L'analyse formelle des systèmes temporisés en pratique

Stavros Tripakis

UNIVERSITÉ JOSEPH FOURIER – GRENOBLE 1
SCIENCES ET GÉOGRAPHIE

THÈSE

pour le grade de
DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

en INFORMATIQUE

présentée et soutenue publiquement par

Stavros TRIPAKIS

16 décembre, 1998

L'Analyse Formelle des Systèmes Temporisés en Pratique

Membres du jury :

Gerard HOLZMANN,	Rapporteur
Amir PNUELI,	Rapporteur
Patrick COUSOT,	Examinateur
Joseph SIFAKIS,	Directeur de thèse
Jacques VOIRON,	Président

He looked towards the horizon that he had come out to see, of which he had seen so little. Now it was quite dark. Yes, now the western sky was as the eastern, which was as the southern, which was as the northern.

Samuel Beckett, *Watt*

Remerciements

Je remercie Joseph Sifakis, mon directeur de thèse, pour m'avoir accueilli à Verimag et pour m'avoir donné la possibilité de faire la recherche avec des personnes extraordinaires. Je le remercie également pour les nombreuses discussions passionantes qu'on a eues. Enfin, un grand merci pour son soutien à tous les niveaux.

Je remercie Sergio Yovine pour son encadrement clairvoyant et efficace. Son esprit pratique et sa persistance infatiguable à poser les bonnes questions m'ont été utiles maintes fois. Un grand merci aussi pour son soutien amical.

Je remercie Ahmed Bouajjani pour avoir la patience de répondre à mes questions, mais aussi pour les sorties très cools en montagne.

La cause de tout cca est probablement mon ex-professeur en Crète, Costas Courcoubetis, qui a organisé CAV'93. C'est là où mon amour pour les automates a commencé. Merci à Costas Courcoubetis.

Merci à Marius, c'était un plaisir de travailler avec lui.

Merci à Dragan pour les discussions très intéressantes, réelles ou virtuelles.

Merci à Maurice pour son aide.

Je remercie Thao et Yannick, pour être, à part de très bons amis, des collègues de bureau irremplaccables. J'ai pensé beaucoup de fois que j'avais de la chance de partager avec eux une grande partie de ma vie. Merci pour leur amitié et tolérance.

Merci à Conrado et Hassen, les rebels without a cause.

Merci à Roberto, Eduardo et Victor, friends at first sight.

Merci à Peter pour ses vélos. Up the flowers, Peter!

Merci à Suzanne pour sa pompe et son vélo, désolé d'avoir cassé ce dernier.

Merci à Dimitris, Georgia, Tasos, Costas, et les autres grecs.

Kiitos paljon, joulutyttö.

À tous les autres membres de Verimag, et les invités, et à tous ceux que j'ai forcément oublié, un grand merci.

Αυτη η δουλεια αφιερωνεται στους γονεις μου.

Stavros Tripakis
December 15, 1998

P.S. The quote from Beckett's "Watt" (next page) has no semantics, it is only chosen for its beauty of syntax.

Contents

Chapter 1

Introduction

1.1 Formal analysis of timed systems

A *timed system* is a system the behavior of which depends on timing constraints. This definition being too general, we restrict our attention mainly to timed systems for which timing constraints are *critical*, that is, the correctness of the system depends on them. Examples of such systems include traffic controllers, chemical-reaction controllers, real-time operating systems and so on. Failure of a timed system can have catastrophic consequences, therefore, it is crucial to ensure its correctness.

Formal methods are gaining popularity as a way to establish system correctness mathematically, that is, by proving that a formal model of the system satisfies a property. As identified in [MP95b] a formal analysis framework should contain the following elements:

- A *semantic model* to capture the behavior of systems.
- A *system-specification language* to describe systems.
- A *property-specification language* to express properties that a system should satisfy.
- *Techniques* for analyzing systems with respect to properties.

The goal of the field of formal methods is to integrate such a framework in the design and engineering process, so that the risk of building systems with faults is reduced.

To achieve its goal, it is key that a formal framework is also *practical*. In particular, it is necessary that

- the models are good in practice, meaning expressive, intuitive and easy to use;
- the algorithms are efficient, if not always, for most practical applications;
- feedback is provided to the user;
- some methodology exists for the modeling and analysis process.

Since the notion of practicality is not strictly mathematical, perhaps the only way for evaluating a formal framework is by testing it on as many case studies as possible.

In this thesis we consider a formal framework for timed systems consisting in the following elements.

Semantic model: Dense Time

We consider systems which evolve in *dense time*, that is, the time domain is the positive reals. Apart from *qualitative* properties talking about the relative order of the events (e.g., a happens before b) this model can also directly express *quantitative* time properties, talking about the delays between the occurrence of events (e.g., a happens 5 times unit before b). Delays can be exact, bounded or unbounded. The model can also capture delays which are arbitrarily close to a given value, thus, it is independent of a specific time unit or time granularity. Therefore, it is suitable for timed systems of *asynchronous* nature, for instance, when modeling a controller interacting with an environment which issues requests in arbitrary moments in time.

The state space induced by dense time is infinite (in fact, uncountable). Nevertheless, there exist methods to reduce this infinite *concrete* space to a finite *abstract* space, while preserving most of the properties we are interested in.

System-description language: Timed Automata

We describe timed systems using *timed automata* (TA) [Dil89, ACD93, HNSY94], that is, non-deterministic finite-state automata extended with a finite number of real-valued *clocks*. A TA alternates between two modes of execution, letting time pass continuously, then taking a step changing its discrete state.

To model a system consisting in more than one components we use a collection of TA executing *in parallel*. Time is assumed to pass *synchronously* for all components, that is, the clocks advance all at the same rate. The discrete steps of the automata happen *asynchronously*, unless some automata need to *communicate*, in which case they synchronize.

Property specification: Linear and Branching timed formalisms

We use a number of formalisms to express system properties. The *linear-time* formalism of *Timed Büchi Automata* (TBA) can specify properties as execution *sequences*. The *branching-time* logic *TCTL* can specify properties as execution *trees*. The logic $\text{ETCTL}^*_\exists$ is a combination of TBA and TCTL and strictly more expressive than both. TBA and TCTL are incomparable in expressiveness.

We also define some properties such as *reachability* ("does p ever hold?") and *deadlock-* or *timelock-freedom* directly, since they cannot be expressed in the above formalisms.

Using a variety of specification languages has a number of advantages. First, a larger class of properties can be expressed. Second, the user if offered a better choice in deciding his or her language of preference: a property theoretically expressible in two languages might be easier to express in one of them, that is, in a shorter or more intuitive manner. Third, the cost and type of output of the analysis techniques often depends on the specification language.

Techniques for Verification and Controller synthesis

We consider two types of analysis, namely:

- *Verification* (or *model checking*): given a timed system and a property, check whether the system satisfies the property.
- *Controller synthesis*: given a timed system embedded in a certain *environment*, and a property restrict the system so that the property is satisfied, no matter how the environment behaves (more about this below).

Decidability and complexity of the above problems has already been studied in previous works. Verification for TA with respect to TCTL or TBA is shown to be PSPACE-hard in [Alu91, ACD93]. Controller synthesis for TA with respect to *safety* properties is shown to be EXPTIME-hard in [HK97, MPS95]. The above results are based on the *region equivalence*, introduced in [Alu91, ACD93], which reduces the dense state space of a TA to a finite graph, the *region graph*. This construction has been very useful in the field of dense-time systems, since it shows that TA can be essentially viewed as finite-state automata. However, the size of the region graph is far too large for any practical purposes. To our knowledge, there is currently no formal-analysis tool based on the region graph.

1.2 The approach and contributions of this thesis

In order to obtain a practical framework, we develop techniques which, despite of the theoretical difficulty of the problems, work efficiently in practice. The main problem to solve is *state explosion*, that is, the fact that the size of the state space that needs to be generated/explored is often prohibitively large. To cope with this problem, we propose *abstractions* which yield in most practical cases a finite graph of reasonable size (many orders of magnitude smaller than the region graph). All abstractions can be computed *automatically* for a given input model, that is, no help from the user is necessary to define the abstraction [1].

In the case of linear-property verification, we show how generating the abstract state space and checking the property can be done at the same time (*on-the-fly*), and how useful feedback can be provided in the form of *concise diagnostics*. In the case of branching properties and controller synthesis the abstract graph has to be generated a-priori, but the analysis can still be done on-the-fly.

In order to demonstrate the practicality of our framework, we develop tools and treat a number of non-trivial case studies.

Abstractions for timed systems

We define a number of different abstractions for timed systems and study the properties they preserve:

- *Time-abstracting bisimulations* are abstractions where the quantitative aspect of time is hidden away: we know that *some* time passes, but not how much. Of the three time-abstracting bisimulations defined, the strong one preserves both linear and branching properties and can also be used for controller synthesis.

- *Time-abstracting simulations* are abstractions where the passage of time is hidden altogether, and only discrete-state changes can be observed. These abstractions are based on the *simulation graph*, which is built by forward reachability and preserves all linear properties, finite or infinite. Weaker simulation abstractions preserve finite linear properties (i.e., reachability) in an exact or conservative manner.

[1] This is in contrast with another type of formal verification techniques, called *deductive*, which consist in using a set of axioms to prove that the system satisfies the property. Semi-algorithmic procedures exist to help the user perform this task with the aid of a computer, however, since they are incomplete, the interaction with the user is often necessary. For more information about deductive techniques the reader is referred, for instance, to [MP95b, Sai97, BMSU97].

The choice of which abstraction to use depends on:

- the property to be checked, which has to be preserved by the abstraction;
- the reduction factor of the abstraction;
- the cost of computing the abstraction.

Theoretically, all abstractions may produce a state space as large as the region graph. However, as shown by our experiments, the reduction is in practice many orders of magnitude better. Regarding the reduction factor of time-abstracting bisimulations compared to simulations, there is no general rule on which is better: the results vary depending on each case. On the other hand, simulations are usually more interesting, since they can be used for on-the-fly verification.

On-the-fly techniques

In *on-the-fly* verification, the property is checked while the state space is generated. Therefore, an answer can be returned as soon as possible, without necessarily generating the entire state space. On-the-fly methods are particularly useful during the first stages of modeling, while the model is still under development and most of the times contains "bugs". Discovering such bugs rapidly permits to correct the model and continue with the analysis without much cost.

In this thesis we develop on-the-fly algorithms for verification based on simulation abstractions. The algorithms perform reachability, deadlock and timelock detection, TBA model checking and full $\mathrm{ETCTL}^{*}_{\exists}$ model checking. We also develop an on-the-fly controller-synthesis algorithm based on strong time-abstracting bisimulation (see below).

Minimization of TA and analysis based on quotient graphs

We use time-abstracting bisimulations to reduce verification and controller synthesis of timed systems to the untimed case. In that way, we can apply a variety of efficient classical (untimed) techniques and also exploit the existing tool infrastructure. Our method works in two steps. First, we compute the system's *quotient* with respect to the strong time-abstracting bisimulation. The quotient is essentially an untimed graph, which preserves the properties of the concrete state space (for this, the property has to be taken sometimes into account while building the quotient). Therefore, existing algorithms for untimed linear- and branching-time model-checking can be applied on the quotient to solve the corresponding timed model-checking problems.

The first step of the above approach requires generating the quotient, a process called *minimization*. To solve the problem for the timed case, we adapt the generic minimization algorithm of [BFH+92] to the TA model. The algorithm of [BFH+92] uses *complementation* on sets of states. Since this operation is expensive for dense-space representations, we develop a minimization technique which avoids complementation.

Controller synthesis

The systems we consider are often *reactive*, that is, they function in a certain environment. To analyze reactive systems, we need to model both the behaviors of the system and the environment. In the case where the environment is unpredictable (or even adversary) and the specification of the system is still incomplete (for instance, during the first stages of design), the

analysis can be see as a *game* between the system and its environment: the system needs to make the right choices so that a given property is satisfied independently of what the environment does. Synthesizing a *controller* comes down to completing the specification of the system, that is, restricting the set of possible choices to the right ones. The controller-synthesis problem is more general than verification. The techniques for controller synthesis are more sophisticated than verification techniques, thus, usually more expensive.

In this thesis we study controller-synthesis for TA, based on the results of [MPS95]. On the theoretical level, we clarify the notion of *strategy* and provide definitions for the parallel composition of TA in the presence of controllability.

On the practical level, we show how to implement the operators used in the fix-point controller-synthesis algorithm of [MPS95]. We also introduce an on-the-fly method in two steps: First, we develop an algorithm for controller synthesis on untimed systems (i.e. finite graphs). This algorithm is on-the-fly in the sense that it can return a strategy as soon as one it is found. Then, we show how to apply the algorithm on the time-abstracting quotient graph of a TA, to solve the problem in the timed case.

Timed diagnostics

TA verification is performed by exploring abstract state spaces, thus, the diagnostics that can be generated directly from such an exploration are also abstract. In particular, such diagnostics usually lack timing information, which can be crucial to understanding why a property does or does not hold. To give more precise feedback we are interested in *timed diagnostics*: these correspond exactly to the semantics of TA, that is, they contain both the discrete state changes as well as the exact time delay between two discrete transitions. We show how to compute timed diagnostics for linear properties, both finite and infinite.

Implementation and Tools

KRONOS is a tool-suite for the analysis of timed systems [Yov93, Oli94, Daw98, DOTY96, BDM+98, Yov97]. KRONOS uses a *symbolic* representation of states, in particular, boolean combinations of simple linear constraints representing sets of possible clock values. Semantic operations on symbolic states are implemented as syntactic transformations of these sets of constraints.

We have implemented the algorithms developed in this thesis on top of KRONOS, using its symbolic-representation library. The functionalities added in the tool include on-the-fly parallel composition of TA, on-the-fly verification with respect to reachability and TBA, timed diagnostics, and controller synthesis for properties of *invariance* ("always p holds") or reachability. We have also implemented a new module for TA minimization, called `minim`. The infinite-diagnostic generation algorithm (section 8.2) and the on-the-fly controller-synthesis algorithm for untimed graphs (section 9.3) are being currently implemented.

Following the philosophy of tools like SPIN [Hol91], we have also connected KRONOS to the untimed-verification software platform OPEN-CAESAR [Gar98] which is part of the CADP tool-suite [FGM+92]. In particular, we developed the module `kronos-open` which acts as a *compiler*, that is, generates C-code for a particular input model. The C-code is then interfaced to OPEN-CAESAR's libraries, to build the final executable which will perform the analysis. In the context of this work, we have developed a symbolic-representation library of *variable-dimension*, where clocks can be dynamically created and deleted. Using this library, we have been able to treat systems with more than 30 clocks.

Case studies

Using KRONOS, we have treated a number of case studies, six of which are presented in this document. Most of them are real-world case studies, namely, two industrial communication protocols (BANG&OLUFSEN's protocol, CNET's protocol), the electronic circuit STARI, and a multimedia-document authoring language developed as part of INRIA's project OPERA. The real-time scheduling case study is interesting on its own, but also for illustrating some of the techniques for model-checking timed *liveness* properties ("p holds infinitely often"). Fischer's protocol serves as a good example for illustrating many of the techniques presented throughout the thesis. Also, along with the STARI circuit, they represent good benchmarks to test the capacity of the tools and perform measurements demonstrating the usefulness of the techniques.

1.3 Related research

A brief history of dense-time verification

The introduction of the model of TA in the early '90s has created a new domain of research which is still expanding.

The first attempt to overcome the inherent state-explosion of the region graph has given in [HNSY94]. The method, following the framework of [Sif82], consists in computing the set of states satisfying a TCTL formula ϕ (called the *characteristic set of* ϕ). The computation is done recursively on the syntax of ϕ, based on a fix-point characterization of the modal operators of TCTL. This approach is practical, since many regions can be encoded as a single set of constraints in a compact way. However, the method still has some important drawbacks:

- It is not on-the-fly, since the fix-point computation must terminate before an answer can be returned.
- It does not provide diagnostics, except in a very primitive form, namely, the characteristic set of a formula. Usually what is needed as diagnostics is sample executions.
- It considers the whole potential state space, whereas what is interesting is only the reachable part of the state space.
- It uses complementation, which results in expensive symbolic representation of sets of states.

As a remedy to the above, on-the-fly methods have been proposed independently in [DOY94] and [TC96]. These methods are based on forward reachability analysis (i.e., the simulation graph). They build the reachable state space and can provide symbolic diagnostics (finite paths in the simulation graph). Their main drawback is that they are limited to timed *safety* properties such as invariance and *bounded response* ("whenever p_1 holds, p_2 will hold after c time units at the latest").

Parallel to the theoretical work, a number of dense-time verification tools have been developed in the past few years. KRONOS has been the first one [Yov93, Oli94, DOY94], followed by real-time COSPAN [CDCT92], UPPAAL [BLL+95], RTSPIN [TC96] and Timed-COSPAN [AK83, AK96]. These tools are more or less based on the same TA model, however, they differ in their

property-specification languages. All the tools perform verification symbolically by representing sets of states by simple linear constraints, although different techniques are sometimes used to encode the constraints.

The existence of tools has had an important impact on the acceptance of the dense-time approach not only by the research community, but also by industrial partners. Real-world case studies such as those treated with UPPAAL [BGK+96, HSLL97] and KRONOS [TY98] have shown that the model, the techniques and the tools have reached a degree of maturity which makes them useful in practice.

State of the art. The field of dense-time verification is quite active and is being continually enriched by new results. Recent works study how *partial-order* reduction techniques [Val90, GW91, Pel94] can be applied to the verification of timed systems [BJLY98, BM98]. [AJ98] exploit the symmetry of systems of a particular type, to verify them for a parameterized number of processes. Research is also being conducted for new representation techniques for dense state spaces [DY96, LLPY97, STA98, LWYP98]. Controller synthesis for timed systems is also a new and quite promising field, especially because of the number of interesting applications involved, such as real-time scheduling.

Other approaches

Many other formal approaches have been used for the analysis of timed systems, differing in both the models and the analysis techniques. *Synchronous languages* such as ESTEREL [BB91b] and LUSTRE [CPHP87] are particularly suited for systems such as synchronous circuits, which are deterministic and work according to a global clock. An extensive bibliography exists on *time Petri nets* (for instance, [Ram74, Mer74, Sif77, BD91]) and timed extensions of *process algebras* (for instance, [NRSV90, BB91a, LL93]). *Hybrid automata* [PV94, ACH+95] are a generalization of the TA model with real variables having more general evolution laws, defined by differential equations. Although most of the interesting problems are undecidable for the general model, semi-algorithms and approximative analyses can still be implemented. HYTECH [HHW97] is a tool performing analysis for hybrid automata where evolution laws are specified by simple linear differential equations. The graphical formalism of *Statecharts* has been extended with timed and hybrid semantics in [KP92]. Timed I/O automata [LA90] provide a model and a methodology for mathematical reasoning about timed systems.

A comparative study of the above approaches is out of the scope of this thesis. On the other hand, it is worth discussing the following two issues, since they are directly related to our approach and results.

Dense versus Discrete Time. There has been a long and still unsettled debate concerning the choice of the right quantitative model for time. The criteria for choosing a time model are essentially two: first, how expressive the model is, that is, how well can it describe reality; second, how efficient it is, that is, what is the cost of the corresponding analysis techniques.

Dense time is strictly more expressive than discrete time. Perhaps the most important feature of dense time is the fact that it abstracts from a specific time quantum, since it can model arbitrary small delays. This property is of particular practical interest in two cases. First, when implementing a model of a timed system, the correctness of the implementation does not depend on the speed (clock frequency) of the machine. Second, composition of timed systems is independent of the time quantum: we only need to normalize the constants appearing

in the constraints of the components, but not their time granularities, as it would be necessary to do in a discrete-time model.

Regarding theoretical results on expressive power, there have been a number of different discrete-time models proposed for timed systems, as well as a number of works comparing them with dense time. Two discrete-time models are considered in [Alu91]. In the first, events are bound to occur along with the clock "ticks", whereas in the second, events can occur anywhere in the real line, but the only quantitative information is how many ticks have passed between two events. [Alu91] compares these models to the dense-time one informally, using the paradigm of asynchronous circuits, and argues in favor of dense time. More formal results can be found in [HMP92, GPV94, AMP98]. A general conclusion of these results is that dense time is strictly more expressive than discrete time. Another conclusion is that in special cases of TA (e.g., when modeling acyclic electronic circuits) discretization preserves a restricted class of properties (e.g., the order of events). Still, what is missing is a methodology to discretize a given dense-time automaton, in particular, how to find the necessary quantum of time.

Regarding efficiency, the discrete-time model is usually thought to be better suited, since it admits powerful untimed verification techniques such as efficient symbolic representation [ABK+97, BMPY97] using *binary decision diagrams* (BDDs) [Bry86, CBM89, BCD+90], or partial orders [BD98]. This conception is true but only to some extent. First, the discrete-state techniques are not always given for free: discretizing the dense-time model or directly modeling in discrete time can result in a less compact specification. In particular, models which involve large constants can result in state explosion when treated in discrete time (see, for example, the case study in section 12.3). Second, the synchronous nature of the passage of time sometimes affects the performance of BDDs as well as partial orders, which work well in systems where actions are as much independent as possible.

Enumerative versus Symbolic techniques. In the untimed context (and especially after the introduction of BDDs) there is a clear distinction between enumerative and symbolic techniques:

- In enumerative techniques, *each* state is represented explicitly, usually as an encoded vector on the system's variables. The state space is generated and explored state by state.

- In symbolic techniques, a *set* of states is represented implicitly, usually by a formula on system's variables: the members of the set are those states satisfying the formula. The reachable state space is usually computed as a fix-point of the transition relation (itself a formula) applied to the formula encoding the initial states.

Both approaches have their pros and cons. Enumerative ones are well adapted for on-the-fly verification, and can easily provide fast answers and diagnostics. Symbolic ones are much more compact, therefore resistant to state explosion.

In dense-time systems, most of the techniques are a mixture of enumerative and symbolic flavor. This is mainly due to the following reasons. First, purely enumerative techniques are impossible, since the clock state space is dense (the most enumerative technique, based on the region graph, would require a symbolic way to encode regions). Second, purely symbolic techniques are not available yet: such techniques would combine in a homogeneous representation continuous as well as discrete variables, since the latter are indispensable also in a timed system. The approach of [Yov93, HNSY94] can be considered as the most symbolic of the existing

approaches, since it uses the most general type of constraints to represent any union of regions, closed with respect to all set-theoretic operators. The drawback of this representation is that it is not *canonical*. Therefore, although in principle it could be used to encode also discrete variables (in a dense space), this is not efficient.

Our approach, like the ones of [Oli94, BLL+95, TC96], is more enumerative, although not as much as the region graph.

1.4 Organization of this document

This document is structured in four parts.

The first part presents the background. Chapter 2 introduces graphs and polyhedra, used through-out the document. In chapter 3 we present our model of TA and in chapter 4 our property-specification languages.

The second part presents the analysis techniques and constitutes the core of the theoretical results of this thesis. In chapter 5 we define time abstractions and study the properies they preserve. The results of this chapter are transformed into techniques for verification in chapters 6 and 7. Chapter 6 shows how to compute the time-abstracting bisimulations of section 5.1 and how to use them for model checking. Chapter 7 presents techniques which are fully on-the-fly to compute the time-abstracting simulations of section 5.2 and perform model checking at the same time. Diagnostics and controller synthesis are presented separately in chapters 8 and 9, respectively.

The third and fourth parts present the main practical contributions of this thesis. Chapter 10 shows how the semantic entities used in the first two parts can be represented effectively. Chapter 11 gives an overview of the tool suite KRONOS and our contributions to its development. The case studies and experimental results are presented in chapter 12.

Regarding the readability of the document, most of the chapters depend on the definitions given in the first part. Chapter 6 depends also on section 5.1 and chapter 7 on section 5.2. Apart from the above dependencies, most chapters are supposed to be self-contained. No special environment for definitions is used. We prefered not to use a special environment for definitions, but to include them in the text. Special terms, operators and symbols appear in *emphasized* font at the moment of their definition, and can be (hopefully) found in the index.

Part I

Timed Formalisms

Chapter 2

Preliminaries

General notations. Through-out this document we write $\mathsf{R}, \mathsf{Z}, \mathsf{N}$ for the sets of non-negative reals, integers and naturals, respectively. We use variables such as x, y, z, δ, t ranging over R, c ranging over Z and i, j, k, l, m ranging over N. *Labels* is a finite set of labels.

If X and Y are sets, the operations of intersection, union, complementation, set difference and cartesian product are denoted $X \cap Y, X \cup Y, \overline{X}, X \setminus Y$ and $X \times Y$, respectively. The empty set is denoted $\emptyset$. Inclusion and strict inclusion are denoted $X \subseteq Y$ and $X \subset Y$, respectively. Membership of x to X is denoted $x \in X$. For a given order on X, $\min X$ and $\max X$ denote the smaller and greater element of X with respect to this order. 2^X is the *powerset* of X, that is, the set of all subsets of X. If X is finite, $|X|$ denotes its cardinal. A relation between sets X and Y is a subset of $X \times Y$. If $\sqsubseteq$ is a relation between X and Y then $\sqsubseteq^{-1}$ denotes the *inverse* relation between Y and X, such that $y \sqsubseteq^{-1} x$ iff $x \sqsubseteq y$. If $\sqsubseteq$ is a relation between X and Y, and $\sqsubseteq'$ is a relation between Y and Z, then $\sqsubseteq \circ \sqsubseteq'$ is the *composition* of $\sqsubseteq$ and $\sqsubseteq'$, defined as the relation $\sqsubseteq''$ on X and Z such that $x \sqsubseteq'' z$ iff there exists $y \in Y$ s.t. $x \sqsubseteq y$ and $y \sqsubseteq' z$. For a function $f : 2^X \mapsto 2^X$, a *fix-point* of f is a set $Y \subseteq X$ such that $f(Y) = Y$. The *greatest* and *least* (with respect to set inclusion) fix-points of f are denoted $\mu Y \,.\, f(Y)$ and $\nu Y \,.\, f(Y)$, respectively.

Logical and, or and not are written $\wedge, \vee$ and $\neg$, respectively. Implication and equivalence are denoted $\Rightarrow$ and $\equiv$ and are defined as usual.

a, b usually denote labels. a^* stands for "a finite repetition of a" (possible no a at all). a^ω stands for "an infinite repetition of a". We use the symbol ∞ for infinity.

2.1 Graphs

A (directed) *graph* G is a pair $(V, \rightarrow)$, where V is a set of *nodes* and $\rightarrow \subseteq V \times V$ is a set of edges. Sometimes the edges are labeled, that is, $\rightarrow \subseteq V \times \mathit{Labels} \times V$, giving a *labeled transition system* (LTS). In this document we use the term graph for both graphs and LTSs.

Given an edge $v \rightarrow u$, the nodes v and u are called the *predecessor* and the *successor* of v, respectively. $\mathsf{preds}(v)$ (resp. $\mathsf{succs}(v)$) denotes the set of predecessors (resp. successors) of v. A *sink* node is a node with no successors.

Paths, cycles, strongly-connected components. A (finite or infinite) *path* is a (finite or infinite) sequence $v_1 \rightarrow v_2 \rightarrow \cdots$. We say that the path *visits* nodes $v_1, v_2, \ldots$. A path of *length* l from node v to node u is a finite path $v = v_1 \rightarrow \cdots \rightarrow v_l = u, l \geq 1$. A *cycle* with *root* v is

a path from v to itself. A cycle is called *elementary* if it visits no node twice, except from the root. A cycle $v_1 \to \cdots \to v_l \to v_1$ can also be viewed as the infinite path $(v_1 \to \cdots \to v_l \to)^\omega$.

A subgraph G' of G is a *strongly-connected component* (SCC) of G if for any node v of G', there is a cycle rooted at v and visiting all nodes of G' and nothing but nodes of G'. A SCC is *maximal* if it is not properly contained in any larger SCC.

Relations, preorders and simulations. Consider a graph $G = (V, \to)$. A *binary relation* on G is a subset $\sim$ of $V \times V$. $\sim$ is *reflexive* if $v \sim v$ for all $v \in V$. $\sim$ is *symmetric* if $v \sim u$ implies $u \sim v$ for all $v, u \in V$. $\sim$ is *transitive* if $v \sim u$ and $u \sim w$ imply $v \sim w$, for all $v, u, w \in V$. Given two relations $\sim_1$ and $\sim_2$, $\sim_1$ is *stronger than* $\sim_2$ if $\sim_1 \subseteq \sim_2$.

A *preorder* on G is a binary relation $\sqsubset \subseteq V \times V$ on the set of nodes of G which is reflexive and transitive.

A binary relation $\sqsubset$ on V is called a *simulation* on G iff for any pair $v_1, v_2 \in V$, if $v_1 \sqsubset v_2$ then for each successor u_1 of v_1, there exists a successor u_2 of v_2 such that $u_1 \sqsubset u_2$. It is easy to see that a simulation is a preorder on G.

Equivalences, Partitions. Consider a graph $G = (V, \to)$. An *equivalence* $\approx$ on G is a preorder on G which is symmetric.

