new algorithm performs much better than the standard Uppaal machinery, their performances are similar when the convex-hull approximation option of the latter is employed. Our result shows that this "approximation" can be easily made exact.

3.6 Related Work

The application of partial-order techniques to timed systems has been subject to several publications [Rok94, RM84, YS97, DRGK98, BJLY98, Min99, Zha02, LNZ05, ZYN03]. Although Theorem 3.1 is related to some of the work mentioned in the next paragraph, it has not been stated explicitly and, moreover, its exploitation by a breadth-first version of the standard timed-automaton reachability algorithm has never been considered.

The algorithm of Rokicki [Rok94, RM84], using a variant of timed Petri nets is similar to ours by computing a zone which corresponds to the interleaving of independent transitions. This work which has been done in parallel with the development of the first verification tools for timed automata, used another terminology and has not proliferated to the timed automaton culture. Two more recent efforts, which are more ambitious with respect to "real" partial-order reductions, are those of Zhao [Zha02] and of Niebert et al. [LNZ05, ZYN03]. Both works use additional clocks in their algorithms and use zones over the extended clock space ("event zones" in the terminology of [LNZ05, ZYN03], "local successors" in the terminology of [Zha02]) that represent all configurations reached by interleavings of independent actions. We use the auxiliary clocks only in the proof of convexity which can be deduced via their results. It is worth noting that our result does not require independence of actions. It would be interesting to compare the reductions provided by the two approaches in terms of scope and performance.

An interesting idea which was first proposed in [BJLY98], inspired by distributed simulation, is to use *local time scales*, that is to compute successors for each automaton separately on its own clock subspace, and somehow combine these local zones upon synchronization. Although the idea is elegant, it suffers from several problems including the implicit global synchronization that takes place at time zero, and the fact that you need to augment each automaton with an additional clock that measures its corresponding total elapsed time. This idea, however, inspired our proof of convexity.

As a final remark, let us note that reducing the number of zones by taking their convex hull has been considered in the past [Tri98] but always as an *over-approximation*. We speculate that the reason for not discovering this result is due to the fact that the systems studied were cyclic, in which the same discrete state could be reached by different paths, not all of which being permutations of the same set of transitions. That is why the possibility of exact convex hull escaped the attention.

Part II

Variables Based Timed Automata and Component Based Systems

Chapter 4

Timed Automata with Discrete State Variables

In the rest of this thesis we will use timed digital circuits as a challenge ("killer application") for timed automata analysis technology. Circuits viewed at gate level may have an enormous number of state variables. For synchronous circuits viewed at the functional level, one needs a Boolean variable for every state-holding element. For asynchronous circuits, or for synchronous circuits viewed at the timed level of abstraction, the situation is worse: a variable is needed for every wire outgoing from a gate. Hence it is very natural to associate the states of the corresponding automata with valuations of these state variables. It is worth mentioning that the traditional theory of automata (and also that of timed automata) is based on a "flattened" state space which does not speak of variables nor has a Cartesian product structure. On the other hand, all practical approaches to verification (for example, a popular tool like SMV [McM92a]) do use a structured state space accessed via variables.

Another advantage of using variables explicitly is the fact that most of contemporary systems, hardware as well as software, are *component based*. A major concept within this framework is that of *encapsulation*, the abstraction of components behaviors to their *interfaces* in order to control the complexity of such systems. Using variables it is easier to speak of the restriction of a system to its interface and about other related notions such as composition, abstraction, projection and refinement. In this chapter we will present some definitions that will allow us to speak comfortably about timed automata as components.

Another deviation from the standard timed automaton model is that we use signals (rather than time-event sequences, see [ACM02]) to define the semantics of timed automata. Hence we will give here also the necessary definitions concerning signals over a set of variables. Most of the definitions apply to an arbitrary domain $\mathbb{D}$ but toward the end we will shift to the Boolean domain $\mathbb{B}$.

4.1 Variables, Valuations and Assignments

Let $X = \{x_1, \ldots, x_m\}$ be a finite set of variables ranging over a domain $\mathbb{D}$. An X-valuation is a function

$$\mathbf{v} : X \to \mathbb{D}$$

assigning to each x_i a value $\mathbf{v}(x_i)$. The set of such X-valuations, which are just elements of $\mathbb{D}^m$, is denoted by $\mathbf{V}_X$.

Every $X' \subseteq X$ defines a *projection* function

$$_{/X'} : \mathbf{V}_X \to \mathbf{V}_{X'}$$

such that $\mathbf{v}' = \mathbf{v}_{/X'}$ if $\mathbf{v}'(x) = \mathbf{v}(x)$ for every $x \in X'$.

In the automata to be defined later in this chapter, where states are associated with X-valuations, each transition corresponds to a change in one or more state variables that we formalize as *assignment functions*.

Definition 4.1 (Assignments) *Let* $\mathbf{v}$ *and* $\mathbf{v}'$ *be two* X*-valuations. The function*

$$f_{\mathbf{v}} : \mathbf{V}_X \to \mathbf{V}_X$$

is the constant function defined for every $\mathbf{u} \in \mathbf{V}_X$ *as* $f_{\mathbf{v}}(\mathbf{u}) = \mathbf{v}$. *The function*

$$f_{\mathbf{v},\mathbf{v}'} : \mathbf{V}_X \to \mathbf{V}_X$$

is such that for every $\mathbf{u}, \mathbf{u}' \in \mathbf{V}_X$, $\mathbf{u}' = f_{\mathbf{v},\mathbf{v}'}(\mathbf{u})$ *when*

$$\mathbf{u}'(x) = \begin{cases} \mathbf{v}'(x) & \text{if } \mathbf{v}(x) \neq \mathbf{v}'(x) \\ \mathbf{u}(x) & \text{if } \mathbf{v}(x) = \mathbf{v}'(x) \end{cases}$$

In other words, $f_{\mathbf{v},\mathbf{v}'}$ acts as the identity on all the variables on which $\mathbf{v}$ and $\mathbf{v}'$ agree and assigns to the other variables their value in $\mathbf{v}'$. Needless to say, $\mathbf{v}' = f_{\mathbf{v},\mathbf{v}'}(\mathbf{v})$. We denote the set of X-assignments by F_X. Given two valuations $\mathbf{v}, \mathbf{v}'$ we let $\rho(\mathbf{v}, \mathbf{v}')$ be the number of variables on which $\mathbf{v}$ and $\mathbf{v}'$ differ. We say that assignment $f_{\mathbf{v},\mathbf{v}'}$ is, respectively, *silent*, *single* or *multiple* if $\rho(\mathbf{v}, \mathbf{v}') = 0$, $\rho(\mathbf{v}, \mathbf{v}') = 1$ or $\rho(\mathbf{v}, \mathbf{v}') > 1$. Clearly, every multiple assignment can be written as a composition of single assignments.The projection $f_{/\mathbf{V}'}$ of an assignment f on a subset X' of the variables is defined naturally by restriction.

Examples Let $X = \{x_1, x_2, x_3, x_4\}$, let $\mathbf{v} = (1, 3, 2, 7)$ and $\mathbf{v}' = (2, 3, 5, 7)$. Then

$$f_{\mathbf{v}'} = \{x_1 := 2, x_2 := 3, x_3 := 5, x_4 := 7\}$$

and

$$f_{\mathbf{v},\mathbf{v}'} = \{x_1 := 2, x_3 := 5\}.$$

When we work in the Boolean domain we may use the simpler notation $x^{\downarrow}$ for $x := 0$ and $x^{\uparrow}$ for $x := 1$. Let $\mathbf{v} = (0, 1, 1, 0)$ and $\mathbf{v}' = (1, 1, 0, 0)$. Then

$$f_{\mathbf{v}'} = \{{x_1}^{\uparrow}, {x_2}^{\uparrow}, {x_3}^{\downarrow}, {x_4}^{\downarrow}\},$$

and

$$f_{\mathbf{v},\mathbf{v}'} = \{{x_1}^{\uparrow}, {x_3}^{\downarrow}\}.$$

In further sections, we will associate variable valuations with automaton states and assignments with transitions. Then, when composing automata we will need to check compatibility between valuations over two non-disjoint sets of variables.

Definition 4.2 (Compatibility) *Let X^1 and X^2 be two sets of variables and let $X = X^1 \cup X^2$ and $\underline{X} = X^1 \cap X^2$. Two valuations $\mathbf{v}^1 \in \mathbf{V}_{X^1}$ and $\mathbf{v}^2 \in \mathbf{V}_{X^2}$ are said to be compatible if they agree on shared variables*

$$\mathbf{v}^1_{/\underline{X}} = \mathbf{v}^2_{/\underline{X}}.$$

From two such compatible valuations we can naturally construct a joint valuation $\mathbf{v}^1 \parallel \mathbf{v}^2 : X \to \mathbf{V}_X$ which agrees with $\mathbf{v}^1$ and $\mathbf{v}^2$ on all variables. Likewise, two assignments $f^1 \in F_{X^1}$ and $f^2 \in F_{X^2}$ are compatible if

$$f^1_{/\underline{X}} = f^2_{/\underline{X}}$$

and one can construct from them a joint assignment $f^1 \parallel f^2 : \mathbf{V}_X \to \mathbf{V}_X$.

4.2 Signals

The behavior of a system defined over X is a function from the time domain to $\mathbf{V}_X$. In our case, the time domain is $\mathbb{R}_+$ and such behaviors are called *signals*.

Definition 4.3 (X-Valued Signals) *A signal over a set X of variables ranging over a discrete domain is a partial function*

$$\xi : \mathbb{R}_+ \rightarrow \mathbf{V}_X$$

whose domain of definition is some interval $[0, t)$, $t \in \mathbb{R}_+ \cup \{\infty\}$, which can be partitioned into a countable sequence of left-closed right-open intervals $\mathcal{J} = J_1, J_2, J_3, \ldots$ each of the form $J_k = [t_{k-1}, t_k)$, such that $\xi(\tau) = \xi(\tau') = \xi(J)$ if τ and τ' belong to the same interval J.

We use $|\xi| = t$ to denote the *duration* of the signal and say that ξ is finite if $|\xi| < \infty$ and infinite otherwise. The coarsest partition of $\mathbb{R}_+$ in which the uniformity condition holds is $\mathcal{J}_\xi$ satisfying $\xi(J_k) \neq \xi(J_{k+1})$. We will denote the set of finite and infinite X-signals by $\mathcal{S}^*(X)$ and $\mathcal{S}^\omega(X)$, and let $\mathcal{S}(X) = \mathcal{S}^*(X) \cup \mathcal{S}^\omega(X)$. The *concatenation* operation $\xi_1 \cdot \xi_2$ is defined naturally for any $\xi_1 \in \mathcal{S}^*(X)$, $\xi_2 \in \mathcal{S}(X)$, as it is defined for sequences.

The more commonly-used semantic domain for timed automata are the *time-event sequences*, consisting of *events* separated by time durations, which are also equivalent to the *timed traces* of [AD94] which are sequences of events with non-decreasing time stamps. Signals can be represented in two forms, one is *state based* and the other is *event based*, where the events correspond to changes in the signal value (assignments). In this form they can be viewed as a special subclass of time-event sequences in which events $x^\uparrow$ and $x^\downarrow$ that specify changes in the same variable should satisfy additional constraints, for example, every two occurrences of $x^\uparrow$ should be separated by an occurrence of $x^\downarrow$. More theoretical background on signals and time-event sequences can be found in [ACM02].

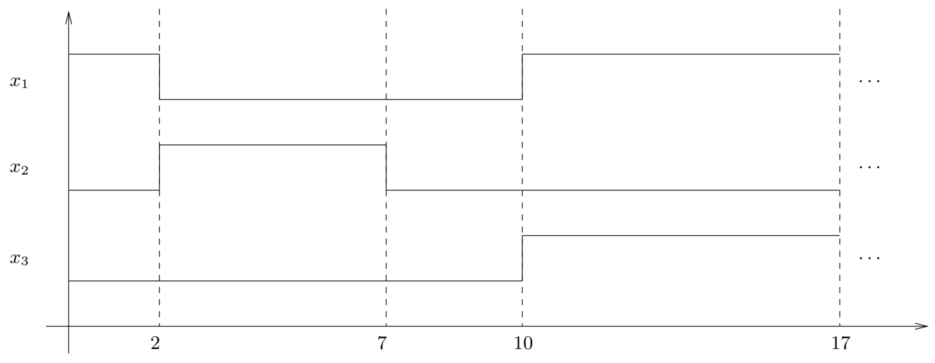


Figure 4.1: A signal over the Boolean variables $\{x_1, x_2, x_3\}$.

State-based Representation of Signals We will use the notation

$$\mathbf{v}_1^{r_1} \cdot \mathbf{v}_2^{r_2} \cdot \mathbf{v}_3^{r_3} \cdots$$

to denote a signal ξ with $\mathbf{v}_k = \xi(J_k)$ and $r_k = |J_k| = t_k - t_{k-1}$, that is a signal whose value is $\mathbf{v}_1$ for duration r_1, then $\mathbf{v}_2$ for duration r_2, etc. The signal appearing in Figure 4.1 will be written in a state-based form as:

$$\xi = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}^2 \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}^5 \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}^3 \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}^7 \cdots, \tag{4.1}$$

Event-based Representation of Signals A signal of the form $\mathbf{v}_1^{r_1} \cdot \mathbf{v}_2^{r_2} \cdot \mathbf{v}_3^{r_3} \cdots$ will be represented as

$$f_{\mathbf{v}_1} \cdot r_1 \cdot f_{\mathbf{v}_1,\mathbf{v}_2} \cdot r_2 \cdot f_{\mathbf{v}_2,\mathbf{v}_3} \cdot r_3 \cdots$$

The signal (4.1) can be written in an event-based form as:

$$\xi = \{{x_1}^{\uparrow}, {x_2}^{\downarrow}, {x_3}^{\downarrow}\} \cdot 2 \cdot \{{x_1}^{\downarrow}, {x_2}^{\uparrow}\} \cdot 5 \cdot \{{x_2}^{\downarrow}\} \cdot 3 \cdot \{{x_1}^{\uparrow}, {x_3}^{\uparrow}\} \cdot 7 \cdots$$

or as

$$\xi = \begin{pmatrix} \uparrow \\ \downarrow \\ \downarrow \end{pmatrix} \cdot 2 \cdot \begin{pmatrix} \downarrow \\ \uparrow \\ = \end{pmatrix} \cdot 5 \cdot \begin{pmatrix} = \\ \downarrow \\ = \end{pmatrix} \cdot 3 \cdot \begin{pmatrix} \uparrow \\ = \\ \uparrow \end{pmatrix} \cdot 7 \cdots$$

It is important to note that in this representation, an assignment which involves several variables is considered *instantaneous*, without any order imposed on the changes in the variables. However, when we consider signals associated with the behaviors of timed automata, an assignment may be realized by a *sequence* of transitions, each performing a single assignment on one variable. In this case the ordering of assignments should be consistent with a natural "causality" partial order among variables, where $x \prec x'$ should hold if x influences x', for example if x is an input variable and x' is a state variable for some automaton (see next section). Assuming a partial order $x_1 \prec x_2, x_1 \prec x_3$, the signal of (4.1), can be written as either

$${x_1}^{\uparrow} \cdot {x_2}^{\downarrow} \cdot {x_3}^{\downarrow} \cdot 2 \cdot {x_1}^{\downarrow} \cdot {x_2}^{\uparrow} \cdot 5 \cdot {x_2}^{\downarrow} \cdot 3 \cdot {x_1}^{\uparrow} \cdot {x_3}^{\uparrow} \cdot 7 \cdots$$

or

$${x_1}^{\uparrow} \cdot {x_3}^{\downarrow} \cdot {x_2}^{\downarrow} \cdot 2 \cdot {x_1}^{\downarrow} \cdot {x_2}^{\uparrow} \cdot 5 \cdot {x_2}^{\downarrow} \cdot 3 \cdot {x_1}^{\uparrow} \cdot {x_3}^{\uparrow} \cdot 7 \cdots$$

The *projection* $\xi' = \xi_{/X'}$ of an X-signal ξ on some $X' \subseteq X$ is defined naturally by either projecting the valuations in a state-based representation, or by deleting variables outside X' from the assignment in an event-based description. As discussed in the previous chapter in the context of local runs, such a projection may reduce the number of changes in the signal value and may result in a time partition $\mathcal{J}_{\xi'}$ coarser than $\mathcal{J}_{\xi}$. The projection of signal (4.1) on $\{x_1, x_3\}$, shown in Figure 4.2, is represented in a state-based form as

$$\xi/_{\{x_1,x_3\}} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}^2 \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix}^5 \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix}^3 \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix}^7 \cdots \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix}^2 \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix}^8 \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix}^7 \cdots,$$

and in an event-based form as

$$\xi/_{\{x_1,x_3\}} = \begin{pmatrix} \uparrow \\ \downarrow \end{pmatrix} \cdot 2 \cdot \begin{pmatrix} \downarrow \\ = \end{pmatrix} \cdot 5 \cdot \begin{pmatrix} = \\ = \end{pmatrix} \cdot 3 \cdot \begin{pmatrix} \uparrow \\ \uparrow \end{pmatrix} \cdot 7 \cdots \equiv \begin{pmatrix} \uparrow \\ \downarrow \end{pmatrix} \cdot 2 \cdot \begin{pmatrix} \downarrow \\ = \end{pmatrix} \cdot 8 \cdot \begin{pmatrix} \uparrow \\ \uparrow \end{pmatrix} \cdot 7 \cdots$$

In this projection the assignment which involved only a change in x_2 became silent.

The *untiming* of a signal is the sequence

$$\mu(\xi) = \mathbf{v}_1 \cdot \mathbf{v}_2 \cdot \mathbf{v}_3 \cdots \equiv f_{\mathbf{v}_1} \cdot f_{\mathbf{v}_1,\mathbf{v}_2} \cdot f_{\mathbf{v}_2,\mathbf{v}_3} \cdots$$

which is similar to the definition of *qualitative behavior* given in the previous chapter. The untiming of the signal (4.1) will be represented as:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \cdots$$

in the state-based representation, and as

$$\begin{pmatrix} \uparrow \\ \downarrow \\ \downarrow \end{pmatrix} \cdot \begin{pmatrix} \downarrow \\ \uparrow \\ = \end{pmatrix} \cdot \begin{pmatrix} \downarrow \\ = \\ = \end{pmatrix} \cdot \begin{pmatrix} \uparrow \\ = \\ \uparrow \end{pmatrix} \cdots$$

or

$$x_1{}^{\uparrow} \cdot x_2{}^{\downarrow} \cdot x_3{}^{\downarrow} \cdot x_1{}^{\downarrow} \cdot x_2{}^{\uparrow} \cdot x_2{}^{\downarrow} \cdot x_1{}^{\uparrow} \cdot x_3{}^{\uparrow} \cdots$$

in the event-based representation.

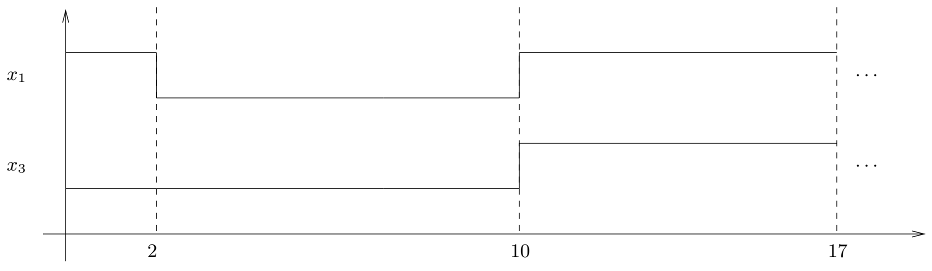


Figure 4.2: The signal of Figure 4.1 projected on $\{x_1, x_3\}$.

Finally let us mention a special class of signals called *ultimately constant* which are signals that at some point "stabilize"and do not change their values. Such signals can be written as

$$\mathbf{v}_1^{t_1} \cdot \mathbf{v}_2^{t_2} \cdots \mathbf{v}_k^{\infty} \quad \text{or} \quad f_{\mathbf{v}_1} \cdot t_1 \cdot f_{\mathbf{v}_1,\mathbf{v}_2} \cdot t_2 \cdots f_{\mathbf{v}_{k-1},\mathbf{v}_k} \cdot \infty.$$

The event-based representation of signals will allow us to stay close to the traditional definitions of timed automata where input and output values are associated with *transitions* rather than giving new definitions where these values are associated with *states*.

4.3 Timed Automata with Variables

In this section we associate every timed automaton with a set X of variables, partitioned into disjoint subsets of *input* and *state* variables $X = X_{in} \uplus X_{st}$. The difference between these two classes of variables is that those in X_{in} are controlled by the *external environment* and the automaton can follow their evolution passively, and only react to changes in their values. On the other hand, variables in X_{st} are the variables "owned" *exclusively* by the automaton which controls their values. A subset $X_{ou} \subseteq X_{st}$ of the state variables, called *output* variables, are observable to the outside world and can serve as input variables for other components. The *interface* variables of the automaton are its input and output variables $X_{io} = X_{in} \uplus X_{ou}$.

The connection between states and variable valuations is defined via the function

$$\lambda : Q \to \mathbf{V}_X$$

associating with every discrete state q of the automaton *one* X-valuation $\mathbf{v} = \lambda(q)$. The projections of λ on the input, state, output and interface variables are denoted by λ_{in}, λ_{st}, λ_{ou}, and λ_{io} respectively.

There are two points worth mentioning: 1) we will allow multiple states to be mapped to the *same* valuation. This is to accommodate for future use in interpreted timed automata where the reachability graph of an automaton may have several symbolic states corresponding to the same discrete state; 2) We will consider the value of input variables to be part of the state, hence *every change* in the input will cause a transition to a different state, and there will be only one valuation associated with each state. This amounts to composing every automaton with an unconstrained generator of its inputs. At a first glance this looks like an unnecessary blow up of the state space, but, when analyzed, the automaton should be composed anyhow with a generator of its inputs, so this overhead is an illusion.

DEFINITION 4.4 (Timed Automata over Variables) *A timed automaton over a set X of variables is a triple $\mathcal{A}_X = (\mathcal{A}, X, \lambda)$ where $\mathcal{A} = (Q, F_X, q_0, C, I, \Delta)$ is a timed automaton, X is a set of variables, and $\lambda : Q \to \mathbf{V}_X$ is a function mapping states to X-valuations. This function induces a mapping between transition labels and X-assignments where every transition (q, a, g, γ, q') such that $\lambda(q) = \mathbf{v}$ and $\lambda(q') = \mathbf{v}'$, is labeled by $a = f_{v,v'}$.*

We require the timed automaton to be *input enabled*, that is, if from every state q and for every $x \in X_{in}$ there is a transition which changes the value of x, that is, a transition to a state q' such that $\lambda(q)(x) \neq \lambda(q')(x)$. This reflects the fact that the input variables are not controlled by the automaton.

The *semantics* of $\mathcal{A}_X$ is defined in terms of *steps* and *runs*, inherited from the definition of $\mathcal{A}$, and in terms of the signals *"carried"* by these runs. With each compound step

$$(q, v) \xrightarrow{f_{\mathbf{v},\mathbf{v}'},t} (q', v')$$

between discrete states q and q' such that $\lambda(q) = \mathbf{v}$ and $\lambda(q) = \mathbf{v}'$, we associate the finite X-signal of duration t, represented, respectively, in state-based and event-based forms as

$$\mathbf{v}'^{t} \quad \text{and} \quad f_{\mathbf{v},\mathbf{v}'} \cdot t.$$

The signal carried by a run is the *concatenation* of the signals carried by its compound steps.

We denote by $[\![\mathcal{A}]\!] \subseteq \mathcal{S}(X)$ the set of signals carried by all runs of $\mathcal{A}$ and use $[\![\mathcal{A}]\!]_{in}$, $[\![\mathcal{A}]\!]_{ou}$ and $[\![\mathcal{A}]\!]_{io}$ for the sets of their projections on the respective sets of variables. In particular, $[\![\mathcal{A}]\!]_{io} \subseteq \mathcal{S}(X_{io})$ is called the *observable behavior* of $\mathcal{A}$. Likewise we will use $\mu([\![\mathcal{A}]\!])$ and $\mu([\![\mathcal{A}]\!]_{io})$ for the untiming of those signal languages (qualitative behaviors).

Let us remark that the observable behavior $[\![\mathcal{A}]\!]_{io}$ of an input-enabled automaton $\mathcal{A}$ can be seen also as a non-deterministic *causal* and *duration-preserving function* (transduction)

$$h_{\mathcal{A}} : \mathcal{S}(X_{in}) \to 2^{\mathcal{S}(X_{ou})},$$

which associates with each input signal one or more output signals. We use the relational definition of $[\![\mathcal{A}]\!]_{io}$, knowing that its projection on the input $[\![\mathcal{A}]\!]_{in}$ is the set of *all* input signals $\mathcal{S}(X_{in})$.

Given two timed automata $\mathcal{A}$ and $\mathcal{A}'$ having the same set X_{io} of interface variables, we say that $\mathcal{A}'$ is an *over approximation* of $\mathcal{A}$, denoted by $\mathcal{A} \sqsubseteq \mathcal{A}'$, if $[\![\mathcal{A}]\!]_{io} \subseteq [\![\mathcal{A}']\!]_{io}$. This will always be the case when $\mathcal{A}'$ is obtained from $\mathcal{A}$ by relaxing some of the timing constraints, or by merging two or more discrete states.

Composition

We will now define the composition of two timed automata with variable sets X^1 and X^2 such that

$$X^1_{st} \cap X^2_{st} = \emptyset.$$

In this setting, the interaction between components is realized via the *shared variables*

$$\underline{X} = X^1 \cap X^2 = (X^1_{in} \cap X^2_{in}) \uplus (X^1_{ou} \cap X^2_{in}) \uplus (X^2_{ou} \cap X^1_{in}).$$

When one automaton takes a transition that changes the value of a shared variable, the other automaton must take a similar transition as well. If the variable is an input for both automata, that is, $x \in X^1_{in} \cap X^2_{in}$, it remains an input of the composed system. On the other hand a variable is an output variable of one automaton and an input for another, it becomes a state variable of the composed system which controls its value.

A composition is *cyclic* is there are “loops” of influence between components:

$$(X^1_{out} \cap X^2_{in} \neq \emptyset) \wedge (X^2_{out} \cap X^1_{in} \neq \emptyset).$$

Cyclic composition may introduce undesired phenomena into the composed systems such as what is called an unstable behavior: spontaneous oscillations between values without any external stimulus. However it is well known that some of these loops are harmless and stable. Our implementation allows such cyclic compositions and uses the algorithm of [Mal94] to check stability. However, to simplify the presentation we will assume acyclic composition. Let us summarize the above by a definition.

DEFINITION 4.5 (Composition of X-Automata: Variable Classification) *The set of variables in a composition of two timed automata over variable sets X^1 and X^2, is $X = X^1 \cup X^2$, where*

$$X_{st} = X^1_{st} \uplus X^2_{st},$$

$$X_{in} = (X^1_{in} - X^2_{ou}) \cup (X^2_{in} - X^1_{ou}),$$

and

$$X_{ou} \subseteq X^1_{ou} \uplus X^2_{ou}.$$

Note that the definition of X_{ou} is not unique, allowing it to be any subset of $X^1_{ou} \uplus X^2_{ou}$. This reflect a liberty in the design of a composed system concerning which variables that will be exposed to the external world and which will be hidden. We do not enter this choice of X_{ou} to the formal definition of the composition in order to keep it associative. When we apply it to circuits we compose all the automata together and specify explicitly the set of output variables of the whole product.

The following definition identifies potential global states of the product as pairs of states whose valuations are compatible.

DEFINITION 4.6 (Consistent Global States) *Let $\mathcal{A}_{X^1} = (\mathcal{A}^1, X^1, \lambda^1)$ and $\mathcal{A}_{X^2} = (\mathcal{A}^2, X^2, \lambda^2)$ be two timed automata. We say that a pair of states $(q^1, q^2) \in Q^1 \times Q^2$ is consistent if $\lambda^1(q^1)_{/\underline{X}} = \lambda^2(q^2)_{/\underline{X}}$.*

We denote the set of consistent global states by $\widehat{Q^1 \times Q^2}$.