A *partition* of the set of nodes V is a set $\mathcal{C} \subseteq 2^V$ of subsets of V such that:

1. For all $C_1, C_2 \in \mathcal{C}$, $C_1 \cap C_2 = \emptyset$, that is, all members of $\mathcal{C}$ are disjoint.
2. For all $v \in V$, there exists $C \in \mathcal{C}$ such that $v \in C$, that is, $\mathcal{C}$ covers V.

The members of a partition are called *classes*.

The following facts can be easily derived from the definitions:

- an equivalence $\approx$ induces a partition $\mathcal{C}_\approx$ of V, where for all $C \in \mathcal{C}_\approx$, $v, u \in C$ iff $v \approx u$;
- inversely, a partition $\mathcal{C}$ induces an equivalence $\approx_\mathcal{C}$, where $v \approx_\mathcal{C} u$ iff v and u belong to the same class.

Let *Props* be a set of atomic propositions and let $P : \mathit{Props} \mapsto 2^V$ be a function associating to each proposition a set of nodes of G. An equivalence $\approx$ *respects* P if for all pairs $v_1, v_2 \in V$ such that $v_1 \approx v_2$, for all $p \in \mathit{Props}$, $v_1 \in P(p)$ iff $v_2 \in P(p)$. A partition respects P if the equivalence induced by the partition respects P.

Given two partitions $\mathcal{C}_1$ and $\mathcal{C}_2$ of V, $\mathcal{C}_1$ is *coarser* than $\mathcal{C}_2$ ($\mathcal{C}_2$ is *finer* than $\mathcal{C}_1$) if the equivalence induced by $\mathcal{C}_2$ is stronger than the one induced by $\mathcal{C}_1$. If $\mathcal{C}_1$ is coarser than $\mathcal{C}_2$, then for each class C_1 of $\mathcal{C}_1$ there exists a class C_2 of $\mathcal{C}_2$ such that $C_2 \subseteq C_1$.

Quotient graphs. The *quotient* of a graph $G = (V, \to)$ with respect to a partition $\mathcal{C}$ is the graph $G' = (\mathcal{C}, \to')$ where $C_1 \to' C_2$ iff there exist $v \in C_1, u \in C_2$ such that $v \to u$. The quotient graph of G with respect to an equivalence $\approx$, denoted $G_\approx$, is the quotient of G with respect to the partition induced by $\approx$. We often call $G_\approx$ the $\approx$-*quotient* of G.

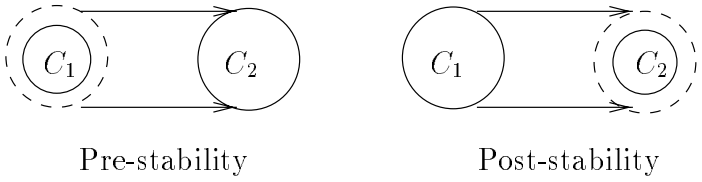


Figure 2.1: Pre- and post-stability.

Stable partitions, Bisimulations. Consider a partition $\mathcal{C}$. Given two classes $C_1, C_2 \in \mathcal{C}$, C_1 is said to be *pre-stable with respect to* C_2 if either $C_1 \subseteq \mathsf{preds}(C_2)$ or $C_1 \cap \mathsf{preds}(C_2) = \emptyset$. C_2 is said to be *post-stable* with respect to C_1 if either $\mathsf{succs}(C_1) \subseteq C_2$ or $\mathsf{succs}(C_1) \cap C_2 = \emptyset$. The two notions of stability are illustrated in figure 2.1. $\mathcal{C}$ is called *pre-stable* (resp. *post-stable*) if all its classes are pre-stable (resp. post-stable) to one-another.

A relation $\approx \subseteq V \times V$ is a *(strong) bisimulation* iff for all pairs $v_1, v_2 \in V$ such that $v_1 \approx v_2$, the following conditions hold:

1. for each successor u_1 of v_1, there exists a successor u_2 of v_2 such that $u_1 \approx u_2$,
2. the above condition also holds if the roles of v_1 and v_2 are reversed.

The definition is illustrated in figure 2.2 (left). We say that two nodes v and u are *bisimilar* if there exists a bisimulation $\approx$ such that $v \approx u$.

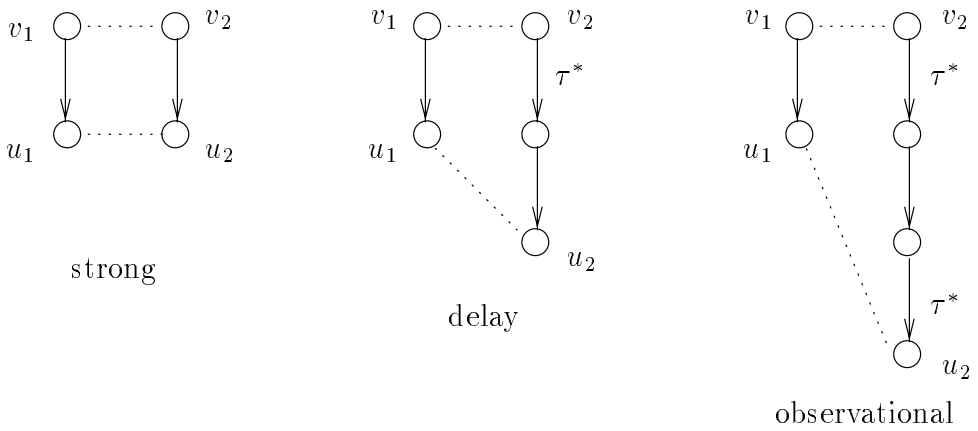


Figure 2.2: Strong, delay and observational bisimulation.

The following facts can be easily derived from the definitions:

- a bisimulation is an equivalence inducing a pre-stable partition;
- inversely, a pre-stable partition induces a bisimulation.

Weaker bisimulations. Sometimes it is useful, for refining or abstracting specifications, to consider weaker bisimulations, which do not take into account parts of the behavior which are *internal* to the system. For the purposes of this document, we consider two weaker bisimulations, namely, the *delay* bisimulation [NMV90] and the *observational* bisimulation [Mil80].

More precisely, consider a graph $G = (V, \rightarrow')$, labeled in $Labels \cup \{\tau\}$, where τ is an internal label. Let $\rightarrow$ be the restriction of $\rightarrow'$ to $V \times Labels \times V$ and $\xrightarrow{\tau}$ be the restriction of $\rightarrow'$ to $V \times \{\tau\} \times V$. Also let $\xrightarrow{\tau^*}$ be the reflexive, transitive closure of $\xrightarrow{\tau}$.

A binary relation $\approx$ on V is an *observational* (resp. *delay*) bisimulation iff for all pairs $v_1, v_2 \in V$ such that $v_1 \approx v_2$, the following conditions hold:

1. for all $u_1 \in V$ such that $v_1 \xrightarrow{\tau} u_1$, there exists $u_2 \in V$ such that $v_2 \xrightarrow{\tau^*} u_2$ and $u_1 \approx u_2$.
2. for all $u_1 \in V$ such that $v_1 \rightarrow u_1$, there exists $u_2 \in V$ such that $v_2 \xrightarrow{\tau^*}\rightarrow\xrightarrow{\tau^*} u_2$ (resp. $v_2 \xrightarrow{\tau^*}\rightarrow u_2$) and $u_1 \approx u_2$.
3. the above two conditions also hold if the roles of u_1 and u_2 are reversed.

The above definitions are illustrated in figure 2.2 (middle and right).

Comparing graphs with respect to bisimulations. Consider two graphs $G_i = (V_i, \rightarrow_i)$, for $i = 1, 2$. The *union* of G_1 and G_2 is defined to be the graph $G_1 \cup G_2 = (V_1 \cup V_2, \rightarrow_1 \cup \rightarrow_2)$. If we identify two nodes $v_1 \in V_1$ and $v_2 \in V_2$ as initial nodes, then G_1 and G_2 are said to be *bisimilar* if there exists a bisimulation on $G_1 \cup G_2$ so that v_1 and v_2 are bisimilar.

Let $\approx$ be a bisimulation on a graph G and let $\approx'$ be the relation between nodes of G and nodes of the quotient $G_{\approx}$, such that $v \approx' C$ iff $v \in C$. Then, it is easy to verify that $\approx'$ is a bisimulation on $G \cup G_{\approx}$. Thus, a graph and its quotient with respect to a bisimulation are themselves bisimilar.

2.2 Dense state spaces

2.2.1 Polyhedra

Clocks and valuations. Let $\mathcal{X} = \{x_1, ..., x_n\}$ be a set of variables in R. In the scope of this document, these variables are called *clocks*. An $\mathcal{X}$*-valuation* is a function $\mathbf{v} : \mathcal{X} \mapsto \mathsf{R}$ assigning to each clock x a non-negative real value $\mathbf{v}(x)$. The set of all valuations is $\mathsf{R}^{\mathcal{X}}$. We write $\mathbf{0}$ for the valuation that assigns zero to all clocks. For a subset X of $\mathcal{X}$, $\mathbf{v}[X := 0]$ is the valuation $\mathbf{v}_1$ such that $\forall x \in X \,.\, \mathbf{v}_1(x) = 0$ and $\forall x \notin X \,.\, \mathbf{v}_1(x) = \mathbf{v}(x)$. Intuitively, $\mathbf{v}[X := 0]$ is obtained from $\mathbf{v}$ by resetting all clocks in X to zero and leaving the rest of the clocks unchanged. For $\delta \in \mathsf{R}$, $\mathbf{v} + \delta$ is the valuation $\mathbf{v}_2$ such that $\forall x \in \mathcal{X} \,.\, \mathbf{v}_2(x) = \mathbf{v}(x) + \delta$. Intuitively, $\mathbf{v} + \delta$ is obtained from $\mathbf{v}$ by advancing all clocks by the same time delay δ. Similarly, $\delta \cdot \mathbf{v}$ is the valuation $\mathbf{v}_3$ such that $\forall x \in \mathcal{X} \,.\, \mathbf{v}_3(x) = \delta \cdot \mathbf{v}(x)$.

Hyperplanes and Polyhedra. An *atomic constraint* on $\mathcal{X}$ is an expression of the form $x \sim c$ or $x \Leftrightarrow y \sim c$, where $x, y \in \mathcal{X}$, $\sim \in \{<, \leq, \geq, >\}$ and $c \in \mathsf{N}$. An $\mathcal{X}$-valuation $\mathbf{v}$ *satisfies* the constraint $x \sim c$ if $\mathbf{v}(x) \sim c$; $\mathbf{v}$ satisfies $x \Leftrightarrow y \sim c$ if $\mathbf{v}(x) \Leftrightarrow \mathbf{v}(y) \sim c$.

An $\mathcal{X}$*-hyperplane* is a set of valuations satisfying an atomic clock constraint. The class $\mathcal{H}_{\mathcal{X}}$ of $\mathcal{X}$*-polyhedra* is defined as the smallest subset of $2^{\mathsf{R}^{\mathcal{X}}}$ which contains all $\mathcal{X}$-hyperplanes and is closed under set union, intersection and complementation.

We often use the following notation for polyhedra: we write $x < 5$ for the hyperplane defined by the constraint $x < 5$, $x < 5 \wedge y = 2$ for the polyhedron defined as the intersection

of $x < 5$ and $y = 2$, and so on. We also write true for $\mathsf{R}^{\mathcal{X}}$ (equivalently, $\bigwedge_{x\in\mathcal{X}} x \geq 0$), false for $\emptyset$ (equivalently, $\bigwedge_{x\in\mathcal{X}} x < 0$) and zero for $\{\mathbf{0}\}$ (equivalently, $\bigwedge_{x\in\mathcal{X}} x = 0$).

Given a polyhedron ζ and a clock x, the predicate $\mathsf{unbounded}(x,\zeta)$ is defined as follows:

$$\mathsf{unbounded}(x,\zeta) \stackrel{\text{def}}{=} \forall t \in \mathsf{R} \,.\, \exists \mathbf{v} \in \zeta \,.\, \mathbf{v}(x) > t$$

A polyhedron ζ is called *convex* if for all $\mathbf{v}_1, \mathbf{v}_2 \in \zeta$, for any $0 < \delta < 1$, $\delta\mathbf{v}_1 + (1 \Leftrightarrow \delta)\mathbf{v}_2 \in \zeta$. It is easy to show that a polyhedron is convex iff it can be defined as the intersection of a finite number of hyperplanes. On the other hand, if ζ is non-convex then it can be written as $\zeta_1 \cup \cdots \cup \zeta_k$, where $\zeta_1, ..., \zeta_k$ are all convex. We denote the set $\{\zeta_1, ..., \zeta_k\}$ by $\mathsf{convex}(\zeta)$.

2.2.2 Operations on polyhedra

By definition, intersection, union and complementation are well-defined operations on polyhedra. Polyhedra difference is defined via complementation as: $\zeta_1 \setminus \zeta_2 = \zeta_1 \cap \overline{\zeta_2}$. The test for inclusion $\zeta_1 \subseteq \zeta_2$ is equivalent to $\zeta_1 \setminus \zeta_2 = \emptyset$. We now define some more operations which will be used throughout this document. Examples are shown in figure 2.3.

Convex hull. Given two $\mathcal{X}$-polyhedra ζ_1 and ζ_2, we define the *convex hull* of ζ_1 and ζ_2, denoted $\zeta_1 \sqcup \zeta_2$, to be the smallest (w.r.t. set inclusion) convex $\mathcal{X}$-polyhedron containing both ζ_1 and ζ_2.

c-equivalence and c-closure. Given $c \in \mathsf{N}$, two valuations $\mathbf{v}$ and $\mathbf{v}'$ are called *c-equivalent* if:

- for any clock x, either $\mathbf{v}(x) = \mathbf{v}'(x)$, or $\mathbf{v}(x) > c$ and $\mathbf{v}'(x) > c$;
- for any pair of clocks x, y, either $\mathbf{v}(x) \Leftrightarrow \mathbf{v}(y) = \mathbf{v}'(x) \Leftrightarrow \mathbf{v}'(y)$, or $\mathbf{v}(x) \Leftrightarrow \mathbf{v}(y) > c$ and $\mathbf{v}'(x) \Leftrightarrow \mathbf{v}'(y) > c$.

Given a convex $\mathcal{X}$-polyhedron ζ, we define $\mathsf{close}(\zeta, c)$ to be the greatest convex $\mathcal{X}$-polyhedron $\zeta' \supseteq \zeta$, such that for all $\mathbf{v}' \in \zeta'$ there exists $\mathbf{v} \in \zeta$ and $\mathbf{v}, \mathbf{v}'$ are c-equivalent. Intuitively, ζ' is obtained by ζ by "ignoring" all constraints which involve constants greater than c (figure 2.3 displays an example).

ζ is said to be *c-closed* if $\mathsf{close}(\zeta, c) = \zeta$.

Lemma 2.1 *1. If ζ is c-closed then it is c'-closed, for any $c' > c$.*

2. If ζ_1 and ζ_2 are c-closed then $\zeta_1 \cap \zeta_2$ is also c-closed.

3. For any ζ, there exists a constant c such that ζ is c-closed.

Proof: Properties 1 and 2 are easily derived from the definitions. For 3, we first prove the result in the special case where ζ is convex, that is, $\zeta = \zeta_1 \cap \cdots \cap \zeta_m$, where $\zeta_1, ..., \zeta_m$ are hyperplanes. Let $c_1, ..., c_m$ be the constants appearing in the atomic constraints defining $\zeta_1, ..., \zeta_m$. It is easy to see that ζ_i is c_i-closed, for each $i = 1, ..., m$. Now, if $c = \max\{c_1, ..., c_m\}$, then $\zeta_1, ..., \zeta_m$ are all c-closed and so is ζ. If ζ is non-convex, then let $\mathsf{convex}(\zeta) = \{\zeta'_1, ..., \zeta'_k\}$. There exists $c'_1, ..., c'_k$ such that ζ'_i is c'_i-closed, for $i = 1, ..., k$. If c' is the maximum of $c'_1, ..., c'_k$, then $\zeta'_1, ..., \zeta'_k$ are all c'-closed (by property 1). Since $\zeta' = \zeta'_1 \cup \cdots \cup \zeta'_k$, by property 2, ζ' is also c'-closed. ■

From now on, $c_{max}(\zeta)$ will denote the smallest constant c such that ζ is c-closed.

Lemma 2.2 *For any constant c, there is a finite number of c-closed convex $\mathcal{X}$-polyhedra.*

Proof: By induction on c. ∎

Projections. Given an $\mathcal{X}$-polyhedron ζ and a subset of clocks $Y \subseteq \mathcal{X}$, we define two *orthogonal projections* of ζ to Y. The *dimension-preserving* projection, denoted $\zeta/_Y$, is the $\mathcal{X}$-polyhedron ζ' such that:

$$\mathbf{v}' \in \zeta' \quad \text{iff} \quad \exists \mathbf{v} \in \zeta \,.\, \forall x \in Y \,.\, \mathbf{v}(x) = \mathbf{v}'(x)$$

The *dimension-restricting* projection, denoted $\zeta\rfloor_Y$, is defined identically to $\zeta/_Y$, except that $\zeta\rfloor_Y$ is a polyhedron on Y instead of $\mathcal{X}$. This type of projection can be extended to valuations in a straightforward way: if $\mathbf{v}$ is a valuation on $\mathcal{X}$, then $\mathbf{v}\rfloor_Y$ is the Y-valuation $\mathbf{v}'$ such that $\mathbf{v}'(y) = \mathbf{v}(y)$ for any $y \in Y$.

The operations $\zeta[Y := 0]$ and $[Y := 0]\zeta$ are defined as:

$$\begin{aligned} \zeta[Y := 0] &\stackrel{\text{def}}{=} \{\mathbf{v}[Y := 0] \mid \mathbf{v} \in \zeta\} \\ [Y := 0]\zeta &\stackrel{\text{def}}{=} \{\mathbf{v} \mid \mathbf{v}[Y := 0] \in \zeta\} \end{aligned}$$

Intuitively, $\zeta[Y := 0]$ contains all valuations which can be obtained from some valuation in ζ by resetting clocks in Y. An example is shown in the bottom-right diagram of figure 2.3. $[Y := 0]\zeta$ is the dual operation. It contains all valuations which, after resetting clocks in Y, yield a valuation in ζ. For example, for the polyhedron ζ_2 of figure 2.3, $[\{y\} := 0]\zeta_2$ is equal to $\zeta_2/_{\{y\}}$ (shown in the bottom-left diagram). On the other hand, $[\{y\} := 0]\zeta_1$ is empty.

We finally define the *backward* and *forward diagonal projections* of an $\mathcal{X}$-polyhedron ζ to be the $\mathcal{X}$-polyhedra $\swarrow\zeta$ and $\nearrow\zeta$, respectively, such that:

$$\begin{aligned} \mathbf{v}' \in \swarrow\zeta \quad &\text{iff} \quad \exists \delta \in \mathsf{R} \,.\, \mathbf{v}' + \delta \in \zeta \\ \mathbf{v}' \in \nearrow\zeta \quad &\text{iff} \quad \exists \delta \in \mathsf{R} \,.\, \mathbf{v}' \Leftrightarrow \delta \in \zeta \end{aligned}$$

The following result is easy to derive from the definitions.

Lemma 2.3 *If ζ is convex and $Y \subseteq \mathcal{X}$ then $\zeta/_Y, \zeta\rfloor_Y, \swarrow\zeta, \nearrow\zeta$ are also convex.*

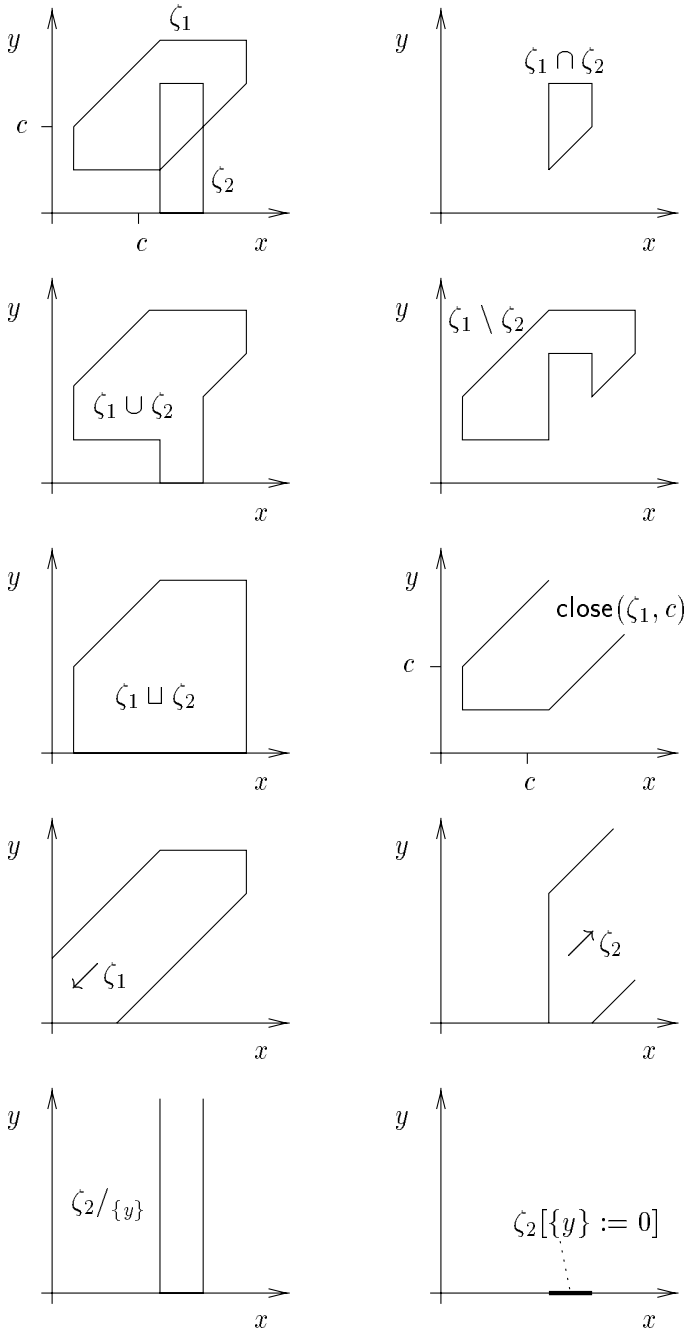


Figure 2.3: Polyhedra on $\{x, y\}$ and their operations.

Chapter 3

Timed Automata

In this chapter we introduce our system-specification language, timed automata. We first present the model informally, through an example which will be used through-out this document for illustrative purposes. Then we present formally the TA syntax and semantics, for a single automaton, as well as for the parallel composition of two or more automata. A special section is devoted to the issue of time progress (or zenoness) which is a characteristic of dense-time systems. The final section is also related to the issue of progress, discussing how simple static analysis can be applied to test the correctness of a TA model.

3.1 From finite-state machines to timed automata

We illustrate the difference of TA with respect to finite-state machines by considering of a gate-regulation system in a railroad crossing. The example is taken from [Alu91].

The system consists of three components, namely, a gate, a controller for the gate and a train. (A more realistic example would comprise more than one components such as trains or gates, and possibly distributed controllers.) The informal specification of the system can be stated in natural language as follows:

> The train sends a signal to the controller at least 2 time units before it enters the crossing, stays there no more than 3 time units and sends another signal to the controller upon exiting the crossing. The controller commands the gate to lower exactly one time unit after it has received the approaching signal from the train and commands the gate to rise again no more than 1 time unit after receiving the exiting signal. The gate takes less than 1 time unit to come down and between 1 and 2 time units to come up.

The system is modeled as a set of TA, one for each component, as shown in figure 3.1. Each automaton has a discrete structure, namely, a set of discrete states (depicted as circles) and a set of edges (depicted as arrows). The discrete states are supposed to capture all information about the current status of the system, except timing information. In this example, the discrete states are used to describe the function mode or the current condition of each of the components. For instance, the gate can be either "up" or "down", the train can be "far" from or "near" to the gate. The edges represent events which change the discrete state of the system. For instance, the train "approaches" the gate changing its state from "far" to "near". Such events are taken to be atomic and instantaneous.

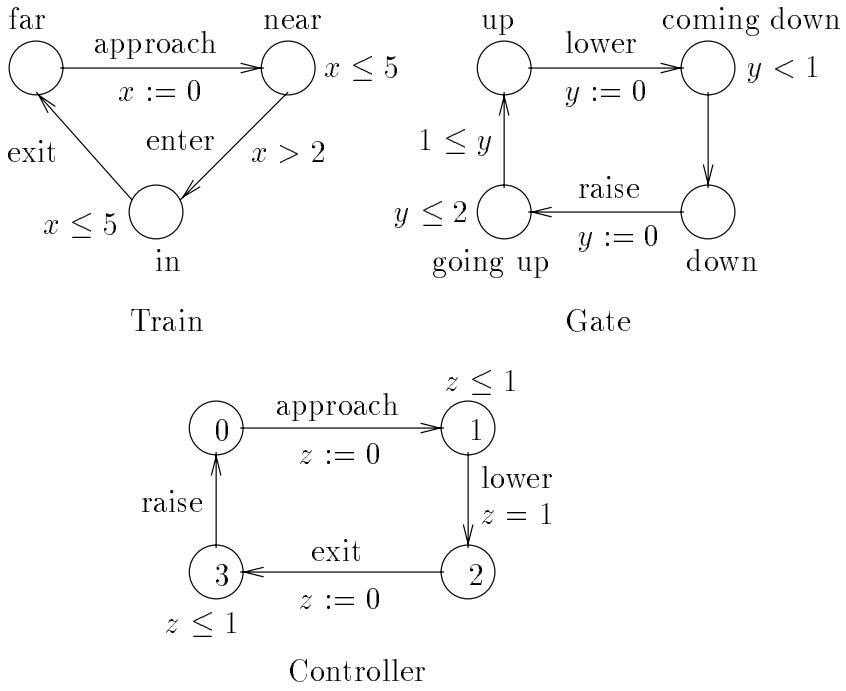


Figure 3.1: The Train–Gate–Controller example.

Time passes while the system remains at the same discrete state. There are three clocks (variables x, y, z) used to constraint the amount of time spent on discrete states and, more generally, the amount of time that passes between two events. For instance, the upper bound $y \leq 2$ at state "going up" of Gate, together with the fact that y is reset upon the edge "raise", models the fact that the Gate does not spend more than 2 time units rising.

Communication of components is modeled by action *synchronization*. Edges which are labeled with the same event correspond to actions which must happen simultaneously. For instance, the fact that the train sends a signal to the controller when it is approaching the gate is modeled by having an edge of the train automaton and an edge of the controller automaton both labeled "approach".

A sample execution of the Train-Gate-Controller (TGC) system is shown in figure 3.2.

Although quite simple, the TGC example illustrates two key features of TA, which distinguish them from finite-state machines.

- First, some executions which would have been possible if timing constraints are ignored are no longer valid because of their "bad timing". For instance, we can prove that in the TGC system above, whenever the train is in the crossing the gate is down.
- Second, quantitative statements can be made about the system's durations. For instance, we can prove that the gate never stays down for more than 5 time units.

3.2 Timed automata syntax and semantics

A *timed automaton* (TA) [ACD93, HNSY94] is a tuple $A = (\mathcal{X}, Q, q_0, E, \mathsf{invar})$, where:

- $\mathcal{X}$ is a finite set of clocks.

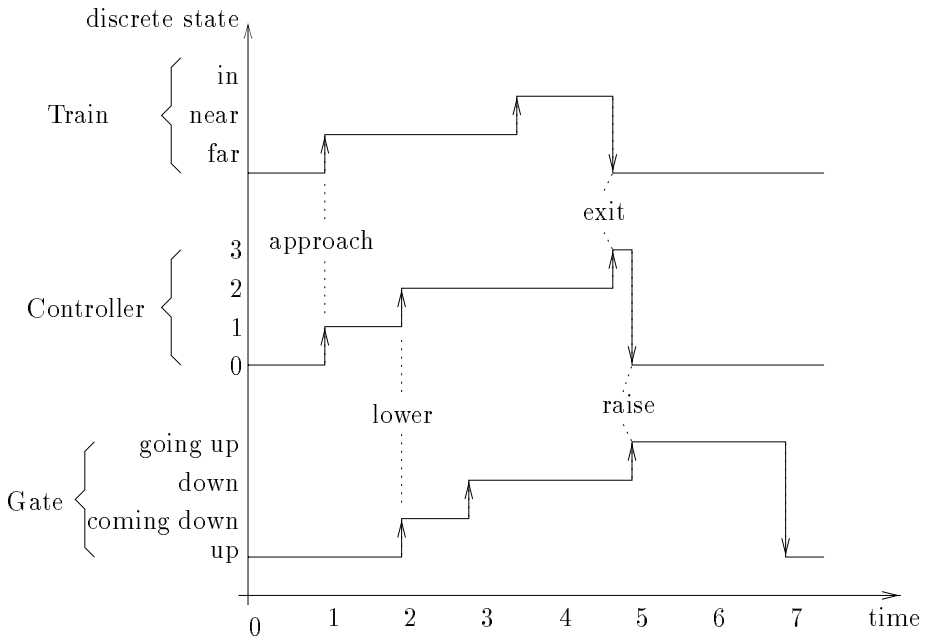


Figure 3.2: A sample execution of the Train-Gate-Controller system.

- Q is a finite set of *discrete states*.
- $q_0 \in Q$ is the *initial* discrete state.
- E is a finite set of *edges* of the form $e = (q, \zeta, a, X, q')$. $q, q' \in Q$ are the *source* and *target* discrete states. $a \in Labels$ is a label. ζ is a conjunction of atomic constraints on $\mathcal{X}$ defining a convex $\mathcal{X}$-polyhedron, called the *guard* of e. $X \subseteq \mathcal{X}$ is a set of clocks to be reset upon crossing the edge.
- invar is a function associating with each discrete state q a convex $\mathcal{X}$-polyhedron called the *invariant* of q.

Given an edge $e = (q, \zeta, a, X, q')$, we write $\mathsf{source}(e)$, $\mathsf{target}(e)$, $\mathsf{guard}(e)$, $\mathsf{label}(e)$ and $\mathsf{reset}(e)$ for q, q', ζ, a and X, respectively. Given a discrete state q, we write $\mathsf{in}(q)$ (resp. $\mathsf{out}(q)$) for the set of edges of the form $(\_, \_, \_, \_, q)$ (resp. $(q, \_, \_, \_, \_)$). We assume that for each $e \in \mathsf{out}(q)$, $\mathsf{guard}(e) \subseteq \mathsf{invar}(q)$. $c_{max}(A)$ is defined as the maximum of $c_{max}(\zeta)$, where ζ is a guard or an invariant of A.

Back to the TGC example of figure 3.1, the constraint $y > 2$ is a guard, $x \leq 5$ is an invariant (true invariants are not shown) and $x := 0$ denotes the set of clocks to reset $\{x\}$.

States. A *state* of A is a pair $(q, \mathbf{v})$, where $q \in Q$ is a location, and $\mathbf{v} \in \mathsf{invar}(q)$ is a valuation satisfying the invariant of q. We write $\mathsf{discrete}(s)$ to denote q, the discrete part of s. The *initial state* of A is $s_0 = (q_0, \mathbf{0})$. Two states $(q, \mathbf{v}_1)$ and $(q, \mathbf{v}_2)$ are c-equivalent if $\mathbf{v}_1$ and $\mathbf{v}_2$ are c-equivalent.

Transitions. Consider a state $(q, \mathbf{v})$. Given an edge $e = (q, \zeta, a, X, q')$ such that $\mathbf{v} \in \zeta$ and $\mathbf{v}' = \mathbf{v}[\mathsf{reset}(e) := 0] \in \mathsf{invar}(q')$, $(q, \mathbf{v}) \xrightarrow{e} (q', \mathbf{v}')$ is a *discrete transition* of A. $(q', \mathbf{v}')$ is called the *e-successor* of $(q, \mathbf{v})$.

A *time transition* from $(q, \mathbf{v})$ has the form $(q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v}+\delta)$, where $\delta \in \mathsf{R}$ and $\mathbf{v}+\delta \in \mathsf{invar}(q)$ [1]. For a state $s = (q, \mathbf{v})$, we simply write $s+\delta$ instead of $(q, \mathbf{v}+\delta)$. $s+\delta$ is the *δ-successor* of s. The *concatenation* of two time transitions $s \xrightarrow{\delta} s+\delta$ and $s+\delta \xrightarrow{\delta'} s+\delta+\delta'$ is a time transition $s \xrightarrow{\delta+\delta'} s+\delta+\delta'$. Inversely, due to the dense nature of the reals, a time transition $s \xrightarrow{\delta} s+\delta$ can be *split* to any number m of consecutive time transitions $s \xrightarrow{\delta_1} s+\delta_1 \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_m} s+\delta$, such that $\delta_1+\delta_2+\cdots+\delta_m = \delta$.

We write $s \xrightarrow{\delta} \xrightarrow{e} s'$ if, either $\delta = 0$ and $s \xrightarrow{e} s'$ is a discrete transition, or $\delta > 0$, $s \xrightarrow{\delta} s+\delta$ is a time transition and $s+\delta \xrightarrow{e} s'$ is a discrete transition.

We associate two kinds of semantics to a TA, namely, a *branching-time* semantics in terms of a labeled graph and a *linear-time* semantics in terms of executions (runs).

Semantic graph. The *semantic graph* of A, denoted G_A, is defined to be the graph which has as nodes the states of A and two types of edges, corresponding to the discrete and time transitions of A. Notice that G_A has generally an uncountable set of nodes and uncountable branching.

Runs. A *run* of A starting from state s is a finite or infinite sequence $\rho = s_1 \xrightarrow{\delta_1} s_1+\delta_1 \xrightarrow{e_1} s_2 \xrightarrow{\delta_2} s_2+\delta_2 \xrightarrow{e_2} \cdots$, such that $s_1 = s$ and for all $i = 1, 2, ...$, $s_i+\delta_i$ is the δ-successor of s_i and s_{i+1} is the e_i-successor of $s_i+\delta_i$. That is, a run is a path in the semantic graph of A where discrete transitions are taken infinitely often and consecutive time transitions are concatenated. The *i-th point* of ρ, denoted $\rho(i)$, is defined to be s_i, for $i = 1, 2, ...$ The *waiting delay of ρ at point i*, denoted $\mathsf{delay}(\rho, i)$, is defined to be δ_i. All states $\rho(i)$ where the run spends no time, that is, where $\mathsf{delay}(\rho, i) = 0$, are called *transient* states. The *elapsed time until point i*, denoted $\mathsf{time}(\rho, i)$, is defined to be the sum $\Sigma_{j<i}\mathsf{delay}(\rho, j)$. The *total elapsed time during ρ*, denoted $\mathsf{time}(\rho)$, is defined to be the limit of the sequence $\mathsf{time}(\rho, i)$, if the sequence converges and ∞ otherwise.

It is worth noticing that the semantics permit discrete transitions to be taken consecutively without any time passing in between. This is convenient sometimes for describing sequences of actions which are *atomic*, or assumed to consume a negligible amount of time. However, these sequences must be bounded, that is, the system cannot engage in a cycle (called a *critical race*) where time cannot progress at all. This issue is discussed in section 3.3 below.

A state s is *reachable* if there exists a finite run $s_0 \xrightarrow{\delta_0} \xrightarrow{e_0} \cdots \xrightarrow{\delta_l} \xrightarrow{e_l} s_l \xrightarrow{\delta} s$, for $l \geq 0$. Let $Reach(A)$ be the set of all reachable states of A.

Parallel composition of TA. A system is usually divided in parts, therefore, it is convenient (if not indispensable) to be able to describe systems *compositionally*, that is, as a set of components which execute in parallel and communicate in a certain way. Our model of parallelism is based on synchronous passage of time for all components and *interleaving* of discrete actions. Communication is modeled via action synchronization.

More precisely, consider two TA $A_i = (\mathcal{X}_i, Q_i, q_i, E_i, \mathsf{invar}_i)$, $i = 1, 2$, such that $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$. Let $Labels_i$ be the set of labels *local* to A_i, that is, $Labels_i = \{\mathsf{label}(e) \mid e \in E_i\}$, for $i = 1, 2$.

Given two edges $e_i = (q_i, a_i, \zeta_i, X_i, q_i') \in E_i$, $i = 1, 2$, we define the following *composite* edges:

[1] For technical convenience, we allow time transitions of zero delay, i.e., $(q, \mathbf{v}) \xrightarrow{0} (q, \mathbf{v})$.

- If $a_1 = a_2 \in \mathit{Labels}_1 \cap \mathit{Labels}_2$, then the synchronization of e_1 and e_2 yields the edge

$$e_1 \| e_2 \stackrel{\text{def}}{=} ((q_1, q_2), a_1, \zeta_1 \cap \zeta_2, X_1 \cup X_2, (q_1', q_2'))$$

  where ζ_1, ζ_2 are viewed as polyhedra on $\mathcal{X}_1 \cup \mathcal{X}_2$ so that the intersection $\zeta_1 \cap \zeta_2$ is well defined.

- If $a_i \notin \mathit{Labels}_1 \cap \mathit{Labels}_2$ for both $i = 1, 2$, then the interleaving of e_1 and e_2 yields the edges

$$\begin{aligned} e_1 \| \perp &\stackrel{\text{def}}{=} ((q_1, q_2), a_1, \zeta_1, X_1, (q_1', q_2)) \\ \perp \| e_2 &\stackrel{\text{def}}{=} ((q_1, q_2), a_2, \zeta_2, X_2, (q_1, q_2')) \end{aligned}$$

Syntactically, the *parallel composition* of A_1 and A_2, denoted $A_1 \| A_2$, is defined to be the TA $(\mathcal{X}_1 \cup \mathcal{X}_2, Q_1 \times Q_2, (q_1, q_2), E, \mathsf{invar})$, where, for $q \in Q_1$ and $q' \in Q_2$, $\mathsf{invar}(q, q') = \mathsf{invar}_1(q) \cap \mathsf{invar}_2(q')$, and the set of edges E contains all composite edges of the form $e_1 \| e_2$, $e_1 \| \perp$, $\perp \| e_2$, for $e_1 \in E_1, e_2 \in E_2$. That is, the two automata synchronize on their common labels and interleave on their local labels.

In order to give the semantic correspondence of $A_1 \| A_2$, we consider the semantic graphs G_1 and G_2 of A_1 and A_2, respectively. The parallel composition of G_1 and G_2, denoted $G_1 \| G_2$, is defined to be the smallest graph G such that:

1. (s_0^1, s_0^2) is a node of G, where s_0^i is the initial state of A_i, $i = 1, 2$.

2. If (s_1, s_2) is a node of G and $s_i \stackrel{\delta}{\to} s_i + \delta$ is an edge of G_i, for $i = 1, 2$, then $(s_1, s_2) \stackrel{\delta}{\to} (s_1 + \delta, s_2 + \delta)$ is an edge of G.

3. If (s_1, s_2) is a node of G and $s_1 \stackrel{e_1}{\to} s_1'$ (resp. $s_2 \stackrel{e_2}{\to} s_2'$) is an edge of G_1 (resp. G_2) such that $\mathsf{label}(e_1) \notin \mathit{Labels}_2$ (resp. $\mathsf{label}(e_2) \notin \mathit{Labels}_1$) then $(s_1, s_2) \stackrel{e_1}{\to} (s_1', s_2)$ (resp. $(s_1, s_2) \stackrel{e_2}{\to} (s_1, s_2')$) is an edge of G.

4. If (s_1, s_2) is a node of G and $s_i \stackrel{e_i}{\to} s_i'$ is an edge of G_i, for $i = 1, 2$, such that $\mathsf{label}(e_1) = \mathsf{label}(e_2)$ then $(s_1, s_2) \stackrel{e_1 \| e_2}{\to} (s_1', s_2')$ is an edge of G.

The second rule says that an amount of time δ passes in the composite TA only if both components can delay δ time units. The third rule says that local actions happen independently (interleaving). The fourth rule says that common actions happen simultaneously (synchronization).

The following lemma relates the syntactic parallel composition of TA with the parallel composition of their semantic graphs. The proof comes easily from the definitions.

Lemma 3.1 *The semantic graph of $A_1 \| A_2$ is identical to the parallel composition of the semantic graphs of A_1 and A_2.*

Since runs are paths in the semantic graph, the semantic correspondence of syntactic parallel composition can be directly extended to runs.

3.3 The requirement of progress in timed systems

Reactive systems are supposed to execute forever [2], which is referred to as the requirement of *progress*.

In untimed systems, progress coincides with absence of *deadlocks*, that is, states with no successors. In timed systems, there are two types of possible evolutions from a state, namely, taking a discrete transition or letting time pass. Accordingly, there are two requirements of progress here: First, it should be possible to take discrete transitions infinitely often (discrete progress). Second, it should be possible to let time pass infinitely often, and this *without upper bound* (time progress). Notice that the requirement of time progress is stronger than one might expect, that is, not only time should be able to pass, but it should also diverge. The time progress requirement is based on our intuition about the physical world we are trying to model, summarized in the following hypothesis:

> Any physical process, no matter how fast, cannot be infinitely fast.

The above hypothesis implies that:

1. only a finite (possibly unbounded) number of events can occur in a certain (positive) amount of time;
2. only a bounded number of events can occur in zero time.

These two requirements are formalized below under the concepts of *non-zenoness* and absence of *critical races*, respectively.

Zeno runs. Consider an infinite run ρ such that $\mathsf{time}(\rho) \neq \infty$, that is, there exists $t \in \mathsf{R}$ such that for all i, $\mathsf{time}(\rho, i) < t$. Such a run is called *zeno*, and corresponds to a pathological situation, since it violates the first of the above time-progress requirements. As an example, consider the TA A_1 shown in figure 3.3. Its run $(q_0, x = 1) \xrightarrow{\frac{1}{2}} \xrightarrow{a} (q_0, x = 1.5) \xrightarrow{\frac{1}{4}} \xrightarrow{a} (q_0, x = 1.75) \cdots$ is zeno. In fact, any run of A_1 taking a-transitions forever is zeno.

Let $NonZenoRuns(s)$ denote the set of all non-zeno runs starting from s.

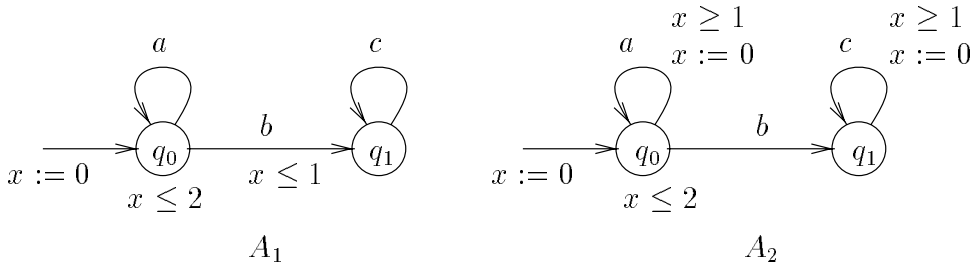


Figure 3.3: A TA with timelocks (left) and a strongly non-zeno TA (right).

[2] This is a convenient simplifying hypothesis, which does not result in loss of generality: if the system can terminate execution in some legal *end state*, the latter can be transformed to a state which has infinite executions by adding to it a "dummy" self-loop transition.

Deadlocks, Timelocks and Critical races. Deadlocks are states violating the discrete-progress requirement. Formally, a state s of a TA A is a *deadlock* if there is no delay $\delta \in \mathsf{R}$ and edge $e \in E$ such that $s \xrightarrow{\delta} \xrightarrow{e} s'$. A is *deadlock-free* if none of its reachable states is a deadlock.

Timelocks are states violating the time-progress requirement. Formally, a state s is a *timelock* if all infinite runs starting from s are zeno. A is *timelock-free* if none of its reachable states is a timelock.

Notice that a deadlock is not necessarily a timelock, neither the reverse. For example, the TA A_1 of figure 3.3 is deadlock-free, but all its states $(q_0, 1 < x \leq 2)$ are timelocks since they are bound to stay to q_0 taking forever a-transitions. On the other hand, if the a-edge was missing, these states would be deadlocks but not timelocks, since they would have no infinite runs starting from them at all.

A infinite run $\rho = s_1 \xrightarrow{\delta_1} \xrightarrow{e_1} s_2 \xrightarrow{\delta_2} \xrightarrow{e_2} \cdots$ is a *critical race* if the following conditions hold:

1. All states of ρ are transient from some point on, that is, $\exists i \,.\, \forall j > i \,.\, \delta_j = 0$.

2. ρ cannot be transformed to a non-transient run, that is, there exists no run $\rho' = s_1 \xrightarrow{\delta'_1} \xrightarrow{e_1} s'_2 \xrightarrow{\delta'_2} \xrightarrow{e_2} \cdots$ such that $\forall i \,.\, \exists j > i \,.\, \delta'_j > 0$.

Critical races correspond to executions violating the second time-progress requirement above. They are not simply zeno runs where time does not pass at all from some point on. Indeed, a critical race cannot be transformed to a (possibly zeno) run where time does pass, even by infinitesimal quantities. A timelock-free TA can have critical races, as shown in figure 3.4. The two TA execute asynchronously in parallel. The sequence of actions $abcdabcd\cdots$ in the composed system corresponds to a critical race, since not time is allowed to pass at all from one b to the next a and from one d to the next c. On the other hand, the sequence $acbdacbd\cdots$ can allow time to progress after every c action.

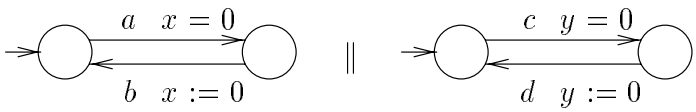


Figure 3.4: A system with critical races.

Strongly non-zeno TA. Consider a TA A. A *structural loop* of A is a sequence of distinct edges $e_1 \cdots e_m$ such that $\mathsf{target}(e_i) = \mathsf{source}(e_{i+1})$, for all $i = 1, ..., m$ (the addition $i + 1$ is modulo m). A is called *strongly non-zeno* if for every structural loop there exists a clock x and some $0 \leq i, j \leq m$ such that:

1. x is reset in step i, that is, $x \in \mathsf{reset}(e_i)$; and

2. x is bounded from below in step j, that is, $(x < 1) \cap \mathsf{guard}(e_j) = \mathsf{false}$.

Intuitively, this means that at least one unit of time elapses in every loop of A. For example, the TA A_2 of figure 3.3 is strongly non-zeno (this would not be the case if any of the guards $x \geq 1$ was missing).

Strong non-zenoness is interesting since it dispenses us with the burden of ensuring time progress. In particular, checking progress is reduced to checking deadlock-freedom, as shown

below. Another nice characteristic of strong non-zenoness is that it is preserved by parallel composition, so that it can be efficiently checked on large systems.

Lemma 3.2 *1. If A is strongly non-zeno then every infinite run of A is non-zeno.*

2. If A, A' are strongly non-zeno, so is $A\|A'$.

Proof: For the first part, let $\rho = s_1 \xrightarrow{\delta_1} \xrightarrow{e_1} s_2 \xrightarrow{\delta_2} \xrightarrow{e_2} \cdots$ be an infinite run of A. Since A has only a finite number of edges, there exist some $i_1, i_2, ..., i_m$ such that $e_{i_1} e_{i_2} \cdots e_{i_m}$ form a structural loop and ρ takes infinitely often every discrete transition e_{i_j}. There exist also a clock x and $j_1, j_2 \in \{i_1, i_2, ..., i_m\}$ such that $x \in \mathsf{reset}(e_{j_1})$ and $(x < 1) \cap \mathsf{guard}(e_{j_2}) = \emptyset$. Now, each time ρ takes an e_{j_1}-transition, clock x is reset to 0. The next time ρ takes an e_{j_2}-transition, at least 1 time unit has passed, since x must be greater or equal to 1 for e_{j_2} to be taken. Since e_{j_1}- and e_{j_2}-transitions are taken infinitely often, an infinite number of 1-time-unit delays are accumulated, thus ρ is non-zeno.

For the second part, observe that any structural loop of $A\|A'$ corresponds to a structural loop of A, A', or both. Therefore, any structural loop of $A\|A'$ satisfies the conditions of part 1, which implies that $A\|A'$ is strongly non-zeno. ■

As a corollary of part 1 of the above lemma, a strongly non-zeno TA is also timelock-free.

Remark 3.3 *When modeling a system, it is often the case that some of its components are untimed, that is, they can be modeled using simple finite-state machines without clocks. These components can be considered strongly non-zeno by convention, so that their parallel composition with the rest of the system does not affect the strong non-zenoness of the global system.*

The meaning of different variants of zenoness. Concerning system modeling, the meaning of zenoness can be summarized as follows:

- Deadlocks, timelocks and critical races correspond to modeling errors, since any TA assumed to capture the behavior of a reactive system correctly should act infinitely often, not block time and execute a bounded number of actions in zero time.
- TA which are not strongly non-zeno model systems where an unbounded number of events can occur in a finite amount of time. For example, the TA A_1 on figure 3.3 can perform an unbounded number of a-transitions in 2 time units. Such systems are useful sometimes, for instance, when modeling a sender which can emit messages arbitrarily fast.
- Strongly non-zeno TA model systems where only a bounded number of events can occur in a finite amount of time. For example, the TA A_2 on figure 3.3 can perform at most two a-transitions in 2 time units. Most systems in practice are strongly non-zeno.

Concerning verification, the impact of zenoness can be summarized as follows:

- Methods to ensure absence of deadlocks, timelocks and critical races should be available so that one gains confidence in the correctness of the model. In section 3.4 we give static tests guaranteeing the absence of the above errors. In sections 6.2.4, 7.1.1 and 7.1.2 we present run-time detection techniques for deadlocks and timelocks.
- Model checking algorithms should ignore zeno runs when verifying a property. For example, the two TA in figure 3.3 do not have the same untimed behaviors, since a^ω corresponds to a non-zeno run of A_2 but only to zeno runs of A_1.

3.4 Static tests for the sanity of timed automata

We propose sufficient but not necessary conditions to ensure absence of deadlocks, timelocks and critical races in a TA. These conditions are *static*, that is, they take into account the discrete structure of the TA but not the reachable state space in the presence of timing constraints. This is why the conditions are not necessary: an untimed behavior of the TA which does not satisfy the conditions might not be valid when the timing constraints are considered.

Deadlocks. Before presenting a static test, we characterize deadlock-freedom of a TA A by a *local* condition on the reachable states of A. Let q be a discrete state of A and define:

$$\mathsf{free}(q) \stackrel{\text{def}}{=} \bigcup_{e \in \mathsf{out}(q)} \swarrow \Big(\mathsf{guard}(e) \cap \big([\mathsf{reset}(e) := 0] \mathsf{invar}(\mathsf{target}(e)) \big) \Big)$$

Intuitively, $\mathsf{free}(q)$ containts all states with discrete part q, which can let some time pass and take a discrete transition exiting q. Then, it is easy to see the following.

Lemma 3.4 *A is deadlock-free iff $\forall (q, \mathbf{v}) \in Reach(A)$. $\mathbf{v} \in \mathsf{free}(q)$.*

Based on this characterization, a sufficient static condition for deadlock-freedom is provided by the following lemma.

Lemma 3.5 *If for each discrete state q of A and for all $e \in \mathsf{in}(q)$, $((\mathsf{guard}(e))[\mathsf{reset}(e) := 0]) \cap \mathsf{invar}(q) \subseteq \mathsf{free}(q)$, then A is deadlock-free.*

Notice that the above condition is not *compositional*, that is, two TA might satisfy the condition while their parallel composition does not.

Timelocks. A sufficient condition for timelock-freedom is strong non-zenoness, by part 1 of lemma 3.2. Checking that a system of TA is strongly non-zeno can be done compositionally, by part 2 of the same lemma. By definition, the test for strong non-zenoness is static.

Critical races. Let A be a TA with set of edges E and set of clocks $\mathcal{X}$. Informally, A has no critical races if no structural loop of A can be "covered" by one or more segments, each of which does not let time pass at all. This is illustrated in figure 3.5, where the loop $\stackrel{e_1}{\rightarrow}\stackrel{e_2}{\rightarrow}$ can be covered by $\stackrel{e_1}{\rightarrow}\stackrel{e_2}{\rightarrow}$ (where x forbids time to pass) and $\stackrel{e_2}{\rightarrow}\stackrel{e_1}{\rightarrow}$ (where y forbids time to pass). If such loops do not exist then absence of critical races can be guaranteed.

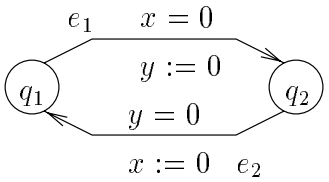


Figure 3.5: A structural loop generating a critical race.

More formally, given an edge e and a clock x, we say that x *is reset in* e if $x \in \mathsf{reset}(e)$ and that x *is zero-bounded in* e if $\mathsf{guard}(e) \subseteq (x = 0)$. Then, consider a structural loop

$q_1 \stackrel{e_1}{\rightarrow} \cdots q_m \stackrel{e_m}{\rightarrow} q_1$. For $1 \leq i,j,k \leq m$, define the ternary relation $<_m (i,j,k)$ (j "is between i and k modulo m) such that, either $i < k$ and $i \leq j < k$ or $i < j \leq k$, or $i \geq k$ and $i \leq j \leq m$ or $1 \leq j \leq k$. For example, if $m > 2$ then $<_m (1,2,m)$ and $<_m (2,m,1)$.

Lemma 3.6 *A has no critical races if in every structural loop $\stackrel{e_1}{\rightarrow} \cdots \stackrel{e_m}{\rightarrow}$, there exists $1 \leq i \leq m$ such that for any clock x which is zero-bounded in e_k, $1 \leq k \leq m$, there exists $1 \leq j \leq m$ such that $<_m (i,j,k)$ and x is reset in e_j.*

Unfortunately, parallel composition does not preserve the absence of critical races, as can be seen in the example of figure 3.4: although both TA shown in the figure are critical-race-free, their parallel composition has the critical race corresponding to the sequence of actions $(abcd)^\omega$.

The test of lemma 3.6 can be costly, since the number of structural loops is exponential on the number of discrete states of a TA, and the latter grows also exponentially with the number of component TA. A cheaper solution is to check for each automaton A in a system of TA that no atomic constraint of the form $x = 0$ appears in any structural loop of A. Then, it is guaranteed that the composite system satisfies the same condition, which implies the one of lemma 3.6.

Relation to the literature

TA were first introduced in [Dil89, Lew90, AD90]. Our TA model differs from the one of [AD90] in that it uses invariants and permits a bounded number of discrete transitions to happen in zero time. Our model is also different from the one of [HNSY94] in that it requires an infinite number of discrete transitions in every infinite run, whereas theirs permits executions where the TA stays forever in the same discrete state. The definition we adopted in this thesis is more general, since it permits to distinguish between the following cases:

(1) an event a occurs eventually but we do not know when; and

(2) an event a may never occur.

We model case (1) by having an edge labeled a going out of a discrete state with invariant `true`. We model case (2) by adding a "dummy" self-loop edge to the state. Using the definition of [HNSY94], case (1) cannot be modeled since a `true` invariant implies that there exists an infinite run staying forever in the corresponding state.

Invariants have been introduced in [HNSY94] to model *time-progress conditions*. Some weaknesses of the model with respect to parallel composition of TA have been first identified in [SY96]. Since then, a new model has been proposed which expresses the urgency information on transitions rather than states, using so-called *deadlines* [BS97, Bor98, BST98].

The notion of non-zenoness was introduced at the same time as TA. [Yov93, HNSY94] introduce the notion of *well-timed* systems to capture deadlock and timelock freedom, without, however, distinguishing between deadlocks and timelocks. To our knowledge, critical races have not been defined previously.

Chapter 4

Property-specification Languages

In the previous chapter we have introduced the formalism of TA for the description of timed systems. In this chapter we present formalisms to express properties of timed systems. We consider two types of formalisms, namely, *linear-* and *branching-time*.

In linear time, properties are viewed as sets of executions, so that specifications are evaluated on runs. In branching time, properties are viewed as sets of execution trees, so that specifications are evaluated on the semantic graph. The two views are incomparable, that is, there are properties expressed in linear time but not in branching time and vice-versa. On the other hand, *safety* properties ("p always holds"), which can capture most-frequently used properties like invariance and bounded response, are expressible in both linear and branching time.

In this thesis we consider both linear-time properties, expressed by Timed Büchi Automata, and branching-time properties, expressed by the logic TCTL. We also consider the automata-based logic $\text{ETCTL}^*_\exists$ which is strictly more expressive than both TBA and TCTL.

4.1 A linear-time formalism: Timed Büchi Automata

Timed Büchi automata have been introduced in [Alu91] as a real-time extension of Büchi automata [Büc62].

Syntax and semantics. A *timed Büchi automaton* (TBA) is a tuple $B = (A, F)$, where $A = (\mathcal{X}, Q, q_0, E, \mathsf{invar})$ is a TA and $F \subseteq Q$ is a set of *repeating* discrete states.

The notions of states, transitions and runs of TA are easily extended to TBA. A state s of B is called repeating if $\mathsf{discrete}(s)$ is repeating. A run ρ of B is called *accepting* if ρ visits repeating states infinitely many times, that is, for all i there exists $j > i$ such that $\mathsf{discrete}(\rho(j)) \in F$. B is said to have *trivial acceptance condition* if for every accepting run ρ of B, ρ remains in F from some point on, that is, there exists i such that for all $j > i$, $\mathsf{discrete}(\rho(j)) \in F$. The *language* of B, denoted $Lang(B)$, is the set of all accepting, non-zeno runs of B starting from its initial state. The *emptiness problem* for a TBA B consists in deciding whether its language is empty, and if not, provide an accepting, non-zeno run.

Let A' be a TA with set of discrete states Q'. Also let $P : Q \mapsto 2^{Q'}$ be a function associating to each discrete state of B a set of discrete states of A'. A run ρ' of A' *satisfies* B with respect to P, written $\rho' \models_P B$, if there exists a run $\rho \in Lang(B)$ such that for all $i = 0, 1, ...$:

1. $\mathsf{delay}(\rho', i) = \mathsf{delay}(\rho, i)$.