DEFINITION 4.7 (Composition of Timed Automata with Variables) *Let $\mathcal{A}_{X^1} = (\mathcal{A}^1, X^1, \lambda^1)$ with $\mathcal{A}^1 = (Q^1, F_{X^1}, q_0^1, C^1, I^1, \Delta^1)$ and $\mathcal{A}_{X^2} = (\mathcal{A}^2, X^2, \lambda^2)$ with $\mathcal{A}^2 = (Q^2, F_{X^2}, q_0^2, C^2, I^2, \Delta^2)$, be two timed automata with variables, such that (q_0^1, q_0^2) is consistent. Their composition $\mathcal{A}_{X^1} \parallel \mathcal{A}_{X^2} = \mathcal{A}_X = (\mathcal{A}, X, \lambda)$ with $\mathcal{A} = (Q, F_X, q_0, C, I, \Delta)$ where*

- *The set of variables is $X = X^1 \cup X^2$ classified according to Definition 4.5*
- *The set of discrete states is $Q = \widehat{Q^1 \times Q^2}$*
- *The initial state is (q_0^1, q_0^2)*
- *The set of clocks is $C = C^1 \uplus C^2$*
- *The staying condition I is defined for every global state as $I((q^1, q^2)) = I^1(q^1) \wedge I^2(q^2))$*
- *The transition relation consists of all transitions of the form*

  $$((q^1, q^2), f, g, \gamma, (q'^1, q'^2))$$

  for which one of the following holds:

  1. *$q'^2 = q^2$ and there is a transition $(q^1, f, g, \gamma, q'^1) \in \Delta^1$ such that $f_{/\underline{X}}$ is the identity function*
  2. *$q'^1 = q^1$ and there is a transition $(q^2, f, g, \gamma, q'^2) \in \Delta^2$ such that $f_{/\underline{X}}$ is the identity function*
  3. *there are two transitions $(q^1, f^1, g^1, \gamma^1, q'^1) \in \Delta^1$ and $(q^2, f^2, g^2, \gamma^2, q'^2) \in \Delta^2$ such that $f^1_{/\underline{X}} = f^2_{/\underline{X}}$ is not the identity function, $f = f^1 \parallel f^2$, $g = g^1 \wedge g^2$ and $\gamma = \gamma^1 \circ \gamma^2$*
- *The mapping from states to variables $\lambda : Q \to X$ is defined for every global state as $\lambda((q^1, q^2)) = \lambda^1(q^1) \parallel \lambda^2(q^2)$.*

The first two cases in the definition of Δ correspond to local transitions of $\mathcal{A}^1$ and $\mathcal{A}^2$, transitions that do not touch shared variables. To these transitions we apply the interleaving semantics, letting them be taken independently. The third case corresponds to two transitions, one in each automaton, whose respective assignments change the values of some shared variables in a consistent way. Hence transitions preserve state consistency.

We mention a special class of timed automata which is closed under acyclic composition.

DEFINITION 4.8 (Input-Dependent Timed Automata) *A timed automaton $\mathcal{A}$ is input dependent if every non-trivial cycle in its transition graph involves at least one transition which changes the value of some input variable.*

Proposition 4.1 (Closure under Acyclic Composition) *If both $\mathcal{A}_{X^1}$ and $\mathcal{A}_{X^2}$ are input-dependent, so is their acyclic composition $\mathcal{A} = \mathcal{A}_{X^1} \parallel \mathcal{A}_{X^2}$.*

PROOF 4.1 *Suppose, without loss of generality, that $X^1_{ou} \cap X^2_{in} \neq \emptyset$. Suppose $\mathcal{A}$ does have a non-trivial cycle without change in its input. Then, at least one of its projections on $\mathcal{A}^1$ or $\mathcal{A}^2$ must be a non-trivial cycle. Such a cycle in $\mathcal{A}^1$ will contradict the fact that $\mathcal{A}^1$ is input-dependent. Otherwise, the projection on $\mathcal{A}^1$ is a trivial cycle without a change in its output, and the induced cycle in $\mathcal{A}^2$ contradicts the fact that $\mathcal{A}^2$ is input-dependent.* □

Proposition 4.2 (Input-Dependent Automata and Ultimately-Constant Inputs) *Let $\mathcal{A}_X$ be an input-dependent timed automaton. Then for every ultimately-constant input signal ξ_{in}, any output signal $\xi_{ou} \in h_{\mathcal{A}_X}(\xi_{in})$ is ultimately constant.*

PROOF 4.2 *Suppose the contrary. Then the run carrying ξ_{ou} (and ξ_{in}) should exercise a non-trivial cycle infinitely-many times. Since the automaton is input-dependent, this will require infinitely-many changes in ξ_{in} which contradicts the assumption about its being ultimately-constant.*

Chapter 5

Timed Circuits

Modern circuits may contain up to hundred million transistors. The development of computer-based tools and methodologies to facilitate the design of such enormously-complex systems is at the center of the EDA (Electronic Design Automation) industry which involves today around billion dollars a year and more than 20,000 employees, not counting those working inside the semiconductor industry itself. This thesis is concerned with two topics related to circuit design and analysis methodology. The first topic is that of *timing*, the interaction between the *physical* properties of the circuit and its *logical* functionalities. The main issue in timing is how to get more computing throughput from the circuit and how to coordinate the speeds of different parts of a complex system. The second topic is *modular* (*hierarchical, component-based*) design and analysis of circuits, which is the only way to manage their complexity. Our main contribution is in creating a computer-supported framework for handling timing aspects in a modular fashion.

The choice of digital circuits as principal experimental framework for evaluating our techniques seems to be good one. It allows us to generate large examples of arbitrary size and benefit from the simplicity to express systems defined over Boolean variables. We recall also that due to the prohibitive costs of post-fabrication bugs, hardware was an important driving force behind practical verification technology and we hope it will play a similar role in timing analysis. Nevertheless, we stress once more that the techniques developed here can be applied to systems whose basic building blocks are more complex than Boolean gates as well as systems whose components are realized in software. In fact, our methodology applies to *any* system consisting of interconnected components each computing some discrete function within some time delay.

The rest of this chapter is organized as follows. In Section 5.1 we discuss some trade-offs between circuit performance and the complexity of analyzing its behavior. In Section 5.2 we describe briefly some hierarchical design principles. Section 5.3 mentions various approaches for modeling delays in Boolean gates while Section 5.4 described the gate delay model that we use throughout the rest of this thesis and the way we represent circuits composed of such gates using timed automata. Finally, Section 5.5 demonstrates how existing timed-automata is used to analyze such circuits.

5.1 Circuit Timing

Digital circuits can be classified into two major categories. The first category includes *combinational* circuits, that is, logic circuits without feedback. Their output is a function of, and only of, the present input values. As such they obey simple laws and the analysis of their functionality is rather straightforward. Such circuits have no memory and cannot maintain a state. When the capability to store information is added, we speak about *sequential* circuits. In this class of circuits, the output depends not only on the *present* input values but also on their *history*.[1] For this kind of circuits, *timing* is an aspect of a major importance that has to be well studied and understood. To be more precise, timing is important as well for combinational circuits but it will not affect their logical functionality: by ignoring timing we still know *what* the circuit computes, if not *when*. Variations in component timing characteristics in sequential circuits may affect *what* they compute and the abstraction of timing aspects which makes the study of combinational circuits so simple, is, unfortunately, no longer possible. Consequently, no simple theory exists for describing and studying the behavior of this type of circuits in a *faithful and efficient* manner.

The most successful design and analysis approach for sequential circuits, and indeed the dominant approach to date, starts by breaking all feedback loops with registers and latches, which are circuit elements that store values presented at their inputs on a shared clock signal. This way the sequential circuit is divided into combinational sections, each of which is easily analyzed. In this approach, time is assumed to be quantized and the *global clock*, which is a *periodic signal*, controls the communication between those combinational sections. The computation of the output of every section must be completed within less than a fixed amount of time called the *clock period*. As long as this condition is met (ignoring certain other details), the circuit is guaranteed to work reliably.

Thanks to the simplicity of this approach, nearly all sequential logic today is *clocked* or *synchronous*. However, this design methodology suffers from many drawbacks, the major among them is the cost associated with the global, periodic clock. The fixed clock period of synchronous circuits is chosen as a result of *worst-case* timing analysis. It is not *adaptive* and

[1] Mathematically speaking, the difference between the two is similar to the difference between *Boolean functions* and *automata* over Boolean variables.

cannot exploit the fact that in many situations the average-case behavior is much better than the worst case.

Even a small circuit like the one of Figure 5.1-(a) can be used to illustrate some of the inefficient utilization of resources in clocked circuits. In this circuit the input is fed in parallel into two inverters, one slow with propagation delay 10 and one fast with delay 2. The output of both are fed into an AND gate with delay 10. The period of the *clock signal* should be greater than 20 time units, which is the maximal propagation delay from input to output.

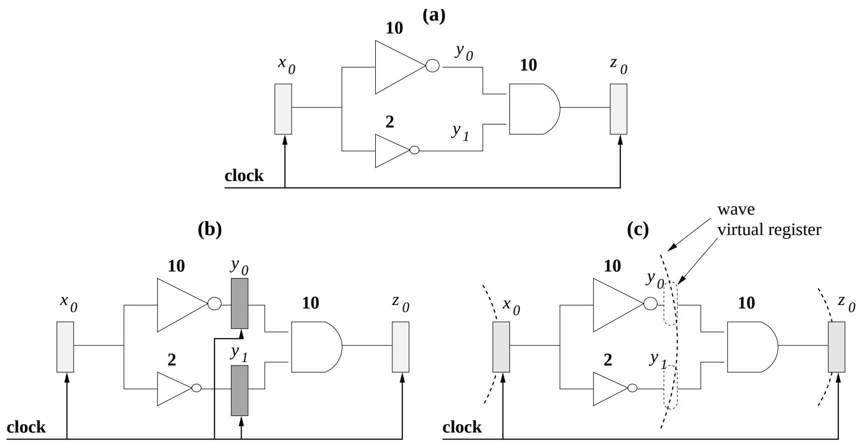


Figure 5.1: Example: (a) a clocked circuit; (b) a pipelined circuits; (c) a wave-pipelined circuits. The numbers indicate the propagation delays associated with the gates.

Many techniques have been introduced to improve the performance of synchronous designs. *Pipelining* is one of the most studied and applied techniques. It consists in inserting clocked registers at specific points of the circuit. The clock rate of the circuit given above, can double by introducing two registers as shown in Figure 5.1-(b), because once the inverters have propagated their values, they can start processing the next input. Another technique for improving the performance of clocked circuits is called *retiming* [LS91]. It consists of certain transformations that move latches across the circuit while preserving its behavior and possibly reducing the minimal clock period.

Wave-pipelining is another method for high-performance circuit design which implements pipelining in logic without the use of intermediate latches or registers. The idea was originally introduced by Cotton [Cot69], based on the observation that the rate at which logic signals can propagate through a circuit depends not on the longest path delay but on the difference between the longest and the shortest path delay. As a result, several computations, called *waves*, that is, logic signals related to *different* clock cycles, can propagate through the logic *simultaneously*. One can also view wave-pipeline as a kind of latch-free pipelining, in which each gate capacitances can serve as a virtual storage element. Figure 5.1-(c) illustrates the idea. Although wave-pipelining may potentially lead to more efficient designs, it is rarely used because the reliability of such circuits is hard to prove and in most (but not all) cases the gain in performance does not justify the complexity and cost of the design compared to classical synchronous logic.

Pipelining and related techniques can prevent certain parts of a circuit from being idle while they can do something useful, but they do not resolve the inherent problem associated with the *worst-case* reasoning underlying clocked circuits, illustrated again using the circuit of Figure 5.1-(a). The maximal propagation delay occurs when the input falls and the AND gate has to wait for the slow inverter to rise in order to start rising by itself. On the other hand, when the input rises, the fall of the fast inverter is sufficient for lowering the AND gate, hence a propagation delay is only 12, much less than the clock period. A clocked circuit cannot adapt to the actual rate and has to be ready for the worst case. Arithmetic circuit performance, for instance, is typically dominated by the propagation delay of carry signals, where the worst-case propagation situation rarely occurs.

In addition to the potential inefficiency of the computation, the maintenance of a global clock is associated with other severe problems such as clock skew, high power consumption, and could make it hard to compose systems.

Clock skew Clock skew is a phenomenon in synchronous circuits in which the clock signal arrives at different components at different times, which may produce a global dysfunction. This can be due to multiple causes, such as wire-interconnect length, temperature variations, capacitive coupling, and so on. As the clock rate of a circuit increases, less variation can be tolerated. This problem is usually hard to deal with and it is typically resolved at the expense of system performance.

Power consumption Synchronous circuits suffer also from an excessive consumption of power. In fact, the global clock is distributed over all the system, at each cycle every sub-system is activated even if it is not needed at a particular time.

Compositionality While efficient solutions are sometimes proposed to deal with power consumption or clock skew problems, compositionality which is a principal aspect of contemporary systems, is always hard to consider with synchronous logic. Indeed, composing modules running at different frequencies is extremely arduous if not impossible. When this is feasible, the speed of the whole system will be imposed by the clock rate of the slowest component, which often means a further decrease in performance.

An alternative *asynchronous* design methodology has been proposed since the first days of circuit design. The basic idea is that a component in the circuit which produces some data to be used by other components, activates these components upon completing its computation. These components, instead of looking periodically for their data, will be notified when the data is ready. This approach has several advantages:

1. The components are activated as soon as their data are ready;
2. The overhead of a central clock (in terms of power consumption and distribution over the circuit) is avoided;
3. Composition is much easier.

The downside of this nice and intuitive idea is that it may lead to unstable electrical behavior due to “races” among parallel processes that require additional circuitry in order to be avoided. Moreover, the mechanisms for coordination among components, such as *handshaking*, introduce a lot of communication overhead that should be added to the circuit, especially if one wants to follow a purely “speed-independent” approach where nothing is assumed about the relative speeds of the components. This overhead can be reduced significantly if timing is taken into account but, as mentioned before, timing of circuits with feedback is complex and cannot benefit from the efficient abstractions used for this purpose in the synchronous methodology. Consequently the design of asynchronous circuits is much more complex (for the same size) and they did not know the same success as their synchronous counterparts, except for some niches where power consumption is important and computations are sporadic rather than periodic.

The success of design based on synchronous logic is not due to circuit performance but mostly for its capacity to abstract time and thus to enormously reduce the complexity and the cost of the design. It is worth mentioning that a similar approach has been advocated by proponents of *synchronous languages* for developing real-time software. The so-called “synchrony hypothesis” according to which the computer is supposed to be much faster than its environment, allows the programmer to focus on the functionality of the software, and delegate the treatment of the timing aspects to others.

Static timing analysis is the most popular timing analysis methodology used in industry. The effectiveness of this technique is due to the abstraction of the functional aspect of the system, while focusing only on the quantitative timing aspect. In other words, a circuit is considered as a network of cells representing delays without logic functions. However, such technique can not be used for the analysis of asynchronous circuits because by separating the two aspects time and logic, several phenomena, such as hazards, become invisible in the model. Even for synchronous circuits, this separation between the two aspects, time and logic, is at the origin of several problems, such as *false paths* which can lead to an over-approximation of the minimal clock period and a decrease in system performance. In what follows, we will show how such problems completely disappear when we use a model which covers both logic and timing simultaneously.

As a final conclusion, existing frameworks for the design and analysis of timed circuit (and systems in general) are not completely finalized. They are often unable to offer at the same time precise modeling formalisms, as well as exhaustive, efficient and automatizable analysis methods.

5.2 Circuit Structure

As detailed in the section 5.2, digital circuits are usually defined hierarchically as component-based systems. Each component is a network of sub-components communicating through a finite set of shared boolean variables. Thus digital circuits expose the characteristics required by our platform. We will exploit then these characteristics in order to better model and analyze them.

Circuits seen at a low level are transistors through which electrical current flows controlled by the voltage present on their terminal wires. This current flow will produce voltage which can increase or decrease in certain specific points of the circuits. Starting from a certain level, depending on the considered technology, the voltage could be considered as *high* or *low*, and indicate then a valuation of a single Boolean variable, also known as a *bit*. By combining transistors in certain ways, one can build circuits realizing *Boolean functions*. Moreover, a given Boolean function can be built in many ways, depending on the considered technology, as well as on other requirements concerning power consumption, speed and size.

More complex Boolean functions can be built from the simpler ones by replicating the corresponding collection of transistors anywhere a basic Boolean function is required. On this new level, called the *logic level*, current and voltage notions are encapsulated in an idealized version of building blocks defining logic operations and basic logical functions. Gates are basic *standard cells* that could at their turn be composed to construct standard cells of much greater complexity, such as a 2-bit full-adder, or muxed D-input flipflop. Those standard cells could at their turn be composed to define even more complex cells, and so on. This *component-based design methodology* was responsible for allowing designers to scale ASICs (application-specific integrated circuits) from comparatively simple single-function circuits (of several thousand gates), to complex multi-million gate devices such as systems-on-a chip (SoC).

The structure of a circuit, could be regarded as a block diagram, a *graph* the nodes of which are components, and the edges, called *wires* represent the communication between the nodes. Component-based design is an approach to circuits development that relies on the concept of *reuse*. Each conceived circuit could be reused as an elementary building block in the design of more complex circuits. Such a component can appear more than once in a single circuit; we speak then about different *instances* of components of the same *type* or *class*. For example, the circuit A shown in Figure 5.2 consists of six components: three instances of the circuit B, two instances of the circuit C, and one instance of the circuit D. Circuits B, C, and D, at their turn, can be defined similarly as networks of more basic components. This recursive description of a circuit can be viewed as a tree (see Figure 5.3) the nodes of which are instances of circuits. The root of this tree is the *main* circuit, the leaves (called *primitive* or *basic* components) are logic gates, and the rest of the nodes are *intermediate* components.

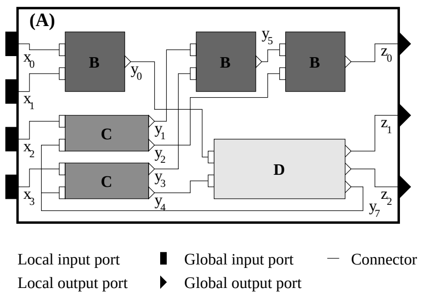


Figure 5.2: Circuit structure: Graph of components.

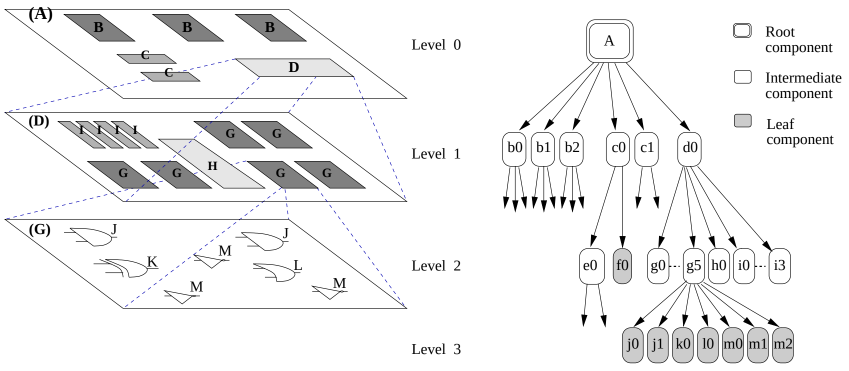


Figure 5.3: Hierarchical structure of a circuit.

Consider a set of interacting components $\mathcal{C} = \{C^1, \ldots C^k\}$ belonging to a level i in the hierarchy. The interaction between these components is via *wires* that connect some *output ports* of one component with some *input ports* of other components. The whole level can then be seen as a more complex component C at level $i-1$ whose set of input-output ports is subset of the ports of the basic components, those that can interact with other high-level components. Other ports of the components in $\mathcal{C}$ become *internal* as they are used for connecting these components at level i but are not visible for interaction at level $i-1$. Not surprisingly, this pattern of interaction is captured by the automata with variables introduced in the previous chapter where variables in X_{in} and X_{ou} represent, respectively, the input and output ports of the component, and composition with shared variables mimics the process of connecting two or more components via wires.

When a circuit is used as a component at a higher level, some abstraction of its internal working can be tolerated and only its interface behavior should be considered. However, such a restriction to the interface does not necessarily mean a significant reduction of its state space because any sequential function has its inherent state space structure through which it is realized. The only hope to obtain a smaller description of the reused component, is to give an *approximation* of its input-output behavior which will be sufficient for fulfilling its functionalities at the higher level. For example, a component may compute the values for two of its output ports in some order, but for the higher-level system this order may not be important. Naturally, such an approximation will be more “non deterministic” and will allow more behaviors than the exact model. Our goal is to to apply this idea of over-approximation to the *timing aspects* of the interface behavior of the component, and in order to do that we have to speak before on timing and delays in the context of the basic components.

5.3 Delays in Digital Circuits

When viewed as mathematical objects, Boolean functions, like any other class of functions, are *timeless*: when we say that $y = f(x)$, we say nothing about the time it takes to compute y once x is given. On the other hand, when we speak of *mechanisms* that compute functions, we need to take into consideration the fact that computation itself is a process that takes time. When computation is realized in software, its execution time is the accumulated duration of the instructions executed between its initiation and completion. When the computation is realized directly in hardware, the computation time, also known as the *propagation delay*, corresponds to the time that elapses between the moment a new input appears at the input ports of the circuit, until the corresponding new value of the function appears at the output ports.

The delay in circuits is due to the characteristics of transistors, of interconnecting wires and many other physical considerations. The estimation of propagation delays, by simulation or by analytic methods is a topic of ongoing research which is outside the scope of this thesis. We focus on the following question: *given* the delay characteristics of the basic components, for example Boolean gates, determine the timing properties of a large network built from such components.

There are many delay models used in academic and commercial tools such as PrimeTime or SPICE, differing from each other by their fidelity to the physical reality, their level of detail and by the complexity of their analysis. Simple *deterministic* delay models associate a single number d with a gate, indicating that the gate output will switch exactly d time after a change in the input. The most rudimentary models will assign the same delay value to all gate types while progressively more sophisticated models, may differentiate between gate types, between delay associated with rising and falling and may even go further and assign specific delay values to each *combination* of changes in the input variables. For example, it may be the case that an AND gate will rise at different speeds when its inputs move to 11 from 01 or from 10.

The advantage of deterministic delays is that they lend themselves easily to simulation, either in isolation or in parallel with transistor-level simulation and as such can be used to have a good feel concerning the behavior of the circuit. The disadvantages of this approach are evident: delays cannot be known precisely and may vary even among gates of the same type due to fabrication variations and the different electrical and physical contexts in which the gates operate. Hence simulation with deterministic delays will cover only one of the uncountable number of behaviors that the circuit may exhibit.

Other models cope with delay uncertainty by assigning to a gate an *interval* of possible delay values. This can be an interval of the form $[0, d_u]$ or, more interestingly, intervals of the form $[d_l, d_u]$ assigning both a lower and an upper bound on the propagation delay. In this model, known as the *bi-bounded delay* model [BS95], a change in the input of the gate which implies a change in its logical function[2] will propagate to the output port within some time $t \in [d_l, d_u]$. Like deterministic delay models, these models can be refined to distinguish between the delays associated with *rising* and *falling*, or between different combinations of input changes. Since our goal is to demonstrate scalability of our technique rather than solve a specific circuit problem, we will not go to this level of detail and use models in which there is one delay interval for rising and one for falling.

Allowing the *definition* of non-deterministic delays does not imply covering all their possible combinations. Using standard simulation tools, it is possible to simulate with different combinations of values taken from the delay intervals, for example combination of lower and upper bounds for each gate, but even the restriction to these "corner cases" cannot be covered by simulation as the number of combinations grows exponentially with the number of gates. Moreover, it is not clear that a bug, if exists, will be exhibited using one of these extreme values. The approach that we take is that of formal verification: using timed automata we cover *all* possible behaviors of the circuit under *any* combination of choices of delay for each gate in its corresponding interval. Of course, this generality has its high cost compared to deterministic simulation.

[2]Not all changes are like that of course, some do not imply a change in the output.

5.4 Circuit Model

We can now introduce the circuit model used in this thesis which is essentially the *inertial* bi-bounded delay model [Dil89, BS95]. Each elementary gate is decomposed into two parts, the first is an *instantaneous Boolean gate* whose output responds *immediately* to changes in its input. However this value is not visible to the external world but serves as an input to the second component, a *delay element* parameterized by the rising and falling delay intervals $[d_l^{\uparrow}, d_u^{\uparrow}]$ and $[d_l^{\downarrow}, d_u^{\downarrow}]$. The modeling of delay elements using timed automata, according to the proposal of [MP95a], is the cornerstone of our approach, that we illustrate via the AND gate model of Figure 5.4.

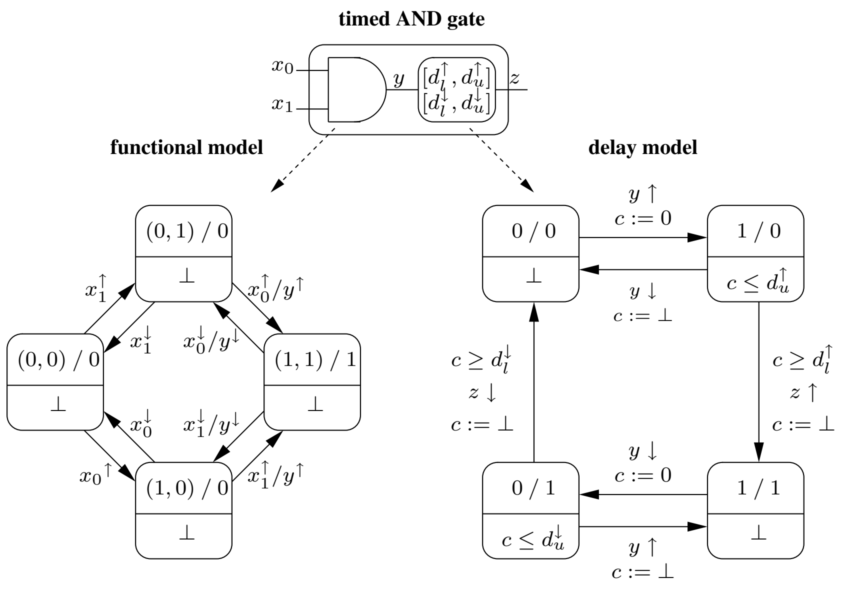

timed AND gate
x_0
x_1
y
$[d_l^{\uparrow}, d_u^{\uparrow}]$
$[d_l^{\downarrow}, d_u^{\downarrow}]$
z
functional model
delay model
$(0,1)\,/\,0$
$(0,0)\,/\,0$
$(1,1)\,/\,1$
$(1,0)\,/\,0$
$\perp$
$x_1^{\uparrow}$
$x_1^{\downarrow}$
$x_0^{\uparrow}/y^{\uparrow}$
$x_0^{\downarrow}/y^{\downarrow}$
$x_0^{\downarrow}$
$x_1^{\downarrow}/y^{\downarrow}$
$x_0^{\uparrow}$
$x_1^{\uparrow}/y^{\uparrow}$
$0\,/\,0$
$1\,/\,0$
$0\,/\,1$
$1\,/\,1$
$c \le d_u^{\uparrow}$
$c \le d_u^{\downarrow}$
$y \uparrow$ $c := 0$
$y \downarrow$ $c := \perp$
$c \ge d_l^{\downarrow}$ $z \downarrow$ $c := \perp$
$c \ge d_l^{\uparrow}$ $z \uparrow$ $c := \perp$
$y \downarrow$ $c := 0$
$y \uparrow$ $c := \perp$

Figure 5.4: A timed gate decomposed into a logical function and a delay element, and the automata corresponding the these components: logical function at the left and a delay element at the right. The states are labeled by pairs of the form $\mathbf{v}_{in}/\mathbf{v}_{ou}$ corresponding to valuations of input and output variables.

The automaton for the logical function is straightforward. It has two input variables x_0 and x_1 and one output variable y. The transitions (triggered by the input) to and from state $(1,1)/1$ change the value of the output variable. The automaton for the delay element has one input variable y and one output variable z. State $0/0$ is a *stable* state in which the input and output agree and the automaton may stay in this state indefinitely as long as the input does not change. Once the input rises, the automaton moves to the *excited* state $1/0$ and resets clock c to zero. This transition is not visible from the outside as the output variable remains low. The automaton may stay in this state as long as the clock has not reached the upper bound $d_u^{\uparrow}$, but may move to the stable state $1/1$ as soon as the clock reaches the lower bound $d_l^{\uparrow}$. This transition is visible from the outside as it changes the value of the output. The "dense" nondeterminism, the ability to switch or not to switch when the clock is in the interval $[d_l^{\uparrow}, d_u^{\uparrow}]$, will generate uncountably-many output signals for every input signal, to express delay uncertainty. If the inputs falls again before its value is propagated to the output, the automaton returns to stable state $0/0$. This way changes in the input that persist for less than $d_l^{\uparrow}$ time are "filtered" away and are not propagated to the output, those that persist for more than $d_u^{\uparrow}$ time *must* be propagated and those that last for some time between $d_l^{\uparrow}$ and $d_u^{\uparrow}$ may be propagated or not. Such a delay is called *inertial*[3] [BS95]. The behavior of the automaton when the input falls at $1/1$ is similar.