2. $\mathsf{discrete}(\rho'(i)) \in P(\mathsf{discrete}(\rho(i)))$.

Condition 1 says that the two runs take their discrete steps at the same time. Condition 2 ensures that at any time instant the discrete state of A' meets the requirements specified by the discrete state of B. Notice that, by definition, only non-zeno runs of A' satisfy B.

A' satisfies B if there exists a run starting from the initial state of A' satisfying B.

Remark 4.1 *Our definition of TBA satisfaction is based on language* intersection *(i.e., there exists an execution of the system which is in the language of the TBA) rather than the usual automata-theoretic definition based on language* inclusion *(i.e., every execution of the system is in the language of the TBA). Since (non-deterministic) TBA are not closed under complementation, the problem of inclusion is generally undecidable [Alu91]. However, the problem of intersection is decidable.*

Defining TBA satisfaction as a problem of language intersection implies that if we want to prove that all *behaviors of a system A satisfy a property ϕ, then we have to use a TBA $B_{\neg\phi}$ expressing the* negation *of ϕ: a behavior is in the language of $B_{\neg\phi}$ iff it does not satisfy ϕ. Then, all the behaviors of A satisfy ϕ iff A does not satisfy $B_{\neg\phi}$. For most interesting properties, $B_{\neg\phi}$ can easily be found in practice (actually, it is sometimes more intuitive to construct a TBA expressing the negation of the property than the property itself).*

Examples of property specification. TBA can be used to express a property either directly, or via its negation. Two examples are shown in figure 4.1. The *liveness* property "there exists an execution where p holds infinitely often" is modeled by the (untimed) Büchi automaton B_1, expressing the property directly. On the other hand, the *bounded-response* property "every instance of p_1 is followed by an instance of p_2 within at most k time units" is modeled indirectly by the TBA B_2, expressing the negation of the property. Notice that B_2 has trivial acceptance condition, but not B_1.

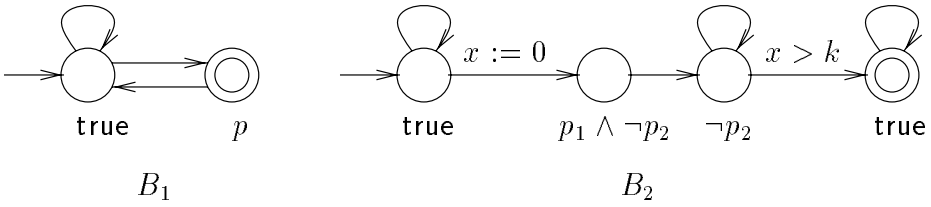


Figure 4.1: Examples of (timed) Büchi automata.

The above examples show the main difference between our definition of satisfaction of TBA and the usual definition of satisfaction of untimed linear formalisms. Instead of requiring that *all* behaviors of the system satisfy the property specified by the TBA, we require that at least *one* of them does so (see also discussion at the end of the chapter). This implies that when we have to express a property of the form "in all behaviors of the system ϕ holds", we have to find a TBA which specifies $\neg\phi$, so that the above property holds iff the system does *not* satisfy the TBA.

Reducing TBA satisfaction to TBA emptiness. As usual, the problem of checking whether a TA A satisfies a TBA specification B can be reduced to the problem of checking whether the *synchronous product* of A and B, denoted $A \times B$, has an empty language.

Formally, let A be $(\mathcal{X}, Q, q_0, E, \mathsf{invar})$ and $B = (\mathcal{X}', Q', q_0', E', \mathsf{invar}', F)$. Also let $P : Q' \mapsto 2^Q$ be a proposition-labeling function as before. $A \times B$ is defined only if $q_0 \in P(q_0')$, as the TBA $(\mathcal{X} \cup \mathcal{X}', Q'', (q_0, q_0'), E'', \mathsf{invar}'', F'')$, where:

- $Q'' = \{(q, q') \in Q \times Q' \mid q \in P(q')\}$.
- E'' contains all composite transitions $e \| e'$ such that $e \in E$, $e' \in E'$, $\mathsf{source}(e) \in P(\mathsf{source}(e'))$ and $\mathsf{target}(e) \in P(\mathsf{target}(e'))$.
- $\mathsf{invar}''(q, q') = \mathsf{invar}(q) \cap \mathsf{invar}'(q')$.
- $F'' = (Q \times F) \cap Q''$.

It is easy to see that if $s_0' \xrightarrow{\delta_0} \xrightarrow{e_0'} s_1' \cdots$ is a run in $Lang(B)$ and $s_0 \xrightarrow{\delta_0} \xrightarrow{e_0} s_1 \cdots$ is a run of A satisfying B, then $(s_0, s_0') \xrightarrow{\delta_0} \xrightarrow{e_0 \| e_0'} (s_1, s_1') \cdots$ is a run in $Lang(A \times B)$. Inversely, any run in $Lang(A \times B)$ can be "projected" in two runs ρ and ρ' such that ρ satisfies ρ'. Then:

Lemma 4.2 *A satisfies B iff the language of $A \times B$ is non-empty.*

4.2 The branching-time logic TCTL

Timed Computation Tree Logic has been introduced in [ACD93] as a real-time extension of the branching-time logic CTL [EC81].

Syntax and semantics. Let $\mathcal{I}$ denote the set of all intervals of R of the form $[c, c']$, $[c, c')$, $(c, c']$, (c, c'), (c, ∞) and $[c, \infty)$, where $c, c' \in \mathsf{N}$. A formula in TCTL is defined according to the following syntax:

$$\phi \quad ::= \quad \mathsf{true} \mid p \mid \neg\phi \mid \phi \vee \phi \mid \exists\, \phi\, \mathcal{U}_I\, \phi \mid \forall\, \phi\, \mathcal{U}_I\, \phi$$

where $p \in \mathit{Props}$ is an atomic proposition and $I \in \mathcal{I}$ is an interval.

Let A be a TA with set of discrete states Q. Also let $P : \mathit{Props} \mapsto 2^Q$ be a function associating to each atomic proposition a set of discrete states of A. TCTL formulae are interpreted over states of A. Given a formula ϕ and a state s, the satisfaction relation $s \models_P \phi$ is defined inductively on the syntax of ϕ as follows (we omit the subscript $_P$ for simplicity):

$$\begin{array}{lll}
s \models \mathsf{true} & & \\
s \models p & \text{iff} & \mathsf{discrete}(s) \in P(p) \\
s \models \neg\phi_1 & \text{iff} & \text{not } s \models \phi_1 \\
s \models \phi_1 \vee \phi_2 & \text{iff} & s \models \phi_1 \text{ or } s \models \phi_2 \\
s \models \exists\, \phi_1\, \mathcal{U}_I\, \phi_2 & \text{iff} & \exists \rho = s \xrightarrow{\delta_1} \xrightarrow{e_1} \cdots \text{ s.t. } \mathsf{time}(\rho) = \infty \text{ and} \\
 & & \quad \exists i \,.\, \Sigma_{j \le i} \delta_j \in I \text{ and } \rho(i) + \delta_i \models \phi_2 \text{ and} \\
 & & \quad\quad \forall j < i \,.\, \forall \delta \le \delta_j \,.\, \rho(j) + \delta \models \phi_1 \vee \phi_2 \\
s \models \forall\, \phi_1\, \mathcal{U}_I\, \phi_2 & \text{iff} & \forall \rho = s \xrightarrow{\delta_1} \xrightarrow{e_1} \cdots \text{ s.t. } \mathsf{time}(\rho) = \infty \,. \\
 & & \quad \exists i \,.\, \Sigma_{j \le i} \delta_j \in I \text{ and } \rho(i) + \delta_i \models \phi_2 \text{ and} \\
 & & \quad\quad \forall j < i \,.\, \forall \delta \le \delta_j \,.\, \rho(j) + \delta \models \phi_1 \vee \phi_2
\end{array}$$

Intuitively, s satisfies the formula $\exists\, \phi_1\, \mathcal{U}_I\, \phi_2$ if there exists a non-zeno run ρ starting from s and a point along the run such that the time spent until that point belongs to the interval I, ϕ_2

holds at that point and ϕ_1 holds continuously until that point. The interpretation for $\forall\, \phi_1 \mathcal{U}_I\, \phi_2$ differs only in the quantification over runs starting from s: here it is required that *all* such runs meet the conditions. The interpretation of boolean operators, atomic propositions, and the trivial formula true is straightforward.

The following abbreviations are defined:

$$\begin{aligned}
\exists \Diamond_I\, \phi &\stackrel{\text{def}}{=} \exists\, \mathsf{true} \mathcal{U}_I\, \phi \\
\forall \Diamond_I\, \phi &\stackrel{\text{def}}{=} \forall\, \mathsf{true} \mathcal{U}_I\, \phi \\
\forall \Box_I\, \phi &\stackrel{\text{def}}{=} \neg \exists \Diamond_I\, \neg \phi \\
\exists \Box_I\, \phi &\stackrel{\text{def}}{=} \neg \forall \Diamond_I\, \neg \phi
\end{aligned}$$

We also simplify notation for intervals, for instance, we write $\exists \Diamond_{\leq 5}\, \phi$ instead of $\exists \Diamond_{[0,5]}\, \phi$ and $\forall \Box\, \phi$ instead of $\forall \Box_{[0,\infty)}\, \phi$.

We say that the TA A satisfies a formula ϕ if the initial state of A satisfies ϕ.

Examples of property specification. We now give some examples of TCTL formulae. The *invariance* property "p always holds" can be expressed by the formula $\forall \Box\, p$. The formula $\forall \Box_{[3,5]}\, p$ requires that p holds only during the interval $[3, 5]$. Bounded response is expressed by the formula $\forall \Box\, (p_1 \Rightarrow \forall \Diamond_{\leq k}\, p_2)$ (the TBA B_2 of figure 4.1 models precisely the negation of this formula). Finally, the *escape-possibility* property stating that "it is always possible for p to hold" can be expressed by the formula $\forall \Box\, \exists \Diamond\, p$.

CTL. It is useful to identify an interesting subclass of TCTL, namely, CTL, the *untimed* subclass of TCTL containing all formulae with *trivial subscript interval* $[0, \infty)$.

4.3 A mixture of branching and linear time: the logic ETCTL$^*_\exists$

The logic ETCTL$^*_\exists$ (*extended* TCTL$^*_\exists$) [BLY96] is a real-time version of the automata-based logic ECTL* introduced in [HT87]. ETCTL$^*_\exists$ is more expressive than both TCTL and TBA (see next section). Intuitively, ETCTL$^*_\exists$ can be seen as an extension of TBA where, instead of associating with each discrete state of the TBA a simple atomic proposition, we associate a general sub-formula. That is, ETCTL$^*_\exists$ is an extension of TBA with nesting.

Formally, the syntax of ETCTL$^*_\exists$ is as follows:

$$\phi \quad ::= \quad \mathsf{true} \;\mid\; p \;\mid\; \neg \phi_1 \;\mid\; \phi_1 \vee \phi_2 \;\mid\; \exists B(\phi_1, \ldots, \phi_n)$$

where $p \in \mathit{Props}$ is an atomic proposition and B is a TBA with set of discrete states $Q = \{q_1, ..., q_n\}$.

The semantics of ETCTL$^*_\exists$ is a combination of those of TCTL and TBA. ETCTL$^*_\exists$ formulae are interpreted over states of a TA A. The satisfaction rules are as for TCTL in the case of atomic propositions and boolean formulae, and similar to those of TBA for the type of formulae $\exists B(\phi_1, \ldots, \phi_n)$. Informally, a state s satisfies $\exists B(\phi_1, \ldots, \phi_n)$ if there is a run ρ' of A starting from s and a run ρ in the language of B, such that ρ' and ρ execute synchronously and at each point in time the state of ρ' satisfies the sub-formula specified by the discrete state of ρ.

Formally, let A be a TA with set of discrete states Q' and $P : \mathit{Props} \mapsto 2^{Q'}$ be a function mapping each atomic proposition to a set of discrete states of A. The satisfaction relation $\models_P$ between a state s of A and an ETCTL$^*_\exists$ formula ϕ is defined inductively on the syntax of ϕ as follows (we omit the subscript P for simplicity):

$$
\begin{array}{lll}
s \models \mathsf{true} & & \\
s \models p & \text{iff} & \mathsf{discrete}(s) \in P(p) \\
s \models \neg\phi_1 & \text{iff} & \text{not } s \models \phi_1 \\
s \models \phi_1 \vee \phi_2 & \text{iff} & s \models \phi_1 \text{ or } s \models \phi_2 \\
s \models \exists B(\phi_1, ..., \phi_n) & \text{iff} & \exists s_0 \xrightarrow{\delta_0} \xrightarrow{e'_0} s_1 \cdots \in \mathit{NonZenoRuns}(s) \ . \\
 & & \exists (q_{j_0}, \mathbf{v}_{j_0}) \xrightarrow{\delta_0} \xrightarrow{e_0} (q_{j_1}, \mathbf{v}_{j_1}) \cdots \in \mathit{Lang}(B) \ . \\
 & & \quad \forall i \ . \ \forall \delta \leq \delta_i \ . \ s_i + \delta \models \phi_{j_i}
\end{array}
$$

As we shall see in the section that follows, ETCTL$^*_\exists$ is strictly more expressive than both TBA and TCTL.

4.4 Comparison of the different specification languages

ETCTL$^*_\exists$ subsumes both TBA and TCTL. Consider the case of TBA first. Let B be a TBA with set of discrete states $Q = \{q_1, ..., q_n\}$, A be a TA with set of discrete states Q' and $P : Q \mapsto 2^{Q'}$ be a function mapping a discrete state of B to a set of discrete states of A. Then, we define Props to be a set of atomic propositions $\{p_1, ..., p_n\}$ and $P' : \mathit{Props} \mapsto 2^{Q'}$ to be such that $P'(p_i) = P(q_i)$, for $i = 1, ..., n$. It is easy to see that $A \models_P B$ iff $A \models_{P'} \exists B(p_1, ..., p_n)$.

As for TCTL, [BLY96] prove that any TCTL formula ϕ can be translated to an ETCTL$^*_\exists$ formula. The translation is done recursively on the syntax of ϕ and has complexity linear on the size of ϕ. As an example, figure 4.2 shows how the TCTL formulae $\exists\, \phi_1 \mathcal{U}_{\leq k}\, \phi_2$ and $\forall\, \phi_1 \mathcal{U}_{\leq k}\, \phi_2$ can be translated to the ETCTL$^*_\exists$ formulae $\exists B_1(\phi_1, \phi_2, \mathsf{true})$ and $\neg\exists B_2(\neg\phi_2, \neg(\phi_1 \vee \phi_2), \mathsf{true})$, respectively.

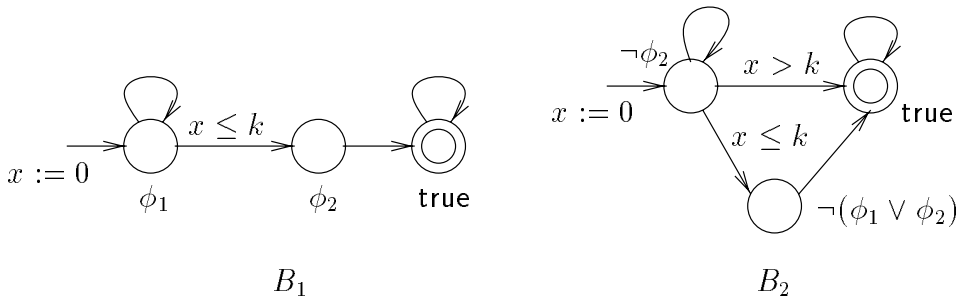


Figure 4.2: Translating TCTL to ETCTL$^*_\exists$.

Some remarks are worth making concerning the translation of TCTL to ETCTL$^*_\exists$.

First, a universally-quantified formulae such as $\forall\psi$ is translated to $\neg\exists B(\cdots)$, where B expresses the negation of the property expressed by ψ. For example, the TBA B_2 above expresses the property: "there exists a run which either fails to satisfy ϕ_2 before k time units, or fails to satisfy ϕ_1 continuously until ϕ_2 becomes true".

Second, for some nested TCTL formulae, more efficient translations can be found than the ones given by the formal translation algorithm. In particular, such TCTL formulae can be captured by $\text{ETCTL}^*_\exists$ formulae without nesting (i.e., to TBA), modulo negation. For example, $\forall\Box\,(p_1 \Rightarrow \forall\Diamond_{\leq k}\, p_2)$ is translated to $\neg\exists B_2(\mathsf{true}, p_1 \wedge \neg p_2, \neg p_2, \mathsf{true})$, where B_2 is the TBA on the right of figure 4.1.

Third, notice that both TBA in figure 4.2 have trivial acceptance conditions. In fact, this is true for any TBA appearing in an $\text{ETCTL}^*_\exists$ formula obtained from the translation of a TCTL formula. This is important in practice for $\text{ETCTL}^*_\exists$ model checking: the latter is based on TBA emptiness, which can be solved more efficiently for TBA with trivial acceptance (see section 7.2).

We finally turn to the comparison of TBA and TCTL. The two formalisms are incomparable in expressiveness, a fact that has to do with the linear nature of TBA and the branching nature of TCTL, rather than their timed features. In particular, the CTL formula $\forall\Box\,\exists\Diamond\, p$ cannot be captured by any TBA. On the other hand, the BA B_1 of figure 4.1 cannot be captured by any TCTL formula. These results are direct extensions of the results for linear- and branching-time incomparability in the untimed case, which can be found, for instance, in [Lam80, EH86].

Relation to the literature

TBA were used for the specification of timed properties in [Alu91], based on the classical automata-theoretic definition of language inclusion. TCTL has also been introduced in [Alu91], although with a slightly different definition of satisfaction using sequences of intervals of the real line mapped to states. Our definition is closer to the one of [HNSY94], with the difference that not any path of the semantic graph is considered, but only those where discrete transitions are taken infinitely often (i.e. runs).

$\text{ETCTL}^*_\exists$ is introduced in [BLY96].

A large number of other timed logics exist in the literature. For a survey, the reader is referred to [AH92].

Regarding the debate between linear and branching time [Lam80, Lam83, EL85, EH86], it seems to be slightly out-of-date, although there is still no consensus as to which view is better. For the reasons mentioned in the introduction, we believe that the two views are really complementary, therefore, both are necessary.

Part II

Analysis Techniques

Chapter 5

Abstractions for Timed Automata

The semantics of TA are given in terms of an infinite (dense) state-space. On the other hand, automatic verification methods require finite (but also reasonably-sized) state spaces. Consequently, analysis techniques for TA rely on *abstractions* of the infinite semantic graph to a finite domain. Apart from reducing the state space, an abstraction also leads to loss of information. The stronger the abstraction, the less the information lost but also the less important the state-space reduction. The crucial question then is to what extent to abstract in order to preserve properties of interest and at the same time keep automatic analysis feasible.

We first recall some generalities on abstractions and introduce them in the timed context.

Abstractions for TA. In the context of this thesis, an *abstraction* is a relation between the *concrete* state space of a TA and an *abstract* space. The concrete space is the semantic graph. The states forming the abstract space are sets of concrete states, called *symbolic states*. We consider abstractions based on:

- *Bisimulations*: the abstract states form a partition of the concrete states, that is, the abstraction is a function. We define a strong and two weak *time-abstracting bisimulations*, where exact delays in time transitions are abstracted away. We also recall the *region equivalence* of [ACD93] and show that it is a strong time-abstracting bisimulation. We prove that all time-abstracting bisimulations preserve TBA emptiness, and that the strong one also preserves CTL (by extension TCTL). These results are modulo non-zenoness, which can be characterized syntactically in symbolic paths.

- *Simulations*: here, the abstract states might be overlapping, that is, the abstraction is a relation. We define the *simulation graph*, where time transitions are eliminated altogether. On top of the simulation graph, we define three weaker abstractions, based on *clock activity*, *symbolic state inclusion* and *convex hull*. These abstractions can be also combined to yield better reduction. The simulation graph, possibly with activity, preserves TBA emptiness. Inclusion preserves TBA emptiness in a conservative way and reachability in an exact way. Convex hull preserves TBA emptiness and reachability conservatively.

All abstractions yield in practice much smaller state spaces than the region graph. The advantage of bisimulations is that they preserve more properties. On the other hand, they have to be computed a-priori, before verification can be applied, as shown in chapter 6. The advantage of the simulations is that they can be computed during the verification of the property, yielding on-the-fly techniques (chapter 7).

Before presenting the abstractions and the preservation results, we define formally symbolic states and their semantic successor and predecessor operations.

Symbolic states and operations

Consider a TA A. A set of states of A is called a *symbolic state*.

Let S be a symbolic state and e an edge of A. We define the following operations on S:

$$\begin{aligned}
\mathsf{time\text{-}succ}(S) &\stackrel{\text{def}}{=} \{s \mid \exists\, s' \in S, \delta \in \mathsf{R} \,.\, s' \xrightarrow{\delta} s\} \\
\mathsf{time\text{-}pred}(S) &\stackrel{\text{def}}{=} \{s \mid \exists\, s' \in S, \delta \in \mathsf{R} \,.\, s \xrightarrow{\delta} s'\} \\
\mathsf{disc\text{-}succ}(e, S) &\stackrel{\text{def}}{=} \{s \mid \exists\, s' \in S \,.\, s' \xrightarrow{e} s\} \\
\mathsf{disc\text{-}pred}(e, S) &\stackrel{\text{def}}{=} \{s \mid \exists\, s' \in S \,.\, s \xrightarrow{e} s'\}
\end{aligned}$$

In words, $\mathsf{time\text{-}succ}(S)$ is the set of all time-successors of states in S and $\mathsf{disc\text{-}succ}(e, S)$ are the e-successors of S. The meaning of $\mathsf{time\text{-}pred}()$ and $\mathsf{disc\text{-}pred}()$ is symmetrical.

A *zone* is a symbolic state S such that:

1. all states of S are associated with the same discrete state, i.e., for all $s, s' \in S$, $\mathsf{discrete}(s) = \mathsf{discrete}(s')$; and
2. the set of valuations $\{\mathbf{v} \mid \exists (q, \mathbf{v}) \in S\}$ is a convex $\mathcal{X}$-polyhedron ζ.

We often write (q, ζ) for the zone S. Also, we use false to denote the empty zone.

Let S_1 be a zone, S_2 a symbolic state, e an edge and c a natural constant. We define the following successor and predecessor operations:

$$\begin{aligned}
\mathsf{post}(e, S_1, c) &\stackrel{\text{def}}{=} \mathsf{close}\Big(\mathsf{time\text{-}succ}\Big(\mathsf{disc\text{-}succ}(e, S_1)\Big), c\Big) \\
\mathsf{pre}(e, S_2) &\stackrel{\text{def}}{=} \mathsf{disc\text{-}pred}\Big(e, \mathsf{time\text{-}pred}(S_2)\Big)
\end{aligned}$$

where we write $\mathsf{close}((q, \zeta), c)$ instead of $(q, \mathsf{close}(\zeta, c))$. Intuitively, $\mathsf{post}(e, S_1, c)$ contains all states (and their c-equivalents) that can be reached from some state in S_1, by taking an e-transition, then letting some time pass; $\mathsf{pre}(e, S_2)$ contains all states that can reach some state in S_2 by taking an e-transition, then letting some time pass.

The following result says that zones are preserved by the above successor and predecessor operations.

Lemma 5.1 *If S is a zone, then* $\mathsf{time\text{-}succ}(S)$, $\mathsf{time\text{-}pred}(S)$, $\mathsf{disc\text{-}succ}(e, S)$, $\mathsf{disc\text{-}pred}(e, S)$, $\mathsf{post}(e, S, c)$ *and* $\mathsf{pre}(e, S)$ *are also zones.*

Proof: Let $S = (q, \zeta)$. Using the definitions of polyhedral operations (section 2.2.2), it is easy

to prove the following equalities:

$$
\begin{aligned}
\zeta[Y := 0] &= (\zeta/_{\overline{Y}}) \cap (\bigwedge_{x \in Y} x = 0) \\
[Y := 0]\zeta &= (\zeta \cap (\bigwedge_{x \in Y} x = 0))/_{\overline{Y}} \\
\mathsf{time\text{-}succ}(q, S) &= \Big(q, \nearrow \zeta \cap \mathsf{invar}(q)\Big) \\
\mathsf{time\text{-}pred}(q, S) &= \Big(q, \swarrow \zeta \cap \mathsf{invar}(q)\Big) \\
\mathsf{disc\text{-}succ}(e, S) &= \begin{cases} \Big(q', \big((\zeta \cap \zeta_e)[X := 0]\big) \cap \mathsf{invar}(q')\Big), & \text{if } e = (q, \zeta_e, \_, X, q') \\ \emptyset, & \text{otherwise} \end{cases} \\
\mathsf{disc\text{-}pred}(e, S) &= \begin{cases} \Big(q', \zeta_e \cap ([X := 0]\zeta)\Big), & \text{if } e = (q', \zeta_e, \_, X, q) \\ \emptyset, & \text{otherwise} \end{cases}
\end{aligned}
$$

The result follows from the fact that polyhedral operations preserve convexity (lemma 2.3) and $\mathsf{close}(\zeta, c)$ is by definition convex. ■

5.1 Time-abstracting bisimulations

Time-abstracting bisimulations are equivalences which abstract away from the quantitative aspect of time: we know that *some* time passes, but not how much.

Before giving the formal definition, we give the intuition through an example. Consider the two systems shown in figure 5.1. The TA A on the left of the figure has two discrete transitions to states satisfying propositions p_1 and p_2 respectively. The first transition is possible immediately and remains possible for one time unit, while the second is possible only after two time units and remains possible forever. The graph G on the right of the figure describes as system which can either move to p_1, or wait some time (modeled by the transition labeled τ) and go to a state where no discrete transition is possible. Then, after some more time, it moves to a state from which it can go to p_2.

The two systems can be considered equivalent modulo statements of the form: "p_1 can be reached immediately while p_2 can be reached only after letting some time pass and meanwhile there is a point when no action is possible". This statement does not impose any exact quantitative timing requirements, apart from "letting some time pass". On the other hand, it imposes conditions on discrete-state changes. This is the idea behind time-abstracting equivalences: exact delays are abstracted away while information on the discrete-state changes of the system is retained. We formalize this in the sequel.

5.1.1 Definition

The Strong Time-Abstracting Bisimulation

Consider a TA A with set of edges E. A binary relation $\approx$ on the states of A is a *strong time-abstracting bisimulation* (STaB) if for all states $s_1 \approx s_2$, the following conditions hold:

1. if $s_1 \xrightarrow{e_1} s_3$, for some $e_1 \in E$, then there exists $e_2 \in E$ such that $s_2 \xrightarrow{e_2} s_4$ and $s_3 \approx s_4$;

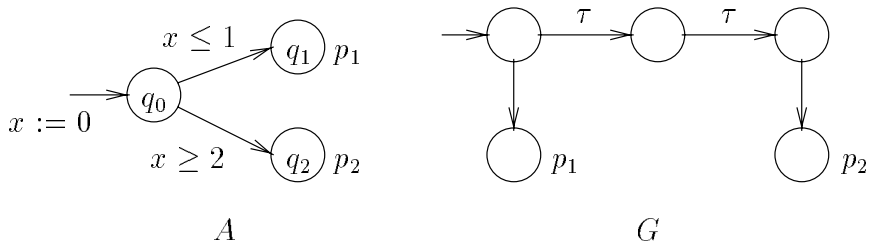


Figure 5.1: Two time-abstracting bisimilar systems.

2. if $s_1 \xrightarrow{\delta_1} s_3$ then there exists $\delta_2 \in \mathsf{R}$ such that $s_2 \xrightarrow{\delta_2} s_4$ and $s_3 \approx s_4$;

3. the above conditions also hold if the roles of s_1 and s_2 are reversed.

The definition is illustrated in figure 5.2 (left). The states s_1 and s_2 are said to be STa-bisimilar. In general, two TA A_1 and A_2 are said to be STa-bisimilar if there exists a STaB $\approx$ on the states of A_1 and A_2, such that $s_0^1 \approx s_0^2$, where s_0^i is the initial state of A_i.

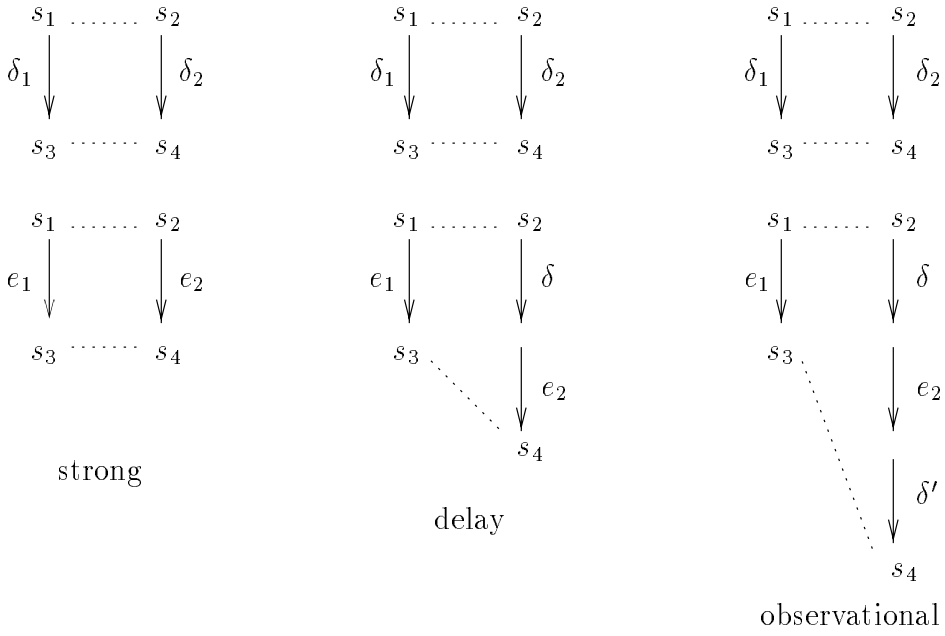


Figure 5.2: Time-abstracting bisimulations.

Consider again the example of figure 5.1, and a STaB respecting atomic propositions p_1 and p_2. The greatest such bisimulation, say $\approx$, induces five classes, namely, $(q_0, x \leq 1)$, $(q_0, 1 < x < 2)$, $(q_0, x \geq 2)$, (q_1, true) and (q_2, true). In fact, the graph G of figure 5.1 is essentially the $\approx$-quotient of A, where time edges are represented symbolically in a compact form (τ edges). This is explained in more detail below.

Time-Abstracting Quotient Graphs

According to the general definition of quotients (section 2.1), the STa-quotient of a TA A is a graph whose nodes are symbolic states (the classes induced by the STaB) and whose edges are

0:	(far,	up,	0,	true)
1:	(near,	up,	1,	$x \leq 1 \wedge z = 1$)
2:	(near,	up,	1,	$z < 1 \wedge x < y + 1 \wedge x \leq z$)
3:	(near,	coming down,	2,	$y < 1 \wedge x \leq y + 1 \wedge x < z + 2$)
4:	(near,	down,	2,	$2 < x \wedge x \leq 5$)
5:	(near,	down,	2,	$x \leq 2$)
6:	(in,	down,	2,	$x \leq 5$)
7:	(far,	down,	3,	$z \leq 1$)
8:	(far,	going up,	0,	$y = 1$)
9:	(far,	going up,	0,	$1 < y \leq 2$)
10:	(far,	going up,	0,	$y = 0$)
11:	(far,	going up,	0,	$0 < y < 1$)
12:	(near,	going up,	1,	$x \leq 1 \wedge 1 \leq y \wedge y \leq 2 \wedge z = 1$)
13:	(near,	going up,	1,	$1 \leq y \wedge z < 1 \wedge y < x + 2 \wedge x \leq z \wedge y \leq z + 1$)
14:	(near,	going up,	1,	$y \leq 2 \wedge x \leq z \wedge z + 1 < y$)
15:	(near,	going up,	1,	$y < 1 \wedge x \leq y \wedge z < x + 1 \wedge y = z$)
16:	(near,	going up,	1,	$y < 1 \wedge x \leq z \wedge z < y$)

Table 5.1: The nodes of the STa-quotient of figure 5.3.

of two types: $C_1 \stackrel{e}{\rightarrow} C_2$, for some edge e of A, when states in C_2 are e-successors of states in C_1; or $C_1 \stackrel{\delta}{\rightarrow} C_2$, for some $\delta \in \mathsf{R}$, when C_2 contains a δ-successor of some state in C_1.

To be used for algorithmic analysis, quotients must have a finite representation. Later we shall prove that the number of classes induced by a STaB is always finite, implying that the quotient has a finite number of nodes. As for the infinite sets of timed edges, they can be represented symbolically using a single edge labeled τ. For instance, all edges $(q_0, x \leq 1) \stackrel{\delta}{\rightarrow} (q_0, 1 < x < 2)$ in the example above are replaced by $(q_0, x \leq 1) \stackrel{\tau}{\rightarrow} (q_0, 1 < x < 2)$. Also, we eliminate τ-edges which can be obtained by reflexive, transitive closure. Thus, there is no edge $(q_0, x \leq 1) \stackrel{\tau}{\rightarrow} (q_0, x \leq 1)$, neither $(q_0, x \leq 1) \stackrel{\tau}{\rightarrow} (q_0, x \geq 2)$. These edges are omitted for reasons of economy, but most importantly, so that classical (untimed) verification techniques can be applied to quotient graphs without modification. The technique is explained in detail in section 6.2.

In the sequel we write $C_1 \rightarrow C_2$ for two classes C_1 and C_2 if either $C_1 \stackrel{\tau}{\rightarrow} C_2$ or $C_1 \stackrel{e}{\rightarrow} C_2$ for some edge e.

Example. The STa-quotient of the TGC system of section 3.1 is shown in figure 5.3. The nodes of the graph are detailed in table 5.1. The quotient has been generated using the minimization technique of section 6.1.2, implemented in the module `minim` (section 11.2). The graph has been drawn using the module `bcg_edit` of the CADP tool suite. In the CADP graph format, τ is denoted "i" (for "internal" or "invisible").

Weak Time-Abstracting Bisimulations

We now define two weaker time-abstracting bisimulations. The *time-abstracting delay bisimulation* or TadB (resp. *time-abstracting observational bisimulation* or TaoB) is a binary relation $\approx$ on the states of A, such that for all pairs $s_1 \approx s_2$, the following conditions hold:

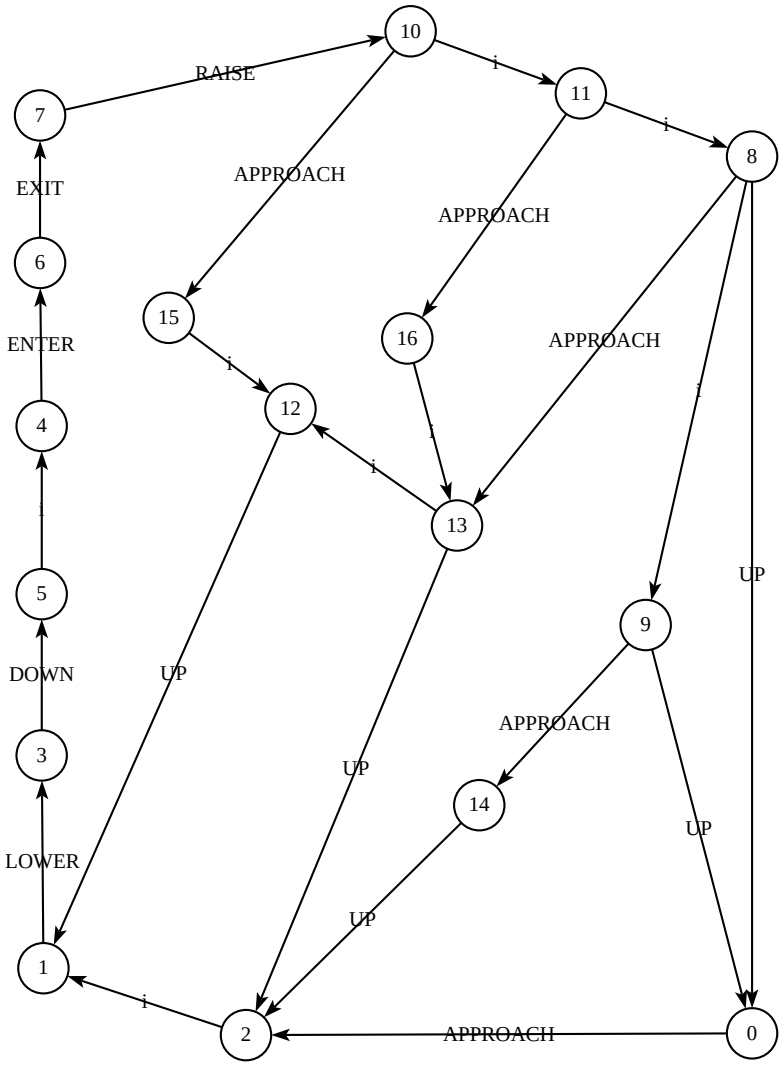


Figure 5.3: The STa-quotient graph of the Train–Gate–Controller example.

1. if $s_1 \stackrel{e_1}{\rightarrow} s_3$, for some $e_1 \in E$, then there exist $e_2 \in E$ and $\delta \in \mathsf{R}$ (resp. $\delta, \delta' \in \mathsf{R}$) such that $s_2 \stackrel{\delta}{\rightarrow}\stackrel{e_2}{\rightarrow} s_4$ (resp. $s_2 \stackrel{\delta}{\rightarrow}\stackrel{e'}{\rightarrow}\stackrel{\delta'}{\rightarrow} s_4$) and $s_3 \approx s_4$;

2. if $s_1 \stackrel{\delta_1}{\rightarrow} s_3$ then there exists $\delta_2 \in \mathsf{R}$ such that $s_2 \stackrel{\delta_2}{\rightarrow} s_4$ and $s_3 \approx s_4$;

3. the above conditions also hold if the roles of s_1 and s_2 are reversed.

The definitions are illustrated in figure 5.2 (middle and right). The notions of bisimilar states or TA and quotient graphs are straightforward to extend to weak TaBs. The quotient graph of the TA of figure 5.1 modulo the greatest TadB or TaoB respecting p_1 and p_2 is shown in figure 5.4. The induced classes are $(q_0, x \leq 1)$, $(q_0, x > 1)$, (q_1, true) and (q_2, true). Notice that states $(q_0, 1 < x < 2)$ are bisimilar to states $(q_0, x \geq 2)$ since they can let time pass and take a discrete transition to q_2.

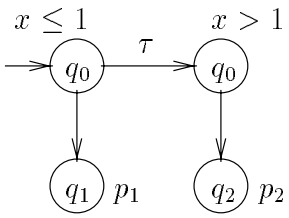


Figure 5.4: Tad- and Tao-quotient graph of the TA of figure 5.1.

Example. The Tao-quotient of the TGC system of section 3.1 is shown in figure 5.5. In fact, the graph has been obtained from the STa-quotient of figure 5.3, further reduced with respect to the untimed observation bisimulation. This is explained in detail in what follows.

Obtaining the weak TaBs from the strong TaB

Weak TaBs correspond to the composition of strong TaB with the (untimed) delay and observational bisimulations defined in section 2.1. More precisely, consider a TA A and its semantic graph G. Let $\approx_{ta}$ be a strong time-abstracting bisimulation on G. Also let G_τ be the graph obtained by G by replacing all labels $\delta \in \mathsf{R}$ on the time transitions of G by τ. Let $\approx_{delay}$ (resp. $\approx_{obs}$) be the greatest delay (resp. observational) bisimulation on G_τ.

Lemma 5.2 *$\approx_{delay} \circ \approx_{ta}$ (resp. $\approx_{obs} \circ \approx_{ta}$) is a time-abstracting delay (resp. observational) bisimulation on G.*

Proof: We only prove the result for $\approx_{delay} \circ \approx_{ta}$. The proof for $\approx_{obs} \circ \approx_{ta}$ is similar. For simplicity, we write $s \rightarrow s'$ if $s \stackrel{e}{\rightarrow} s'$ for some edge e.

Let $(s_1, s_2) \in \approx_{delay} \circ \approx_{ta}$, i.e., $s_1 \approx_{delay} s$ and $s \approx_{ta} s_2$, for some state s. Now, assume that $s_1 \rightarrow s_1'$. From the fact that $s_1 \approx_{delay} s$, there exist $s \stackrel{\tau^*}{\rightarrow} s'' \rightarrow s'$ such that $s_1' \approx_{delay} s'$. Observe that $s \stackrel{\tau^*}{\rightarrow} s''$ implies $s \stackrel{\delta}{\rightarrow} s''$, for some $\delta \in \mathsf{R}$ (this is by definition of the graph G_τ). From the fact that $s \approx_{ta} s_2$, there exist $s_2 \stackrel{\delta_2}{\rightarrow} s_2''$ such that $s'' \approx_{ta} s_2''$. Thus, there exist $s_2'' \rightarrow s_2'$ such that $s' \approx_{ta} s_2'$. Summarizing, we have $s_2 \stackrel{\delta_2}{\rightarrow}\rightarrow s_2'$ such that $(s_1', s_2') \in \approx_{delay} \circ \approx_{ta}$. The case of time transitions is similar.

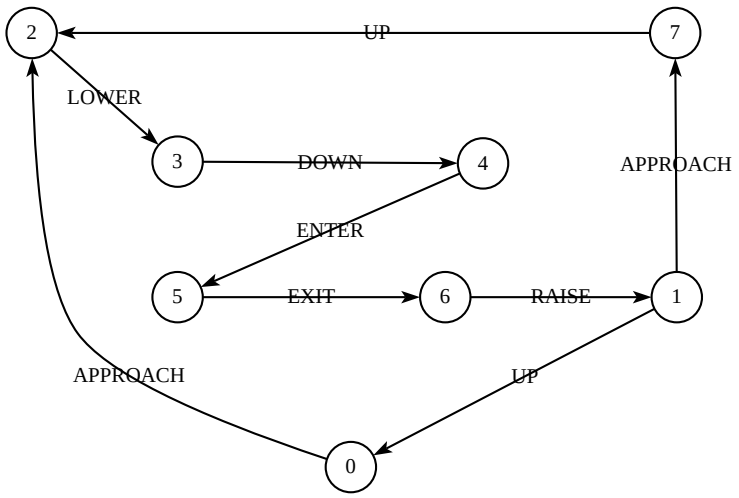


Figure 5.5: The Tao-quotient graph of the Train–Gate–Controller example.

Inversely, assume that $s_2 \xrightarrow{\delta_2} s_2'$. From the fact that $s \approx_{ta} s_2$, there exist $s \xrightarrow{\delta} s'$ such that $s' \approx_{ta} s_2'$. Observe that $s \xrightarrow{\delta} s'$ implies $s \xrightarrow{\tau} s'$. From the fact that $s_1 \approx_{delay} s$, there exist $s_1 \xrightarrow{\tau^*} s_1'$ such that $s_1' \approx_{delay} s'$. Now, $s_1 \xrightarrow{\tau^*} s_1'$ implies $s_1 \xrightarrow{\delta_1} s_1'$, for some $\delta_1 \in \mathsf{R}$, and we have $(s_1', s_2') \in \approx_{delay} \circ \approx_{ta}$. The case of discrete transitions is similar. This completes the proof. ■

The above result is used in section 6.2, where we show how to compute the weak Ta-quotient of a TA from its STa-quotient and how to compare two TA with respect to weak TaBs by comparing their STa-quotients with respect to weak untimed bisimulations.

Comparison of the three TaBs

Given a TA A, let $\approx_{tao}$, $\approx_{tad}$ and $\approx_{ta}$ be the greatest TaoB, TadB and STaB on A, respectively. By definition, $\approx_{ta} \subseteq \approx_{tad} \subseteq \approx_{tao}$, that is, $\approx_{ta}$ is stronger than $\approx_{tad}$ which is in turn stronger than $\approx_{tao}$. We now show that the above inclusions are strict. The example of figures 5.1 and 5.4 shows that $\approx_{tad} \not\subseteq \approx_{ta}$.

To see that $\approx_{tao} \not\subseteq \approx_{tad}$, consider the TA of figure 5.6 and assume that $\approx_{ta}$ and $\approx_{tao}$ respect propositions p_1 and p_2. First, observe that both $\approx_{tad}$ and $\approx_{tao}$ distinguish states $(q_2, x \leq 1)$ and $(q_2, x > 1)$. Now, $\approx_{tad}$ distinguishes states (q_1, true) and $(q_1', x > 1)$, since the latter can move to $(q_2, x > 1)$ by a discrete transition, whereas the former cannot. On the other hand, $\approx_{tao}$ does not distinguish these states, since (q_1, true) can more to $(q_2, x > 1)$ by taking the discrete transition and then delaying until $x > 1$.

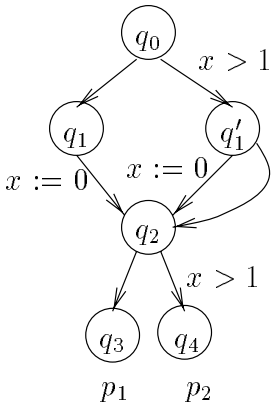


Figure 5.6: Example showing that TadB is strictly stronger than TaoB.

The Region equivalence: a strong time-abstracting bisimulation

The region equivalence has been introduced in [ACD93] in order to prove decidability of TA model checking. The equivalence has two important properties: first, it preserves all formalisms presented in the previous chapter; second, it induces a finite partition of the state space. Here we show that the region equivalence is a strong TaB. This implies in particular that the quotient of a TA A with respect to the greatest STaB, TadB or TaoB defined on A is finite.

Informally, two states $(q, \mathbf{v})$ and $(q, \mathbf{v}')$ are region equivalent if $\mathbf{v}$ and $\mathbf{v}'$ agree on the integral parts of all clock values and have the same ordering of the fractional parts of all pairs of clock values.

More formally, let $\lfloor \delta \rfloor$ (the integral part of δ) be the greatest integer smaller than δ, for $\delta \in \mathsf{R}$. Let $\langle \delta \rangle$ (the fractional part of δ) be $\delta \Leftrightarrow \lfloor \delta \rfloor$. Consider a TA A with set of clocks $\mathcal{X}$ and let $c \geq c_{max}(A)$. Two $\mathcal{X}$-valuations $\mathbf{v}$ and $\mathbf{v}'$ are *region equivalent*, denoted $\mathbf{v} \simeq_c \mathbf{v}'$ if they satisfy the following conditions:

1. For each clock x, either $\lfloor \mathbf{v}(x) \rfloor = \lfloor \mathbf{v}'(x) \rfloor$ or both $\mathbf{v}(x)$ and $\mathbf{v}'(x)$ are greater than c.
2. For all pairs of clocks x, y, either $\lfloor \mathbf{v}(x) \Leftrightarrow \mathbf{v}(y) \rfloor = \lfloor \mathbf{v}'(x) \Leftrightarrow \mathbf{v}'(y) \rfloor$ or both differences $\mathbf{v}(x) \Leftrightarrow \mathbf{v}(y)$ and $\mathbf{v}'(x) \Leftrightarrow \mathbf{v}'(y)$ are greater than c.

It can be checked that $\simeq_c$ is indeed an equivalence relation, independently of c. The equivalence classes induced by $\simeq_c$ are called *regions*. Part of the region space for two clocks x, y and $c = 2$ is shown in figure 5.7.

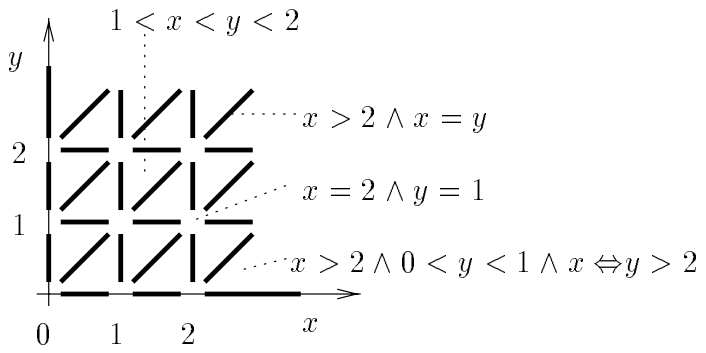


Figure 5.7: A partition of the clock space in 54 regions.

The region equivalence can be extended to states of A so that $(q, \mathbf{v})$ is equivalent to $(q', \mathbf{v}')$ if $q = q'$ and $\mathbf{v} \simeq_c \mathbf{v}'$. Then, we have the following result.

Lemma 5.3 *The region equivalence is a strong time-abstracting bisimulation.*

Proof: Let $(q, \mathbf{v}) \simeq_c (q, \mathbf{v}')$. Observe that:

1. for any c-closed $\mathcal{X}$-polyhedron ζ (in particular, any guard or invariant of A), $\mathbf{v} \in \zeta$ iff $\mathbf{v}' \in \zeta$;
2. for any set of clocks $X \subseteq \mathcal{X}$, $\mathbf{v}[X := 0] \simeq \mathbf{v}'[X := 0]$;
3. for any $\delta \geq 0$ there exists $\delta' \geq 0$ such that $\mathbf{v} + \delta \simeq \mathbf{v}' + \delta'$.

Now, if $(q, \mathbf{v}) \xrightarrow{e} (q_1, \mathbf{v}_1)$ is a discrete transition, then $(q, \mathbf{v}') \xrightarrow{e} (q_1, \mathbf{v}_1')$ is also a discrete transition, since both $\mathbf{v}$ and $\mathbf{v}'$ satisfy $\mathsf{guard}(e)$. Also, $(q_1, \mathbf{v}_1) \simeq (q_1, \mathbf{v}_1')$, since $\mathbf{v}[\mathsf{reset}(e) := 0] \simeq \mathbf{v}'[\mathsf{reset}(e) := 0]$. Let $(q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v} + \delta)$ be a time transition. There exists $\delta' \geq 0$ such that $\mathbf{v} + \delta \simeq \mathbf{v}' + \delta'$. $\mathbf{v}'$ and $\mathbf{v}' + \delta'$ satisfy $\mathsf{invar}(q)$, since $\mathbf{v}$ and $\mathbf{v} + \delta$ do. $\mathbf{v}' + \delta''$ satisfies $\mathsf{invar}(q)$ for any $\delta'' < \delta'$, by convexity of $\mathsf{invar}(q)$. Thus, $(q, \mathbf{v}') \xrightarrow{\delta'} (q, \mathbf{v}' + \delta')$ is also a time transition. ■

Using a combinatorial argument, [ACD93] have shown that the number of regions has the following upper bound:

$$n! \cdot 2^n \cdot (2c + 2)^n$$

where $n = |\mathcal{X}|$ is the number of clocks. In fact, the lower bound is on the same order of magnitude, which implies that the number of regions is too large for any practical purpose. For example, the TGC system of section 3.1 has a region space in the order of 10^4 regions per discrete state.

As a corollary of lemma 5.3 and the above upper bound, we conclude that the quotient of a TA with respect to the greatest STaB (thus, also TadB, TaoB) is finite.

c-equivalence: a strong time-abstracting bisimulation

The preservation results of section 5.2, and the correctness of the algorithms of chapters 7 and 8 depends on the following result.

Lemma 5.4 *Let A be a TA and $c \geq c_{max}(A)$. Then, c-equivalence is a strong time-abstracting bisimulation.*

Proof: Observe that c-equivalence is stronger than the region equivalence. The result follows by lemma 5.3. ■

The intuition is that the values of those clocks which have grown greater than $c_{max}(A)$ are not relevant, since there is no guard or invariant of A which can distinguish between such values. Therefore, adding $c_{max}(A)$-equivalent states to the set of reachable states of A would not affect the satisfaction of any property.

5.1.2 Properties preserved by time-abstracting bisimulations

In this section we show that all TaBs preserve linear-time properties, while only the strong TaB preserves branching-time properties. The results are modulo non-zenoness, that is, they hold only for strongly non-zeno systems. For the general case, we give necessary and sufficient conditions guaranteeing non-zenoness in quotient graphs. These conditions are "syntactic", that is, they apply on the structure of the nodes and edges of the graph, and resemble the conditions for *strong fairness* of [MP95b].

We start by presenting the fundamental property of Ta-quotient graphs.

Pre-stability. Pre-stability is a property of quotients induced by TaBs, similar to the one holding in quotients induced by untimed bisimulations (section 2.1). The difference is that there are two types of timed pre-stability, depending on whether a class is a discrete or time successor of another class. More precisely, consider a TA A, a TaB $\approx$ on A and two classes C_1 and C_2 in the $\approx$-quotient graph of A. Then, by definition:

- If $C_1 \xrightarrow{\tau} C_2$ then for each state $s_1 \in C_1$ there exists $s_2 \in C_2$ such that $s_1 \xrightarrow{\delta} s_2$, for some $\delta \in \mathsf{R}$.
- If $C_1 \xrightarrow{e} C_2$, for some edge e, then:
  - if $\approx$ is a STaB then for each state $s_1 \in C_1$ there exists $s_2 \in C_2$, such that $s_1 \xrightarrow{e} s_2$;
  - if $\approx$ is a TadB then for each state $s_1 \in C_1$ there exist $\delta \in \mathsf{R}$, $s_2 \in C_2$, such that $s_1 \xrightarrow{\delta}\xrightarrow{e} s_2$;

– if $\approx$ is a TaoB then for each state $s_1 \in C_1$ there exist $\delta_1, \delta_2 \in \mathsf{R}$, $s_2 \in C_2$, such that $s_1 \xrightarrow{\delta_1} \xrightarrow{e} \xrightarrow{\delta_2} s_2$.

The pre-stability property related to strong TaBs is illustrated in figure 5.8.

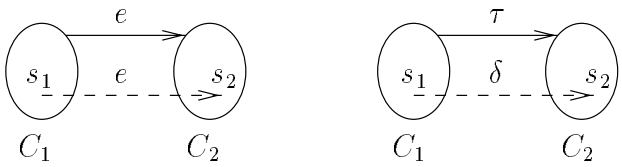


Figure 5.8: Pre-stability in strong time-abstracting bisimulations.

Non-zenoness. TaBs do not preserve non-zenoness. Figure 5.9 presents a counter-example: although TA A_1 and A_2 are STa-bisimilar, only A_1 has non-zeno runs. This is not surprising, since TaBs are insensitive to exact delays. However, we can still use the information contained in the equivalence classes induced by the bisimulation, as well as in the edges of the TA, to check whether there is a clock blocking time or not. This motivates the following definitions.

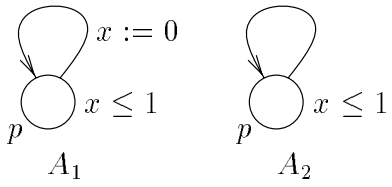


Figure 5.9: Time-abstracting bisimulations do not preserve non-zenoness.

First, we extend the predicate $\mathsf{unbounded}(x, S)$, for a clock x and a symbolic state S, as follows:

$$\mathsf{unbounded}(x, S) \stackrel{\text{def}}{=} \forall t \in \mathsf{R} \,.\, \exists (q, \mathbf{v}) \in S \,.\, \mathbf{v}(x) > t$$

Now, let $G_{\approx}$ be the quotient graph of a TA A with respect to a TaB $\approx$ and let $\pi = C_1 \to C_2 \to \cdots$ be an infinite path in $G_{\approx}$. π is called *non-zeno* if for each clock $x \in \mathcal{X}$:

- either x is reset infinitely often in π, that is, $\forall i \geq 1 \,.\, \exists j > i, e \in E \,.\, C_j \xrightarrow{e} C_{j+1} \wedge x \in \mathsf{reset}(e)$,
- or x remains *unbounded* in π from some point on, that is, $\exists i \geq 1 \,.\, \forall j > i \,.\, \mathsf{unbounded}(x, C_j)$.

As we prove below, non-zeno paths correspond to non-zeno runs, and vice versa.

Linear-time preservation. In this paragraph we show how runs are related to symbolic paths. Given a path $\pi = C_1 \to C_2 \to \cdots$ of $G_{\approx}$ and a run $\rho = s_1 \to s_2 \to \cdots$, we say that ρ *is inscribed in* π if for all $i \geq 1$:

- $s_i \in C_i$,

- if $C_i \xrightarrow{\tau} C_{i+1}$ then there exists $\delta > 0$ such that $s_i \xrightarrow{\delta} s_{i+1}$,
- if $C_i \xrightarrow{e} C_{i+1}$ then $s_i \xrightarrow{e} s_{i+1}$.

Lemma 5.5 *Every run (resp. non-zeno run) ρ is inscribed in a unique path (resp. non-zeno path) π in $G_\approx$. Inversely, if $\pi = C_1 \to C_2 \to \cdots$ is a path (resp. non-zeno path) in $G_\approx$ then for all $s_1 \in C_1$ there exists a run (resp. non-zeno run) ρ starting from s_1 and inscribed in π.*

Proof: A straightforward modification of the proof of lemma 3.35 of [Alu91]. ∎

This result will be used in section 6.2.1 to show how to perform TBA model checking on quotient graphs.

Branching-time preservation. Only the strong TaB preserves branching-time properties. For simplicity, we consider here the untimed fragment of TCTL, CTL. The extension for full TCTL is given in section 6.2.2.

We first need to prove an important property of STaB related to the passage of time. Consider a TA A and a STaB $\approx$ on A. Given a time transition of A, $s \xrightarrow{\delta} s+\delta$, and m different classes $C_1, ..., C_m$, we say that the transition *traverses* $C_1, ..., C_m$ if:

1. $s \in C_1$ and $s + \delta \in C_m$.
2. For all $0 < \delta' < \delta$, there exists $1 \leq i \leq m$ such that $s + \delta' \in C_i$.

Figure 5.10 presents an example of a time transition traversing three classes C_1, C_2, C_3.

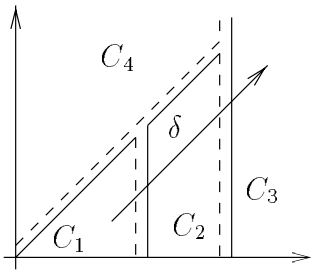


Figure 5.10: A time transition traversing classes C_1, C_2, C_3.