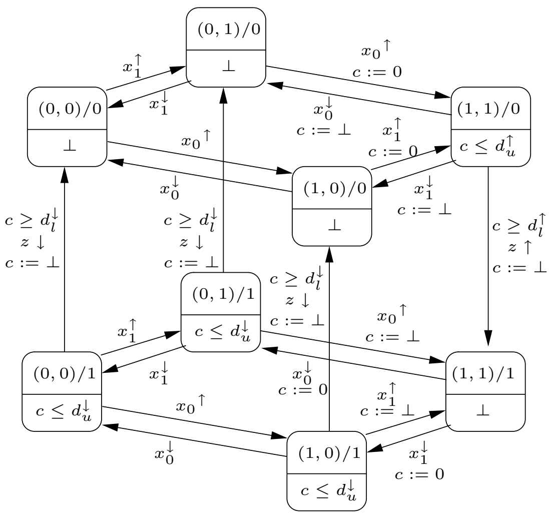

$(0,1)/0$
$\perp$
$x_0{}^{\uparrow}$ $c := 0$
$(0,0)/0$
$\perp$
$x_1^{\uparrow}$
$x_1^{\downarrow}$
$x_0^{\downarrow}$ $c := \perp$
$x_1^{\uparrow}$ $c := 0$
$(1,1)/0$
$c \leq d_u^{\uparrow}$
$x_0{}^{\uparrow}$
$x_0^{\downarrow}$
$(1,0)/0$
$\perp$
$x_1^{\downarrow}$ $c := \perp$
$c \geq d_l^{\downarrow}$ $z\downarrow$ $c := \perp$
$c \geq d_l^{\downarrow}$ $z\downarrow$ $c := \perp$
$c \geq d_l^{\uparrow}$ $z\uparrow$ $c := \perp$
$(0,1)/1$
$c \leq d_u^{\downarrow}$
$c \geq d_l^{\downarrow}$ $z\downarrow$ $c := \perp$
$x_1^{\uparrow}$
$x_0{}^{\uparrow}$ $c := \perp$
$(0,0)/1$
$c \leq d_u^{\downarrow}$
$x_1^{\downarrow}$
$x_0^{\downarrow}$ $c := 0$
$x_1^{\uparrow}$ $c := \perp$
$(1,1)/1$
$\perp$
$x_0{}^{\uparrow}$
$(1,0)/1$
$c \leq d_u^{\downarrow}$
$x_0^{\downarrow}$
$x_1^{\downarrow}$ $c := 0$

Figure 5.5: The timed automaton for an AND gate with propagation delay.

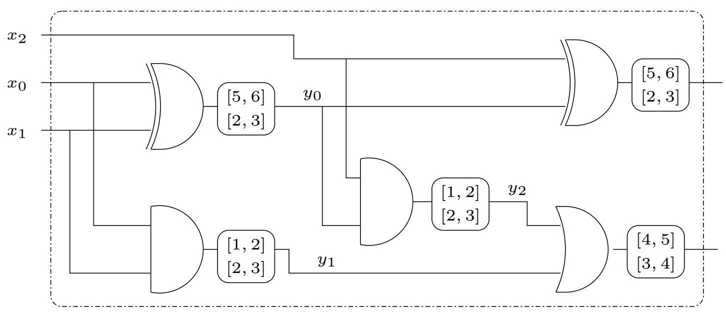
Composing these two automata we obtain the automaton of Figure 5.5 with input variables x_0 and x_1 and output variable z. This automaton is our model of an AND gate with propagation delay. The construction of an automaton for a whole circuit is done by a straightforward composition of the automata corresponding to its gates with shared variables for ports connected via wires. This way we obtain a model in which both *timing* and *logic* are coupled together, in contrast with approaches that separate them.

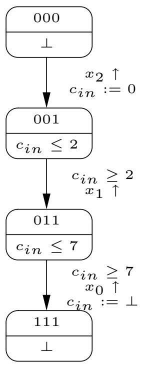
5.5 Analysis

Once an automaton for a circuit is constructed, it can be subject to the verification technology of timed automata described in Chapter 2. To illustrate how the analysis of a whole circuit looks like, consider the adder circuit of Figure 5.6-(a) consisting of two AND gates, two XOR gates

[3]One may think of other, more physically-correct, ways to model this situation, for example, moving to an error state or adding more clocks and states to model the return of an excited gate to its stable state.

and one OR gate, each type with its specific delay intervals. After composing these automata we obtain a global timed automaton with three input variables x_0, x_1 and x_2 and two output variables z_0 and z_1. At this point, the obtained automaton is completely *open* in the sense that there is no constraint whatsoever on the input signals which are allowed to be arbitrary. Of course, no circuit is supposed to function in the presence of this most general environment, and inputs are typically constrained both logically (for example, their values encode some restricted domain, or they follow some protocol) and in terms of timing (for example, bounded variability). These restrictions on the input are easily represented by a timed automaton (the *input generator*) which is composed with the circuit to obtain a model of the circuit with its environment. The size of the automaton obtained by composing the circuit automaton with a restricted input generator will typically (but not always) be much smaller than the automaton obtained by composing with the most general environment. This is because in *dense time*, the completely-unrestricted input generator can generate wild behaviors with unbounded variability.





(a) (b)

Figure 5.6: A circuit example: (a) a full adder; (b) an input generator.

To keep the presentation manageable we compose the circuit with a very simple input generator which generates only one input signal, a scenario in which x_2 rises at time zero, x_1 rises at time 2 and x_1 rises at time 7. This input is modeled by the automaton of Figure 5.6-(b). Although the input is deterministic, there will be a lot of non-determinism in the circuit behavior due to delay uncertainty. We compose the input model with the automaton for the circuit and run the forward reachability algorithm to obtain the interpreted timed automaton depicted in Figure 5.8. This automaton captures *all* the possible behaviors of the circuit in the presence of this particular input, under *all* possible combinations of delays for the individual gates. Let us stress that this type of analysis can be done with more general input models, including unrestricted input generators, for which the result will certainly be larger.

Each path in the automaton of Figure 5.8 corresponds to one qualitative behavior, a set of runs that go through the same sequence of transitions. All runs start from a stable state where all variables are low and all lead eventually to the same stable state where both output variables are high. To relate the runs to the behavior of the circuit let us look at two sample runs. The first run

$$\xi_1 : s_0 \xrightarrow{x_2{\uparrow}} s_1 \xrightarrow{2} s_1 \xrightarrow{x_1{\uparrow}} s_2 \xrightarrow{4} s_2 \xrightarrow{z_0{\uparrow}} s_3 \xrightarrow{1} s_3 \xrightarrow{x_0{\uparrow}} s_5 \xrightarrow{2} s_5 \xrightarrow{y_1{\uparrow}} s_7 \xrightarrow{4} s_7 \xrightarrow{z_1{\uparrow}} s_{11}$$

starts with the rising of x_2 which excites the XOR gate z_0, followed by the rising of x_1 which excites y_0. At time 6, z_0 rises, "choosing" it maximal propagation delay 6, followed by the rise of x_0 at 7. This rise aborts the excitation of y_0 as both inputs of the XOR gate are high. This abortion is possible because only 5 time units have elapsed since the excitation of y_0, which is less than its upper bound. The rise of x_0 also excites y_1 which rises after 2 more time units, exciting z_1 which finally rises at time 13, reaching the stable final state.

The second run

$$\xi_2 : \begin{array}{l} s_0 \xrightarrow{x_2{\uparrow}} s_1 \xrightarrow{2} s_1 \xrightarrow{x_1{\uparrow}} s_2 \xrightarrow{4} s_2 \xrightarrow{z_0{\uparrow}} s_3 \xrightarrow{1} s_3 \xrightarrow{y_0{\uparrow}} s_4 \xrightarrow{0} s_4 \xrightarrow{x_0{\uparrow}} s_6 \xrightarrow{2} s_6 \xrightarrow{y_1{\uparrow}} s_{10} \xrightarrow{1} \\ s_{10} \xrightarrow{y_2{\uparrow}} s_{15} \xrightarrow{1} s_{15} \xrightarrow{z_0{\downarrow}} s_{21} \xrightarrow{0} s_{21} \xrightarrow{y_0{\downarrow}} s_{28} \xrightarrow{3} s_{28} \xrightarrow{y_2{\downarrow}} s_{35} \xrightarrow{0} s_{35} \xrightarrow{z_1{\uparrow}} s_{31} \xrightarrow{2} s_{31} \xrightarrow{z_0{\uparrow}} s_{11} \end{array}$$

starts with the same timed sequence of events $x_2{}^{\uparrow}$, $x_1{}^{\uparrow}$ and $z_0{}^{\uparrow}$, but this time the excited y_0 chooses its minimal delay 5 and rises at time 7 just *before* the rise of x_0. We will soon see that contrary to naive intuition, this early stabilization of one gate may postpone the stabilization of the whole circuit. The rising of y_0 excites z_0 to fall and y_2 to rise while the rise of x_0 excites y_1 to rise and y_0 to fall. The subsequent events are the rise of y_1 which excites z_1 (as in ξ_1), the rise of y_2 and the fall of z_0. Only then y_0 falls exciting z_0 to rise again which happens, (after the rising of z_1) only at time 15. The signals corresponding to both runs are depicted in Figure 5.7.

This example is a small illustration of the complexities associated with a network of timed components working in parallel. It also shows the impressive analysis power of timed automata. An automaton like the one in Figure 5.8 tells us *everything* we might want to know about the behavior of the circuit coupled with a given environment, including which sequences of events are possible and the maximal stabilization time (the duration of the longest run). It can serve as well as a basis for checking properties expressed in real-time temporal logics or as "public" characterization of the circuit. The only remaining question is that of *scalability*: how far can we go in terms of circuit size, and in terms of environment complexity? The answer is "not very". The ability to produce reachability graphs for timed automata is restricted to some dozens of gates for acyclic input generators like the one presented in this example, and much less so for cyclic inputs. This is the problem we are going to attack in the next two chapters.

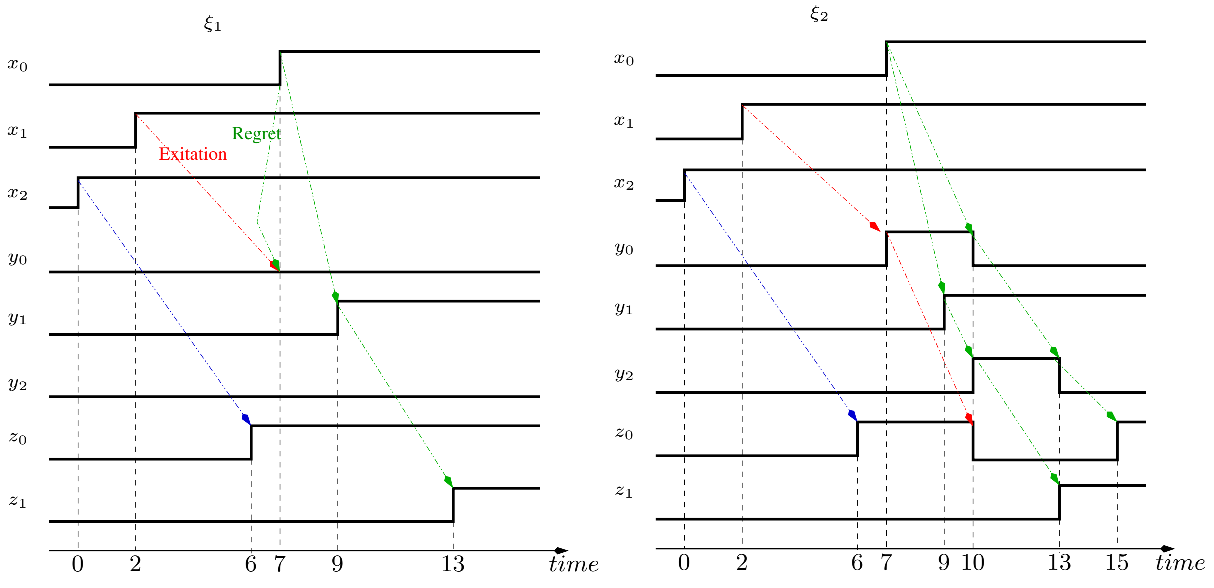


Figure 5.7: The signals associated with runs ξ_1, and ξ_2.

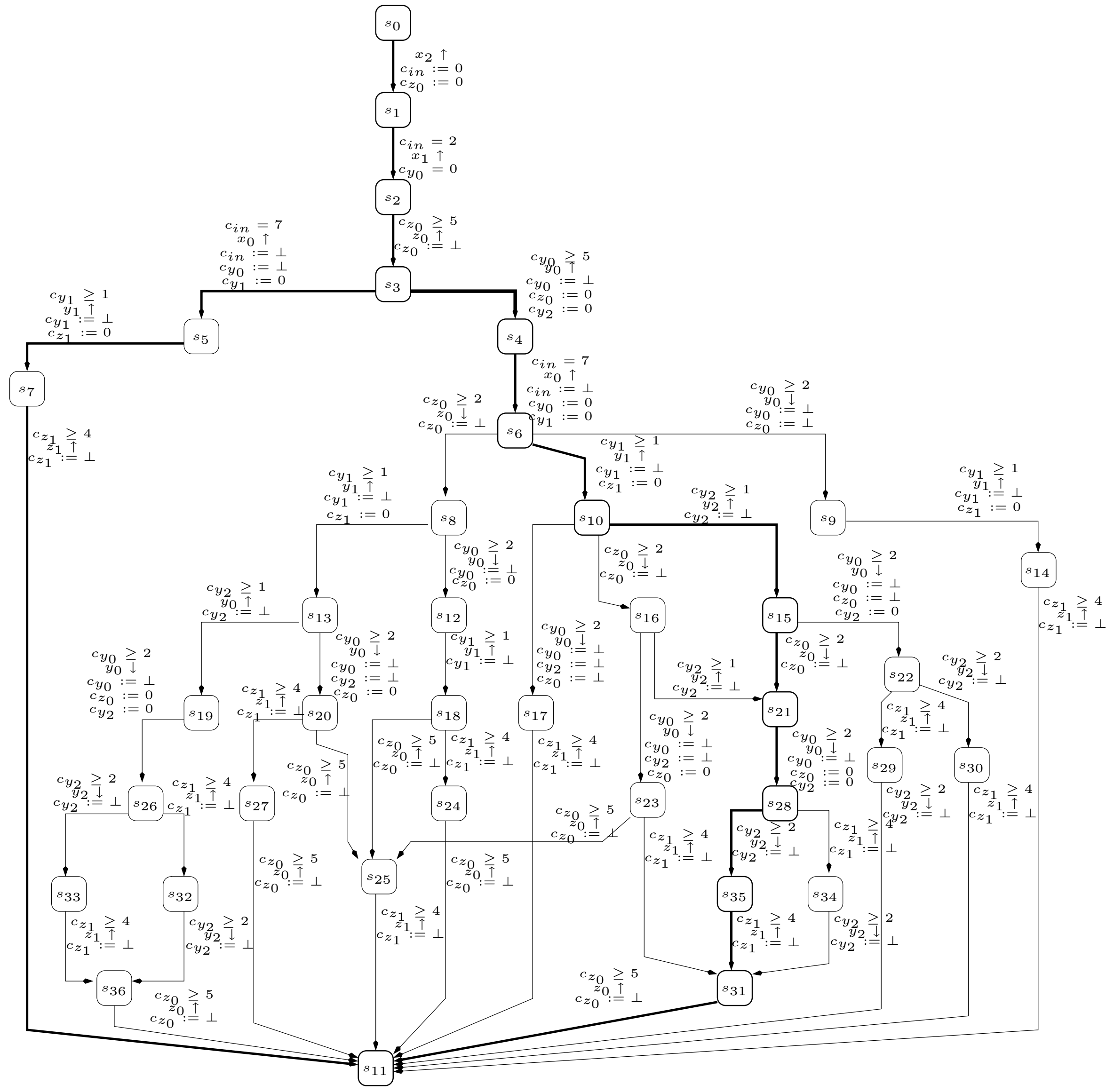


Figure 5.8: The interpreted timed automaton obtained for the circuit composed with the input generator. Runs ξ_1 and ξ_2 are marked by thick lines.

Chapter 6

Abstraction: Acyclic Environments

We have seen in the previous chapter how to build a precise model of a timed circuit made of gates with bounded delay uncertainties. This model generates all the behaviors that the circuit may exhibit. However, this model, an interpreted timed automaton $\mathcal{A}$, is quite large, having one clock for every gate and a number of discrete states exponential in the number of gates. On the other hand, when this circuit will be considered as a component in a larger system, its interaction will be restricted to its *interface variables*. Our goal is to approximate $\mathcal{A}$ by a smaller automaton $\mathcal{A}'$ with fewer states and fewer clocks satisfying $[\![\mathcal{A}]\!]_{io} \subseteq [\![\mathcal{A}']\!]_{io}$, that is, it will exhibit all the behaviors of $\mathcal{A}$ and possibly some more. If we are lucky, and $\mathcal{A}'$ is indeed much smaller than $\mathcal{A}$, but still preserves the properties that make $\mathcal{A}$ correct and usable, we can replace $\mathcal{A}$ by $\mathcal{A}'$ in the composition with the rest of the system, apply the same approximation procedure to the composed system and so on. Using this divide-and-conquer methodology, we may end up analyzing systems much larger than we could, if we tried to apply reachability computation *directly* to the whole system. However, since this process accumulates approximations in every step, we should be careful.

6.1 Acyclic Input Generators

Ideally, we would like to build the model (and then the reduced model) of a component relative to its *most general* environment. This way we can reuse the model in different contexts. However, when facing complexity issues one should make some pragmatic compromises. In this chapter we will focus on a restricted subclass of environments (input generators), those that are modeled by *acyclic* timed automata where all states except the terminal ones have bounded invariants. All the input signals generated by such automata are *ultimately constant* and when composed with input-dependent automata, they will lead to acyclic automata that exhibit ultimately-constant behaviors.

Such restricted environments may fit well into the synchronous design methodology discussed in the previous chapter. According to this approach a circuit is decomposed into a *combinational* part and a *memory* part. The inputs are presented to the combinational part in one burst at time zero (or in a finite sequence of such bursts as in the example of the previous chapter) and no further inputs are presented until the circuit stabilizes. Hence the input presented to the circuit during one clock cycle can be modeled as an acyclic generator. Other software or hardware systems that assume "well-behaving" inputs that wait for the systems to process previous input instances, or that have *admission controllers* that do not allow new inputs before stabilization, can be modeled this way as well.

A nice feature of such acyclic automata is that they reach a stable state after a finite amount of time, bounded by the longest path (in the circuit or in the automaton). Hence if we add an auxiliary clock $\hat{c}$ which is reset at time zero, and never reset again, its domain will be bounded. Such a clock measures *absolute time* and its value in any configuration is the time elapsed since the beginning. Hence the maximal value of $\hat{c}$ over all unstable reachable configurations is the maximal propagation delay of the circuit. Such a clock has been used for similar purposes in [AAM06] and elsewhere for solving acyclic optimal scheduling problems.

The abstraction that we propose for this setting takes advantage of this clock. The reduced model will be an automaton having only $\hat{c}$ as a clock, and output transitions will be thus guarded by constraints on $\hat{c}$, which in this case will denote *intervals* of absolute time. For instance, a transition labeled $z^{\uparrow}$ guarded by $\hat{c} \in [t_1, t_2]$ indicates that along this path z will rise between t_1 and t_2 time since the beginning. However, the road to obtain this automaton is long as will be detailed below. The major idea is to use the other clocks and state variables, those associated with the gates, to eliminate impossible qualitative behavior and only then throw these clocks and variables away by *projecting* the timing constraints on $\hat{c}$, and *hiding* transitions that do not change the value of any observable variable.

6.2 The Abstraction Technique

6.2.1 Summary

The abstraction methodology that we develop in this chapter starts with the automaton $\mathcal{A}_X$ modeling the composition of the circuit with its input generator, having a set X of variables. It can be summarized by the sequence of steps

$$\mathcal{A}_X \overset{1}{\Rightarrow} \mathcal{A}_X^{+\hat{c}} \overset{2}{\Rightarrow} \mathcal{A}_X^{r} \overset{3}{\Rightarrow} \mathcal{A}_X^{\hat{c}} \overset{4}{\Rightarrow} \mathcal{A}_{X_{ou}} \overset{5}{\Rightarrow} \mathcal{A}_{X_{ou}}^{m} \tag{6.1}$$

which is explained below.

1. From $\mathcal{A}_X$ we construct $\mathcal{A}_X^{+\hat{c}}$ by adding an auxiliary clock which does not participate in transition guards or invariants hence it only *observes* the dynamics of the automaton and measures absolute time.

2. We apply the forward reachability algorithm to $\mathcal{A}_X^{+\hat{c}}$ to obtain the interpreted timed automaton $\mathcal{A}_X^r$ having the same semantics as $\mathcal{A}_X^{+\hat{c}}$, but with additional useful properties as described in Chapter 2.

3. We then *project* the timing constraints of $\mathcal{A}_X^r$ on clock $\hat{c}$, thus relaxing them somewhat, to obtain $\mathcal{A}_X^{\hat{c}}$. This is the most crucial part of our methodology.

4. We project $\mathcal{A}_X^{\hat{c}}$ on the *output* variables to obtain $\mathcal{A}_{X_{ou}}$ thus making certain transitions *silent* .

5. We then reduce the discrete state space of $\mathcal{A}_{X_{ou}}$ by merging states which are essentially equivalent in terms of the untimed behaviors that they generate and after merging also state invariants, transitions and transition guards we obtain the reduced automaton $\mathcal{A}_{X_{ou}}^m$.

Recalling the fact that $[\![\mathcal{A}]\!] = [\![\mathcal{A}']\!]$ implies $[\![\mathcal{A}]\!]_{/X'} = [\![\mathcal{A}']\!]_{/X'}$ for any subset X' of variables, we will have the following relationships between the observable (output) semantics of these automata:

$$[\![\mathcal{A}_X]\!]_{ou} = [\![\mathcal{A}_X^{+\hat{c}}]\!]_{ou} = [\![\mathcal{A}_X^r]\!]_{ou} \subseteq [\![\mathcal{A}_X^{\hat{c}}]\!]_{ou} = [\![\mathcal{A}_{X_{ou}}]\!] \subseteq [\![\mathcal{A}_{X_{ou}}^m]\!]$$

The corollary of this is

$$[\![\mathcal{A}_X]\!]_{ou} \subseteq [\![\mathcal{A}_{X_{ou}}^m]\!]$$

which by itself is not a very strong claim, since it holds for any abstraction, but if we look at the qualitative semantics we have

$$\mu([\![\mathcal{A}_X]\!]_{ou}) = \mu([\![\mathcal{A}_X^{+\hat{c}}]\!]_{ou}) = \mu([\![\mathcal{A}_X^r]\!]_{ou}) = \mu([\![\mathcal{A}_X^{\hat{c}}]\!]_{ou}) = \mu([\![\mathcal{A}_{X_{ou}}]\!]) = \mu([\![\mathcal{A}_{X_{ou}}^m]\!]).$$

In other words, we preserve the qualitative semantics of the automaton and relax, to some extent, its timing constraints. As we shall see, this relaxation of timing constraints is much less violent than discarding them altogether and taking the untimed reachability graph (Definition 2.4) as the reduced model.

6.2.2 Interpreted Timed Automata with an Additional Global Clock

We start with the automaton $\mathcal{A}_X$ which models the circuit composed with some acyclic input generator, and in which all paths reach a stable state after a finite amount of time. The first two steps are the augmentation of $\mathcal{A}_X$ with the auxiliary clock $\hat{c}$ and computing its interpreted automaton via reachability computation. The first step is trivial for acyclic input generators, but as we see in the next chapter, its adaptation to cyclic generators requires novel ideas. Automaton $\mathcal{A}_X^{+\hat{c}}$ is almost identical to $\mathcal{A}_X$ with clock $\hat{c}$ reset at the first transition; the clock $\hat{c}$ does not appear in any time constraint of $\mathcal{A}_X^{+\hat{c}}$, and it is thus inactive in every state of this automaton. The runs of $\mathcal{A}_X^{+\hat{c}}$ are isomorphic to the runs of $\mathcal{A}_X$ with the valuation of $\hat{c}$ serving as a time stamp. That is, for any compound step of the form

$$(q, v) \xrightarrow{a,d} (q', v' + d)$$

in $\mathcal{A}_X$ there will be a compound step of the form

$$(q, v, t) \xrightarrow{a,d} (q', v' + d, t + d)$$

in $\mathcal{A}_X^{+\hat{c}}$ if that step in $\mathcal{A}_X$ could start at absolute time t.

We apply forward reachability to obtain the interpreted timed automaton $\mathcal{A}_X^r$. Again, this automaton is similar to the one obtained for $\mathcal{A}_X$ except for the fact that the zones associated with its invariants and guards will have one additional dimension.

6.2.3 Projection on Global Time

The next step is to project all the zones appearing as invariants and guards of $\mathcal{A}_X^r$ on clock $\hat{c}$.

DEFINITION 6.1 (Clock Projection) *Let $\mathcal{A}_X = (\mathcal{A}, X, \lambda)$ be a timed automaton, where $\mathcal{A} = (Q, F_X, q_0, C, I, \Delta)$, and let C' be a subset of C. The projection of A_X on C' is the timed automaton $\mathcal{A}_X/_{C'} = (\mathcal{A}/_{C'}, X, \lambda)$, where $\mathcal{A}/_{C'} = (Q, F_X, q_0, C', I', \Delta')$ such that for every $q \in Q$, $I'(q) = I(q)/_{C'}$ and Δ' consists of transitions of the form $(q, a, g/_{C'}, \gamma/_{C'}, q')$ for every $(q, a, g, \gamma, q') \in \Delta$.*

Since such a projection *relaxes* the timing constraints, we have $[\![\mathcal{A}_X]\!] \subseteq [\![\mathcal{A}_X/_{C'}]\!]$, and since $\mathcal{A}_X^r$ is an interpreted timed automaton, its projection on global time, $\mathcal{A}_X^{\hat{c}} = \mathcal{A}_X^r/_{\{\hat{c}\}}$ preserves the qualitative semantics, $\mu([\![\mathcal{A}_X^r]\!]) = \mu([\![\mathcal{A}_X^{\hat{c}}]\!])$.

To gain some intuition on the kind of spurious behaviors added through this projection operation, let us look at the example depicted in Figure 6.1. It consists of two components B_1 and B_2, which simply propagate the value of input x_0 after some delay $d_1 \in [l_1, u_1]$ to z_1, and later, after a second relative delay $d_2 \in [l_2, u_2]$ to z_2. Let c_1 and c_2 be the clocks of the components, and let $\hat{c}$ be the auxiliary clock. To simplify the discussion, consider an input generator with only one input event, the rising of x_0 at time 0. The corresponding interpreted timed automaton $\mathcal{A}^r_X$ is shown in Figure 6.1-(b), while its projection $\mathcal{A}^{\hat{c}}_X$ appears in Figure 6.1-(c). The respective behaviors of these automata are illustrated in Figures 6.1-(d) and 6.1-(e).

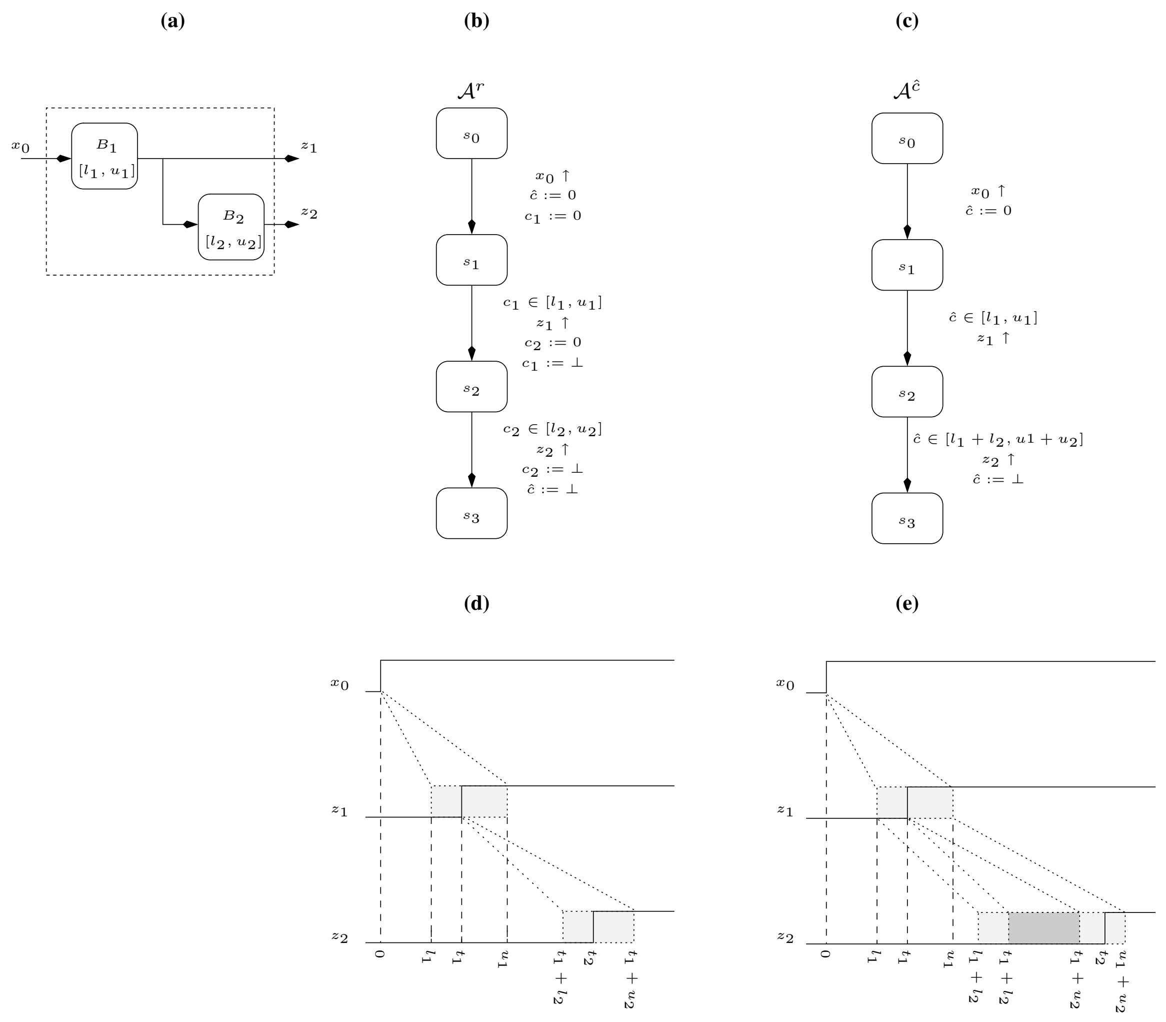

(a)
(b)
(c)
x_0
B_1
$[l_1, u_1]$
z_1
B_2
$[l_2, u_2]$
z_2
$\mathcal{A}^r$
s_0
$x_0 \uparrow$ $\hat{c} := 0$ $c_1 := 0$
s_1
$c_1 \in [l_1, u_1]$ $z_1 \uparrow$ $c_2 := 0$ $c_1 := \perp$
s_2
$c_2 \in [l_2, u_2]$ $z_2 \uparrow$ $c_2 := \perp$ $\hat{c} := \perp$
s_3
$\mathcal{A}^{\hat{c}}$
s_0
$x_0 \uparrow$ $\hat{c} := 0$
s_1
$\hat{c} \in [l_1, u_1]$ $z_1 \uparrow$
s_2
$\hat{c} \in [l_1 + l_2, u1 + u_2]$ $z_2 \uparrow$ $\hat{c} := \perp$
s_3
(d)
(e)
x_0
z_1
z_2
0 l_1 t_1 u_1 $t_1 + l_2$ t_2 $t_1 + u_2$
x_0
z_1
z_2
0 l_1 t_1 u_1 $l_1 + l_2$ $t_1 + l_2$ $t_1 + u_2$ t_2 $u_1 + u_2$

Figure 6.1: (a) A simple circuit; (b) The interpreted timed automaton $\mathcal{A}^r$ with the additional clock $\hat{c}$; (c) The projection of $\mathcal{A}^{\hat{c}}$ on the global clock; (d) A behavior of $\mathcal{A}^r$; (e) A behavior of $\mathcal{A}^{\hat{c}}$ in which $t_2 \in [l_1 + l_2, u_1 + u_2]$ but $t_2 \notin [t_1 + l_2, t_1 + u_2]$.

Both behaviors consist of $x_0^{\uparrow}$ at time 0, then $z_1^{\uparrow}$ at some t_1 and $z_2^{\uparrow}$ at time t_2. In the original automaton, these should satisfy $t_1 \in [l_1, u_1]$ and $t_2 \in [t_1 + l_2, t_1 + u_2]$. The first condition is due to the guard $c_1 \in [l_1, u_1]$ of the transition from s_1 to s_2, because at any time t, $c_1 = t$. The second condition is imposed by the guard $c_2 \in [l_2, u_2]$ of the transition from s_2 to s_3, because at any t, $c_2 = t - t_1$. After the projection, the *relation between* t_1 and t_2 is *broken*, and the projected guard $\hat{c} \in [l_1 + l_2, u_1 + u_2]$ allows $z_2^{\uparrow}$ to occur at any $t_2 \in [l_1 + l_2, u_1 + u_2]$, regardless of t_1. Roughly speaking, for each path, the projected automaton allows runs in which transitions occur at times $t_1, t_2, \ldots$ if for *each* t_i there *exists* a run of the original automaton in which *this transition* occurs at t_i. This over-approximation is the price we pay for reducing the number of clocks. A similar reduction has been proposed in [ZMM03] for timed Petri nets.

Before moving further, to reduce the number of states let us note that after the projection, clock $\hat{c}$ changes its role from a *passive observer* of the dynamics to the sole guardian of the (relaxed) timing constraints. Let us also repeat the important fact that this clock is never reset after time zero. The result of applying this step to the full-adder example of the previous chapter is shown in Figure 6.2.

6.2.4 Projection on Output Variables

All the previous transformations have led to timed automata defined on the same original set of variables X. From now on we focus the output variables X_{ou} of the component. Let us define the projection of a timed automaton on a set of variables.

DEFINITION 6.2 (Variables Projection) *Let $\mathcal{A}_X = (\mathcal{A}, X, \lambda)$ be a timed automaton, where $\mathcal{A} = (Q, F_X, q_0, C, I, \Delta)$, and let $X' \subseteq X$. The projection of $\mathcal{A}_X$ on X' is the timed automaton $\mathcal{A}_X/_{X'} = (\mathcal{A}/_{X'}, X', \lambda/_{X'})$, where $\mathcal{A}/_{X'} = (Q, F_{X'}, q_0, C, I, \Delta')$ where Δ' is constructed from Δ by replacing every transition of the form (q, g, f, γ, q') by a transition (q, g, f', γ, q') where $f' = f/_{X'}$ is the restriction of f to X'.*

In other words, this projection is a kind of a *hiding* operation that has no effect on the *dynamics* of the automaton, only on the *labeling* of states and transitions. In particular, transitions that do not change any variable in X' are labeled by the identity function that we denote by τ. Such transitions are called *silent*. It is not hard to see that variable projection commutes with the semantics:

$$[\![\mathcal{A}_X/_{X'}]\!] = [\![\mathcal{A}_X]\!]/_{X'}.$$

In our particular case we project automaton $\mathcal{A}_X^{\hat{c}}$ on the output variables X_{ou} to obtain $\mathcal{A}_{X_{ou}} = \mathcal{A}_X^{\hat{c}}/_{X_{ou}}$. To illustrate the semantics of the projected system let us look at the signal carried by run ξ_1 of full adder of the previous chapter, presented here in an event-based form:

$$\xi_1 : x_2^{\uparrow} \cdot 2 \cdot x_1^{\uparrow} \cdot 3 \cdot z_0^{\uparrow} \cdot 1 \cdot y_0^{\uparrow} \cdot 0 \cdot x_0^{\uparrow} \cdot 1 \cdot y_1^{\uparrow} \cdot 1 \cdot y_2^{\uparrow} \cdot 1 \cdot z_0^{\downarrow} \cdot 0 \cdot y_0^{\downarrow} \cdot 3 \cdot y_2^{\downarrow} \cdot 2 \cdot z_1^{\uparrow} \cdot 1 \cdot z_0^{\uparrow}$$

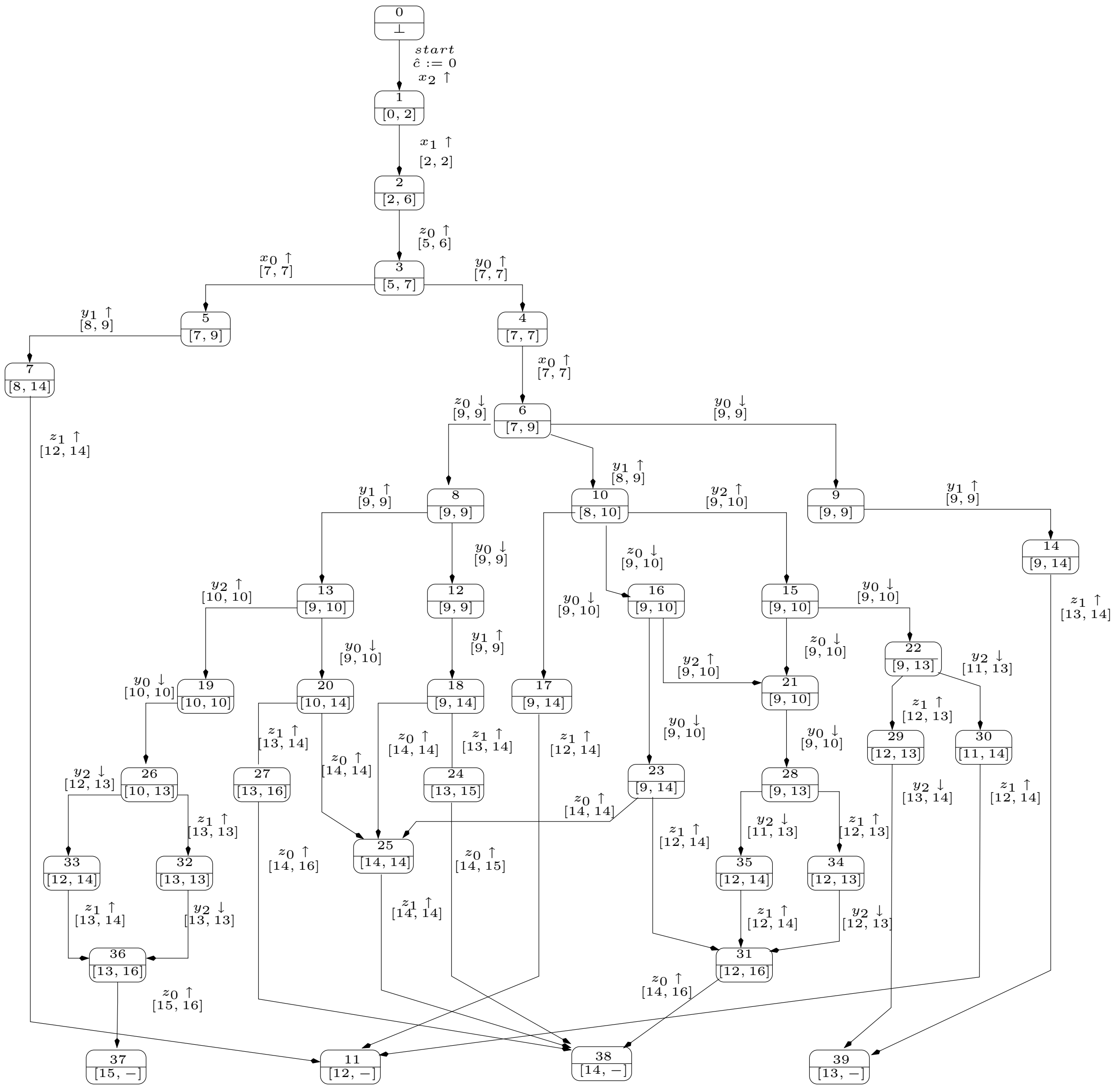


Figure 6.2: Projecting on the global clock $\hat{c}$: automaton $\mathcal{A}_X^{\hat{c}}$. Invariants and guards are presented as intervals.

Projecting on the output variables $\{z_0, z_1\}$ we first obtain the signal

$$\xi_1' : \tau \cdot \underbrace{2 \cdot \tau \cdot 3}\cdot z_0^{\uparrow} \cdot \underbrace{1 \cdot \tau \cdot 0 \cdot \tau \cdot 1 \cdot \tau \cdot 1 \cdot \tau \cdot 1}\cdot z_0^{\downarrow} \cdot \underbrace{0 \cdot \tau \cdot 3 \cdot \tau \cdot 2}\cdot z_1^{\uparrow} \cdot 1 \cdot z_0^{\uparrow}$$

in which several transitions become silent. Collapsing each sequence of silent transitions into a single time step we obtain the signal

$$\xi_1/_{\{z_0,z_1\}} : 5 \cdot z_0^{\uparrow} \cdot 4 \cdot z_0^{\downarrow} \cdot 5 \cdot z_1^{\uparrow} \cdot 1 \cdot z_0^{\uparrow}$$

So far the projected automaton $\mathcal{A}_{X_{ou}}$ has the same structure (and observable semantics) as $\mathcal{A}_X^{\hat{c}}$, see Figure 6.3

6.2.5 Minimization

The next step is the reduction of the discrete state space by merging states which are equivalent in some sense in terms of the qualitative behaviors that can be generated from them. The process will transform automaton $\mathcal{A}_{X_{ou}}$ into the timed automaton $\mathcal{A}_{X_{ou}}^m$ having the same qualitative semantics. The quantitative semantics of $\mathcal{A}_{X_{ou}}^m$ will be, in certain cases equivalent to that of $\mathcal{A}_{X_{ou}}$ and will over-approximate it in other cases. In fact, a slight modification of our minimization procedure that we mention at the end of the section can guarantee semantic equivalence. This work is inspired by ideas appearing in [BFG+91] and [TY01].

Since all the steps in the minimization procedure merge states with *identical* variable valuations, we will present it in terms of transformations on an ordinary timed automaton, with an action alphabet Σ containing the special "silent" symbol τ. To facilitate the presentation we will use the notation $q \xrightarrow{a} q'$ for the existence of a transition labeled by a from q to q' and let $q \xrightarrow{\tau} q'$ indicate the existence of a silent transition. The former will imply the existence of some compound step $(q, v) \xrightarrow{t,a} (q', v')$ in the automaton while the other implies a compound step $(q, v) \xrightarrow{t,\tau} (q', v')$ which looks like a pure time step from the outside.

We present minimization as a sequence of three steps

$$\mathcal{A} \overset{1}{\Rightarrow} \mathcal{A}^o \overset{2}{\Rightarrow} \mathcal{A}^b \overset{3}{\Rightarrow} \mathcal{A}^m$$

explained below.

1. To remove silent transitions from $\mathcal{A}$ we collapse every sequence of transitions of the form $a\tau^*$ into a single transition labeled by a. To do so we introduce states that correspond to various *chains* of silent transitions. The resulting automaton $\mathcal{A}^o$ will have much more

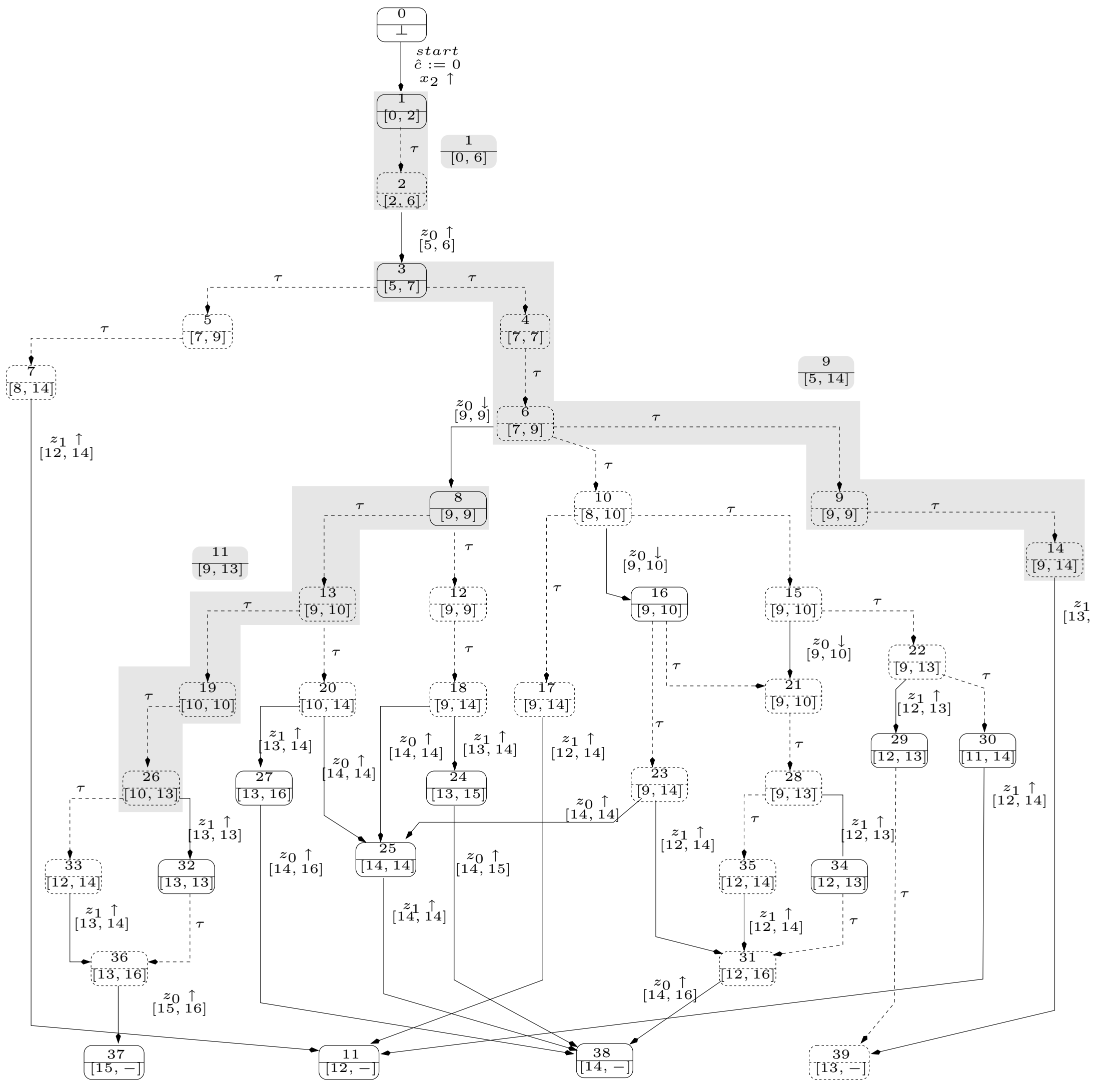


Figure 6.3: Projection on output variables, automaton $\mathcal{A}_{X_{ou}}$. The chains $\langle 0, 2\rangle$, $\langle 2, 14\rangle$, and $\langle 6, 26\rangle$ are shaded in preparation for their transformation into states 1, 9 and 11 of $\mathcal{A}^o$ in the next step.

non-determinism in the discrete transitions but every *sequence of silent transitions* will be replaced by a *time step inside a single state*.

2. We compute the (qualitative) bisimulation relation $\sim$ on states of $\mathcal{A}^o$ and let the states of $\mathcal{A}^b$ be the equivalence classes of $\sim$. The invariant of each class is the *convex hull* of the invariants of its members.

3. For every pair of states of $\mathcal{A}^b$ we merge all the transitions between them having the same label into one transition whose guard is the *convex hull* of the guards of the individual transitions.

Eliminating Silent Transitions

Let $\mathcal{A} = (Q, q_0, \Sigma, I, \Delta)$ be a timed automaton with silent transitions. A state $q \in Q$ is *directly observable* if it is the initial state q_0 or there is some observable a and q' such that $q \xrightarrow{a} q'$. We denote by $D(Q)$ the set of directly observable states. Given two states $q, q' \in D(Q)$ we say that q' is a *chain-successor* of q if there is a path

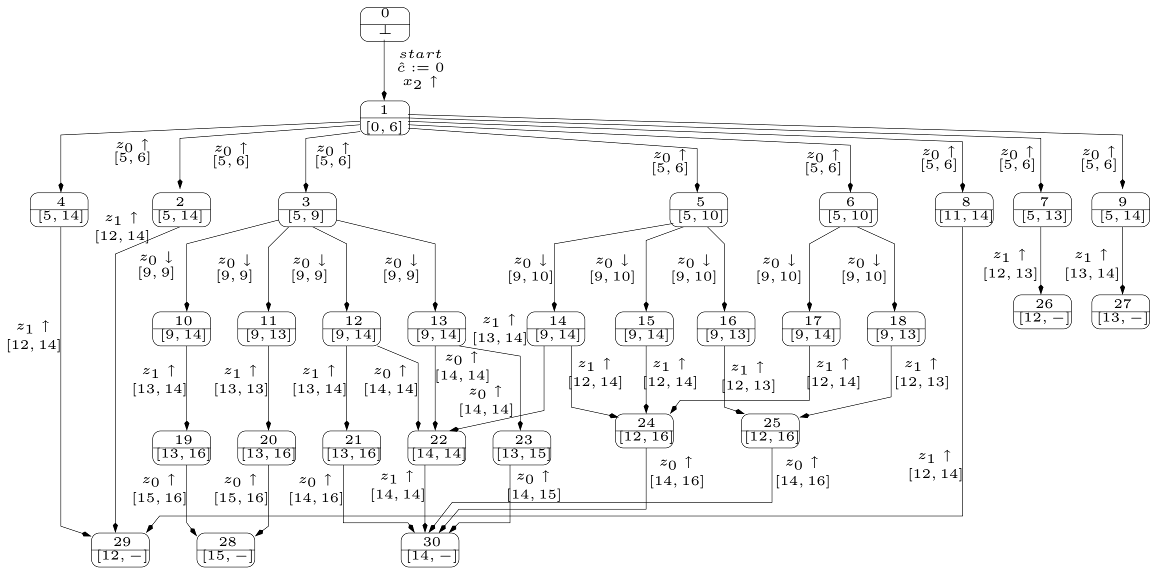
$$q \xrightarrow{a} q_1 \xrightarrow{\tau} q_2 \cdots q_k \xrightarrow{\tau} q'$$

in the automaton. We denote by $\langle q, q' \rangle$ the set of states $\{q_1, \ldots, q_k, q'\}$ that appear in this silent chain. Note that all states in $\langle q, q' \rangle$ admit the same variable valuation since the transitions inbetween them are silent.

Definition 6.3 (Automaton $\mathcal{A}^o$) *The automaton $\mathcal{A}^o = (Q^o, q_0, \Sigma, I^o, \Delta^o)$ is constructed from $\mathcal{A} = (Q, q_0, \Sigma, I, \Delta)$ as follows:*

- $Q^o = \{q_0\} \cup \{\langle q, q' \rangle : q, q' \in D(Q)\}$
- $I^o(\langle q, q' \rangle) = \bigcup_{p \in \langle q, q' \rangle} I(p)$
- Δ^o *contains a transition* $(\langle q, q' \rangle, a, g, \gamma, \langle p, p' \rangle)$ *for every transition* $(q', a, g, \gamma, p) \in \Delta$.

It is not hard to see that the invariants thus obtained are convex, because clock $\hat{c}$ is never reset and the intervals associated with states appearing in a chain are contiguous. Hence $[\![\mathcal{A}^o]\!]$ and $[\![\mathcal{A}]\!]$ coincide. The result of applying silent-transition removal to the full-adder example is shown in Figure 6.4.



$1 = \{1, 2\}$
$2 = \{3, 5, 7\}$
$3 = \{3, 4, 6\}$
$4 = \{3, 4, 6, 10, 17\}$
$5 = \{3, 4, 6, 10\}$
$6 = \{3, 4, 6, 10, 15\}$
$7 = \{3, 4, 6, 10, 15, 22\}$
$8 = \{3, 4, 6, 10, 15, 22, 30\}$
$9 = \{3, 4, 6, 9, 14\}$
$10 = \{8, 13, 19, 26, 33\}$
$11 = \{8, 13, 19, 26\}$
$12 = \{8, 13, 20\}$
$13 = \{8, 12, 18\}$
$14 = \{16, 23\}$
$15 = \{16, 21, 28, 35\}$
$16 = \{16, 21, 28\}$
$17 = \{21, 28, 35\}$
$18 = \{21, 28\}$
$19 = \{36\}$
$20 = \{32, 36\}$
$21 = \{27\}$
$22 = \{25\}$
$23 = \{24\}$
$24 = \{31\}$
$25 = \{31, 34\}$
$26 = \{29, 39\}$
$27 = \{39\}$
$28 = \{37\}$
$29 = \{11\}$
$30 = \{38\}$

Figure 6.4: Removing silent transitions, automaton $\mathcal{A}^o$. The mapping of states of Q^o to silent chains of $\mathcal{A}$ is indicated at the bottom.

Merging Bisimilar States

Let $\mathcal{A} = (Q, q_0, \Sigma, I, \Delta)$ be a timed automaton without silent transitions. The *qualitative bisimulation* relation over Q is the largest equivalence relation satisfying

$$q \sim q' \equiv \forall a \in \Sigma \begin{cases} q \xrightarrow{a} p \Rightarrow \exists p'(q' \xrightarrow{a} p' \wedge p \sim p') \\ q' \xrightarrow{a} p' \Rightarrow \exists p(q \xrightarrow{a} p \wedge p \sim p') \end{cases}$$

In other words $q \sim q'$ if the same qualitative observable behaviors can be generated from both. We let $[q]$ denote the $\sim$ equivalence classes of q and denote the set of such classes by $Q/\sim$. Note again that all elements of $[q]$ have the same variable assignment.

DEFINITION 6.4 (Automaton $\mathcal{A}^b$) *The automaton $\mathcal{A}^b = (Q^b, q_0^b, \Sigma, I^b, \Delta^b)$ is constructed from $\mathcal{A}^o = (Q^o, q_0, \Sigma, I^o, \Delta^o)$ as follows:*

- $Q^b = Q^o/\sim$
- $q_0^b = [q_0]$
- $I^b([q]) = \bigsqcup_{q' \in [q]} I^o(q')$

- Δ^b *contains a transition* $([q], a, g, \gamma, [q'])$ *for every transition* (p, a, g, γ, p') *such that* $p \in [q]$ *and* $p' \in [q']$.

This transformation, by definition, preserves qualitative behavior, and may over-approximate the timed behavior due to the use of convex hull.

Merging Transitions

The last step is to merge transitions between every pair of states which agree on the transition label.

DEFINITION 6.5 (Automaton $\mathcal{A}^m$) *Automaton* $\mathcal{A}^m = (Q^b, q_0^b, \Sigma, I^b, \Delta^m)$ *is obtained from* $\mathcal{A}^b = (Q^b, q_0^b, \Sigma, I^b, \Delta^b)$ *by letting* Δ^m *consist of transitions of the form* (q, a, g, γ, q') *where*

$$g = \bigsqcup \{g' \mid (q, a, g', \gamma, q') \in \Delta^b\}.$$

As in the previous step, qualitative semantics is preserved while the timed semantics may grow due to the use of convex hull. Let us remark that in many cases, including our example, $\bigcup g'$ is already convex and the timed semantics is preserved. Moreover, we can guarantee preservation of timed semantics if we restrict the merging of transitions and states to those whose corresponding unions are convex and refine the equivalence relation accordingly. Let us note that in the acyclic case, convexity of the union will be more frequent because we are dealing here with *one-dimensional* zones (intervals) where the convexity of $Z_1 \cup Z_2$ is equivalent to $Z_1 \cap Z_2 \neq \emptyset$. The result of applying the last two steps to the full-adder example are shown in Figure 6.5.

As one can see, the reduction is quite significant. Starting from a timed automaton with 39 states and 5 clocks we end up with an automaton with 8 states and one clock which still gives a reasonable representation of the behavior of the circuit. In the next section we show how this reduction procedure can be used in a compositional analysis framework and report some experimental results on a synthetic example.