Lemma 5.6 *1. Any time transition traverses a unique (finite) number of classes.*

2. If $s \approx s'$ then for any time transition $s \xrightarrow{\delta} s+\delta$, there exists a time transition $s' \xrightarrow{\delta'} s'+\delta'$ such that $s + \delta \approx s' + \delta'$ and the two transitions traverse the same classes.

Proof: For the first part, consider a time transition $s \xrightarrow{\delta} s + \delta$ and let m be the number of different "points" $0 < \delta_1 < \cdots < \delta_m < \delta$ such that $s + \delta_i$ and $s + \delta_{i+1}$ belong to different classes (there is a finite number of such points since the quotient is finite). The proof is by induction on m. If $m = 0$, then $s, s + \delta \in C_1$, for some class C_1. We shall show that for all $0 < \delta_1 < \delta$, $s + \delta_1 \in C_1$. Assume the opposite, i.e., $s + \delta_1 \in C_2$ for some $C_2 \neq C_1$. Then, since $C_1 \xrightarrow{\tau} C_2$ (from the fact that $s \xrightarrow{\delta_1} s + \delta'$) and $C_2 \xrightarrow{\tau} C_1$ (from the fact that $s + \delta_1 \xrightarrow{\delta - \delta_1} s + \delta$), we can build

an infinite sequence $C_1 \xrightarrow{\tau} C_2 \xrightarrow{\tau} C_1 \xrightarrow{\tau} C_2 \cdots$. But this is not possible, since we assumed $\approx$ to be weaker than the region equivalence $\simeq_c$ and after the upper bound c all states are equivalent. The induction step is straightforward.

For part 2 of the lemma, let $C_1, ..., C_m$ be the classes traversed by $s \xrightarrow{\delta} s + \delta$. The proof is again by induction on m. If $m = 1$, then $s \approx s + \delta$ and it suffices to take $\delta' = 0$. For the sake of simplicity, instead of proving the general induction step, we assume that $m = 2$, that is, $s \xrightarrow{\delta} s + \delta$ traverses classes C_1, C_2. The extension to any $m > 1$ is easy using the induction hypothesis.

We have: $s, s' \in C_1$, $s + \delta, s + \delta' \in C_2$, for some δ'. We want to show that for all $\delta'_1 < \delta'$, $s' + \delta'_1 \in C_1 \cup C_2$. Assume this is not the case, that is, $s' + \delta'_1 \in C$ and C is different from C_1, C_2. Since $s' \approx s$, there exists $s \xrightarrow{\delta_1} s + \delta_1$ such that $s + \delta_1 \in C$. From the fact that $s \xrightarrow{\delta} s + \delta$ traverses C_1, C_2 and condition 2 of the definition of traversal, it must be that $\delta_1 > \delta$. Thus, $C_2 \xrightarrow{\tau} C$ (from the fact that $s + \delta \xrightarrow{\delta_1 - \delta} s + \delta_1$). On the other hand, $C \xrightarrow{\tau} C_2$ (from the fact that $s' + \delta'_1 \xrightarrow{\delta' - \delta'_1} s + \delta'$). As previously, we can build an infinite sequence $C_2 \xrightarrow{\tau} C \xrightarrow{\tau} C_2 \xrightarrow{\tau} C \cdots$, contradicting the hypotheses. ■

We are now ready to prove the main result, namely, CTL preservation.

Lemma 5.7 *Let A be a strongly non-zeno TA and $\approx$ be a strong time-abstracting bisimulation on A. For any CTL formula ϕ and any pair of states $s \approx s'$, $s \models \phi$ iff $s' \models \phi$.*

Proof: The proof is by induction on the syntax of ϕ. The basis comes directly from the hypothesis that $\approx$ respects P. The interesting induction steps are for $\phi = \forall\, \phi_1 \mathcal{U} \phi_2$ or $\phi = \exists\, \phi_1 \mathcal{U} \phi_2$. We only consider the latter case, the former being similar.

Assume that $s \models \exists\, \phi_1 \mathcal{U} \phi_2$. Then, there exists a non-zeno run $\rho = s \xrightarrow{\delta_1} \xrightarrow{e_1} \cdots$ and some point i along ρ such that $\rho(i) + \delta_i \models \phi_2$ and for all $j < i$, $\delta \leq \delta_j$, $\rho(j) + \delta \models \phi_1 \vee \phi_2$.

From the fact that $s \approx s'$, we can build a run $\rho' = s' \xrightarrow{\delta'_1} \xrightarrow{e'_1} \cdots$, such that $s_j \approx s'_j$ and $s_j + \delta_j \approx s'_j + \delta'_j$, for all j. From the strongly non-zeno hypothesis, ρ' is non-zeno. From the induction hypothesis, $\rho'(i) + \delta'_i \models \phi_2$ and for all $j < i$, $\rho'(j) + \delta'_j \models \phi_1 \vee \phi_2$. It remains to show that $\rho'(j) + \delta' \models \phi_1 \vee \phi_2$, for all $\delta' \leq \delta'_j$. By lemma 5.6, for any $\delta' \leq \delta'_j$, there exists $\delta \leq \delta_j$ such that $s'_j + \delta' \approx s_j + \delta$. The result follows from the induction hypothesis. ■

The assumption for strong non-zenoness is indispensable, as can be seen by the example of figure 5.9, where the two TA are bisimilar, however, only A_1 satisfies the CTL formula $\exists \Box\, p$.

Regarding the two weaker time-abstracting bisimulations, they do not generally preserve CTL. Figure 5.11 gives a counter-example. The TA shown in the top-left part of the figure yields the Tad-quotient shown in the bottom. The CTL formula $\exists\, (\exists \Diamond\, p_1) \mathcal{U}\, p_2$ is satisfied at state $s_2 = (q_1, x = 2.5, y = 0.7)$ (in fact, at all states $(q_1, x \Leftrightarrow y > 1 \wedge y \leq 2)$) but not at state $s_1 = (q_1, x = 1.5, y = 0.7)$ (in fact, at no state $(q_1, 0 \leq x \Leftrightarrow y \leq 1 \wedge y \leq 2)$) although s_2 and s_1 are Tad-bisimilar. As a corollary, CTL is not preserved by TaoB either, since TaoB is weaker than TadB.

5.2 Abstractions based on simulations

In these abstractions, only information about discrete transitions is kept, while time transitions are completely ignored. Intuitively, every abstract state S is "closed" under the passage of time,

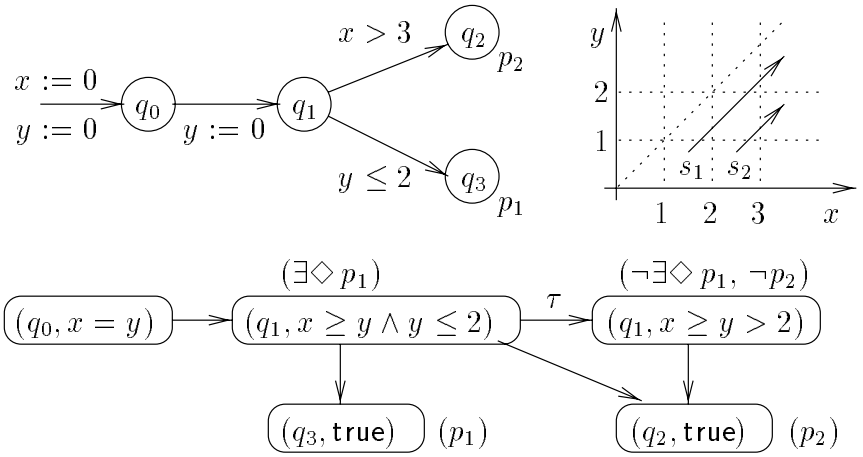


Figure 5.11: Weak time-abstracting bisimulations do not preserve CTL.

that is, if a concrete state s belongs to S then so do all the time successors of s.

Simulation abstractions are based on the notion of *simulation graph*, a forward-reachability graph where nodes are zones and successor nodes are obtained using the $\mathsf{post}()$ operator.

5.2.1 The Simulation Graph

Consider a TA A. In the rest of this section, we assume that c is a natural constant greater or equal to $c_{max}(A)$.

The *simulation graph* of A with respect to c, denoted $SG(A,c)$, is the smallest graph (in terms of nodes and set inclusion among nodes) such that:

1. $\mathsf{time\text{-}succ}(S_0)$ is a node of $SG(A,c)$, where $S_0 = (q_0, \mathsf{zero})$;
2. if S is a node of $SG(A,c)$ and e is an edge of A, then $S' = \mathsf{post}(e,S,c)$ is a node of $SG(A,c)$ and $S \xrightarrow{e} S'$ is an edge of $SG(A,c)$.

The following lemma shows that the above definition yields a finite graph.

Lemma 5.8 *For any TA A and any constant c, $SG(A,c)$ is finite.*

Proof: By definition, all nodes in $SG(A,c)$ are c-closed. The result is then obtained by lemma 2.2 and the fact that the discrete states and edges of A are finite. ∎

In the worst case, the nodes of the simulation graph can be as many as the classes induced by the region equivalence. In practice, however, the size of the simulation graph is orders of magnitude less. For example, the simulation graph of the TGC system (section 3.1) is shown in figure 5.12. The nodes are detailed in table 5.2. The simulation graph has 11 zones, made up of 8 distinct discrete-state vectors and 11 distinct polyhedra. The number of regions for the same system would be in the order of 10^4 per discrete-state vector (although not all of them would be reachable).

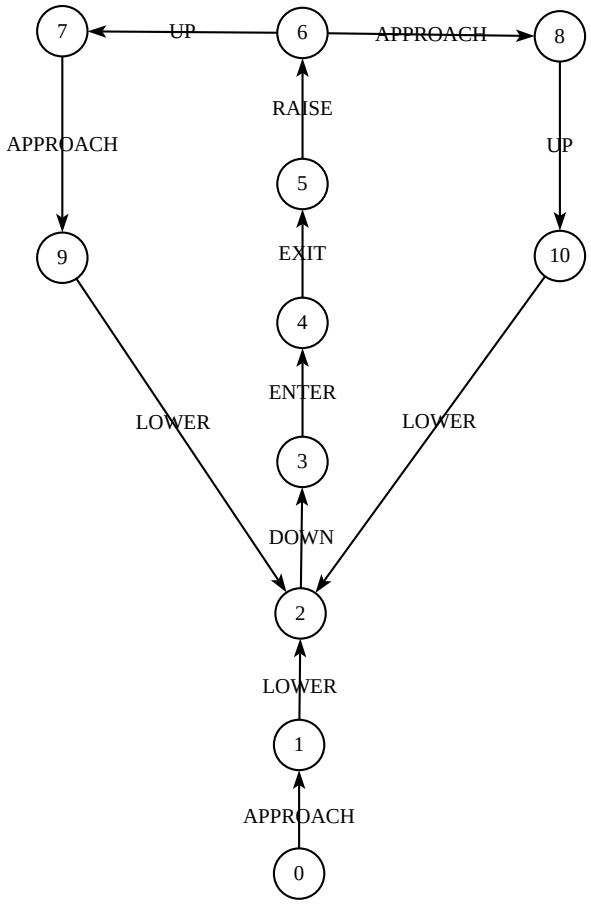


Figure 5.12: The simulation graph of the Train–Gate–Controller example.

0:	(far,	up,	0,	$x = y = z)$
1:	(near,	up,	1,	$x = z \leq 1 \wedge x \leq y)$
2:	(near,	coming down,	2,	$1 \geq x = z = y + 1 \wedge y < 1)$
3:	(near,	down,	2,	$1 \geq x = z \leq 5 \wedge x = y + 1)$
4:	(in,	down,	2,	$2 < x = z \leq 5 \wedge x = y + 1)$
5:	(far,	down,	3,	$z \leq 1 \wedge z + 2 < x \leq z + 5 \wedge x = y + 1)$
6:	(far,	going up,	0,	$y \leq 2 \wedge z + 2 < x \leq z + 5 \wedge y \leq z \leq y + 1)$
7:	(far,	up,	0,	$1 \leq y \wedge z + 2 < x \leq z + 5 \wedge y \leq z \leq y + 1)$
8:	(near,	going up,	1,	$x = z \leq 1 \wedge x \leq y \leq 2)$
9:	(near,	up,	1,	$x = z \leq 1 \wedge x + 1 \leq y)$
10:	(near,	up,	1,	$x = z \leq 1 \wedge x \leq y \leq x + 2)$

Table 5.2: The nodes of the simulation graph of figure 5.12.

5.2.2 Properties preserved in the simulation graph

The simulation graph preserves linear-time properties. As in the case of TaBs, non-zenoness is not preserved, however, we give necessary and sufficient syntactic conditions for the existence of non-zeno runs in paths of the simulation graph.

We start by presenting the fundamental property of the simulation graph.

Post-stability. Consider a TA A and an edge $S_1 \xrightarrow{e} S_2$ in the simulation graph $SG(A, c)$. By definition of $\mathsf{post}()$, we have the following two *post-stability* properties in the simulation graph:

- for any state $s_1 \in S_1$, if $s_1 \xrightarrow{e}\xrightarrow{\delta} s_2$, then $s_2 \in S_2$;
- for any state $s_2' \in S_2$, there exist $s_1 \in S_1$, $\delta \in \mathsf{R}$ and $s_2 \in S_2$ such that $s_1 \xrightarrow{e}\xrightarrow{\delta} s_2$, $s_2 \Leftrightarrow \delta \in S_2$, and s_2, s_2' are c-equivalent.

The properties are illustrated in figure 5.13. The arrow $\xrightarrow{e}$ drawn in solid line corresponds to the symbolic edge $S_1 \xrightarrow{e} S_2$. The arrows drawn in dashed lines correspond to semantic discrete or timed transitions.

Notice that, due to the second property above, simulation-graph post-stability is stronger than general post-stability (section 2.1). Also notice that pre-stability is not a simulation-graph property: indeed, there might be states in S_1 which have no e-successors at all (states $(q, \mathbf{v})$ where $\mathbf{v} \notin \mathsf{guard}(e)$).

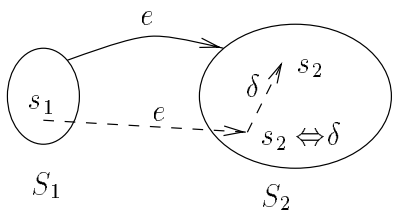


Figure 5.13: Post-stability in the simulation graph.

Non-zeno paths. The example of figure 5.9 can be re-used for showing that the simulation graph does not preserve non-zenoness: although TA A_1 and A_2 generate the same simulation graph, only A_1 has non-zeno runs. As in the case of TaBs, we give a syntactic definition of non-zenoness. Call a path in the simulation graph a *zone path*. An infinite zone path $\pi = S_1 \stackrel{e_1}{\rightarrow} S_2 \stackrel{e_2}{\rightarrow} \cdots$ is called non-zeno if for each clock $x \in \mathcal{X}$:

- either x is reset infinitely often in π, that is, $\forall i \,.\, \exists j > i \,.\, x \in \mathsf{reset}(e_j)$,
- or x remains unbounded in π from some point on, that is, $\exists i \,.\, \forall j > i \,.\, \mathsf{unbounded}(x, \zeta_j) \wedge \mathsf{unbounded}(x, \mathsf{guard}(e_j))$.

Linear-time preservation. Runs are related to zone paths in a similar way as runs are related to paths in quotient graphs (lemma 5.5). Consider a (finite or infinite) zone path $\pi = (q_1, \zeta_1) \stackrel{e_1}{\rightarrow} (q_2, \zeta_2) \stackrel{e_2}{\rightarrow} \cdots$. A run $\rho = (q_1, \mathbf{v}_1) \stackrel{\delta_1}{\rightarrow} (q_1, \mathbf{v}'_1) \stackrel{e_1}{\rightarrow} (q_2, \mathbf{v}_2) \stackrel{\delta_2}{\rightarrow} (q_2, \mathbf{v}'_2) \stackrel{e_2}{\rightarrow} \cdots$ is said to be inscribed in π if for all $i = 1, 2, ...$, $\mathbf{v}_i, \mathbf{v}'_i \in \zeta_i$.

Lemma 5.9 *Every run (resp. non-zeno run) of A is inscribed in a unique path (resp. non-zeno path) in $SG(A, c)$. Inversely, for every path (resp. non-zeno path) π in $SG(A, c)$, there is a run (resp. non-zeno run) inscribed in π.*

Sketch of proof: The idea behind the proof is illustrated in figure 5.14. The proof shows a zone path $S_1 \rightarrow (S_2 \rightarrow S_3 \rightarrow)^\omega$, ending in a cycle (zones are depicted as large ellipses). Each zone can be seen as a set of regions (depicted as small circles). When two zones are connected by an edge, say, $S_2 \rightarrow S_3$, this means that some region in S_2 has a discrete successor region in S_3. Also, some regions in a zone might have time successors in the same zone. By the post-stability property of the simulation graph, starting from any region in any zone and following edges backwards, we inevitably find a cycle of regions (for instance, trying to move backwards starting from the upper-most region of S_3, we find a cycle visiting the two lower-most regions of S_2 and S_3). A cycle of regions implies the existence of an infinite run, according to lemma 5.5. The complete proof is given in the appendix. ■

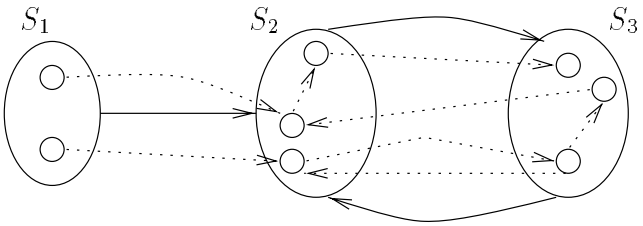


Figure 5.14: Why in every zone cycle there is an inscribed infinite run.

The main difference of the above result with the one of lemma 5.5 is that, given a node S in a zone path π, there exists a run inscribed in π starting from *some* states in S, but not necessarily all of them.

Lemma 5.9 implies the existence of two simulations between the semantic graph of a TA A and its simulation graph, one for each direction. The relation $\in$ between states of A and nodes of $SG(A, c)$ is a simulation in the sense that for each run ρ starting from a state s, there exists

a zone S such that $s \in S$ and ρ is inscribed in a path starting from S. The inverse relation is also a simulation, in the sense that for each path π starting from a zone S, there exists a state $s \in S$ and a ρ starting from s, such that ρ is inscribed in π.

These results will be used in chapter 7 for performing on-the-fly different types of analysis, namely, reachability, deadlock and timelock detection, and TBA and $\text{ETCTL}^*_\exists$ model checking.

Branching-time non-preservation. We use the same counter-example for weak TaBs (the TA of figure 5.11) to show that CTL is not preserved by the simulation graph. Recall that states $(q_1, x \Leftrightarrow y > 1 \wedge y \leq 2)$ satisfy the formula $\exists\,(\exists\Diamond\, p_1)\,\mathcal{U}\, p_2$ while states $(q_1, 0 \leq x \Leftrightarrow y \leq 1 \wedge y \leq 2)$ do not. However, all these states are "merged" in a single node (q_1, true) in the simulation graph, shown in figure 5.15.

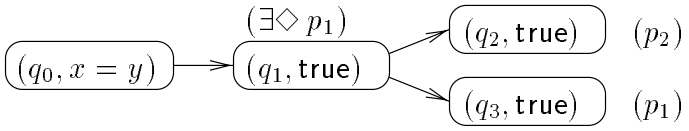


Figure 5.15: The simulation graph of the TA of figure 5.11.

5.2.3 Clock Activity

This abstraction considers clocks only when they are *usefully* counting time (we say that they are *active*). Intuitively, a clock is active from the point where it is reset up to all points where it is tested (in a guard or invariant of the TA), without being reset meanwhile. Inactive clocks do not affect the behavior of the TA, thus, they can be ignored.

More formally, consider a TA $A = (\mathcal{X}, Q, E, q_0, \mathsf{invar})$, where $\mathcal{X} = \{x_1, ..., x_n\}$. Given a discrete state $q \in Q$, the set of clocks *tested in* q, $\mathsf{clocks}(q) \subseteq \mathcal{X}$, is defined to be the set of clocks x such that x is constrained either in $\mathsf{invar}(q)$ or in $\mathsf{guard}(e)$, for some edge $e \in \mathsf{out}(q)$.

The function $\mathtt{act} : Q \mapsto 2^{\mathcal{X}}$, associating with each location q the set of active clocks in q, is defined as the least fix-point of the following system of equations (one equation for each location q):

$$\mathtt{act}(q) \;=\; \mathsf{clocks}(q) \;\cup \bigcup_{(q,\_,\_,\_,q') \in E} \mathtt{act}(q') \setminus X$$

Intuitively, x is active in q iff it is either tested in q or it is active in a discrete state q' which can be reached from q by a sequence of edges, so that x is never reset along the sequence.

As an example, consider the Train–Gate–Controller system shown in figure 3.1. The activity functions for each of the three TA of the system are as follows:

Train:	$\mathtt{act}(\text{far}) = \{\}$
	$\mathtt{act}(\text{near}) = \mathtt{act}(\text{in}) = \{x\}$
Gate:	$\mathtt{act}(\text{up}) = \mathtt{act}(\text{down}) = \{\}$
	$\mathtt{act}(\text{going up}) = \mathtt{act}(\text{coming down}) = \{y\}$
Controller:	$\mathtt{act}(0) = \mathtt{act}(2) = \{\}$
	$\mathtt{act}(1) = \mathtt{act}(3) = \{z\}$

0:	(far,	up,	0,	)
1:	(near,	up,	1,	$x = z \leq 1$)
2:	(near,	coming down,	2,	$1 \geq x = y + 1 \wedge y < 1$)
3:	(near,	down,	2,	$1 \geq x \leq 5$)
4:	(in,	down,	2,	$2 < x \leq 5$)
5:	(far,	down,	3,	$z \leq 1$)
6:	(far,	going up,	0,	$y \leq 2$)
7:	(near,	going up,	1,	$x = z \leq 1 \wedge x \leq y \leq x + 2$)

Table 5.3: The zones of the activity graph of figure 5.16.

It is interesting to see that none of the clocks is active all the time. For instance, clock y of the gate serves only at states "going up" or "coming down", where it is necessary to count the time needed for these operations.

An algorithm to compute $\mathtt{act}$ is given in [DY96]. This algorithm works on the syntactic structure of the automaton (i.e., discrete states and edges) and can be used compositionally to compute the active clocks of the parallel composition of two or more TA. More precisely, let A_1 and A_2 be two TA with disjoint sets of clocks $\mathcal{X}_1$ and $\mathcal{X}_2$, and sets of discrete states Q_1, Q_2, respectively. Let $\mathtt{act}_i : Q_i \mapsto 2^{\mathcal{X}_i}$, be the activity function of A_i, for $i = 1, 2$. Then, it is easy to prove that the activity function of $A_1 \| A_2$ is the function $\mathtt{act} : Q_1 \times Q_2 \mapsto 2^{\mathcal{X}_1 \cup \mathcal{X}_2}$, defined as follows:

$$\mathtt{act}(q_1, q_2) = \mathtt{act}_1(q_1) \cup \mathtt{act}_2(q_2)$$

Consider a TA A with set of discrete states Q and set of clocks $\mathcal{X}$. Given an activity function $\mathtt{act} : Q \mapsto 2^{\mathcal{X}}$, the *activity abstraction* with respect to $\mathtt{act}$ is defined to be the function α_{act} mapping each zone (q, ζ) in the simulation graph of A to the zone $(q, \zeta\rfloor_{\mathtt{act}(q)})$. That is, all inactive clocks are projected away, so that the dimension of the polyhedron associated to q is reduced from $\mathcal{X}$ to $\mathtt{act}(q)$.

The *activity graph* of A with respect to α_{act}, denoted $AG(A, c)$, is defined as follows:

- For each node S of $SG(A, c)$, $\alpha_{act}(S)$ is a node of $AG(A, c)$.
- For each edge $S_1 \xrightarrow{e} S_2$ of $SG(A, c)$, $\alpha_{act}(S_1) \xrightarrow{e} \alpha_{act}(S_2)$ is an edge of $AG(A, c)$.

The activity graph of the TGC system is shown in figure 5.16 (notice that it is smaller than the simulation graph of the same system). The zones are shown in table 5.3. Observe that no polyhedron is associated to node 0, since no clocks are active in the initial discrete-state vector.

Properties preserved in the activity graph

The activity graph preserves the same properties as the simulation graph. The only slight difference is in the definitions of post-stability, non-zeno symbolic paths, and inscription of runs to paths, due to the fact that the clock space induced by activity has variable dimension. We make explicit these differences in what follows.

Post-stability. Let A be a TA with set of clocks $\mathcal{X}$. Consider an edge $S_1 \xrightarrow{e} S_2$ in $AG(A, c)$, where $S_i = (q_i, \zeta_i)$, $i = 1, 2$ (recall that ζ_i is a polyhedron on $\mathtt{act}(q_i)$). Post-stability here is expressed as follows:

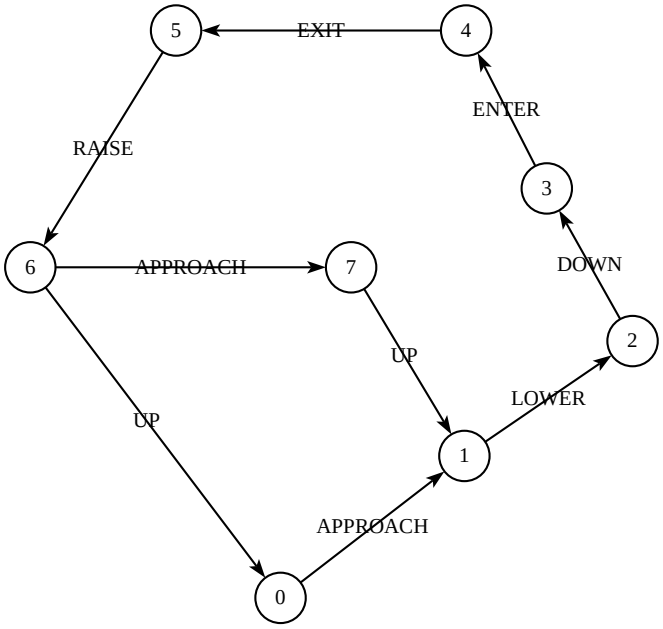


Figure 5.16: The activity graph of the Train–Gate–Controller example.

- for any state $\mathbf{v}_1 \in \zeta_1$, if there exist $\mathcal{X}$-valuations $\mathbf{v}'_1, \mathbf{v}'_2$ such that $(q_1, \mathbf{v}'_1) \xrightarrow{e} \xrightarrow{\delta} (q_2, \mathbf{v}'_2)$ and $\mathbf{v}_i = \mathbf{v}'_i\rfloor_{\mathtt{act}(q_i)}$, $i = 1, 2$, then $\mathbf{v}_2 \in \zeta_2$;
- for any $\mathbf{v}_2 \in \zeta_2$, there exist $\mathbf{v}_1 \in \zeta_1$ and $\mathcal{X}$-valuations $\mathbf{v}'_1, \mathbf{v}'_2, \mathbf{v}''_2$, such that:
  - $\mathbf{v}_i = \mathbf{v}'_i\rfloor_{\mathtt{act}(q_i)}$, for $i = 1, 2$.
  - $\mathbf{v}'_2$ and $\mathbf{v}''_2$ are c-equivalent.
  - $(q_1, \mathbf{v}'_1) \xrightarrow{e} \xrightarrow{\delta} (q_2, \mathbf{v}''_2)$, for some $\delta \in \mathsf{R}$.

The above property is proved directly from the definitions. The difference with simulation-graph post-stability is that $\mathbf{v}_1, \mathbf{v}_2$ are dimension-restricting projections of the "real" valuations $\mathbf{v}'_1, \mathbf{v}'_2$.

Non-zeno paths. An infinite path $\pi = (q_1, \zeta_1) \xrightarrow{e_1} (q_2, \zeta_2) \xrightarrow{e_2} \cdots$ in $AG(A, c)$ is called non-zeno if for each clock $x \in \mathcal{X}$:

- either x is active and reset infinitely often in π, that is, $\forall i \,.\, \exists j > i \,.\, x \in \mathtt{act}(q_j) \wedge x \in \mathsf{reset}(e_j)$,
- or x remains active and unbounded in π from some point on, that is, $\exists i \,.\, \forall j > i \,.\, x \in \mathtt{act}(q_j) \wedge \mathsf{unbounded}(x, \zeta_j) \wedge \mathsf{unbounded}(x, \mathsf{guard}(e_j))$.

Relating runs to symbolic paths. A run $\rho = (q_1, \mathbf{v}_1) \xrightarrow{\delta_1} (q_1, \mathbf{v}_1 + \delta_1) \xrightarrow{e_1} \cdots$ is said to be inscribed in a path $(q_1, \zeta_1) \xrightarrow{e_1} \cdots$ of the activity graph, if for all $i \geq 1$, there exists $\mathbf{v}'_i \in \zeta_i$ such that $\mathbf{v}'_i + \delta_i \in \zeta_i$ and $\mathbf{v}'_i = \mathbf{v}_i\rfloor_{\mathtt{act}(q_i)}$.