6.3 Applications

6.3.1 A Divide and Conquer Analysis Algorithm

Suppose we have a big combinational circuit with interface variables X_{in} and X_{ou}. Assuming a restricted input modeled by an acyclic timed automaton $\mathcal{A}_{in}$, we want to build a timed model $\mathcal{A}_{ou}$

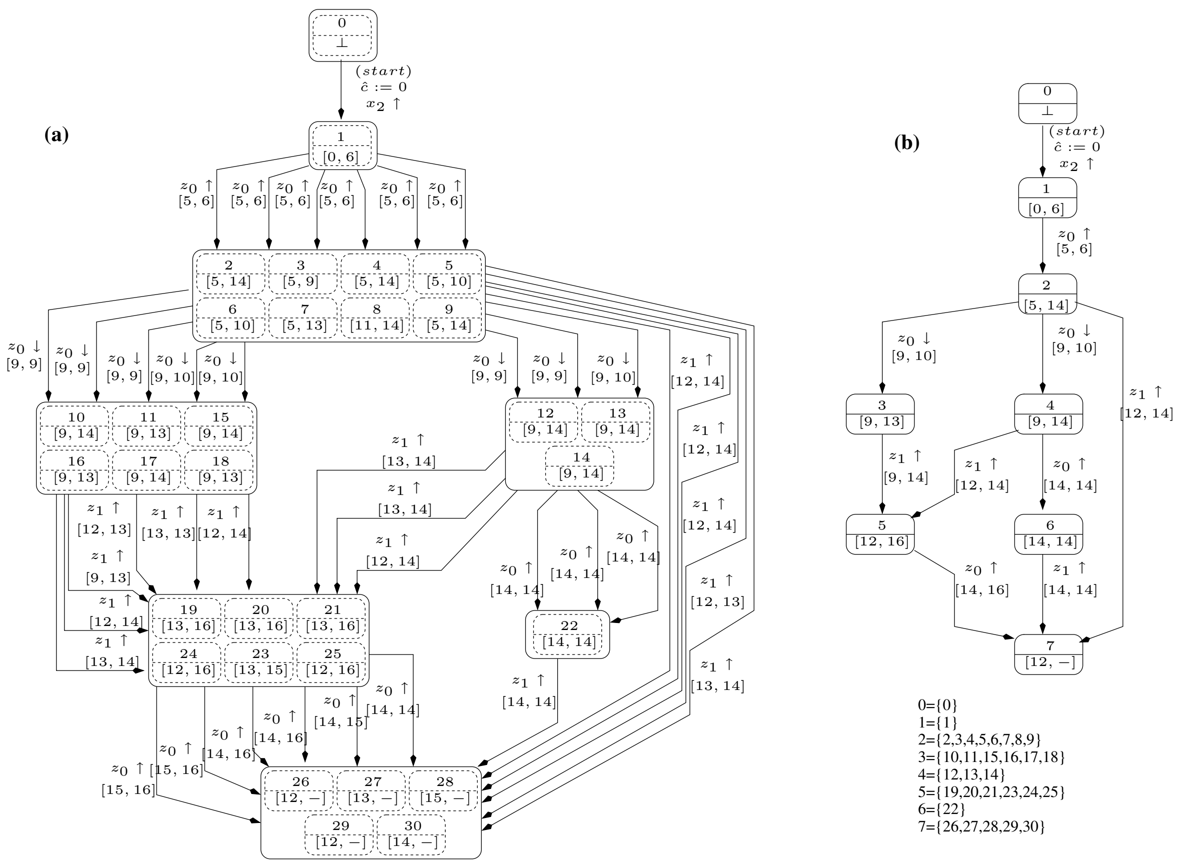


Figure 6.5: (a) Merging bisimilar states, automaton $\mathcal{A}^b$; (b) Merging transitions, automaton $\mathcal{A}^m_{X_{ou}}$, the final result.

of its *output behavior*. Such a model can, for example, tell us what is the maximal stabilization time of the circuit. As the circuit is too big to be analyzed as a whole, we apply the following *divide-and-conquer* approach. We partition the circuit into slices $\{P^1, \cdots, P^n\}$, and model each sub circuit P^i by a timed automaton $\mathcal{A}^i$ such that the (primary) input of the circuit is the input of $\mathcal{A}^1$, the output of each $\mathcal{A}^i$, $i < n-1$, is the input of $\mathcal{A}^{i+1}$ and the output of $\mathcal{A}^n$ is the output of the circuit, see Figure 6.6. The model of the whole circuit is then

$$\mathcal{A}_{in} \parallel \mathcal{A}^1 \parallel \mathcal{A}^2 \parallel \cdots \parallel \mathcal{A}^n.$$

Our idea is simple. Going from left to right we compose first $\mathcal{A}_{in}$ with $\mathcal{A}^1$, apply the reduction technique to the resulting automaton, use it as an *input generator* for $\mathcal{A}^{i+1}$. After composition we apply the reduction technique to obtain the input generator for $\mathcal{A}^{i+1}$ and so on. This procedure is summarized in Algorithm 6.1, using ρ to denote the reduction operation described in this chapter.

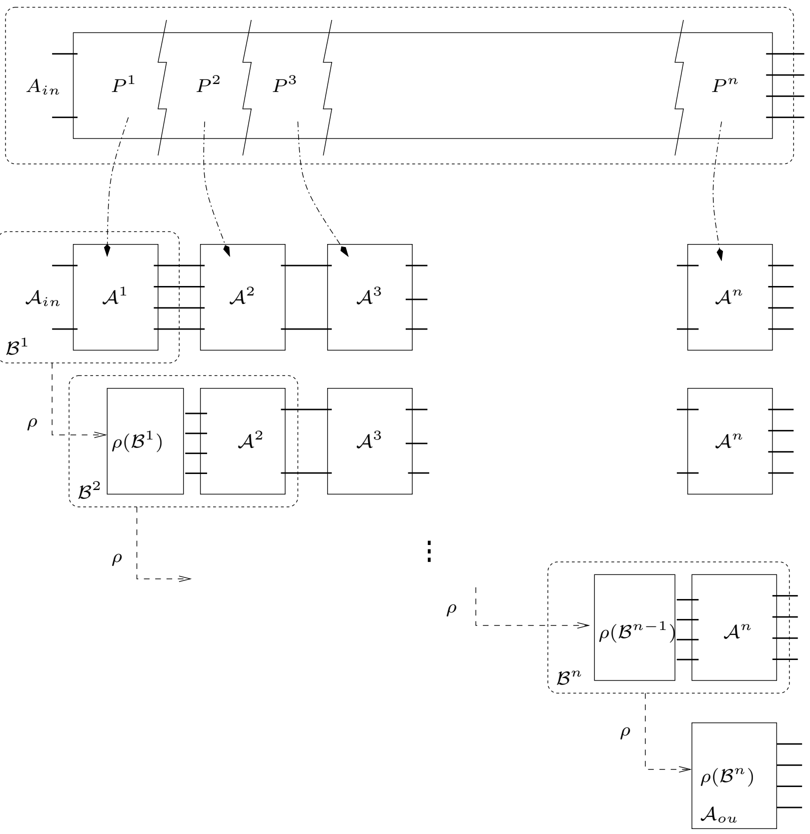


Figure 6.6: A compositional analysis framework for acyclic circuits with acyclic inputs.

Let us remark that this technique is not restricted to purely-sequential decomposition, but may admit parallel components. What is important is that the size of the slices will be small enough so that their reachability graph can be computed before being reduced. Secondly, many graph

Algorithm 6.1 Divide an Conquer Analysis

```
$\mathcal{B} := \mathcal{A}_{in} \parallel \mathcal{A}^1$
**for** $i = 1$ to $n - 1$ **do**
    $\mathcal{B}^{i+1} := \rho(\mathcal{B}^i) \parallel \mathcal{A}^{i+1}$
**end for**
$\mathcal{A}_{ou} := \rho(\mathcal{B}^n)$
```

partitioning algorithms can be used to decompose a circuit and, moreover, if the circuit was constructed hierarchically, some hints for the decomposition may be already given by the circuit structure.

6.3.2 Experimental Results

To illustrate experimentally the capabilities of this technique, we consider a synthetic family of examples, based on a 4-gate circuit, see Figure 6.7. This circuit has 2 inputs and 2 outputs so we can concatenate as many instances of it as we want. A concatenation of i instances will thus have $4 \cdot i$ gates and $4 \cdot i$ clocks. We assume an input generator which generates a single scenario and compare our compositional technique with a direct approach. Table 6.1 summarizes the results: The first column gives the number of gates in the circuit. The next three columns present the results obtained through the compositional method: the first column gives for each line i the number of states in automaton $\mathcal{B}^i$ before the final reduction, the second column gives the number of states obtained for the final reduced model $\mathcal{A}_{ou}$ while the third indicates the total time to compute $\mathcal{A}_{ou}$ via successive applications of all the steps in our reduction procedure. The last two columns give the results obtained when we proceed by modeling the whole system as a single block. It is easy to see that, facing the whole circuit, we reach very quickly the limitations of current timed automata technology (less than 20 gates) while using our method we can analyze automata with almost 100 state variables and clocks.

6.4 Summary

We have developed and implemented an abstraction technique that can reduce dramatically the size of the automaton corresponding to an acyclic composition of components, subject to acyclic input generators, which preserves much of the timing characteristics of the system. We have shown how using this technique we can analyze systems with almost 100 clocks and state variables. Let us discuss briefly the limitations of this approach.

The first limitation is related to the *width* of the circuit and the number of its input wires. The number of possible input scenarios, even if we restrict ourselves to a single burst of switches at

Gates	Compositional analysis			Direct analysis	
	$\mathcal{B}^i$ states	$\mathcal{A}_{ou}$ states	time(h:m:s)	$\mathcal{A}$ states	time(h:m:s)
4	162	5	0 .58	162	0 .58
8	133	6	1 .28	1074	2 .20
12	756	9	2 .02	8172	14 .55
16	2690	11	3 .24	137548	7:41 .34
20	4080	30	11 .41	*	*
28	50543	39	29 .47	*	*
32	73502	48	39 .57	*	*
36	95619	57	1:53 .68	*	*
40	117736	66	3:12 .01	*	*
44	139853	75	5:07 .23	*	*
48	161970	84	7:31 .53	*	*
52	184087	93	10:04 .82	*	*
56	206204	102	14:41 .57	*	*
60	228321	111	20:38 .43	*	*
64	250438	120	28:14 .93	*	*
68	272555	129	37:45 .58	*	*
72	294672	138	49:35 .27	*	*
76	316789	147	1:04:03 .58	*	*
80	338906	156	1:04:03 .58	*	*
84	361023	165	1:42:58 .68	*	*
88	383140	174	1:58:55 .42	*	*
92	405257	183	2:30:30 .08	*	*
96	*	*	*	*	*

Table 6.1: Comparing the performance of direct and compositional analysis.

time zero, will grow exponentially with the number of inputs. Hence the technique is limited to systems with a relatively-small number of input wires. When this is not the case, we can still apply the following heuristics. We can focus each time on a small set of output variables, compute backwards their "cones of influence" and extract the sub circuit that is responsible for their behavior. This way we can obtain separate models for each output but loose some information concerning their inter-dependence. Many other heuristics are possible and their empirical evaluation on real-life examples is a topic for further research.

The other limitation of this approach is that it is *not associative* as it has to proceed from left to right (or *from inputs to outputs*). The reason is that the reduced model is always a *generator* rather than a *transducer*. Consequently, even if there is a component appearing in several places in the circuit, we cannot build its reduced model once for all, and then reuse it everywhere this component appears. Hence it cannot be employed within a hierarchical development methodology. This will be possible with the reduction technique to be presented in the next chapter, where we consider arbitrary input models which may keep on changing indefinitely.

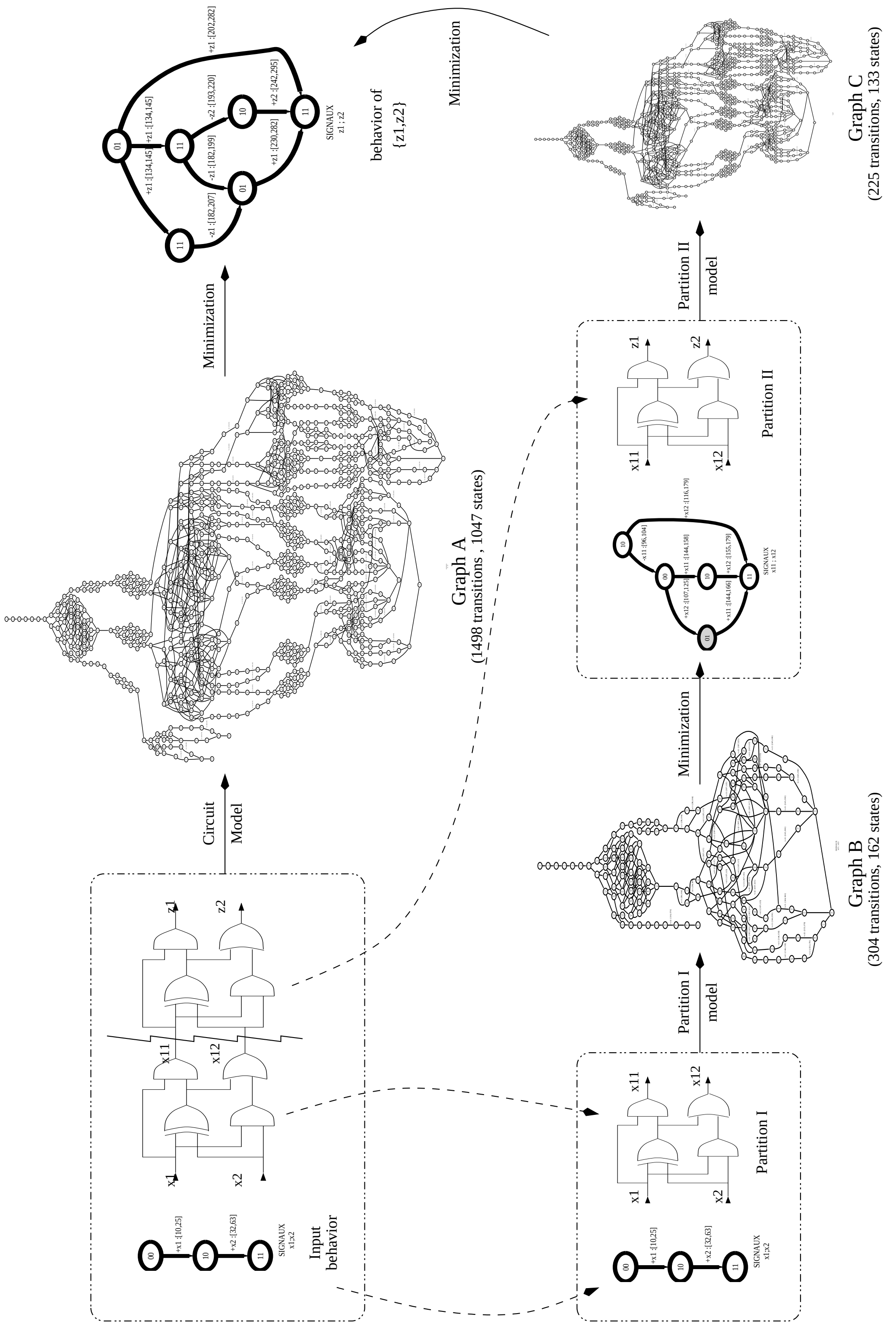


Figure 6.7: A global view of the compositional approach. Graph A is the reachability graph of the concatenation of 2 instances of the basic components; graph B is the reachability graph of one instance which, after reduction and composition with a second instance yields graph C which is finally reduced to a 6-state automaton.

Chapter 7

Abstraction: General Environments

In this chapter we move on to the more challenging task of generating small-size abstractions of timed components in the presence of more general environments that generate *infinite streams* of input events. Section 7.1 motivates the technique by illustrating how it can be used as a major ingredient in a hierarchical component-based development methodology and describes the major difficulties in treating such environments. Section 7.2 describes the major novel aspect of the technique, the dynamic creation and removal of auxiliary clocks associated with input events. These clocks will define the temporal frame of reference relative to which output events will be defined in the abstract model. The adaptation of the other ingredients of our abstraction procedure (clock projection, variable projection and minimization) to this setting are described in Section 7.3.

7.1 Introduction

7.1.1 Motivation

Consider the component-based system illustrated in Figure 7.1 constructed as a composition

$$\mathcal{A}_8 = \mathcal{A}_5 \parallel \mathcal{A}_6 \parallel \mathcal{A}_7$$

in which the components $\mathcal{A}_5$, $\mathcal{A}_6$ and $\mathcal{A}_7$ are themselves constructed from more basic components:

$$\mathcal{A}_5 = \mathcal{A}_1 \parallel \mathcal{A}_3\ ,\ \mathcal{A}_6 = \mathcal{A}_1 \parallel \mathcal{A}_2\ ,\ \mathcal{A}_7 = \mathcal{A}_3 \parallel \mathcal{A}_4 \parallel \mathcal{A}_6.$$

In cases when the system is too large to be analyzed as a whole, we would like to develop an abstraction operator ρ that will allow us to analyze the system in a divide-and-conquer manner, that is, generate a small approximating model of $\mathcal{A}_8$ using the following steps:

$$\begin{gathered}\mathcal{B}_1 = \rho(\mathcal{A}_1)\ ,\ \mathcal{B}_2 = \rho(\mathcal{A}_2)\ ,\ \mathcal{B}_3 = \rho(\mathcal{A}_3)\ ,\ \mathcal{B}_4 = \rho(\mathcal{A}_4)\\ \mathcal{B}_5 = \rho(\mathcal{B}_1 \parallel \mathcal{B}_3)\ ,\ \mathcal{B}_6 = \rho(\mathcal{B}_1 \parallel \mathcal{B}_2)\ ,\ \mathcal{B}_7 = \rho(\mathcal{B}_3 \parallel \mathcal{B}_4 \parallel \mathcal{B}_6)\\ \mathcal{B}_8 = \rho(\mathcal{B}_5 \parallel \mathcal{B}_6 \parallel \mathcal{B}_7)\end{gathered}$$

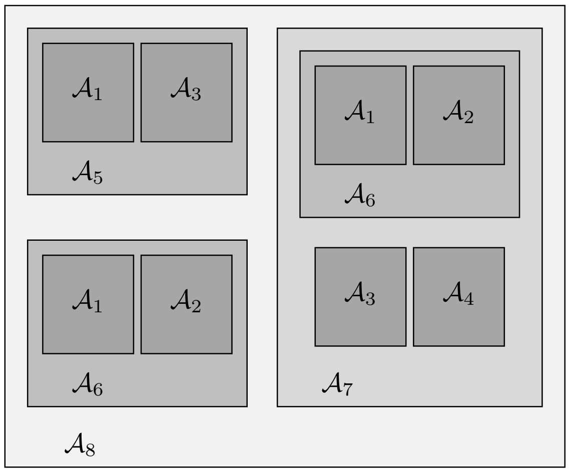


Figure 7.1: Example of Component Based System.

We want to use the same type of abstraction for a component, regardless of its environment. For example, we want to use the same reduced model $\mathcal{B}_6$ for $\mathcal{A}_6$ when it is composed with $\mathcal{A}_3$ and $\mathcal{A}_4$ and when it is composed with $\mathcal{A}_5$ and $\mathcal{A}_7$. This is different from the situation described in the previous chapter where we cut a circuit into slices with fixed location and environment.

As before we would like the abstraction to focus on the *interface* behavior of the components, relating the timings of input and output events. In the acyclic setting, since we had only a *finite* number of events, all arriving within a *bounded* amount of time after the beginning, we could let the reduced model characterize all output events in terms of their *absolute* time, their distance from time zero as measured by the auxiliary clock $\hat{c}$. This is no longer the case with cyclic input generators due to the following reasons. First, the system is modeled as working *indefinitely* and the temporal distance between time zero to events is *unbounded* and cannot be represented by bounded clocks. Secondly, since the arrival pattern of input events is *sporadic*, it does not make much sense to relate output events to time zero. We would rather relate each output event to the *input event* that has *triggered* it. This way the reduced model will give an approximation of the time it takes the system to respond to input events.

To this end, we let input events *create their own clocks* which, like clock $\hat{c}$ of the preceding chapter, will serve only for monitoring purposes, recording the "age" of the corresponding events.

As we will see, the number of events "alive" in the system in any given moment is bounded by the number and every event is propagated to the output (or aborted) within a finite amount of time. When this happens, its corresponding *input clock* can be discarded ("killed") and reused for new events.

After performing reachability computation and computing the interpreted timed automaton associated with this automaton, we apply clock projection, but this time we project on the input clocks. Hence, in the reduced model obtained after projection, hiding and minimization we will have input transitions that reset input clocks, followed downstream by output transitions guarded by conditions on the input clocks associated with the event that triggered them. This way the relation between the timing of input and output events will be captured in the reduced model, whose number of clocks will depend on the maximal number of live events in the system rather than on the number of components.

7.1.2 An Overview of the Abstraction Procedure

The abstraction starts with the automaton $\mathcal{A}_X$ modeling a network of timed components without an input generator. This amounts to assuming the *most general* environment. It can be summarized by the sequence of steps

$$\mathcal{A}_X \Rightarrow \mathcal{A}_X^{+\hat{C}} \Rightarrow \mathcal{A}_X^r \Rightarrow \mathcal{A}_X^{\hat{C}} \Rightarrow \mathcal{A}_{X_{io}} \Rightarrow \mathcal{A}_{X_{io}}^m \tag{7.1}$$

which is explained below.

1. From $\mathcal{A}_X$ we construct $\mathcal{A}_X^{+\hat{C}}$ by adding auxiliary input clocks which do not participate in transition guards or invariants hence they only *observe* the dynamics of the automaton and measure the time elapsed since each input event. Each of these clocks is discarded after a finite amount of time when all the reactions that the input event has triggered in the system terminate or are aborted.

2. We apply the forward reachability algorithm to $\mathcal{A}_X^{+\hat{C}}$ to obtain the interpreted timed automaton $\mathcal{A}_X^r$ having the same semantics, but with the additional property that each of its paths corresponds to a realizable qualitative behavior.

3. We relax the timing constraints of $\mathcal{A}_X^r$ by *projecting* them on clocks in $\hat{C}$ to obtain $\mathcal{A}_X^{\hat{C}}$. To be more precise, each transition guard is projected on the clock associated with the input event that has triggered it.

4. We project $\mathcal{A}_X^{\hat{C}}$ on the *input* and *output* variables to obtain $\mathcal{A}_{X_{io}}$ thus making internal transitions *silent*.

5. We then reduce the discrete state space of $\mathcal{A}_{X_{io}}$ by merging states which are essentially equivalent in terms of the untimed behaviors that they generate and after merging also state invariants, transitions and transition guards we obtain the reduced automaton $\mathcal{A}^m_{X_{io}}$.

At the end we will have the following relationships between the observable (input-output) semantics of these automata:

$$[\![\mathcal{A}_X]\!]_{io} = [\![\mathcal{A}_X^{+\hat{C}}]\!]_{io} = [\![\mathcal{A}_X^r]\!]_{io} \subseteq [\![\mathcal{A}_X^{\hat{C}}]\!]_{io} = [\![\mathcal{A}_{X_{io}}]\!] \subseteq [\![\mathcal{A}^m_{X_{io}}]\!]$$

and the following relation between the qualitative semantics

$$\mu([\![\mathcal{A}_X]\!]_{io}) = \mu([\![\mathcal{A}_X^{+\hat{C}}]\!]_{io}) = \mu([\![\mathcal{A}_X^r]\!]_{io}) = \mu([\![\mathcal{A}_X^{\hat{C}}]\!]_{io}) = \mu([\![\mathcal{A}_{X_{io}}]\!]) = \mu([\![\mathcal{A}^m_{X_{io}}]\!]).$$

In other words, we preserve the qualitative semantics of the automaton and relax, to some extent, its timing constraints. As a running example we will use the circuit of Figure 7.2 having one input variable x, fed into two delay elements, whose outputs, the internal variables y_1 and y_2, are the inputs for an AND gate whose output z is the output of the circuit. The full timed automaton for this circuit is depicted in Figure 7.3. The automaton has 16 states and 3 clocks, $\{c_x, c_{y_1}, c_{y_2}\}$.

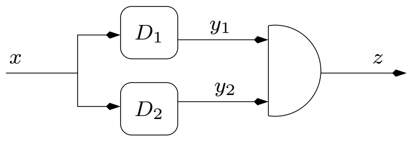


Figure 7.2: An example of circuit

7.2 Adding Input Clocks

7.2.1 Life and Death of Events

In this section we discuss the propagation of input events in an acyclic network of timed components. To facilitate the discussion we assume that all components have identical lower and upper bounds l and u on their reaction time. When such an input event occurs it may excite one or more of the components to which it is a direct input. In the absence of additional events, each of those components will stabilize within some $t \in [l, u]$ and emit a change in its output, which may trigger reactions in some further components, and so on. Due to acyclicity, a component which has reacted to an input event cannot be influenced anymore by the same event. Consequently all input events leave the system within a finite amount of time, bounded by $d \cdot u$, where d is the *depth* of the network, the maximal number of sequentially-connected components.

A second observation is that for an input event to be alive in the system there must be at least one active clock triggered by it, and since each clock is reset by one event, the number of live events is bounded by the number of clocks in the system which, in the case of circuits, is equal to the number of gates. This is an upper bound and in practice the maximal number of live events can be much smaller due to logical interference or additional bounded-variability assumptions concerning the external environment. In the sequel we assume m to be the upper bound on the number of live input events for each input variable.

In the circuit model that we use every two changes in the value of the same input variable must be separated by at least l time in order for both to be alive in the system, otherwise the second change aborts the first via a "regret" transition. Hence the maximal number of live events for each input variable is bounded by $m = d \cdot u/l$. Similar bounds will hold for other approaches for treating excessive input variability and, in fact, one may see that the number of living events in a network is always bounded by the number of its components.

7.2.2 From $\mathcal{A}$ to $\mathcal{A}^{+\hat{C}}$

From now on we assume additional properties of the timed automata that we obtain for circuits (and other well-structured networks of timed components) and which are preserved under composition. Let us recall that we consider timed automata with variables $\mathcal{A}_X = (\mathcal{A}, X, \lambda)$ where $\mathcal{A} = (Q, F_X, q_0, C, I, \Delta)$ is a timed automaton, X is a set of variables partitioned into input and state variables, $X = X_{in} \uplus X_{st}$, with $X_{in} = \{x_1, \ldots, x_n\}$, and a mapping $\lambda : Q \to \mathbf{V}_X$ from states to variable valuations. Transitions in Δ are of the form (q, f, g, γ, q') where f is the assignment which changes the variable valuation from $\lambda(q)$ to $\lambda(q')$. The additional assumptions are:

1. Every transition in Δ changes the value of *exactly one* variable. Consequently we can partition Δ into sets of *input* and *state* transitions $\Delta = \Delta_{in} \uplus \Delta_{st}$.

2. The automata are input-enabled (or receptive) in the sense that every change of an input variable is possible in every state and all transitions in Δ_{in} are guarded by $true$.

3. All transitions in Δ_{st} are guarded by clock constraints of the from $c \in [l, u]$ involving only a *single* clock.

It is not hard to see that the automata for the basic gates and delay elements described in Chapter 5 satisfy these properties, and that the composition described in Definition 4.5 preserves them because we use *interleaving* semantics for transitions of state variables and the only synchronized transitions are those that correspond to a change in an *input variable* shared by two or more components.

For every transition $\delta \in \Delta$ we let $\chi(\delta)$ be the set of clocks reset to zero by the transition. We say that a transition is *exciting* if $\chi(\delta) \neq \emptyset$. An exciting transition is a transition which triggers processes that will eventually be concluded by transitions guarded by clocks in $\chi(\delta)$.

We will augment every discrete state of the automaton with additional machinery that keeps track of the events which are alive in the system, and relates pending changes of states (represented by active clocks) to these events. Let $M = \{0, \ldots, m\}$ where m is the maximal number of live events in the automaton. The additional structure consists of:

- An *event-recording table* (ℓ, θ) consisting of
  - A *live-event counter* $\ell : X_{in} \to M$ indicating for each input variable x how many of its events are alive.
  - An *association function* $\theta : C \to (X_{in} \times M) \cup \{\bot\}$ relating every active clock with the input event which is responsible for its activation. Event (x, i) is understood to be the i^{th} oldest x-event which is still alive in the system.

  We denote the (finite) set of all event-recording tables for given X_{in} and M by $\mathcal{E}$.
- A set of *auxiliary input clocks*

$$\hat{C} = \{c_x[i] \mid x \in X_{in}, i \in M\}$$

  with the intended meaning that $c_x[i]$ is the time elapsed since the occurrence of event (x, i). A valuation of these clocks is a function $u : \hat{C} \to \mathbb{R}_\bot$. We will refer to the original clocks of the component as *internal* clocks.

We can now proceed to the construction of the automaton. Since this construction does not affect the mapping of states to variable valuations we define it in terms of $\mathcal{A}$ from which we construct $\mathcal{A}^{+\hat{C}} = (Q^{+\hat{C}}, F_X, q_0^{+\hat{C}}, C \cup \hat{C}, I^{+\hat{C}}, \Delta^{+\hat{C}})$ where $Q^{+\hat{C}} = Q \times \mathcal{E}$, $q_0^{+\hat{C}} = (q_0, \ell_0, \theta_0)$ where $\ell_0(x) = 0$ for every x and all clocks are inactive and $\theta_0(c) = \bot$ for every c.

The transition relation $\Delta^{+\hat{C}}$ consists of transitions of the form $\delta^{+\hat{C}} = ((q, \ell, \theta), f, g, \hat{\gamma}, (q', \ell', \theta'))$ where $\hat{\gamma}$ is a clock assignment on $C \cup \hat{C}$ whose projection on C is identical to γ. The updated event recoding table (ℓ', θ') and the extended assignment $\hat{\gamma}$ are computed according to the following algorithm.