Based on the above definitions, lemma 5.9 can be easily re-proven for the activity graph.

Lemma 5.10 *Every run (resp. non-zeno run) of A is inscribed in a unique path (resp. non-zeno path) in $AG(A, c)$. Inversely, for every path (resp. non-zeno path) π in $AG(A, c)$, there is a run (resp. non-zeno run) inscribed in π.*

5.2.4 Inclusion abstraction

The inclusion abstraction is intended to preserve reachability. It is based on the following observation: if for two zones S_1 and S_2, $S_1 \subseteq S_2$, then S_1 can be ignored, since any state in S_1 belongs also to S_2 and any successor of S_1 is also a successor of S_2.

More precisely, let $\mathcal{Z}$ be a set of zones. A total function $\alpha_{inc} : \mathcal{Z} \mapsto \mathcal{Z}$ is an *inclusion abstraction* on $\mathcal{Z}$ if for any $S \in \mathcal{Z}$, $S \subseteq \alpha_{inc}(S)$.

Now, consider a TA A and let α_{inc} be an inclusion abstraction on the set of nodes of the simulation graph of A. The *inclusion graph* of A with respect to α_{inc}, denoted $IG(A, c)$, is defined as follows:

- For each node S of $SG(A, c)$, $\alpha_{inc}(S)$ is a node of $IG(A, c)$.
- For each edge $S_1 \xrightarrow{e} S_2$ of $SG(A, c)$, $\alpha_{inc}(S_1) \xrightarrow{e} \alpha_{inc}(S_2)$ is an edge of $IG(A, c)$.

Returning to the TGC example, observe that in the simulation graph of figure 5.12, zones 9 and 10 are subsets of zone 1. Then, we can define an inclusion abstraction that maps zones 9 and 10 to 1 and every other zone to itself. This abstraction induces the inclusion graph of figure 5.17, which contains two nodes less than the simulation graph.

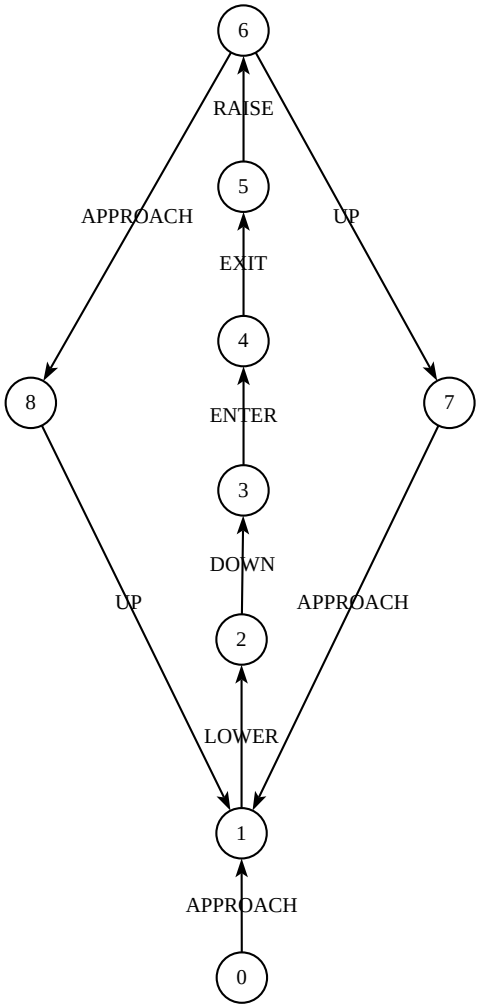


Figure 5.17: The inclusion graph of the Train–Gate–Controller example.

Optimal inclusion. According to the definition above, there might be many inclusion abstractions possible for a given simulation graph. For the previous example, we could have eliminated only one of the nodes 9 and 10 instead of both, and this would still be a valid abstraction. An inclusion abstraction α_{inc} is said to be *optimal* if for any other inclusion abstraction α'_{inc}, $\alpha_{inc}[\mathcal{Z}] \subseteq \alpha'_{inc}[\mathcal{Z}]$, where $\alpha[\cdot]$ denotes the image of a relation α. For the previous example, the abstraction merging both 9 and 10 to 1 is optimal.

It is easy to prove that an optimal inclusion abstraction always exists (since the simulation graph is finite), although it might not be unique (since a zone can be a subset of more than one incomparable zones). The reason why we have not considered just optimal inclusion abstractions is that they cannot be always computed on-the-fly: finding which zone is included in which other should be done while the simulation graph is generated, which depends on the order of traversal of the graph (see also discussion in section 7.1 about on-the-fly generation of abstract graphs).

Properties preserved in the inclusion graph

Inclusion preserves linear properties in a conservative manner and reachability in an exact manner, modulo c-equivalence. More precisely:

Lemma 5.11 *1. For each state $s \in Reach(A)$ there exists a node S in $IG(A,c)$ such that $s \in S$. Inversely, for each $s \in S$, there exists a c-equivalent state $s' \in Reach(A)$.*

2. Every run (resp. non-zeno run) of A is inscribed in a unique path (resp. non-zeno path) in $IG(A,c)$.

The proof of the lemma follows directly from the properties below:

- (Post-stability): If $S_1 \xrightarrow{e} S_2$ is an edge of $IG(A,c)$, then for each $s_1 \in S_1$, if $s_1 \xrightarrow{e}\xrightarrow{\delta} s_2$ then $s_2 \in S_2$.
- Every node S of $IG(A,c)$ is also a node of $SG(A,c)$, therefore, there exists a zone path $S_0 \xrightarrow{e_0} \cdots S$.
- For any path $\pi = S_1 \xrightarrow{e_1} \cdots S_l$ in $SG(A,c)$, there exists a path $\pi' = S'_1 \xrightarrow{e_1} \cdots S'_l$ in $IG(A,c)$, such that $S_i \subseteq S'_i$, for $i = 1, ..., l$. Moreover, if π is non-zeno, then π' is also non-zeno.

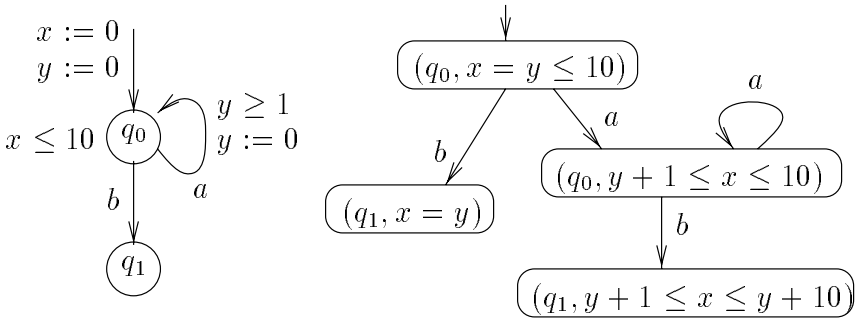


Figure 5.18: A TA and its inclusion graph.

Lemma 5.11 will be used in section 7.1 to check reachability. To see that the inverse of part 2 of the lemma does not always hold, look at the example of figure 5.18. In any run of the TA the a-transition is taken at most 10 times, however, in the inclusion graph it can be taken an unbounded number of times.

5.2.5 Convex hull

The convex-hull abstraction is intended to be an over-approximation of the set of reachable states. The idea is to perform a forward reachability analysis, as in the simulation graph, but keep a single zone (q, ζ) for each discrete state q. If another zone (q, ζ') is found reachable, then ζ is updated to $\zeta \sqcup \zeta'$ (notice that $\zeta \cup \zeta'$ is not generally convex, so, replacing ζ by $\zeta \cup \zeta'$ in (q, ζ) doesn't yield a zone).

More precisely, consider a TA A with set of discrete states Q. The *convex-hull graph* of A, denoted $CHG(A, c)$, is defined to be the smallest graph (in terms of nodes and set inclusion among nodes) such that:

- All nodes of $CHG(A, c)$ are c-closed zones.
- For each $q \in Q$, $CHG(A, c)$ has at most one node (q, ζ).
- For each node S in $SG(A, c)$, $CHG(A, c)$ has a node $S' \supseteq S$.
- For each node S in $CHG(A, c)$ and each edge e such that $\mathsf{post}(e, S, c) \neq \mathsf{false}$, $CHG(A, c)$ has a node $S' \supseteq \mathsf{post}(e, S, c)$ and an edge $S \xrightarrow{e} S'$.

In other words, $CHG(A, c)$ is generated by taking the "closure" of $SG(A, c)$ with respect to the convex-hull and $\mathsf{post}()$ operators. By the fact that all nodes of $CHG(A, c)$ are c-closed and lemma 2.2, $CHG(A, c)$ is finite. The *convex-hull abstraction*, denoted α_{ch}, is the function mapping each node (q, ζ') of $SG(A, c)$ to the (unique) node (q, ζ) of $CHG(A, c)$.

As an example, the convex-hull graph of the TGC system is identical to its optimal inclusion graph (figure 5.17). Notice that no unreachable states are added in this case.

Properties preserved in the convex-hull graph

The convex-hull graph preserves linear properties and discrete-state reachability in a conservative manner. More precisely, we have the following result.

Lemma 5.12 *1. For each state $s \in Reach(A)$ there exists a node S in $CHG(A, c)$ such that $s \in S$.*

2. Every run (resp. non-zeno run) of A is inscribed in a unique path (resp. non-zeno path) in $CHG(A, c)$.

The result follows directly from the post-stability of $CHG(A, c)$, namely, that for each edge $S_1 \xrightarrow{e} S_2$ of $CHG(A, c)$, for each $s_1 \in S_1$, if $s_1 \xrightarrow{e}\xrightarrow{\delta} s_2$ then $s_2 \in S_2$.

Lemma 5.12 is used in checking reachability, for instance, of error states. Since the convex-hull graph is an over-approximation of the concrete state space, this method can mainly be used in proving that some discrete state q is *not* reachable: if q is not reachable in the abstract graph, it is certainly not reachable in the concrete graph either; on the other hand, if q is reachable in the abstract graph, no conclusion can generally be made (a partial remedy to this is discussed in section 7.1).

	Reachability	TBA	CTL (TCTL)
Strong TaB	$\surd$	$\surd$	$\surd$
Weak TaBs	$\surd$	$\surd$	$\times$
Simulation graph	$\surd$	$\surd$	$\times$
Activity	$\surd$	$\surd$	$\times$
Inclusion	$\surd$	conservatively	$\times$
Convex hull	conservatively	conservatively	$\times$

Table 5.4: Summary of property preservation by abstractions.

5.2.6 Combination of activity, inclusion and convex hull

Apart from being applied separately on top of the simulation graph of a TA, the activity, inclusion and convex-hull abstractions can be also combined, in order to give a better reduction of the state space.

There are two meaningful combinations, obtained by applying activity on top of either the inclusion or the convex-hull graph. The *activity-inclusion graph* (resp. *activity-convex-hull graph*) is defined similarly to $AG(A, c)$ with the difference that α_{act} is replaced by the composition $\alpha_{act} \circ \alpha_{inc}$ (resp. $\alpha_{act} \circ \alpha_{ch}$).

The rest of the possible combinations are either not well-defined (for instance, applying first activity and then inclusion is not possible, since activity changes the dimension of polyhedra) or not interesting (for instance, applying inclusion on top of convex hull or the inverse is identical to applying just convex hull).

Concerning the properties preserved by the two combined abstractions, the activity-inclusion graph preserves all properties preserved in the inclusion graph, while the activity-convex-hull graph preserves all properties preserved in the convex-hull graph. These results can be derived easily by combining the preservation properties of activity, inclusion and convex hull.

Summary of preservation results

Table 5.4 summarizes the preservation results of this chapter. The sign $\surd$ means that the abstraction preserves the corresponding class of properties and $\times$ means that it does not. The note "conservatively" means that the abstraction preserves the property only in one direction, i.e.

- for reachability, if a state is reachable in the concrete system then it is also reachable in the abstract one;
- for TBA, if the concrete system has a non-empty language then the abstract graph has a non-zeno accepting cycle.

All results are modulo non-zenoness. The results of the column for CTL also hold for TCTL, modulo the construction described in section 6.2.2.

Relation to the literature

The *abstract interpretation* framework for program analysis using abstractions has been introduced in [CC77], and has been used extensively for model checking in the untimed context

(see, for instance, [CGL94, LGS+95, DGG97]). This framework is quite powerful, however, when used for model-checking of infinite-state systems, an abstraction towards a finite domain is not easy to find. It is often the responsibility of the user to define the abstraction, which can be a non-trivial task requiring a lot of intuition in the input model. For instance, [TAKB96] study timed simulations and propose a compositional approach to prove that a timed system simulates another one, by applying *assume-guarantee* proof rules on the system components. However, no methodology is provided for finding the right system simulating a given one.

The time-abstracting observational bisimulation has been introduced in [LY93]. The authors study TaoB from an algebraic point of view, proving that it is a congruence with respect to the real-time process calculus of [Yi90]. Its properties with respect to logical formalisms are not examined. The strong TaB has been introduced in [TY96]. Stronger timed bisimulations not abstracting away from exact time delays have been studied in the literature, often associated to extensions of real-time process calculi (see, for instance, [RR88, NRSV90, Č92]).

The simulation graph has been introduced independently in KRONOS [Oli94] and RT-SPIN [TC96], along with c-closure. In fact, the latter operation is necessary only to ensure that the simulation graph is finite, therefore, in practice, it is an option of the forward reachability algorithm. Inclusion has been used implemented independently in the reachability analysis of KRONOS and UPPAAL [BGK+96]. Convex-hull abstractions have been also used for approximate reachability analysis by [Hal93, WTD94, WT95, Bal96]. Clock activity has been introduced in [DY96]. These techniques have been formalized in the framework of abstractions in [DT98]. A related approach can be found in [SV96]. [AIKY92] present a technique based on over-approximations: the method consists in attempting to prove the property on an abstract system where some clocks are ignored; if this attempt fails, then clocks are re-introduced progressively until either the property is proven on the abstract system, or all the clocks have been re-introduced.

Chapter 6

Verification based on Minimization

In this chapter we show how the time-abstracting bisimulations introduced in section 5.1.1 can be used for verification. The goal is to demonstrate how classical (untimed) verification techniques can be used also for verifying timed systems.

More precisely, given a system described as a TA A, the following cases are possible:

- If the specification of A is given in terms of a TBA B, model-checking A against B is reduced to checking the $\approx$-quotient of $A \times B$ for emptiness; $\approx$ can be any TaB, since they all preserve linear properties.
- If the specification of A is given in terms of a CTL formula ϕ, model-checking A against ϕ is reduced to model-checking the STa-quotient of A against ϕ. If ϕ is a TCTL formula, it is transformed to a CTL formula, A is extended with a set of auxiliary clocks, and the same technique applies (see section 6.2.2).
- If the specification of A is given behaviorally, that is, in terms of another (timed or untimed) automaton A', then checking that A and A' are bisimilar with respect to a TaB is reduced to checking that their STa-quotients are bisimilar with respect to an untimed bisimulation.

In any of the above cases, the Ta-quotient of A needs to be computed. For reasons of efficiency, we are interested in the *minimal* quotient, that is, the one corresponding to the greatest bisimulation. Computing the minimimal quotient is called *minimization* and is dealt with in section 6.1. Then, in section 6.2 we show how Ta-quotients can be used for verification, in any of the three cases discussed above.

6.1 Minimization of Timed Automata

Based on the fact that a bisimulation induces a pre-stable partition and vice versa (section 2.1), minimization is done by partition *refinement*: given an (untimed) graph $G = (V, \rightarrow)$, we compute the coarsest pre-stable partition of V, starting from an initial partition $\mathcal{C}$ and successively refining it until it becomes pre-stable. Refining $\mathcal{C}$ consists in choosing two classes C_1, C_2 such that C_1 is unstable with respect to C_2, and then replacing C_1 by $C_1 \cap \mathsf{preds}(C_2)$ and $C_1 \setminus \mathsf{preds}(C_2)$.

A partition-refinement algorithm [PT87] is shown in figure 6.1. The algorithm takes as input the initial partition $\mathcal{C}_0$ and computes the coarsest stable partition which is finer than $\mathcal{C}_0$.

```
Refine (C_0) {
  C := C_0 ;
  while (∃C_1, C_2 ∈ C . C_1 ∩ preds(C_2) ∉ {C_1, ∅}) do
    C_{C_1} := {C_1 ∩ preds(C_2), C_1 \ preds(C_2)} ;
    C := (C \ {C_1}) ∪ C_{C_1} ;
  end-while
  return (C) ;
}
```

Figure 6.1: A simple partition-refinement algorithm.

$\mathcal{C}_0$ can be either $\{V\}$ (the entire set of nodes) or a partition of V respecting a set of atomic propositions.

The algorithm of figure 6.1 does not take into account class reachability. A class is reachable if it contains at least one reachable node. In the above algorithm, the final partition is pre-stable with respect to all classes, whether or not reachable. The *minimal-model generation algorithm* (MMGA) proposed in [BFH+92] combines refinement and reachability, by refining only reachable classes and updating reachability information meanwhile [1].

The MMGA is shown in figure 6.2. It starts with an initial partition $\mathcal{C}_0$ and uses three sets of classes, namely, the current partition $\mathcal{C}$, the set of reachable classes $Access \subseteq \mathcal{C}$ (the initial node is v_0) and the set of stable classes $Stable \subseteq \mathcal{C}$. If there exists a reachable class C_1 which may be unstable (i.e., $C_1 \in Access \setminus Stable$), the algorithm attempts to refine it. If it succeeds, C_1 is removed from $\mathcal{C}$, replaced by its two sub-parts. Otherwise, C_1 is inserted in *Stable*. The sets *Access* and *Stable* are updated accordingly. In the first case, all predecessor classes of C_1 which were stable are considered unstable, thus, are removed from *Stable*. Also, a sub-part of C_1 is considered unreachable, unless it contains the initial node. In the second case, a single-step reachability is performed, to add to the reachable classes all successors of the newly-found stable class.

6.1.1 Adapting for TA the partition-refinement algorithm of [BFG+92]

MMGA can be adapted to infinite state spaces, assuming that they admit effective representations of classes and decision procedures for computing intersection, set-difference and predecessors of classes, and testing whether a class is empty. For termination, it must be ensured that a pre-stable partition always exists.

The state space of TA falls in this category, with the difference that there are two types of predecessors, corresponding to discrete and time transitions of the TA. Taking this observation into account, the adapted algorithm called time-abstracting MMGA (TA-MMGA) is shown in figure 6.3.

In TA-MMGA, the initial partition $\mathcal{C}_0$ is a set of symbolic states. The test for stability of a class S_1 is done in two phases: first stability is checked with respect to time-successors and

[1] There exists two alternatives, namely, either to perform refinement followed by reachability to keep only reachable classes, or to compute the set of reachable states first and then refine it. The drawback of the first approach is that unreachable classes are uselessly refined, while in the second approach, computing the set of reachable states might lead to explosion if a compact representation is not available, like in the case of TA.

```
ReachRefine (C_0) {
  C := C_0 ;
  Access := {C ∈ C_0 | v_0 ∈ C} ;
  Stable := {} ;
  while (∃C_1 ∈ Access \ Stable) do
    if (∃C_2 ∈ C . C_1 ∩ preds(C_2) ∉ {C_1, ∅}) then
      C_{C_1} := {C_1 ∩ preds(C_2), C_1 \ preds(C_2)} ;
      Access := (Access \ {C_1}) ∪ {C ∈ C_{C_1} | v_0 ∈ C} ;
      Stable := Stable \ {C' ∈ C | C' ∩ preds(C) ≠ ∅} ;
      C := (C \ {C_1}) ∪ C_{C_1} ;
    else
      Stable := Stable ∪ {C_1} ;
      Access := Access ∪ {C' ∈ C | C ∩ preds(C') ≠ ∅} ;
    end-if
  end-while
  return (Access) ;
}
```

Figure 6.2: The Minimal Model Generation Algorithm [BFH+92].

then with respect to discrete successors. The updates of sets *Stable* and *Access* are modified accordingly.

TA-MMGA is parameterized by the discrete-predecessor function $\mathsf{disc\text{-}pred}_{\approx}$, so that it can be used for refinement with respect to any TaB. The function is defined as follows:

$$\mathsf{disc\text{-}pred}_{\approx}(S) \stackrel{\text{def}}{=} \begin{cases} \bigcup_{e \in E} \mathsf{disc\text{-}pred}(e, S), & \text{for the STa-quotient} \\ \bigcup_{e \in E} \mathsf{time\text{-}pred}(\mathsf{disc\text{-}pred}(e, S)), & \text{for the Tad-quotient} \\ \bigcup_{e \in E} \mathsf{time\text{-}pred}(\mathsf{disc\text{-}pred}(e, \mathsf{time\text{-}pred}(S))), & \text{for the Tao-quotient} \end{cases}$$

The definition of $\mathsf{disc\text{-}pred}_{\approx}$ corresponds to the definition of the three TaBs, since the STaB distinguishes states according to their immediate discrete successors, while in the weak TaBs a delay can be afforded before or after the discrete transition.

Correction of the algorithm follows from the definitions of the above predecessor operators. Termination is ensured by lemma 5.3: in the worst case, the algorithm will generate the partition induced by the region equivalence.

6.1.2 A partition-refinement technique that preserves convexity

The main drawback of TA-MMGA is that set-difference of symbolic states is an expensive operation to implement. Indeed, this operation is reduced to complementation of polyhedra which is exponential in the number of clocks, as shown in section 10.3. We show how to avoid complementation by starting from an initial partition of the state space in zones and applying a refinement technique that preserves zones, i.e., convexity of polyhedra. The idea is the following. Each time a zone S is to be refined, it is refined with respect to either all its discrete successors by some edge e, or all its time-successors. If all successor classes are zones, then S will be "split" in a number of sub-zones, and all we need to make sure is that these sub-zones are disjoint and cover S.

```
TimeAbstractingReachRefine (C_0, disc-pred_≈()) {
  C := C_0 ;
  Access := {S ∈ C_0 | (q_0, 0) ∈ S} ;
  Stable := {} ;
  while (∃S_1 ∈ Access \ Stable) do
    /* First test stability w.r.t. time transitions */
    if (∃S_2 ∈ C . S_1 ∩ time-pred(S_2) ∉ {S_1, ∅}) then
      C_{S_1} := {S_1 ∩ time-pred(S_2), S_1 \ time-pred(S_2)} ;
      Access := (Access \ {S_1}) ∪ {S ∈ C_{S_1} | (q_0, 0) ∈ S} ;
      Stable := Stable \ {S ∈ C | S ∩ time-pred(S_1) ≠ ∅}
                       \ {S ∈ C | S ∩ disc-pred_≈(S_1) ≠ ∅} ;
      C := (C \ {S_1}) ∪ C_{S_1} ;
    /* Then test stability w.r.t. discrete transitions */
    else if  (∃S_2 ∈ C . S_1 ∩ disc-pred_≈(S_2) ∉ {S_1, ∅}) then
      C_{S_1} := {S_1 ∩ disc-pred_≈(S_2), S_1 \ disc-pred_≈(S_2)} ;
      Access := (Access \ {S_1}) ∪ {S ∈ C_{S_1} | (q_0, 0) ∈ S} ;
      Stable := Stable \ {S ∈ C | S ∩ time-pred(S_1) ≠ ∅}
                       \ {S ∈ C | S ∩ disc-pred_≈(S_1) ≠ ∅} ;
      C := (C \ {S_1}) ∪ C_{S_1} ;
    else
      Stable := Stable ∪ {S_1} ;
      Access := Access ∪ {S ∈ C | S_1 ∩ time-pred(S) ≠ ∅}
                       ∪ {S ∈ C | S_1 ∩ disc-pred_≈(S) ≠ ∅} ;
    end-if
  end-while
  return (Access) ;
}
```

Figure 6.3: The Time-Abstracting Minimal Model Generation Algorithm.

More formally, consider a TA A and let S and S' be two symbolic states of A. The *continuous* time predecessors of S' to S are defined as follows:

$$\mathsf{until}(S, S') \stackrel{\text{def}}{=} \{s \in S \mid \exists \delta \in \mathsf{R} \,.\, s + \delta \in S' \wedge \forall \delta' < \delta \,.\, s + \delta' \in S \cup S'\}$$

Intuitively, $\mathsf{until}(S, S')$ contains all states of S which can let time pass and reach S', continuously staying in S, that is, without traversing other classes in-between. We say that S has *immediate* time successors in S' if $\mathsf{until}(S, S') \neq \emptyset$. For an example of $\mathsf{until}()$, look at figure 6.4. The result of $\mathsf{until}(S, S_1)$ is empty, whereas $\mathsf{until}(S, S_2) = S$. Also, we have $\mathsf{until}(S_2, S_1) = S_4$, $\mathsf{until}(S_2, S_3) = S_5$, and so on.

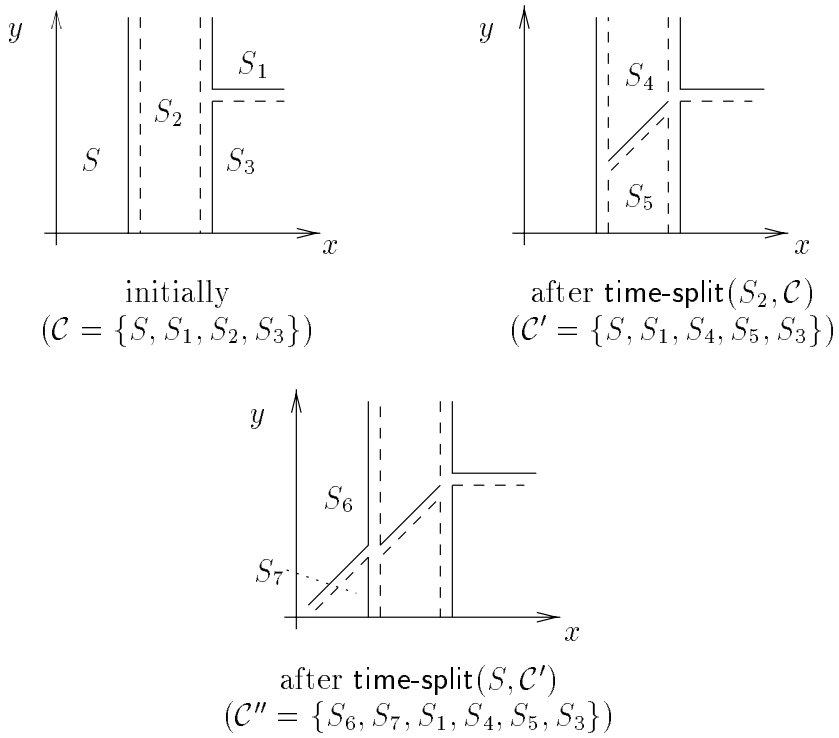


Figure 6.4: Refinement with respect to immediate time-successors.

The following lemma shows that $\mathsf{until}()$ preserves convexity.

Lemma 6.1 *If S and S' are zones then $\mathsf{until}(S, S')$ is a zone.*

Proof: Let $S = (q, \zeta)$, $S' = (q, \zeta')$ and $\mathsf{until}(S, S') = (q, \zeta'')$ (in all other cases $\mathsf{until}(S, S') = \emptyset$). We have to show that if $\mathbf{v}_1, \mathbf{v}_2 \in \zeta''$ then $\mathbf{v} = \beta \mathbf{v}_1 + (1 \Leftrightarrow \beta)\mathbf{v}_2 \in \zeta''$, for any $0 < \beta < 1$. $\mathbf{v}_1, \mathbf{v}_2 \in \zeta''$ implies that $\mathbf{v}_1, \mathbf{v}_2 \in \zeta$ and there exist $\delta_1, \delta_2 \in \mathsf{R}$ such that $\mathbf{v}_1 + \delta_1, \mathbf{v}_2 + \delta_2 \in \zeta'$ and $\forall \delta_1' < \delta_1, \delta_2' < \delta_2 \,.\, \mathbf{v}_1 + \delta_1', \mathbf{v}_2 + \delta_2' \in \zeta \cup \zeta'$.

Let $\delta = \beta\delta_1 + (1 \Leftrightarrow \beta)\delta_2$. Then, $\mathbf{v} + \delta = \beta(\mathbf{v}_1 + \delta_1) + (1 \Leftrightarrow \beta)(\mathbf{v}_2 + \delta_2)$, implying that $\mathbf{v} + \delta \in \zeta'$, since ζ' is convex. We have to show that $\forall \delta' < \delta \,.\, \mathbf{v} + \delta' \in \zeta \cup \zeta'$. Given $\delta' < \delta$, we can write δ' as $\beta\delta_3 + (1 \Leftrightarrow \beta)\delta_4$, for some $\delta_3 \leq \delta_1$ and $\delta_4 \leq \delta_2$. We have $\mathbf{v}_1 + \delta_3, \mathbf{v}_2 + \delta_4 \in \zeta \cup \zeta'$. If both $\mathbf{v}_1 + \delta_3, \mathbf{v}_2 + \delta_4 \in \zeta$ or $\mathbf{v}_1 + \delta_3, \mathbf{v}_2 + \delta_4 \in \zeta'$, we are done, since ζ and ζ' are both convex.