Algorithm 7.1 Computing $\Delta^{+\hat{C}}$

```
Input: An extended state $(q, \ell, \theta) \in Q^{+\hat{C}}$ and a transition $\delta = (q, f, g, \gamma, q') \in \Delta$
Output: A transition $\delta^{+\hat{C}} = ((q, \ell, \theta), f, g, \hat{\gamma}, (q', \ell', \theta')) \in \Delta^{+\hat{C}}$

$\gamma' := \gamma; \ell' := \ell; \theta' := \theta$

if $\chi(\delta) \neq \emptyset$
  if $\delta \in \Delta_{in}$ changing input variable $x$   /* new event creation */
    $\ell'(x) := \ell'(x) + 1$
    $e := (x, \ell'(x))$
    $\gamma' := \gamma' \cup \{c_x[\ell(x)] := 0\}$
  elsif $\delta \in \Delta_{st}$ guarded by clock $c$   /* event propagation */
    $e := \theta(c)$
  for each $c' \in \chi(\delta)$
    $\theta'(c') := e$

for every non-resetting clock assignment in $\gamma$   /* book keeping */
  if of the form $c := \bot$
    $\theta'(c) := \bot$
  elsif of the form $c_1 := c_2$
    $\theta'(c_1) := \theta(c_2)$

for each $x \in X_{in}$   /* clock killing and shifting */
  for $i = 1$ to $\ell'(x)$
    if $\theta'^{-1}(x, i) = \emptyset$
      for $j = i$ to $\ell'(x) - 1$
        $\gamma' := \gamma' \cup \{c_x[j] := c_x[j+1]\}$
        for each $c \in \theta'^{-1}(x, j+1)$
          $\theta'(c) := (x, j)$
      $\ell'(x) := \ell'(x) - 1$
```

This procedure has four major parts.

1. Initialization: γ' inherits the clock assignments for the internal clocks from γ and the initial event-recording table is the same as in the source state.

2. Treatment of excitation: when a transition triggers new internal processes by resetting clocks, these clocks should be associated with the input event responsible for their excitation. There are two cases:

   (a) New event generation: the transition is due to a new input event on x and in this case we increment the x event counter, initialize a new input clock and keep the responsible event in a temporary variable e.

(b) Event propagation: the exciting transition was not related to a new input event, and in this case the responsible input event is inherited from the clock c which guarded the transition.

Then every clock reset by the transition is associated with the input event in e.

3. Association book keeping: when clocks become inactive or when they are shifted, their association is updated as well.[1]

4. Event death detection: if no clock points via θ to event (x, i) as its excitation reason, the event is no longer alive and we discard clock $c_x[i]$. To keep the number of clocks finite we "shift" clocks $c_x[i+1], \ldots, c_x[\ell(x)]$ to the left and modify the association function accordingly. We will use $kill(x, i)$ as a shorthand for this operation.

One can see that this procedure maintains the event-detection table well-formed in the following sense:

1. Only active internal clocks are associated with input events:
$$\theta(c) \neq \bot \text{ iff } v(c) \neq \bot.$$

2. These clocks are associated only with live events:
$$\theta(c) = (x, i) \text{ only if } i \leq \ell(x).$$

3. Each live event has at least one internal clock associated with it:
$$i \leq \ell(x) \text{ only if } \exists c\, \theta(c) = (x, i).$$

4. Only input clocks associated with live events are active:
$$u(c_x[i]) \neq \bot \text{ iff } i \leq \ell(x).$$

The automaton thus obtained for our example is shown in Figure 7.4. Since two x-events can be alive simultaneously in the circuit, the automaton has two additional clocks, $c_x[1]$ and $c_x[2]$. Note that two states of the original automaton (s_{11} and s_{12}) are split into two copies each, due to

[1]Shifting does not occur in the basic automata that model circuits, but they appear in the resulting abstraction (see below) and should be treated when we apply the abstraction procedure recursively.

different values of the event-recording table. To understand that consider the path

$$(s_9, 1111) \xrightarrow{x\downarrow} (s_{10}, 0111) \xrightarrow{y_2\downarrow} (s_{12}, 0101) \xrightarrow{x\uparrow} (s_{15}, 1101) \xrightarrow{x\downarrow} (s'_{12}, 0101)$$

starting from a stable state where all variables are high. The fall of x creates a new input event $(x, 1)$ with which both c_{y_1} and c_{y_2} are associated. Then when y_2 falls, it excites z and its clock is associated with $(x, 1)$ as well. When x rises again it cancels the excitation of y_1 but triggers a new excitation of y_2, this time related to the new event $(x, 2)$. Finally when x falls once more it aborts the excitation of y_2 (and hence kills event $(x, 2)$) and triggers a new excitation of y_1 associated with a new input event $(x, 2)$. The difference between s_{12} and s'_{12} is hence because $\theta(c_{y_1}) = (x, 1)$ in the former and $\theta(c_{y_1}) = (x, 2)$ in the latter.

7.3 Reachability, Projection and Minimization

We now describe briefly and informally the other steps of the procedure which are similar to the corresponding steps for the acyclic environment.

7.3.1 Reachability Computation

Automaton $\mathcal{A}^{+\hat{C}}$ thus constructed is an ordinary timed automaton except for the fact that the denotation of its input clocks may change from one state to another due to clock shifting. Its configurations are of the form $((q, \ell, \theta), (v, u))$ where (v, u) is a joint valuation of the internal and the input clocks.

After reachability computation we obtain the interpreted timed automaton whose states are of the form $((q, \ell, \theta), Z))$ where Z is a zone over both types of clocks. As described in Chapter 2, we intersect invariants and transition guards with these zones to obtain the interpreted timed automaton $\mathcal{A}^r_X = (\mathcal{A}^r, X, \lambda)$ with $\mathcal{A}^r = (Q^r, F_X, q^r_0, C \cup \hat{C}, I^r, \Delta^r)$.

7.3.2 Clock Projection

The automaton $\mathcal{A}^{\hat{C}} = (Q^r, F_X, q^r_0, \hat{C}, I^{\hat{C}}, \Delta^{\hat{C}})$ is constructed from $\mathcal{A}^r$ by projecting the timing constraints on clocks $\hat{C}$. For each state $p = ((q, \ell, \theta), Z) \in Q^r$ let $\hat{C}(p)$ be the set of input clocks active at p. The invariant of p in $\mathcal{A}^{\hat{C}}$ is

$$I^{\hat{C}}(p) = I^r(p)/_{\hat{C}(p)}.$$

For every transition $\delta = (p, f, g, \gamma, p') \in \Delta^r$ we define a transition $\delta^{\hat{C}} = (p, f, g^{\hat{C}}, \gamma^{\hat{C}}, p') \in \Delta^{\hat{C}}$ by letting

$$\gamma^{\hat{C}} = \gamma/_{\hat{C}}$$

and

$$g^{\hat{C}} = \begin{cases} \gamma/_{\hat{C}} & \text{if } \delta \in \Delta^r_{in} \\ \gamma/_{\theta(c)} & \text{if } \delta \in \Delta^r_{in} \text{ and is guarded by } c \end{cases}$$

The automaton obtained after reachability computation and projection on the input clocks is shown in Figure 7.5.

7.3.3 Variable Projection and Minimization

We now project $\mathcal{A}^{\hat{C}}$ on the set of interface variables $X_{io} = X_{in} \uplus X_{ou}$. This operation hides variable assignments that change internal variables (see Figure 7.6). However not all such transitions can be considered silent because they may kill an input clock. Suppose, for example, that input event (x, i) is responsible for the excitation of one internal variable y. It may happen that when y changes its value, this has no influence on the value of subsequent variables and hence event (x, i) dies and its clock should be removed. The change in clock denotation due to the shifting operation should be visible in the reduced model in order to preserve its intended semantics.

Let $\mathcal{A}_{X_{io}} = (Q, F_{X_{io}}, q_0, \hat{C}, I, \Delta)$ be the automaton obtained after the projection on the interface variables. The reduced model $\mathcal{A}_{X_{io}} = (Q^m, F_{X_{io}}, q_0^m, \hat{C}, I^m, \Delta^m)$ is obtained from $\mathcal{A}_{X_{io}}$ by performing the three minimization steps described in the previous chapter (collapsing silent transitions, merging bisimilar states and merging transitions). The only difference is that the action alphabet with respect to which we do minimization is $\Sigma = F_{X_{io}} \times \Gamma(\hat{C})$ consisting of pairs of the form (f, γ) with f being an assignment over interface variable and γ an assignment over input clocks. A transition is considered silent only if *both* f and γ are the identity functions.

After performing minimization with respect to this alphabet, we let the invariant of a merged state be the convex hull of the invariants of the individual states. Likewise we let the guard of a merged transition be the convex hull of the individual guards. Unlike the acyclic case where invariants were intervals, we do not have a guarantee for exact preservation of the quantitative semantics and additional behaviors may be added. The qualitative semantics is, however, preserved. It is worth mentioning that the clocks are not reset to zero by any silent transition.

The abstract model obtained at the end of the process is depicted in Figure 7.7. It has 11 states and 2 clocks and gives an over-approximation of the timed input-output behavior of the circuit. The reduction in this example is not so impressive because the circuit has only two internal variables and will be much more significant for larger circuits. Note also that we have assumed the most general environment that may switch with unbounded frequency, and adding bounded-variability assumptions will reduce the number of live events in a system. All steps in the procedure described in this chapter, except for the minimization have already been implemented.

7.4 Conclusions

We have developed a new original technique for abstracting the behavior of timed components. The essence of this technique is to *use the internal clocks* to compute what the component can and cannot do and then *get rid of these clocks* by projecting on the observable input clocks that we introduce into the model. As a result we obtain a model which focuses on what the potential users of the component care about: the *relation between the timing of input and output events*. This model is faithful to the qualitative semantics of the component but may relax the temporal correlation between output events that depend on the same internal event. Such a reduced model can serve as a specification of the component in a component library. The major advantage of the abstraction operator ρ defined in this chapter is that it can be used naturally within a component-based *hierarchical* development framework as described below.

It is not hard to see that ρ preserves the three properties defined at the beginning of Section 7.2.2, namely that each transition changes one variable, that input transitions are unguarded and that any state transition is guarded by a single-clock constraint. Hence, the procedure can be applied *recursively* at *all levels* of a component-based system. We can summarize the application of this procedure to a hierarchically designed system by the following recursive abstraction algorithm $\hat{\rho}$.

Algorithm 7.2 Hierarchical Abstraction $\hat{\rho}$.

```
Input: A component-based system $\mathcal{A}$
Output: A timed over-approximating abstraction of $\mathcal{A}$

if $\mathcal{A}$ is a basic component
  return$(\rho(\mathcal{A}))$
elsif $\mathcal{A} = A_1 \parallel A_2 \parallel \cdots \parallel \mathcal{A}_k$
  return$(\hat{\rho}(\hat{\rho}(A_1) \parallel \hat{\rho}(A_2) \parallel \cdots \parallel \hat{\rho}(\mathcal{A}_k)))$
```

Moreover, once an abstraction of a particular component has been computed, it can be stored in a library and be used each time that component appears rather than going down the recursion again.

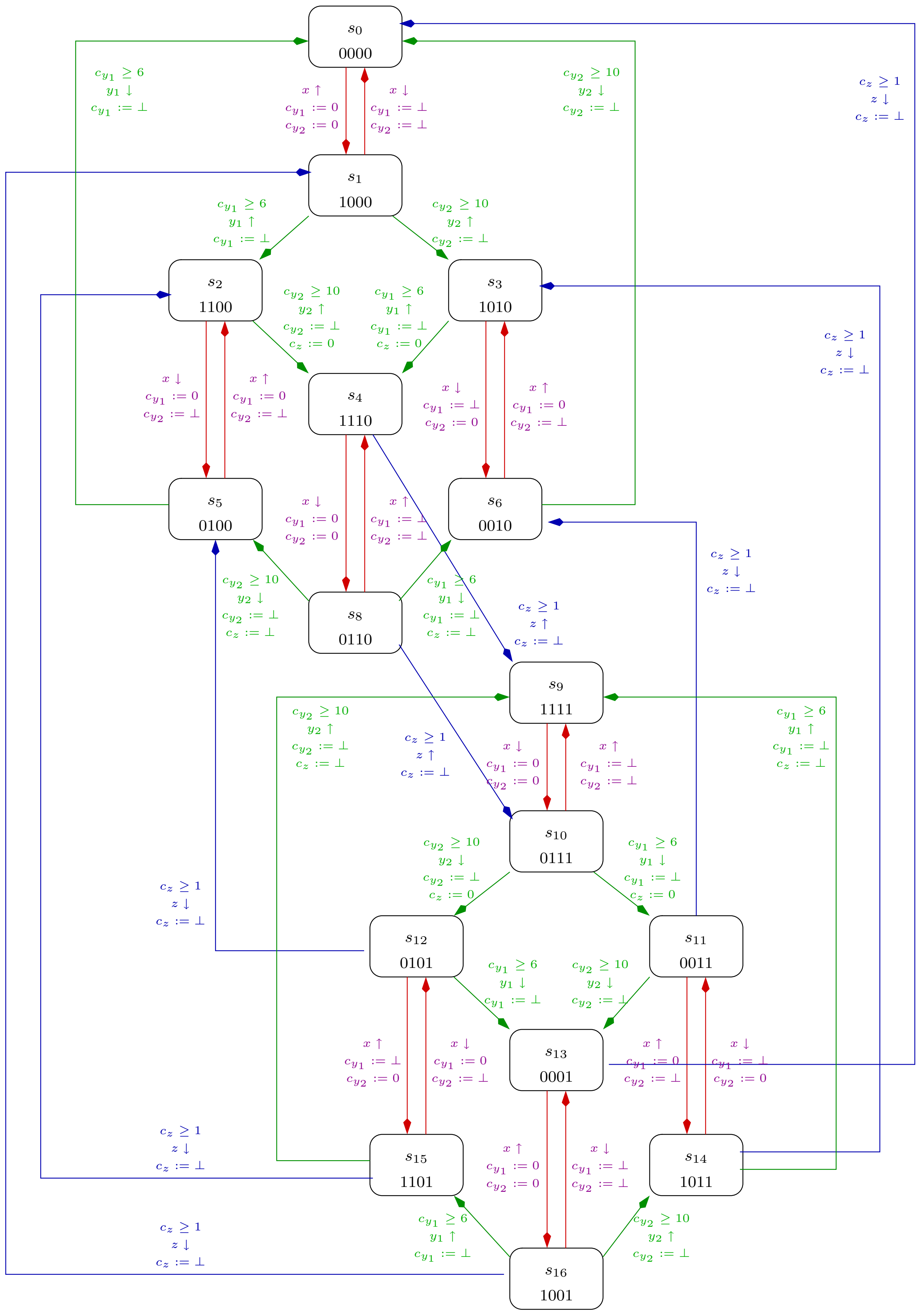

s_0 0000
s_1 1000
s_2 1100
s_3 1010
s_4 1110
s_5 0100
s_6 0010
s_8 0110
s_9 1111
s_{10} 0111
s_{11} 0011
s_{12} 0101
s_{13} 0001
s_{14} 1011
s_{15} 1101
s_{16} 1001
$c_{y_1} \geq 6$ $y_1 \downarrow$ $c_{y_1} := \perp$
$x \uparrow$ $c_{y_1} := 0$ $c_{y_2} := 0$
$x \downarrow$ $c_{y_1} := \perp$ $c_{y_2} := \perp$
$c_{y_2} \geq 10$ $y_2 \downarrow$ $c_{y_2} := \perp$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$
$c_{y_1} \geq 6$ $y_1 \uparrow$ $c_{y_1} := \perp$
$c_{y_2} \geq 10$ $y_2 \uparrow$ $c_{y_2} := \perp$
$c_{y_2} \geq 10$ $y_2 \uparrow$ $c_{y_2} := \perp$ $c_z := 0$
$c_{y_1} \geq 6$ $y_1 \uparrow$ $c_{y_1} := \perp$ $c_z := 0$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$
$x \downarrow$ $c_{y_1} := 0$ $c_{y_2} := \perp$
$x \uparrow$ $c_{y_1} := 0$ $c_{y_2} := \perp$
$x \downarrow$ $c_{y_1} := \perp$ $c_{y_2} := 0$
$x \uparrow$ $c_{y_1} := 0$ $c_{y_2} := \perp$
$x \downarrow$ $c_{y_1} := 0$ $c_{y_2} := 0$
$x \uparrow$ $c_{y_1} := \perp$ $c_{y_2} := \perp$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$
$c_{y_2} \geq 10$ $y_2 \downarrow$ $c_{y_2} := \perp$ $c_z := \perp$
$c_{y_1} \geq 6$ $y_1 \downarrow$ $c_{y_1} := \perp$ $c_z := \perp$
$c_z \geq 1$ $z \uparrow$ $c_z := \perp$
$c_{y_2} \geq 10$ $y_2 \uparrow$ $c_{y_2} := \perp$ $c_z := \perp$
$c_z \geq 1$ $z \uparrow$ $c_z := \perp$
$x \downarrow$ $c_{y_1} := 0$ $c_{y_2} := 0$
$x \uparrow$ $c_{y_1} := \perp$ $c_{y_2} := \perp$
$c_{y_1} \geq 6$ $y_1 \uparrow$ $c_{y_1} := \perp$ $c_z := \perp$
$c_{y_2} \geq 10$ $y_2 \downarrow$ $c_{y_2} := \perp$ $c_z := 0$
$c_{y_1} \geq 6$ $y_1 \downarrow$ $c_{y_1} := \perp$ $c_z := 0$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$
$c_{y_1} \geq 6$ $y_1 \downarrow$ $c_{y_1} := \perp$
$c_{y_2} \geq 10$ $y_2 \downarrow$ $c_{y_2} := \perp$
$x \uparrow$ $c_{y_1} := \perp$ $c_{y_2} := 0$
$x \downarrow$ $c_{y_1} := 0$ $c_{y_2} := \perp$
$x \uparrow$ $c_{y_1} := 0$ $c_{y_2} := \perp$
$x \downarrow$ $c_{y_1} := \perp$ $c_{y_2} := 0$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$
$x \uparrow$ $c_{y_1} := 0$ $c_{y_2} := 0$
$x \downarrow$ $c_{y_1} := \perp$ $c_{y_2} := \perp$
$c_{y_1} \geq 6$ $y_1 \uparrow$ $c_{y_1} := \perp$
$c_{y_2} \geq 10$ $y_2 \uparrow$ $c_{y_2} := \perp$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$

Figure 7.3: A timed automaton modeling the circuit of Figure 7.2.

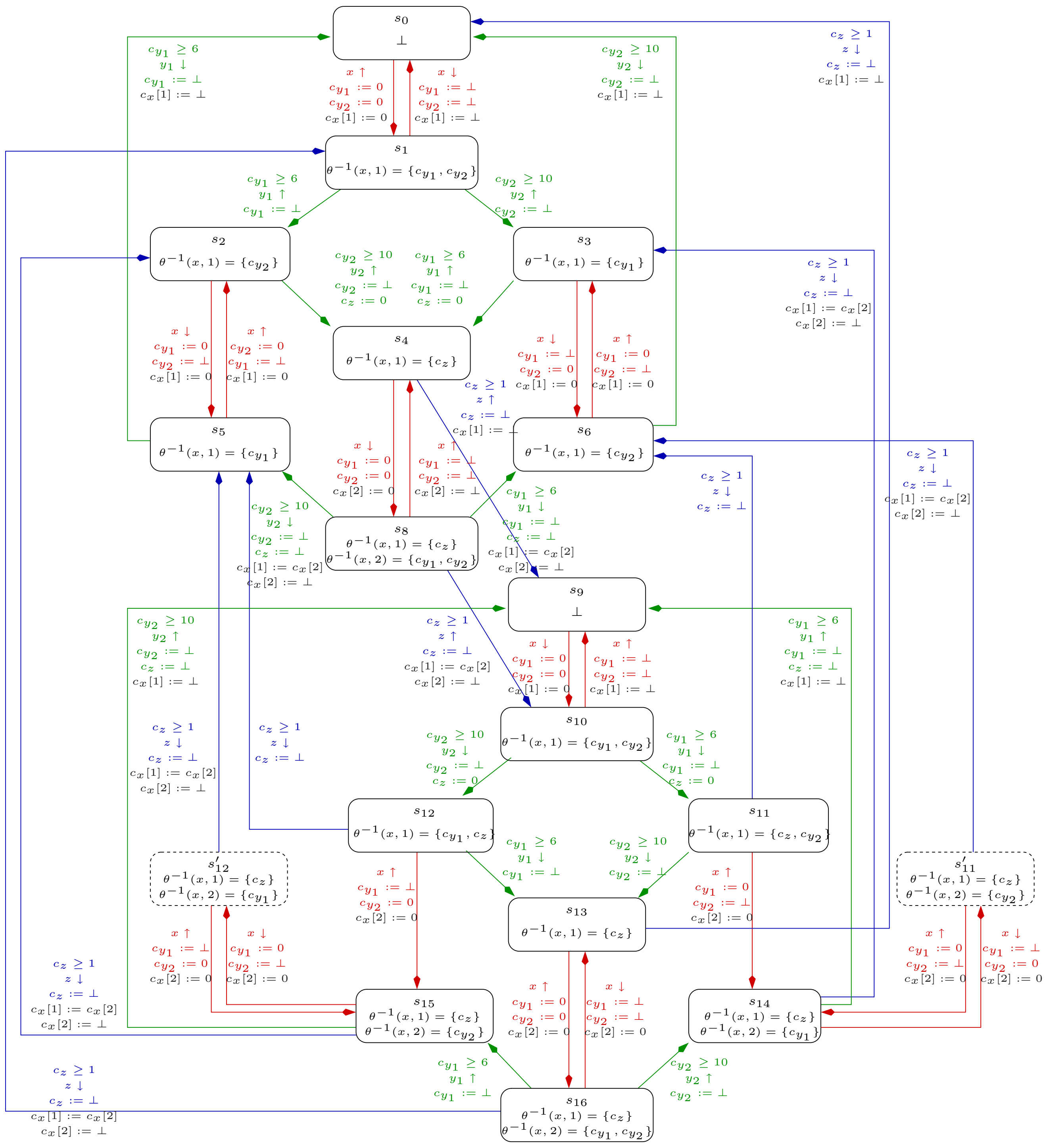

s_0
$\perp$
s_1
$\theta^{-1}(x,1) = \{c_{y_1}, c_{y_2}\}$
s_2
$\theta^{-1}(x,1) = \{c_{y_2}\}$
s_3
$\theta^{-1}(x,1) = \{c_{y_1}\}$
s_4
$\theta^{-1}(x,1) = \{c_z\}$
s_5
$\theta^{-1}(x,1) = \{c_{y_1}\}$
s_6
$\theta^{-1}(x,1) = \{c_{y_2}\}$
s_8
$\theta^{-1}(x,1) = \{c_z\}$
$\theta^{-1}(x,2) = \{c_{y_1}, c_{y_2}\}$
s_9
$\perp$
s_{10}
$\theta^{-1}(x,1) = \{c_{y_1}, c_{y_2}\}$
s_{11}
$\theta^{-1}(x,1) = \{c_z, c_{y_2}\}$
s_{12}
$\theta^{-1}(x,1) = \{c_{y_1}, c_z\}$
s'_{12}
$\theta^{-1}(x,1) = \{c_z\}$
$\theta^{-1}(x,2) = \{c_{y_1}\}$
s'_{11}
$\theta^{-1}(x,1) = \{c_z\}$
$\theta^{-1}(x,2) = \{c_{y_2}\}$
s_{13}
$\theta^{-1}(x,1) = \{c_z\}$
s_{14}
$\theta^{-1}(x,1) = \{c_z\}$
$\theta^{-1}(x,2) = \{c_{y_1}\}$
s_{15}
$\theta^{-1}(x,1) = \{c_z\}$
$\theta^{-1}(x,2) = \{c_{y_2}\}$
s_{16}
$\theta^{-1}(x,1) = \{c_z\}$
$\theta^{-1}(x,2) = \{c_{y_1}, c_{y_2}\}$
$x \uparrow$ $c_{y_1} := 0$ $c_{y_2} := 0$ $c_x[1] := 0$
$x \downarrow$ $c_{y_1} := \perp$ $c_{y_2} := \perp$ $c_x[1] := \perp$
$c_{y_1} \geq 6$ $y_1 \downarrow$ $c_{y_1} := \perp$ $c_x[1] := \perp$
$c_{y_2} \geq 10$ $y_2 \downarrow$ $c_{y_2} := \perp$ $c_x[1] := \perp$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$ $c_x[1] := \perp$
$c_{y_1} \geq 6$ $y_1 \uparrow$ $c_{y_1} := \perp$
$c_{y_2} \geq 10$ $y_2 \uparrow$ $c_{y_2} := \perp$
$c_{y_2} \geq 10$ $y_2 \uparrow$ $c_{y_2} := \perp$ $c_z := 0$
$c_{y_1} \geq 6$ $y_1 \uparrow$ $c_{y_1} := \perp$ $c_z := 0$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$x \downarrow$ $c_{y_1} := 0$ $c_{y_2} := \perp$ $c_x[1] := 0$
$x \uparrow$ $c_{y_2} := 0$ $c_{y_1} := \perp$ $c_x[1] := 0$
$x \downarrow$ $c_{y_1} := \perp$ $c_{y_2} := 0$ $c_x[1] := 0$
$x \uparrow$ $c_{y_1} := 0$ $c_{y_2} := \perp$ $c_x[1] := 0$
$c_z \geq 1$ $z \uparrow$ $c_z := \perp$ $c_x[1] := \perp$
$x \downarrow$ $c_{y_1} := 0$ $c_{y_2} := 0$ $c_x[2] := 0$
$x \uparrow$ $c_{y_1} := \perp$ $c_{y_2} := \perp$ $c_x[2] := \perp$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$c_{y_2} \geq 10$ $y_2 \downarrow$ $c_{y_2} := \perp$ $c_z := \perp$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$c_{y_1} \geq 6$ $y_1 \downarrow$ $c_{y_1} := \perp$ $c_z := \perp$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$c_{y_2} \geq 10$ $y_2 \uparrow$ $c_{y_2} := \perp$ $c_z := \perp$ $c_x[1] := \perp$
$c_z \geq 1$ $z \uparrow$ $c_z := \perp$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$x \downarrow$ $c_{y_1} := 0$ $c_{y_2} := 0$ $c_x[1] := 0$
$x \uparrow$ $c_{y_1} := \perp$ $c_{y_2} := \perp$ $c_x[1] := \perp$
$c_{y_1} \geq 6$ $y_1 \uparrow$ $c_{y_1} := \perp$ $c_z := \perp$ $c_x[1] := \perp$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$
$c_{y_2} \geq 10$ $y_2 \downarrow$ $c_{y_2} := \perp$ $c_z := 0$
$c_{y_1} \geq 6$ $y_1 \downarrow$ $c_{y_1} := \perp$ $c_z := 0$
$c_{y_1} \geq 6$ $y_1 \downarrow$ $c_{y_1} := \perp$
$c_{y_2} \geq 10$ $y_2 \downarrow$ $c_{y_2} := \perp$
$x \uparrow$ $c_{y_1} := \perp$ $c_{y_2} := 0$ $c_x[2] := 0$
$x \uparrow$ $c_{y_1} := 0$ $c_{y_2} := \perp$ $c_x[2] := 0$
$x \uparrow$ $c_{y_1} := \perp$ $c_{y_2} := 0$ $c_x[2] := 0$
$x \downarrow$ $c_{y_1} := 0$ $c_{y_2} := \perp$ $c_x[2] := 0$
$x \uparrow$ $c_{y_1} := 0$ $c_{y_2} := \perp$ $c_x[2] := 0$
$x \downarrow$ $c_{y_1} := \perp$ $c_{y_2} := 0$ $c_x[2] := 0$
$x \uparrow$ $c_{y_1} := 0$ $c_{y_2} := 0$ $c_x[2] := 0$
$x \downarrow$ $c_{y_1} := \perp$ $c_{y_2} := \perp$ $c_x[2] := 0$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$c_z \geq 1$ $z \downarrow$ $c_z := \perp$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$c_{y_1} \geq 6$ $y_1 \uparrow$ $c_{y_1} := \perp$
$c_{y_2} \geq 10$ $y_2 \uparrow$ $c_{y_2} := \perp$

Figure 7.4: Adding input clocks to the automaton of Figure 7.3. Variable valuations of states are the same as in Figure 7.3 and the association function is written inside the states.

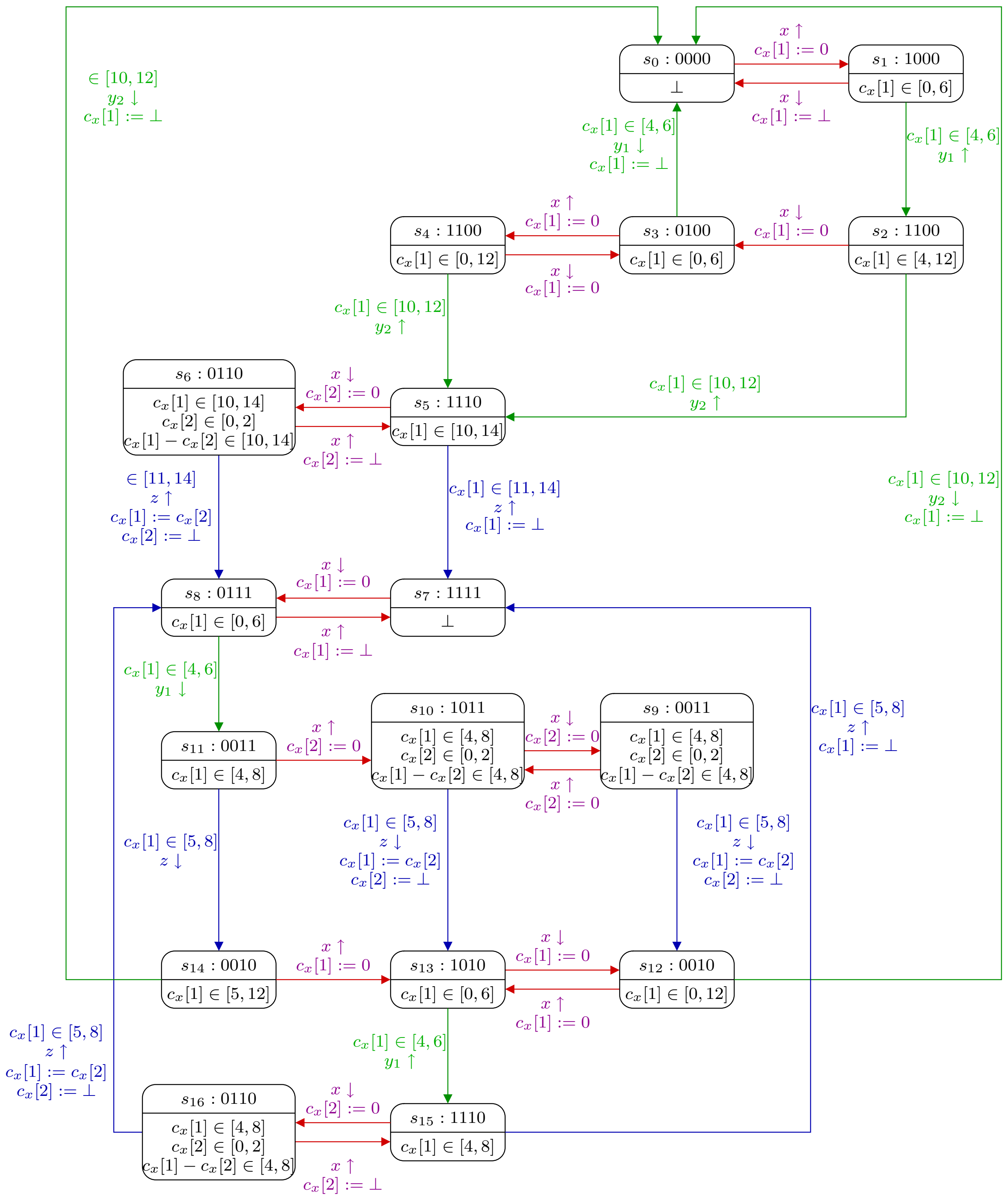


Figure 7.5: The automaton obtained from the automaton of Figure 7.4 by reachability computation and projection on the input clocks.

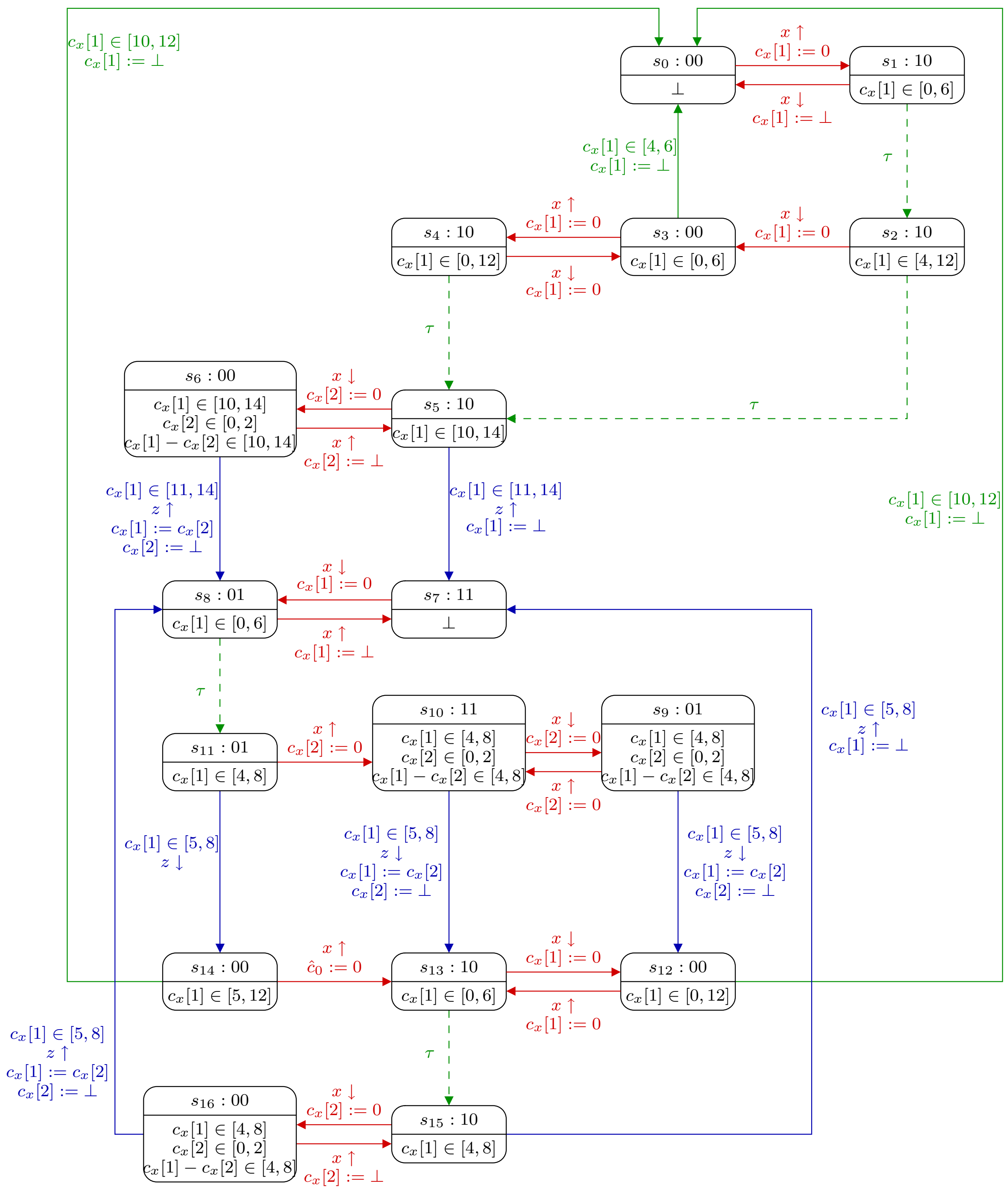
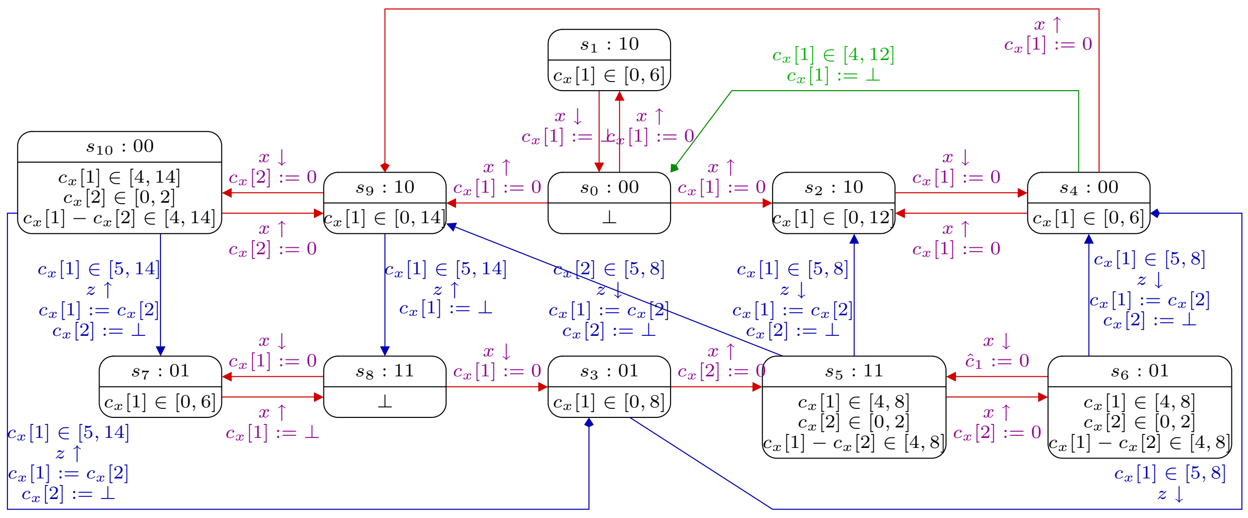

$s_0 : 00$
$\perp$
$s_1 : 10$
$c_x[1] \in [0,6]$
$s_2 : 10$
$c_x[1] \in [4,12]$
$s_3 : 00$
$c_x[1] \in [0,6]$
$s_4 : 10$
$c_x[1] \in [0,12]$
$s_5 : 10$
$c_x[1] \in [10,14]$
$s_6 : 00$
$c_x[1] \in [10,14]$ $c_x[2] \in [0,2]$ $c_x[1] - c_x[2] \in [10,14]$
$s_7 : 11$
$\perp$
$s_8 : 01$
$c_x[1] \in [0,6]$
$s_9 : 01$
$c_x[1] \in [4,8]$ $c_x[2] \in [0,2]$ $c_x[1] - c_x[2] \in [4,8]$
$s_{10} : 11$
$c_x[1] \in [4,8]$ $c_x[2] \in [0,2]$ $c_x[1] - c_x[2] \in [4,8]$
$s_{11} : 01$
$c_x[1] \in [4,8]$
$s_{12} : 00$
$c_x[1] \in [0,12]$
$s_{13} : 10$
$c_x[1] \in [0,6]$
$s_{14} : 00$
$c_x[1] \in [5,12]$
$s_{15} : 10$
$c_x[1] \in [4,8]$
$s_{16} : 00$
$c_x[1] \in [4,8]$ $c_x[2] \in [0,2]$ $c_x[1] - c_x[2] \in [4,8]$
$c_x[1] \in [10,12]$ $c_x[1] := \perp$
$x\uparrow$ $c_x[1] := 0$
$x\downarrow$ $c_x[1] := \perp$
τ
$c_x[1] \in [4,6]$ $c_x[1] := \perp$
$x\uparrow$ $c_x[1] := 0$
$x\downarrow$ $c_x[1] := 0$
$x\downarrow$ $c_x[1] := 0$
τ
$x\downarrow$ $c_x[2] := 0$
$x\uparrow$ $c_x[2] := \perp$
τ
$c_x[1] \in [11,14]$ $z\uparrow$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$c_x[1] \in [11,14]$ $z\uparrow$ $c_x[1] := \perp$
$c_x[1] \in [10,12]$ $c_x[1] := \perp$
$x\downarrow$ $c_x[1] := 0$
$x\uparrow$ $c_x[1] := \perp$
τ
$x\uparrow$ $c_x[2] := 0$
$x\downarrow$ $c_x[2] := 0$
$x\uparrow$ $c_x[2] := 0$
$c_x[1] \in [5,8]$ $z\uparrow$ $c_x[1] := \perp$
$c_x[1] \in [5,8]$ $z\downarrow$
$c_x[1] \in [5,8]$ $z\downarrow$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$c_x[1] \in [5,8]$ $z\downarrow$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$x\uparrow$ $\hat{c}_0 := 0$
$x\downarrow$ $c_x[1] := 0$
$x\uparrow$ $c_x[1] := 0$
τ
$c_x[1] \in [5,8]$ $z\uparrow$ $c_x[1] := c_x[2]$ $c_x[2] := \perp$
$x\downarrow$ $c_x[2] := 0$
$x\uparrow$ $c_x[2] := \perp$

Figure 7.6: The automaton obtained from the one of Figure 7.5 after hiding internal variables.



$s_0 = \{s_0\}$
$s_1 = \{s_1\}$
$s_2 = \{s_2, s_4, s_{13}\}$
$s_3 = \{s_{11}\}$
$s_4 = \{s_3, s_{12}, s_{14}\}$
$s_5 = \{s_{10}\}$

$s_6 = \{s_9\}$
$s_7 = \{s_8\}$
$s_8 = \{s_7\}$
$s_9 = \{s_5, s_{15}, s_{17}\}$
$s_{10} = \{s_6, s_{16}\}$

Figure 7.7: The final automaton obtained by minimization from the automaton of Figure 7.6. Bisimulation equivalence classes are indicated at the bottom of the figure. The over-approximated the behavior of the circuit in Figure 7.2 and the automaton of Figure 7.3.

Part III

Achievements and Conclusions

Chapter 8

TCA: Timed Circuits Analyzer Toolbox

8.1 Introduction

It is fair to say that the experimental platform implemented during this thesis has been the driving force of the development of our ideas. Major contributions introduced in this work has been observed through examples of systems studied thinks to the implementation process.

This chapter introduces the tool TCA that represents an important contribution of this work. The first version of the tool has more strongly focused on the study of digital circuits. The first version of this tool implements the idea described in Chapter 6. It accepts a timed digital circuit description, extract the cone of influence of a given variable set to be studied, and then splits the resulting circuit into several partitions to model them one after one as explained before. The ideas evolved, as well as the state of the tool. This chapter expose only the actual version of the tool, which is related to the ides exposed in the last chapter.

8.2 Global View of the TCA Tool

The main input of the tool TCA is a structural description of a system. This description should be given as a file having, by convention, the same name as the component with ".tc" as extension. The goal of the tool is to generate a timed automaton describing the behavior of the interface of this component. The internal functioning of the tool as well as the properties of the automaton to be generated depend on a certain configuration of the tool. This configuration could be given by the user through a specified file. The configuration file is perhaps the most difficult description to write for a beginner. In order to make the tool accessible to larger class of users the command

```
$  tca -config [fileName.cf]
```

generates a standard configuration file `fileName.cf.` The current version of this file is given by appendix 9.2. The user can start from this well commented file and then adapt it according to its will. This technique is used in many tools, such as Doxigen. It is often the least expensive solution to hide the complexity of using certain tools.

Let `configFile.cf` be the configuration file fixed by the user, and let "`ComponentName.tc`" be the file describing the structure of the component to model, the command to start the modeling process and end up generating the behavior of the interface of the component is the following:

```
$  tca configFile.cf componentName.tc
```

In the framework of this thesis, the behavior of a component, or its interface, are given through a timed automaton. This one is expressed as an IF process. This process is delivered as a file having by convention the same name as the modeled component, with ".ifp" as extension. For example, the file generated by the above command will be named "`componentName.ifp`". This file will be saved for a later use. In fact, this model expressed in a such format, could be directly exploited by the tool IF in order to analyze some properties of the component. It could be also reused to model other components that incorporate it. As an example, the appendix 9.2 gives the model of a xor gate into the IF format. This could give a first idea about the syntax of this format. For better definition of this syntax we invite the reader to refer to [BSGS04].

Structural Description To have an idea about the format of a structural description of a component, we consider the component $\mathcal{C}_4$ given by Figure 8.2. Its description is given by the file *C4.tc* below.[1]

The structural description of a component should first specify the global interface of the component. This interface is given by the first line expressing the set of the input variables, together with the second line expressing the set of output variables. The rest of the file describes, one after one, all the sub-components of the system. Each sub-component is described in one line, and that is in one of two possible formats:

1. The first syntax is specific to logic gates with well specified delays. This could be given by the following syntax.

   ```
   output-variable : [rising-delay][falling-delay] logic-function;
   ```

[1]This choice of the extension ".tc" for structural descriptions has historical reasons. The first version of the tool does not consider component based systems, but just "**t**imed **c**ircuits" where all the components are simple logic gates with delay elements.

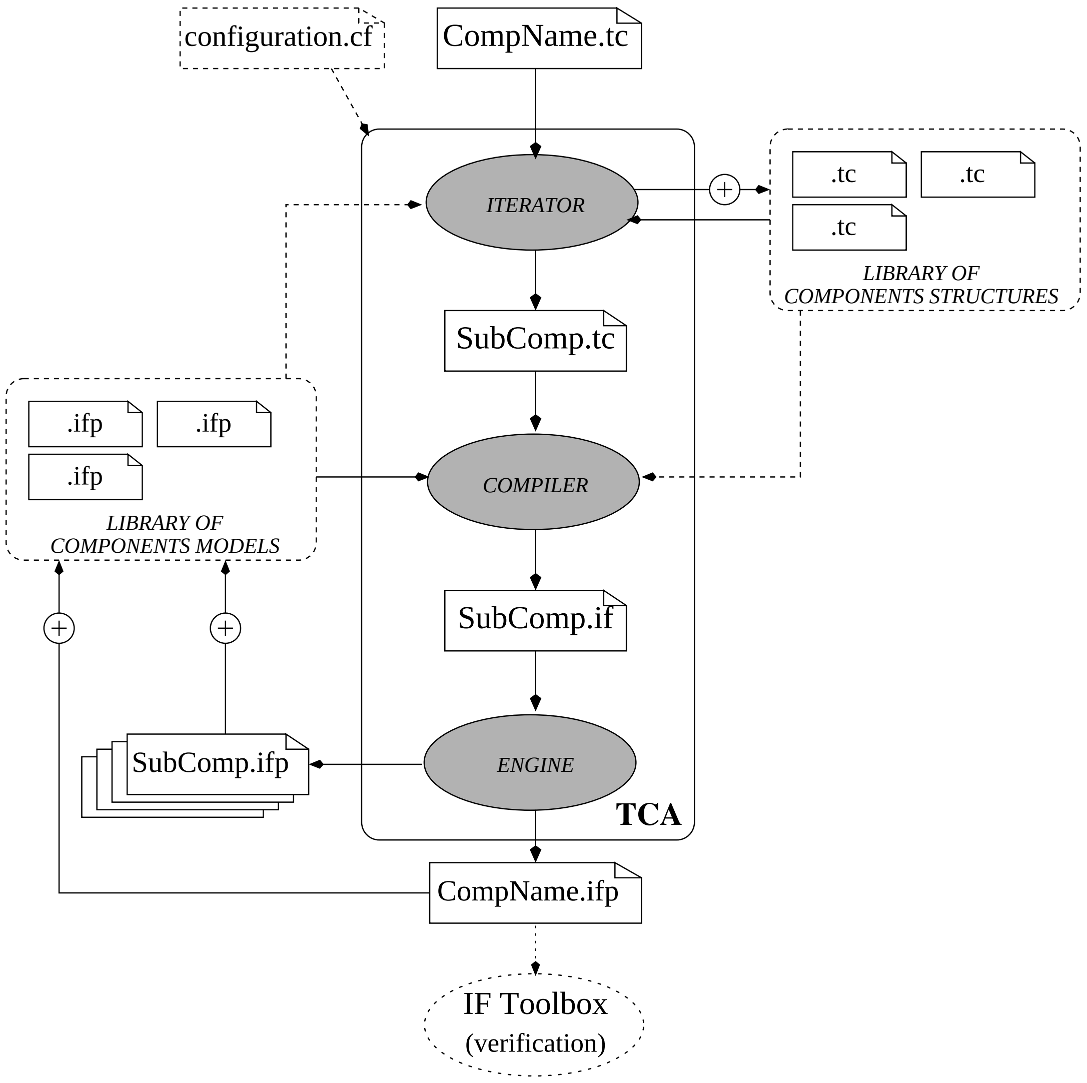
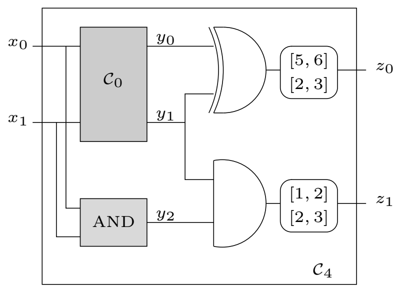


Figure 8.1: A Global view of the TCA tool.



C4.tc :

```
input    {x0; x1}
output   {z0; z1}

(y0, y1)    :  C0 (x0, x1);
y2          :  AND (x0, x1);
z0          :  [5,6][2,3] ((~y0*y1)+(y0*~y1));
z1          :  [1,2][2,3] (y1*y2);
```

Figure 8.2: A digital circuit and its structural description.

The two last lines of the file *C4.tc*, are examples of this description. Such components have a single output variable, the timing of the rising of its value is given by the first interval, the timing of the falling is given by the second interval. The functional aspect of the gate is expressed as logic function on its input variables. The operators “~”, “+”, and “*”, used in such description are denoting the "NOT", the "OR" and the "AND" operations.

2. The second syntax used to describe components is more appropriate for component based systems. The syntax of this description is as follow:

```
output-var-ordered-list : sub-comp-name (input-var-ordered-list);
```

   This syntax is describing only the connection of the sub-component to the rest of the network. The two first sub-components of the file *C4.tc* are described in this way: the input variables of the sub-component $\mathcal{C}_0$ of $\mathcal{C}_4$ are then x_0 and x_1, and its output variables are y_0 and y_1. Note that the order of the input and output variables lists of a sub-component are important.

In the case of simple logic gates, note that the two above description could be used. For instance an AND gate could be specified in the two different ways:

- `y2 : [1,2][2,3] (x0*x1);`
- `y2 : AND (x0,x1);`

The first description is giving the timing of the gate, then it is possible for the tool to generate its model automatically. The generated model could be based on the one emphasized in this thesis, defined by [MP95a], but other options exist. This can make the use of the tool easier, because with the second description of the AND gate, the model of this component should be given by the user. Note that the second way is more flexible, since very specific models, could be written for a given gate.

8.3 Deeper View of the TCA Tool

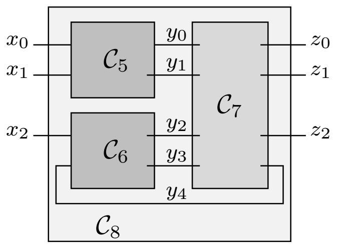
As expressed on the diagram of Figure 8.1, the system is based on three major modules. This section gives an idea on the role that every module of them is playing and how they are collaborating.

8.3.1 The ITERATOR

The task of the ITERATOR is to provide the COMPILER with a sequence of structural description of components to model. At each time, all the sub-components of the provided component structure should be already modeled. In other words, this module will perform the algorithm 7.2

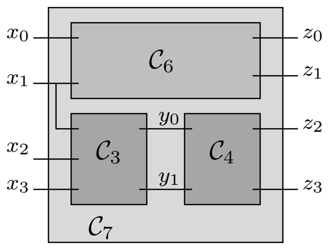
of the last chapter. Its role is to verify if the behavior descriptions of every sub-component of considered component exists in the models library. If this is the case, it will send the structural description of the component to the COMPILER; otherwise it will start first by giving the order to built the missing models. For that purpose the ITERATOR will consider every sub-component, which is not modeled yet, and then continue recursively. Since we assume that at least the models of the basic elements of the considered structures are given, the ending of the ITERATOR process is guaranteed.

Example Let us consider the component $\mathcal{C}_8$, the structure of which is given by the file *C8.tc*. This component is based on the components $\mathcal{C}_4$, $\mathcal{C}_5$, $\mathcal{C}_6$, and $\mathcal{C}_7$, given by their structural descriptions *C4.tc*, *C5.tc*, *C6.tc*, and *C7.tc*.[2] We assume that the models of the components AND, $\mathcal{C}_0$, $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ are already given, for they are basic components, or because their models have been computed in a previous modeling process.



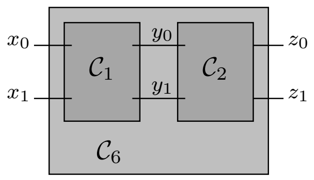
C8.tc :

```
input    {x0; x1; x2}
output   {z0; z1; z2}

(z0, z1, z2, y4)  :  C7
(y1, y2, y3, y4);
(y2 ,y3)          :  C6 (x2, y4);
(y0 ,y1)          :  C5 (x0, x1);
```



C7.tc :

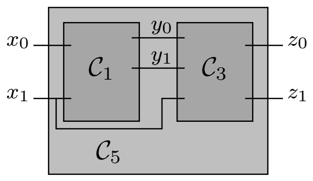
```
input    {x0; x1; x2; x3}
output   {z0; z1; z2; z3}

(z0, z1)          :  C6 (x0, x1);
(z2 ,z3)          :  C4 (y0, y1);
(y0 ,y1)          :  C3 (x1, x2, x3);
```



C6.tc :

```
input    {x0; x1}
output   {z0; z1}

(z0, z1)          :  C2 (y0, y1);
(y0 ,y1)          :  C1 (x0, x1);
```

[2]The component $\mathcal{C}_4$ is described by the example of the section 8.2.



C5.tc :

```
input    {x0; x1}
output   {z0; z1}


(z0, z1)          :  C3 (y0, y1, x1);
(y0 ,y1)          :  C1 (x0, x1);
```

Figure *8.3* gives an idea on the arborescence in the structure of this component. On this figure, components for which a model already exists are colored.

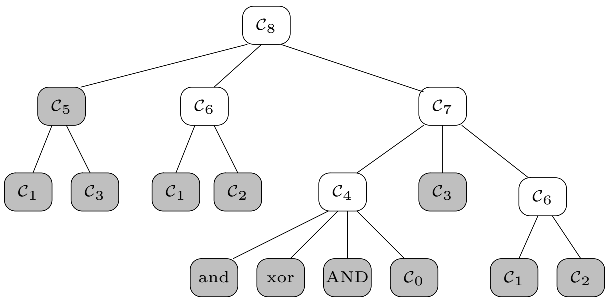


Figure 8.3: The arborescent structure of a component based system.

To model the component $\mathcal{C}_8$ the ITERATOR should send the file *C8.tc* to the COMPILER module. But to do so, the sub-components $\mathcal{C}_6$ and $\mathcal{C}_7$ should be modeled first. For the same reasons, $\mathcal{C}_7$ can not be modeled before $\mathcal{C}_6$ and $\mathcal{C}_4$. Finally, one decision that the ITERATOR can take is to send the non-modeled component to the COMPILER, in the following ordered *C6.tc, then C4.tc, then C7.tc, then C8.tc*.

8.3.2 The COMPILER

The COMPILER is supposed to receive from the ITERATOR a structural description of a component. The models of all the sub-components referred by this description should have been already computed, or given by the user, or possible to be generated automatically.

After verifying the consistency of the structure given at its input the compiler start translating each sub-component of the network to an IF process. If the component is expressed as a logic gate with rising and falling delay, the model of the gate will be generated directly. Unless, the component should be an instance of a class for which the model has been computed and saved. In this case, a simple renaming process will generate the corresponding IF Process.

8.3.3 The ENGINE

This module represents the major technical effort. It accepts at its input an IF file. This file describe the models of set of communicating components. The role of the ENGINE is to generate

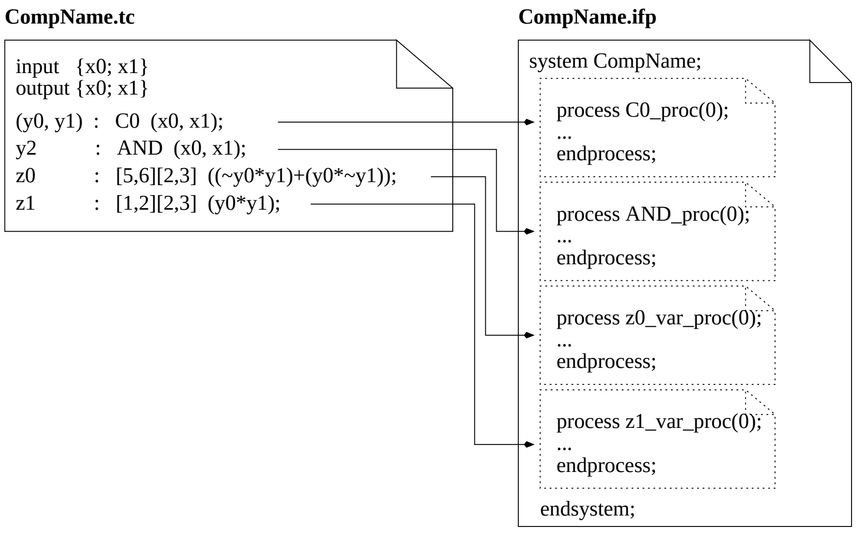


Figure 8.4: The transformation performed by the COMPILER module.

a model of the interface of their product. Part of this module is built on the top of the IF-Engine. So before giving an overview of this module, we first give a quick idea on the IF toolbox.

IF Tool Box

IF toolbox is an environment developed in Verimag, for modeling and validation of communicating real-time systems. The toolset is built upon a formalism, the IF language, allowing structured automata-based system representations. Since its definition in 1998, the IF language was continuously improved and become the interchange format between a set of validation tools dedicated to real time systems. The whole IF toolset architecture is given in figure 8.5. More details about different components of this architecture can be found in [BSGS04].

A part from its functionality, the quality and the modularity of the API that this toolbox presents made from it the starting point of many projects inside and outside the laboratory Verimag. To implement the the module ENGINE of TCA, we reused mainly the libraries of the IF-Engine (represented by the colored box on Figure 8.5).

Overview of the the ENGINE

The input of the ENGINE module, as shown on figure 8.6, is an IF specification, that describes a system as a set of timed automata, each is expressed as a process. The role of this module is to perform the process of abstraction exposed in the last chapter.

As explained formally in the last chapter, the abstraction is performed as a sequence of steps. The starting step consists of generating the interpreted timed automata of the product with the

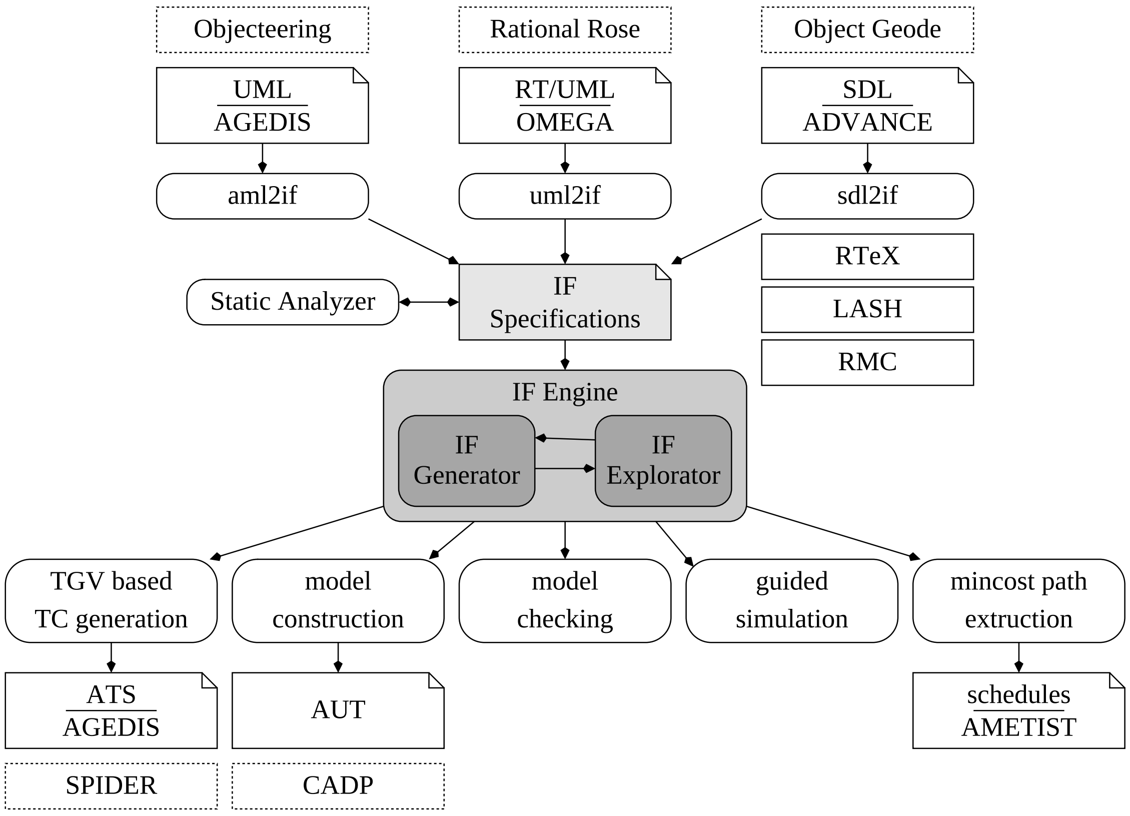


Figure 8.5: The IF-toolbox.

extra global clocks. This step is done by the collaboration between the two modules: TCA-Generator and TCA-Explorator. Briefly, the role of the Explorator is to generate the automaton, asking continuously the Generator module to compute the successors of a given symbolic state.

The TCA-Generator is based on the libraries of the IF-Generator, but is different from this former in several aspect. The major difference, is that the synchronization considered in the TCA-Generator is based on shared variables, which is not the case with IF. The second major difference is the dynamic aspect of the clocks. With IF-Generator, the number of the clocks is static and must be declared a priory. However, with TCA-Generator, the global number of clocks needed including the clocks related to the input events, can not really be decided at the beginning. It computes clocks dynamically, killing off inactive ones, and ensure a canonical form of the DBMs to control to the maximum the size of the model.

The TCA-Explorator is also based on the libraries of IF-Explorator of Figure 8.5. The two major aspects added in TCA-Explorator are:

1. The first is the Forward Reachability Algorithm introduced in Chapter 3, used to reduce the state explosion related to the interleaving semantics. This algorithm merges, on the fly, states having the same shuffle expression given by Definition 3.3. That induces a dynamic

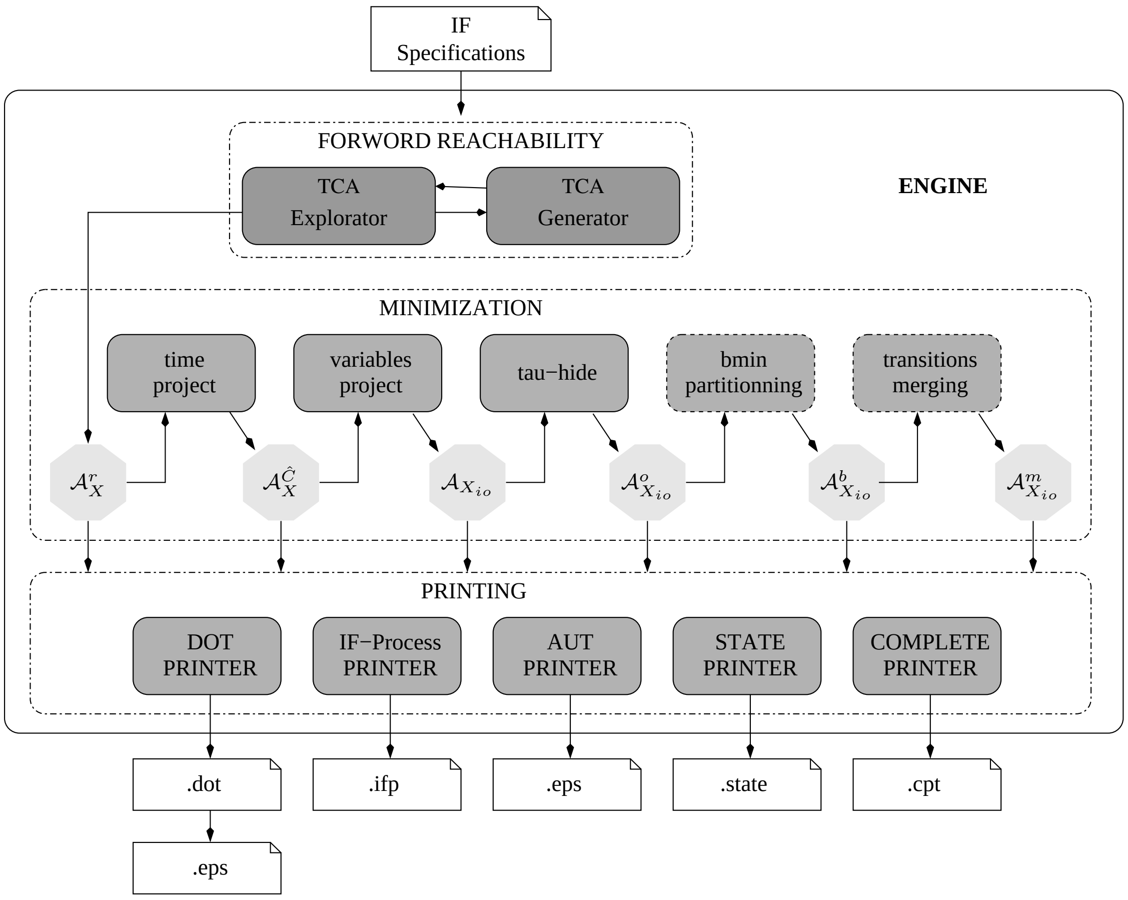


Figure 8.6: An overview of the ENGINE module.

management of the symbolic states, which has not been necessary with IF.

2. The second difference is that TCA-Explorator should generate the whole structure of the generated automaton for later steps, while the IF-Explorator can print it on the fly without saving any transition data. Only reached states should be saved, by the IF-Explorator, while the whole structure should be saved by TCA. To control the size of such systems, that can grow very quickly, efficient and complex data structures are needed.

The built model $\mathcal{A}^r_X$ generated by the TCA-Exlporator will follow a sequence of transformations: $\mathcal{A}^r_X$ will be transformed first by projecting the time constraints, represented as DBMs, on well defined global clocks. The resulting model $\mathcal{A}^{\hat{C}}_X$ is projected on the interface variables, which leads to the model $\mathcal{A}_{X_{io}}$. This model will be minimized after in three steps, as shown on Figure 8.5.

All the models $\mathcal{A}^r_X$, $\mathcal{A}^{\hat{C}}_X$, $\mathcal{A}_{X_{io}}$, $\mathcal{A}^o_{X_{io}}$, $\mathcal{A}^b_{X_{io}}$ and $\mathcal{A}^m_{X_{io}}$ are based on the same structures, every one of them could be printed in several formats using five different PRINTERS modules:

1. The DOT format: for viewing.

2. Le IFP format: expressing a timed automaton in the IF format.
3. The LTS format: which is recognized by several tools such as aldebaran.
4. The STATE format: describing in details every state of the automaton.
5. The CPT format: useful in the final minimization steps. At this level, a state could be a partition including several states of a previous model. A transition could be also a group of transitions. This file gives an idea on the content of every transition and state.

8.4 Conclusion

All in all the implementation effort consist of 113 classes and 35K lines of code. Even not completely finalized, the TCA tool has been at the base of the ideas introduced in this thesis. The tool consists of several modules. At the opposite of the earlier modules of the process, which have been tested and improved several times, the last two modules, especially the module performing the bisimulation refinement is still to be finalized. This represents actually the most urgent thing to complete.

Chapter 9

Conclusions and Future Work

We believe this work lays down the foundation for an eventual solution of the major problem of timed automata technology, the fact that it is not scalable. Below we summarize the achievements and discuss some of the work that still has to be done.

9.1 Summary

In this thesis we started by explaining the importance of extedning verification and analysis methodology to timed systems. Timed automata provide a model in which the *performance* of a system, hardware and software alike, can be naturally expressed. Then we have described the state-of-the-art in the analysis of timed automata using a zone-based forward reachability algorithm. We proposed a major improvement to this algorithm, taking advantage of the fact that the the union of zones reached by different interleavings of independent action is convex. Although this algorithm allowed us to analyze systems larger than what we could before, the problem of unscalability remained and in the rest of the thesis we have opted for a *compositional* divide-and-conquer methodology.

To facilitate the description of such a modular approach we have introduced timed automata with discrete state variables whose sharing is the way the automata interact. We have then described digital circuits with bi-bounded delays as the application domain on which we experiment with our techniques, although it applies as well to other systems consisting of interacting timed components. We then developed two abstraction techniques that take such a network of timed components, modeled as a network of timed automata, and generates from it reduced models with less states and clocks. The reduced models over-approximate the timed behavior while preserving its qualitative untimed behavior.

The first technique dealt with the special case of acyclic environments that generate a finite number of events within a bounded time. For such systems we produce a reduced model which is by itself an acyclic generator for the system's output. The essence of the technique is in introducing an auxiliary clock which measures absolute time, and then projecting the zones obtained from reachability computation on this clock. The final reduced model is obtained by hiding internal transitions and performing minimization.

The second technique is aimed at more general situations where the environment may produce an infinite stream of events. In this case we introduce a set of dynamically-created clocks associated with input events which are discarded and reused according to the propagation of events throuhout the system. Performing a similar type of reachability computation, projecting on these input clocks and minimizing we obtain models that relate, in an approximative manner, the timing of input and output events. We have proposed a recursive abstraction algorithm based on this technique which applies naturally to hierarchically-structured component-based systems.

This thesis involved a lot of implementation effort not all of which is reported. It includes the major ingredients of a hierarchical analysis methodology for network of timed components starting from high-level specifications of circuits, via translation to timed automata, an efficient analysis algorithm and an implementation of most of the steps of the abstraction techniques.

9.2 Future Work

We now mention what remains to be done, both in terms of completing the implementation and further theoretical and empirical investigations.

- The most urgent thing is to complete the implementation of the minimization step for the model with input clocks and thus complete a full tool chain.
- Once this is done we will experiment with examples, both synthetic and real-life in order to assess the complexity reduction obtained by the abstraction technique and estimate how far we can go in terms of size.
- Develop a methodology to assess the over-approximation incurred by the abstraction process. For the moment there is no analytical way to compare the "distance" between the semantics of a system and its approximation. Perhaps a method based on random simulation can be found useful.
- The minimization procedure should be studied more thouroughly. In particular, since circuits are non-blocking, bisimulation is a stronger relation than needed and it can be repalced by a weaker notion.

- The semantic interference between the convexity reduction and the abstraction procedure should be studied in order to integrate the two procedures.

- The effect adding bounded variability assumptions should be studied empiricially. They will certainly allow to analyze larger systems.

- Other modifications and improvements of the abstraction techniques are possible. In particular, using a different pool of clocks for each input variable rather than a joint pool ordered by arrival has its advanatages and disadvantages in terms of facilitating state merging and this aspect should be empirically studied.

- The implementation should be extended to treat non-Boolean discrete variables as well as systems whose events are not necesssarily related to changes in variables.

Bibliography

[AAM06] Y. Abdeddaim, E. Asarin, and O. Maler. Scheduling with timed automata. *Theoretical Computer Science*, 354:272–300, 2006.

[ABB$^+$01] T. Amnell, G. Behrmann, J. Bengtsson, P.R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K.G. Larsen, M.O. Möller, P. Pettersson, C. Weise, and W. Yi. UPPAAL - Now, Next, and Future. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modelling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science Tutorial, pages 100–125. Springer–Verlag, 2001.

[ACM02] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.

[AD90] R. Alur and D.L. Dill. Automata for modeling real-time systems. In *ICALP*, pages 322–335, 1990.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BBD$^+$02] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL implementation secrets. In *Proc. of 7*th *International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002.

[BBF$^+$01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[BCC$^+$03] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[BCM^{+}90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[BCOQ93] F. Baccelli, G. Cohen, G.J Olsder, and J.P. Quadrat. *SYNCHRONIZATION AND LINEARITY:An Algebra for Discrete Event Systems*. 1993.

[BD91] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. volume 17, pages 259–273, 1991.

[Beh03] Gerd Behrmann. *Data Structures and Algorithms for the Analysis of Real Time Systems*. PhD thesis, Aalborg University, November 2003.

[BFG^{+}91] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In *Proceedings of 18th ICALP*. Springer Verlag, 1991.

[BJLY98] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reductions for timed systems. In *CONCUR'98*, pages 485–500, 1998.

[BLL^{+}95] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verication of realtime systems. pages 22–24, October 1995.

[BM83] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time petri nets. pages 41–46, 1983.

[Bou02] P. Bouyer. Timed automata may cause some troubles. Research Report LSV-02-9, Laboratoire Spécification et Vérification, ENS Cachan, France, July 2002. 29 pages.

[BS95] J.A. Brzozowski and C.J.H. Seger. *Asynchronous Circuits*. Springer, March 1995.

[BSGS04] M. Bozga, I. Ober S. Graf, I. Ober, and J. Sifakis. Tools and applications ii: The if toolset. In Springer-Verlag, editor, *SFM'04*, volume 3185, 2004.

[BST97] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *COMPOS*, pages 103–129, 1997.

[BW94] B. A. Brandin and W. M. Wonham. Scheduling algorithms for multiprogramming in a hard-real-time environment. *IEEE transactions on automatic control*, 39(2):329–342, 1994.

[CE81] E. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. volume 131. Springer, 1981.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[CL06] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Cot69] L. W. Cotton. Maximum-rate pipeline system. 34:581–586, 1969.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[Daw98] C. Daws. *Methodes d'analyse de systemes temporises : de la theorie a la pratique*. PhD thesis, Institut National Polytechnique de Grenoble, France, October 1998.

[dBHdRR92] J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg, editors. *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*, volume 600 of *Lecture Notes in Computer Science*. Springer, 1992.

[Dil89] D. Dill. Timing assumptions and verification of finite-state concurrent systems. pages 197–212. Springer-Verlag New York, Inc., 1989.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, volume 1066, pages 208–219, Rutgers University, New Brunswick, NJ, USA, 22–25 October 1995. Springer.

[DR95] V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995.

[DRGK98] D. Dams, B. Knaack R. Gerth, and R. Kuiper. Partial-order reduction techniques for real-time model checking. volume 10, pages 469–482, 1998.

[DY92] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. pages 73–81, December 1992.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. 2006.

[God96] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[HK90] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. volume 69, pages 45–59, 1990.

[HMP92] T. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, volume 600, pages 226–251. Springer-Verlag, 1992.

[HNSY94] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. volume 111, pages 193–244, 1994.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[Kur94] R. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach.* Princeton University Press, 1994.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LNZ05] D. Lugiez, P. Niebert, and S. Zennou. A partial order semantics approach to the clock explosion problem of timed automata. volume 345, pages 27–59, 2005.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, New York, NY, USA, 1985. ACM Press.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in Lecture Notes in Computer Science, pages 62–88, August 1995.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[LS91] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[Mal94] S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design*, 13(7):950–956, 1994.

[McM92a] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, 1992.

[McM92b] K.L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[MF76] P. M. Merlin and D. J. Farber. Recoverability of communication protocols. volume 24, September 1976.

[Min99] M. Minea. Partial order reduction for model checking of timed automata. In *CONCUR'99*, volume 1664, pages 431–446, 1999.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. 1991.

[MP95a] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In P.E. Camurati and H. Eveking, editors, *Correct Hardware Design and Verification Methods*, volume 987, pages 189–205. Springer-Verlag, 1995.

[MP95b] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. 1995.

[Oli94] A. Olivero. *Modelisation et analyse de systemes temporises et hybrides*. PhD thesis, Institut National Polytechnique de Grenoble, France, september 1994.

[Pet99] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata:Theory and Practice*. PhD thesis, Uppsala University, department of Computer Systems, February 1999.

[Pnu77] A. Pnueli. The temporal logic of programs. pages 46–67, 1977.

[Pnu81] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[PPH97] D.A. Peled, V.R. Pratt, and G.J. Holzmann, editors. *POMIV '96: Proceedings of the DIMACS workshop on Partial order methods in verification*, New York, NY, USA, 1997. AMS Press, Inc.

[QS82] J-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar international symposium on programming. volume 137 of *LNCS*, pages 337–351, 1982.

[Ram74] C. Ramchandani. *Analysis of asynchronous concurrent systems by Petri nets*. PhD thesis, L.C.S. M.I.T. Cambridge, MA, February 1974. Ph.D. Thesis.

[RM84] T. Rokicki and C.J. Myers. Automatic verification of timed circuits. In *CAV'94*, pages 468–480, 1984.

[Rok94] T.G. Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, 1994.

[SY96] J. Sifakis and S. Yovine. Compositional specification of timed systems (extended abstract). In *STACS*, pages 347–359, 1996.

[TC96] S Tripakis and C. Courcoubetis. Extending promela and spin for real time. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 329–348, 1996.

[Tri98] S. Tripakis. *The analysis of timed systems in practice*. PhD thesis, Universite Joseph Fourier, Grenoble, France, December 1998.

[TY01] S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.

[Upp] Uppaal benchmarks. (www.it.uu.se/research/group/darts/uppaal/benchmarks).

[VW86] M.Y. Vardi and P. Wolper. An Automata-theoretic Approach to Automatic Program Verification. In *Proc. LICS'86*, pages 322–331. IEEE, 1986.

[Yov93] S. Yovine. *Methodes et outils pour la verification symbolique de systemes temporises*. PhD thesis, Institut National Polytechnique de Grenoble, France, May 1993.

[Yov97] S. Yovine. Kronos: A verification tool for real-time systems. volume 1, pages 123–133, October 1997.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the* 7th *Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.

[YS97] T. Yoneda and B.H. Schlingloff. Efficient verification of parallel real-time systems. volume 11, pages 187–215, 1997.

[Zha02] J. Zhao. Partial order path technique for checking parallel timed automata. In *FTRTFT'02*, pages 417–432, 2002.

[ZMM03] H. Zheng, E. Mercer, and C.J. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Trans. on CAD*, 22, 2003.

[ZYN03] S. Zennou, M. Yguel, and P. Niebert. Else: A new symbolic state generator for timed automata. In *FORMATS'03*, pages 273–280, 2003.

Appendix A: The Default Configuration File

```
# TCA 1.0
# This file describes the settings to be used by the TCA tool to work on a given circuit
#
# All text after a hash (#) is considered a comment and will be ignored
# Values that contain spaces should be placed between quotes (" ")
#
#------------------------------------------------------------------------
# ITERATOR configuration
#------------------------------------------------------------------------
# Path of the library of the behavioral models (the files *.ifp)
# (By default it is the current repository)
#
TC_MODELS_LIBRARY_PATH = .
#
# Path of the library of the structural models (the files *.tc)
# (By default it is the current repository)
#
TC_STRUCTURES_LIBRARY_PATH = .
#
# For closed systems the behavior of the environment should be given as an IF process (*.ifp)
# such models should exist in a given library (By default it is the current repository)
#
TC_INPUTS_MODELS_PATH = .
#
#------------------------------------------------------------------------
# COMPILER configuration
#------------------------------------------------------------------------
```

```
# The generation of the model could start from a stable state of the system.
# But this is not obligatory
#
CIRCUIT_STABILIZATION = YES       # you can change it by:    NO
#
# When a component is a logic gate and when its description is given as a logic
# function with a delays, the compiler can generate two kinds of models:
# (a) DEFAULT: leads to the Oded&Amir 4-states-model
# (b) HAZARD: leads to a similar model without regret and with an additional error state
#
COMPONENT_MODELS = DEFAULT       # you can change it by:    HAZARD
#
#---------------------------------------------------------------------------
# ENGINE configuration
#---------------------------------------------------------------------------
# GENERATOR configuration
#--------------------------
# This tool is built on the IF tool libraries, so we still have its functionality.
# When the following variable is set to "NO", the module GENERATOR will behave exactly
# like the IF GENERATOR. When this option is set to "YES" the GENERATOR will consider a
# synchronization between processes based on common variables.
#
TCA_SYNCH = YES      # you can change it by:    NO
#
# Adding dynamic global clocks:  If this is the case, the tool will be based on the variable
# synchronization, and the variable TCA_SYNCH will be set to YES.
#
WITHWAVES = YES      # you can change it by:    NO
#
#--------------------------
# EXPLORATOR configuration
#--------------------------
# The Explorator of this tool still have all the functionality of the IF tool.
# The TCA is implementing also two new exploration algorithms:  to :
# CU : reduce the state space explosion caused by the interleaving semantics.
# CU_IT: As the CU option but with Inclusion Test.
#
GRAPH-GENERATION = CU      # you can change it by:    CU_IT | DFS | BFS | DEBUG
```

```
#
#-------------------------
# MINIMIZATION configuration
#-------------------------
# Projection of the clocks:
#
PROJECT_CLOCKS = YES      # you can change it by:    NO
#
# Hiding of the non observable action, and perform the (a.tau*) reduction:
#
HIDE_NON_OBSERVABLE = YES      # you can change it by:    NO
#
# Bisimulation partitioning algorithm could be stopped at different steps:
# After the discrete splitting:  The default one, or
# continuing the splitting considering the overlapping between transitions
#
SEMANTICS_BISIMULATION = DEFAULT      # you can change it to:    OVERLAPPING
#
# The last step of the process is to compute the convex hull of invariant and guards
#
# CONVEX_HULL_TIME_MERGING = YES      # you can change it to:    NO
#
#-------------------------
# Printing configuration
#-------------------------
# The ENGINE role is to compute several transforming on an input automaton
# At each step the tool offer the possibility to print the intermediate automaton and
# that is in many possible formats.  This set of files could be generated in a single run.
#
# Print the interpreted automaton before continuing the abstraction process
#
PRINT_INTERPRETED_IFP   = NO      # you can change it to:  YES
PRINT_INTERPRETED_AUT   = NO      # you can change it to:  YES
PRINT_INTERPRETED_STATE = NO      # you can change it to:  YES
#
# Print the automaton after time projection
#
PRINT_CGLBS_IFP   = NO      # you can change it to:  YES
```

```
PRINT_CGLBS_AUT   = NO      # you can change it to:  YES
PRINT_CGLBS_STATE = NO      # you can change it to:  YES
#
# Print the automaton after variable hiding
#
PRINT_IO_IFP   = NO      # you can change it to:  YES
PRINT_IO_AUT   = NO      # you can change it to:  YES
PRINT_IO_STATE = NO      # you can change it to:  YES
#
# Print the automaton after (a.tau*) reduction
#
PRINT_TAUHIDE_IFP   = NO      # you can change it to:  YES
PRINT_TAUHIDE_AUT   = NO      # you can change it to:  YES
PRINT_TAUHIDE_STATE = NO      # you can change it to:  YES
PRINT_TAUHIDE_COMPLET = NO      # you can change it to:  YES (for a complete description)
#
# Print the automaton after bisimulation partitioning
#
PRINT_BMIN_IFP     = NO      # you can change it to:  YES
PRINT_BMIN_AUT     = NO      # you can change it to:  YES
PRINT_BMIN_STATE   = NO      # you can change it to:  YES
PRINT_BMIN_COMPLET = NO      # you can change it to:  YES (for a complete description)
#
# Print the automaton after splitting partitions based on overlapping transitions
#
PRINT_OVERLAP_IFP     = NO      # you can change it to:  YES
PRINT_OVERLAP_AUT     = NO      # you can change it to:  YES
PRINT_OVERLAP_STATE   = NO      # you can change it to:  YES
PRINT_OVERLAP_COMPLET = NO      # you can change it to:  YES (for a complete description)
#
# Complete process of reduction
#
PRINT_IFP     = NO      # you can change it to:  YES
PRINT_AUT     = NO      # you can change it to:  YES
PRINT_STATE   = NO      # you can change it to:  YES
```

Appendix 2: IF Process of an XOR Gate

```
process __z0(1);
     var value boolean := false public;
     var clk0 clock;
/*----- RISING ------*/
state STABLE_0 ;
     deadline eager;
     provided ( (({__x0}0).value and (not ({__x1}0).value ))
               or (({__x1}0).value and (not ({__x0}0).value )));
          informal "{__z0:0+}";
          set clk0 := 0;
          nextstate EXITEE_0;
endstate;
state EXITEE_0 ;
     deadline eager;
     provided not ( (({__x0}0).value and (not ({__x1}0).value ))
                    or (({__x1}0).value and (not ({__x0}0).value )));
          informal "{__z0:0}";
          reset clk0;
          nextstate STABLE_0;
     deadline delayable;
     provided ( (({__x0}0).value and (not ({__x1}0).value ))
               or (({__x1}0).value and (not ({__x0}0).value )));
     when clk0 >= 2 and clk0 <= 5;
          informal "{__z0+}";
          task value := not value;
          reset clk0;
          nextstate STABLE_1;
endstate;
/*----- FALLING ------*/
```

```
state STABLE_1 ;
     deadline eager;
     provided not ( (({__x0}0).value and (not ({__x1}0).value ))
                   or (({__x1}0).value and (not ({__x0}0).value )));
          informal "{__z0:1-}";
          set clk0 := 0;
          nextstate EXITEE_1;
endstate;
state EXITEE_1 ;
     deadline eager;
     provided ( (({__x0}0).value and (not ({__x1}0).value ))
              or (({__x1}0).value and (not ({__x0}0).value )));
          informal "{__z0:1}";
          reset clk0;
          nextstate STABLE_1;
     deadline delayable;
     provided not ( (({__x0}0).value and (not ({__x1}0).value ))
                   or (({__x1}0).value and (not ({__x0}0).value )));
     when clk0 >= 2 and clk0 <= 5;
          informal "{__z0-}";
          task value := not value;
          reset clk0;
          nextstate STABLE_0;
endstate;

endprocess;
```