Consider the case $\mathbf{v}_1 + \delta_3 \in \zeta$ and $\mathbf{v}_2 + \delta_4 \in \zeta'$ (the case $\mathbf{v}_1 + \delta_3 \in \zeta'$ and $\mathbf{v}_2 + \delta_4 \in \zeta$ is symmetrical). Let γ be the smallest positive real such that $\mathbf{v}_2 + \delta_4 \Leftrightarrow \gamma \in \zeta$ or $\mathbf{v}_1 + \delta_3 + \gamma(1 \Leftrightarrow \frac{1}{\beta}) \in \zeta'$.

Assume the first case (notice that only one of the cases is possible). We have $\mathbf{v}_1+\delta_5, \mathbf{v}_2+\delta_6 \in \zeta$, for $\delta_6 = \delta_4 \Leftrightarrow \gamma$ and $\delta_5 = \delta_3 + \gamma(1 \Leftrightarrow \frac{1}{\beta})$. Moreover, $\delta' = \beta\delta_5 + (1 \Leftrightarrow \beta)\delta_6$, which means that $\mathbf{v} + \delta' = \beta(\mathbf{v}_1 + \delta_5) + (1 \Leftrightarrow \beta)(\mathbf{v}_2 + \delta_6)$. By convexity of ζ, $\mathbf{v} + \delta' \in \zeta$. ■

Now, let $\mathcal{X}$ be the clocks of A and Q its discrete states. A *convex partition respecting* A is a finite partition $\mathcal{C}$ of $Q \times \mathsf{R}^{\mathcal{X}}$ in zones, such that for any $(q, \zeta) \in \mathcal{C}$:

- either $\zeta \subseteq \mathsf{invar}(q)$ or $\zeta \cap \mathsf{invar}(q) = \emptyset$ (i.e., $\mathcal{C}$ respects the invariants of A);
- for all $e \in \mathsf{out}(q)$, either $\zeta \subseteq \mathsf{guard}(e)$ or $\zeta \cap \mathsf{guard}(e) = \emptyset$ (i.e., $\mathcal{C}$ respects the guards of A).

Given a class S of $\mathcal{C}$ and an edge e of the TA, we define the following refinement functions:

$$\begin{aligned} \mathsf{time\text{-}split}(S, \mathcal{C}) &\stackrel{\text{def}}{=} \{\mathsf{until}(S, S') \neq \emptyset \mid S' \in \mathcal{C}\} \\ \mathsf{disc\text{-}split}(S, e, \mathcal{C}) &\stackrel{\text{def}}{=} \{S \cap \mathsf{disc\text{-}pred}(e, S') \neq \emptyset \mid S' \in \mathcal{C}\} \end{aligned}$$

Intuitively, time-split() and disc-split() refine S with respect to immediate time-successors and e-successors, respectively.

Using these split functions, we develop the convex TA-MMGA, shown in figure 6.5. The algorithm takes as input a convex partition $\mathcal{C}_0$ respecting A and computes the coarsest partition *Access* which is finer than $\mathcal{C}_0$, such that: (1) all classes in *Access* are zones and contain at least one reachable state; (2) *Access* induces a STaB. To prove this formally, we need to show that:

1. first, the current partition $\mathcal{C}$ is stable iff no splits are successful, that is, for any $S \in \mathcal{C}$ and any edge e, $\mathsf{disc\text{-}split}(S, e, \mathcal{C}) \in \{\{S\}, \emptyset\}$ and $\mathsf{time\text{-}split}(S, \mathcal{C}) \in \{\{S\}, \emptyset\}$;
2. second, each time S is split in a number of subsets, these are disjoint and they *cover* S (i.e., their union gives S).

For the first point above, it is obvious that if a split is successful then $\mathcal{C}$ is unstable. Inversely, by definition, if no discrete split is possible then S is stable with respect to its discrete successors. For the time-successors, a similar fact is not generally true: $\mathsf{time\text{-}split}(S, \mathcal{C})$ might yield $\emptyset$ even if S is unstable with respect to a class S_1. However, this is possible only if S_1 does not contain any immediate time successors of S, that is, if there is at least one class S_2 traversed between S and S_1 (see figure 6.4 for an example). Then, eventually, S_2 will be effectively split, and the refinement will be propagated to S.

The second of the above points is proven by the following lemma.

Lemma 6.2 *Each of* $\mathsf{time\text{-}split}(S, \mathcal{C})$ *and* $\mathsf{disc\text{-}split}(S, e, \mathcal{C})$ *forms a partition of* S *in zones.*

Proof: Consider $\mathcal{C}_1 = \mathsf{time\text{-}split}(S, \mathcal{C})$ first. By lemma 6.1, all members of $\mathcal{C}_1$ are zones. It remains to show that they are disjoint and that their union yields S. Let $S_i \in \mathcal{C}_1$, $S_i = \mathsf{until}(S, S'_i)$, where $S'_i \in \mathcal{C}$, for $i = 1, 2$. Since $\mathcal{C}$ is a partition, S, S'_1 and S'_2 are all disjoint. Assume $s \in S_1 \cap S_2$. For $i = 1, 2$, there exist $\delta_i \in \mathsf{R}$ such that $s + \delta_i \in S'_i$ and $\forall \delta'_i < \delta_i \,.\, S \cup S'_i$. Observe that $\delta_1 \neq \delta_2$, since S'_1 and S'_2 are disjoint. Without loss of generality, assume $\delta_1 < \delta_2$. We have that $s+\delta_1 \in S'_1$ and $s+\delta_1 \in S \cup S'_2$, that is, either $s+\delta_1 \in S'_1 \cap S$ or $s+\delta_1 \in S'_1 \cap S'_2$, which contradicts the fact that S, S'_1 and S'_2 are all disjoint. This proves that S_1 and S_2 are disjoint. Now, let $s \in S$. We can find $\delta \in \mathsf{R}$ and $S' \in \mathcal{C}$ such that $s + \delta \in S'$ and $\forall \delta' < \delta \,.\, s + \delta' \in S \cup S'$ (since $\mathcal{C}$ is a finite partition). By definition, $s \in \mathsf{until}(S, S')$.

```
ConvexStrongTimeAbstractingReachRefine (C_0) {
  C := C_0 ;
  Access := {S ∈ C_0 | s_0 ∈ S} ;
  Stable := {} ;
  while (∃S ∈ Access \ Stable) do
    /* First try to refine w.r.t. time transitions */
    C_S := time-split(S, C) ;
    /* If S is stable w.r.t. time transitions,
       try to refine w.r.t. discrete transitions */
    if (C_S ∈ {{S}, ∅}) then
      for each (e ∈ E) do
        C_S := disc-split(S, e, C) ;
        if (C_S ∉ {{S}, ∅}) then break ;
      end for each
    end if
    if (C_S ≠ {S}) then
      Access := (Access \ {S}) ∪ {S' ∈ C_S | s_0 ∈ S'} ;
      Stable := (Stable \ {S' | until(S', S) ≠ ∅})
                        \ {S' | ∃e ∈ E . S' ∩ disc-pred(e, S) ≠ ∅} ;
      C := (C \ {S}) ∪ C_S ;
    else
      Stable := Stable ∪ {S} ;
      Access := Access ∪ {S' | until(S, S') ≠ ∅}
                        ∪ {S' | ∃e ∈ E . S ∩ disc-pred(e, S') ≠ ∅} ;
    end-if
  end-while
  Access := {(q, ζ) ∈ Access | ζ ⊆ invar(q)} ;
  return (Access) ;
}
```

Figure 6.5: A partition-refinement algorithm preserving convexity.

Now, consider $\mathcal{C}_2 = \mathsf{disc\text{-}split}(S, e, \mathcal{C})$, for some edge e. By lemma 5.1, all members of $\mathcal{C}_2$ are zones. By the distributivity of $\mathsf{disc\text{-}pred}$ over union ($\mathsf{disc\text{-}pred}(e, S_1 \cup S_2, e) = \mathsf{disc\text{-}pred}(e, S_1) \cup \mathsf{disc\text{-}pred}(e, S_2)$) members of $\mathcal{C}_2$ cover S. It remains to show that they are disjoint. Let $S_i \in \mathcal{C}_2$, $S_i = S \cap \mathsf{disc\text{-}pred}(e, S'_i)$, where $S'_i \in \mathcal{C}$, for $i = 1, 2$. Since $\mathcal{C}$ is a partition, S'_1 and S'_2 are disjoint. Assume $s \in S_1 \cap S_2$. Recall that the e-successor of s, say s', is unique. Since $s \in \mathsf{disc\text{-}pred}(e, S'_1) \cap \mathsf{disc\text{-}pred}(e, S'_2)$, it must be that $s' \in S'_1 \cap S'_2$, which contradicts the fact that S'_1 and S'_2 are disjoint. ■

It should be noted that the greatest STaB with respect to an arbitrary initial partition does not generally induce a convex partition. This is why the convex TA-MMGA must be initialized with a convex partition respecting the TA in question, although this partition might be finer than necessary (for instance, to respect a set of atomic propositions). The benefit of avoiding complementation outweighs the overhead of having a finer partition at the end.

The convex TA-MMGA has been implemented in the KRONOS module `minim`, discussed in section 11.2.

6.2 Verification using Quotient Graphs

Quotient graphs are essentially untimed graphs, since all quantitative-time information has been abstracted away. Therefore, the existing infrastructure in algorithms and tools for untimed verification can be exploited, in order to treat timed systems as a particular case of untimed systems. The purpose of this section is to show how this can be done using STa-quotient graphs. For the sake of brevity, we do not consider weak Ta-quotients. The latter can be generated from STa-quotients using untimed minimization techniques (section 6.2.5) and can be used in the place of STa-quotients for TBA model checking or deadlock detection.

First, we make explicit how the STa-quotient graph G of a TA A is computed from the result returned by the convex TA-MMGA:

- The set of nodes of G is the set of classes *Access* returned by the algorithm.
- If for two distinct classes $S_1, S_2 \in \mathit{Access}$, $\mathsf{until}(S_1, S_2) \neq \emptyset$, then $S_1 \xrightarrow{\tau} S_2$ is an edge of G.
- If for two distinct classes $S_1, S_2 \in \mathit{Access}$ and an edge e, $S_1 \cap \mathsf{disc\text{-}pred}(e, S_2) \neq \emptyset$, then $S_1 \xrightarrow{e} S_2$ is an edge of G.

As explained in section 5.1.1, G does not contain any τ-edges which can be obtained by reflexive, transitive closure (for instance, there are no τ-self-loops). The interest of this construction will become clear in section 6.2.2 below, where we describe how to perform CTL model-checking on STa-quotients.

6.2.1 Timed Büchi Automata model checking

Consider a TA A, a TBA B and a function P associating to each discrete state of B a set of discrete states of A. We want to check whether A satisfies B with respect to P. By lemma 4.2, this comes down to checking whether the language of $A \times B$ is non-empty.

Now, consider a STaB $\approx$ on $A \times B$, such that:

- $\approx$ respects P: if $(q_1, q'_1, \mathbf{v}_1) \approx (q_2, q'_2, \mathbf{v}_2)$ then $q'_1 \in P(q_1)$ iff $q'_2 \in P(q_2)$;

- $\approx$ respects the set of repeating states of B: if $(q_1, q_1', \mathbf{v}_1) \approx (q_2, q_2', \mathbf{v}_2)$ then q_1 is a repeating state iff q_2 is a repeating state.

Let G be the $\approx$-quotient of $A \times B$. A node of G is called repeating if it contains only repeating states. Notice that if a node is not repeating then it contains no repeating states, by the fact that $\approx$ respects the repeating states.

A SCC of G is called non-zeno if for each clock x of $A \times B$, either x is reset in some edge of the SCC or x is unbounded in all nodes of the SCC.

Lemma 6.3 *$A \times B$ has a non-empty language iff G has a maximal SCC which is non-zeno and contains a repeating node.*

Proof: Let ρ be a non-zeno accepting run of $A \times B$. By lemma 5.5, ρ is inscribed in a non-zeno path π in G. Since G contains a finite number of nodes, π defines a cycle λ, and since π is non-zeno, λ is non-zeno. Since ρ is accepting, λ contains a repeating node. Now, consider the maximal SCC containing λ. This SCC contains also the repeating node of λ. Moreover, all clocks reset in λ are also reset in the SCC. Now, let x be a clock not reset in λ, thus x is unbounded in every node of λ. By the pre-stability property of G, we conclude that x is unbounded in every node of the SCC. Otherwise, let $C \xrightarrow{e} C'$ be such that x is unbounded in C and bounded in C', say by a constant c. Let $s = (q, q', \mathbf{v}) \in C$ be such that $\mathbf{v}(x) > c$. Since the value of a clock cannot decrease except if reset to zero and x is not reset in e, s has no discrete successor in C', which violates the fact that C is pre-stable with respect to C'. Thus, the SCC is also non-zeno.

Inversely, let G' be a non-zeno maximal SCC of G containing a repeating node. Let x be a clock not reset in G', thus, x must be unbounded in all nodes of G'. Now, we can build a cycle λ which visits the repeating node of G' and at least one edge resetting a clock which is reset in G'. Since any other clock not reset in π is unbounded in all nodes of λ, λ is non-zeno, and we can extract from it a non-zeno accepting run, using lemma 5.5. ■

Based on the above result, the following algorithm is derived for checking $A \models_P B$:

1. Generate the STa-quotient G of $A \times B$. This is done using the convex TA-MMGA, starting from an initial convex partition respecting $A \times B$, P and the repeating states of $A \times B$.

2. Use an on-the-fly algorithm (for example, the one of [Tar72]) to find all maximal SCCs of G. For each SCC found, check whether it is non-zeno and whether it contains a repeating node. If so, A satisfies B and the SCC can be output as diagnostics. If no such SCC is found, then A does not satisfy B.

Regarding complexity, the SCC search is linear in the size of the STa-quotient, which can be exponential in the number of clocks, as well as in the number of components making up A. Consequently, the bottle-neck of the algorithm is the generation of the quotient, which has to be done before-hand.

In practice, generating G is done using the module `minim`, presented in section 11.2. For the SCC search, one can use a number of linear-time verification tools, such as SPIN [Hol91], COSPAN [HK89] or TLV [PS96], after interfacing the output of `minim` to their input format.

6.2.2 CTL model checking

Consider a TA A, a CTL formula ϕ on a set of atomic propositions *Props* and a function P mapping each atomic proposition to a set of discrete states of A. We want to check whether A satisfies ϕ. We assume that A is deadlock-free and strongly non-zeno.

Let $\approx$ be a STaB on A respecting P, that is, if $(q_1, \mathbf{v}_1) \approx (q_2, \mathbf{v}_2)$ then $q_1 \in P(p)$ iff $q_2 \in P(p)$, for any $p \in Props$. Let G be the $\approx$-quotient of A. A formula is said to hold in a node C of G if it is satisfied in some state of C (by lemma 5.7, this implies that the formula is satisfied in any state of C).

To check $A \models_P \phi$, we use the function $\mathsf{ctl\text{-}eval}(\phi)$, which computes all nodes of G where ϕ holds. The function is defined recursively on the syntax of ϕ, as shown below:

$$
\begin{array}{lcl}
\mathsf{ctl\text{-}eval}(p) & = & \{C \mid \exists (q, \mathbf{v}) \in C \,.\, q \in P(p)\} \\
\mathsf{ctl\text{-}eval}(\phi_1 \vee \phi_2) & = & \mathsf{ctl\text{-}eval}(\phi_1) \vee \mathsf{ctl\text{-}eval}(\phi_2) \\
\mathsf{ctl\text{-}eval}(\exists\, \phi_1 \mathcal{U}\, \phi_2) & = & \mu \mathcal{C} \,.\, \mathsf{ctl\text{-}eval}(\phi_2) \cup (\mathsf{ctl\text{-}eval}(\phi_1) \cap \mathsf{preds}(\mathcal{C})) \\
\mathsf{ctl\text{-}eval}(\forall\, \phi_1 \mathcal{U}\, \phi_2) & = & \mu \mathcal{C} \,.\, \mathsf{ctl\text{-}eval}(\phi_2) \cup (\mathsf{ctl\text{-}eval}(\phi_1) \setminus \mathsf{preds}(\overline{\mathcal{C}}))
\end{array}
$$

At the end of the algorithm, it is checked whether the initial node of G is contained in $\mathsf{ctl\text{-}eval}(\phi)$. The following lemma proves correctness of the model-checking procedure (the proof is given in the appendix).

Lemma 6.4 *$C \in \mathsf{ctl\text{-}eval}(\phi)$ iff for all $s \in C$, s satisfies ϕ.*

A number of branching-time verification tools can be used for performing CTL model-checking on G, such as the `evaluator` module of CADP, or SMV [BCD+90]. Not all of these tools use the fix-point technique described above. Other CTL model-checking algorithms exist, for example, exploring the graph using a SCC search, also compatible with STa-quotients. For an example of CTL model-checking on STa-quotients, the reader is referred to the case study of section 12.1.

6.2.3 TCTL model checking

TCTL model checking can be reduced to CTL model checking using a technique similar to the one used in [Alu91, ACD93] for the region graph. The idea is the following. Given a TA A to be checked against a TCTL formula ϕ, first we extend A with a set of clocks, to obtain a new automaton A^+; then we transform ϕ to a CTL formula ϕ_{CTL}; finally, we generate the STa-quotient of A^+ and model check it against ϕ_{CTL}, as described in the previous section.

More precisely, Q and $\mathcal{X}$ be the set of discrete states and set of clocks of A. Also let $I_1, ..., I_m$ be the set of non-trivial intervals appearing in ϕ. A^+ has exactly the same structure as A, except that it has an augmented set of clocks $\mathcal{X}^+ = \mathcal{X} \cup \{y_1, ..., y_m\}$. The set of atomic propositions *Props* is also augmented with two propositions, namely, $p_{y_j=0}$ and $p_{y_j \in I_j}$, for each $j = 1, ..., m$. Finally, the formula ϕ is transformed to ϕ_{CTL} recursively as follows:

$$
\begin{array}{lll}
p & \text{is transformed to} & p \\
\neg \phi' & \text{is transformed to} & \neg \phi'_{CTL} \\
\phi' \vee \phi'' & \text{is transformed to} & \phi'_{CTL} \vee \phi''_{CTL} \\
\exists\, \phi' \mathcal{U}_{I_j}\, \phi'' & \text{is transformed to} & p_{y_j=0} \Rightarrow \exists\, \phi'_{CTL} \mathcal{U}\, (\phi''_{CTL} \wedge p_{y_j \in I_j}) \\
\forall\, \phi' \mathcal{U}_{I_j}\, \phi'' & \text{is transformed to} & p_{y_j=0} \Rightarrow \forall\, \phi'_{CTL} \mathcal{U}\, (\phi''_{CTL} \wedge p_{y_j \in I_j})
\end{array}
$$

Now, let $\approx$ be a STaB on A^+, respecting the proposition function P as previously, but also all intervals $I_1, ..., I_m$, as well as the constraints $y_j = 0$, $j = 1, ..., m$. For instance, if $[c, c')$ is an interval and $(q_1, \mathbf{v}_1) \approx (q_2, \mathbf{v}_2)$, then for all $j = 1, ..., m$, $c \leq \mathbf{v}_1(y_j) < c'$ iff $c \leq \mathbf{v}_2(y_j) < c'$ and $\mathbf{v}_1(y_j) = 0$ iff $\mathbf{v}_1(y_j) = 0$. The function ctl-eval() can be extended for atomic propositions of the form $p_{y_j=0}$ and $p_{y_j \in I_j}$ in a straightforward way, for instance, ctl-eval$(p_{y_j=0})$ returns the set of all nodes C containing a state $(q, \mathbf{v})$ such that $\mathbf{v}(y_j) = 0$. Then it is easy to prove the following.

Lemma 6.5 *$(q, \mathbf{v}) \models \phi$ iff there exists a state $(q, \mathbf{v}^+)$ of A^+ such that if C is the class of $(q, \mathbf{v}^+)$ in G then $C \in$ ctl-eval(ϕ_{CTL}).*

Notice that $\mathbf{v}$ is a $\mathcal{X}$-valuation while $\mathbf{v}^+$ is an $\mathcal{X}^+$-valuation. This means that, in practice, if we want the states of A which satisfy ϕ, then we have to eliminate clocks $\{y_1, ..., y_m\}$. This can be done by projecting each node C in ctl-eval(ϕ_{CTL}) to $C' = C\rfloor_{\mathcal{X}}$. Then C' contains all states of A satisfying ϕ.

6.2.4 Deadlock and Timelock detection

Consider a TA A and let G be the quotient of A with respect to a STaB $\approx$.

Concerning deadlocks, we can easily prove the following, using the definition of a deadlock and the pre-stability property of $\approx$.

Lemma 6.6 *A is deadlock-free iff there is no reachable sink node in G.*

We can use this lemma for deadlock-detection: first, we generate the STa-quotient G of A and then we perform a DFS on G looking for sink nodes. More interestingly, the two steps can be combined so that sink nodes are reported on-the-fly during the construction of G. This can be done by a straightforward modification of the convex TA-MMGA.

Concerning timelocks, by lemma 3.2, if A is strongly non-zeno, then it is also timelock-free. Otherwise, we can use the following result of [Yov93, HNSY94].

Lemma 6.7 *A is timelock-free iff it satisfies the TCTL formula $\forall\Box\, \exists\Diamond_{\geq 1}$ true.*

Therefore, if A is not strongly non-zeno, we can test whether it is timelock-free by checking it against $\forall\Box\, \exists\Diamond_{\geq 1}$ true. This can be done using the TCTL model-checking procedure of the previous section.

6.2.5 Combination with untimed bisimulations and simulations

The untimed bisimulations introduced in section 2.1, namely, strong, delay, and observational bisimulation, can be combined with TaBs for a number of reasons:

1. For *computing the weak Ta-quotients of a TA A*. If G is the STa-quotient of A, then, by lemma 5.2, the Tad- (resp. Tao-) quotient of A is the quotient of G with respect to the delay (resp. observational) bisimulation.

2. For *comparing two TA A_1 and A_2 with respect to a TaB*. If G_1, G_2 are the STa-quotients A_1, A_2, then A_1 and A_2 are STa-bisimilar iff G_1 and G_2 are bisimilar with respect to the strong bisimulation (again, this is justified by lemma 5.2). Similarly, A_1 and A_2 are Tad- (resp. Tao-) bisimilar iff G_1 and G_2 are bisimilar with respect to the delay (resp. observational) bisimulation.

The first approach is useful for reducing furthermore the size of the quotient graph, while preserving a set of properties (e.g., linear properties). The second approach is applied for *behavioral verification*, where the specification of a system A_1 is not given in terms of a formula, but in terms of another system A_2. Then, it is required that A_1 should behave "like" A_2, where the notion of "like" is captured formally by a bisimulation or simulation. Typically, A_1 is the TA modeling the system and A_2 is a (timed or untimed) automaton corresponding to the specification.

Instead of the delay or observational bisimulations, a number of other untimed bisimulations or simulations can be also used, such as the *safety* bisimulation [Mou93]. Although we have not thoroughly studied the properties preserved when combining such relations with STaBs, we have often used them in practice, to get a better reduction of the model and gain intuition in its behavior.

Regarding the algorithms used in approaches 1 and 2 above, minimization with respect to untimed bisimulations can be done using the MMGA or the algorithm of figure 6.1. Comparison of two graphs for bisimilarity can be also reduced to minimizing their union graph, and then checking whether their initial nodes belong to the same class. A more efficient on-the-fly comparison method has been proposed in [FM91]. All these algorithms have been implemented in the module `aldebaran`, part of the untimed-verification tool-suite CADP. Figure 11.3 in section 11.2 summarizes the connection of `minim` to CADP. Sections 12.1 and 12.2 provide examples of the combination of time-abstracting and untimed bisimulations for verification.

Relation to the literature

A straightforward adaptation of the algorithm of [BFH+92], for the time-abstracting delay bisimulation is given in [ACH+92]. Another generic minimization algorithm is proposed in [LY92], and is adapted for timed systems in [YL93]. Our algorithm is inspired from both above works, in particular, the idea to avoid complementation is borrowed from [YL93], however, our solution is different. Experimental results from an implementation of the algorithms of [ACH+92, YL93] are given in [ACD+92]. In section 12.1 we compare these results to ours.

Chapter 7

On-the-fly Verification

In this chapter we propose verification techniques based on the simulation graphs introduced in section 5.2. These techniques are entirely on-the-fly, that is, the property is checked while the graph is being generated. They can be applied for discrete-state reachability, deadlock and timelock detection, TBA model-checking and full ETCTL$^*_\exists$ model-checking.

The algorithms for reachability and TBA model-checking have been implemented in KRONOS (chapter 11) and used in a number of case studies (chapter 12).

7.1 Reachability

Reachability is the most widely used type of analysis, since, first, it is sufficient for expressing a large class of interesting properties such as invariance and bounded-response, and second, it is not expensive to implement.

For simplicity, we consider reachability of discrete states of a TA A. This is not less general than checking reachability of a set of states of A specified by a zone (q, ζ): we can always transform A to a new TA A' with an extra discrete state q' and an extra edge $(q, \zeta, \_, \emptyset, q')$, and check whether q' is reachable in A'.

More precisely, if Q is the set of discrete states of A, given a set of *target* discrete states $\hat{Q} \subseteq Q$, we consider three problems of reachability:

- *Yes/No reachability*: check whether there exists $(q, \mathbf{v}) \in \mathit{Reach}(A)$, such that $q \in \hat{Q}$.
- *Partial reachability*: given a set of initial states represented as a zone S, find $S' \subseteq S$, such that every state in S' can reach some target state $(q, \mathbf{v}), q \in \hat{Q}$.
- *Total reachability*: given a set of initial states represented by a zone S, find the largest subset $S' \subseteq S$, such that no state in S' can reach any target state $(q, \mathbf{v}), q \in \hat{Q}$.

Yes/No reachability, examined in section 7.1.1, can be applied to verify properties of the form $\exists\Diamond\, p$, where p is an atomic proposition [1]. It can also be applied to deadlock detection.

Partial and total reachability, examined in section 7.1.2, are used as intermediate steps for verifying properties of the form $\forall\Box\, \exists\Diamond\, p$. More precisely, partial reachability can be iteratively

[1] Formally speaking, none of the specification languages defined in chapter 4 can generally capture reachable states. This is because the latter are defined by finite runs, whereas formalisms such as TBA and TCTL are evaluated over infinite (in fact, non-zeno) runs. For instance, if all states of a TA A where p holds are deadlocks, then $\exists\Diamond\, p$ is not satisfied by A, although some of these states might be reachable.

applied for solving total reachability. Then, a nested application of yes/no reachability and total reachability is used for model checking $\forall\Box\,\exists\Diamond\, p$. As an application, we show how this technique can be used for timelock detection.

All algorithms presented in this section are based on a depth-first generation and exploration of a simulation graph, possibly combined with the activity, inclusion or convex-hull abstractions, or their combinations (section 5.2). In the sequel, we use α to denote the abstraction used; α can be the identity function $\mathbf{1}$ (if we are using just the simulation graph), or one of α_{act}, α_{inc}, α_{ch}, $\alpha_{act} \circ \alpha_{inc}$ and $\alpha_{act} \circ \alpha_{ch}$.

7.1.1 Yes/No reachability

The algorithm for yes/no reachability is shown in figure 7.1. It is based on a depth-first (DF) procedure `Reach`$^{\alpha}$, parameterized by the abstraction α used. Depending on α, functions post$^{\alpha}$, `store`$^{\alpha}$, `is_visited`$^{\alpha}$ and `return`$^{\alpha}$ are instantiated differently, and this is how the α-graph is generated on-the-fly.

The instantiations are shown in the lower part of the figure. Activity is implemented by coupling α_{act} to the post function. Inclusion and convex hull are implemented by managing differently the set of visited nodes *Visit*. Given a successor S' of the current node, the test `is_visited` decides whether S' is to be considered a new node or not. Normally, S' is new if it is not in *Visit*, except when inclusion or convex hull is used: in that case, S' is new if it is not a subset of a node already in *Visit*.

When a new node is found, procedure `store` updates the set *Visit*. The new node is inserted in *Visit*, except when convex hull is used: in that case, a single node (q, ζ) is kept per discrete state q and when a new node (q, ζ') is found, (q, ζ) is replaced by $(q, \zeta \sqcup \zeta')$.

`return`$^{\alpha}$ gives a non-conclusive answer in case convex hull is applied.

It is worth making the following remarks.

- The algorithm can be easily extended to provide diagnostics. The DF procedure is often implemented using a stack to eliminate recursion. During execution of the algorithm, the stack contains the symbolic path currently explored. From such a path, we can extract more detailed diagnostics (i.e. a finite run) as shown in chapter 8.

- In the case of a hit during a convex-hull search, we can do better than just return a "maybe" answer. Since the stack contains the sequence of edges $e_1, ..., e_l$ forming the path currently explored, we can generate the exact zone path corresponding to these edges, using normal post operations. If the whole path can be generated (up to l), then $\hat{Q}$ is indeed reachable even in the simulation graph. Otherwise, the search can continue. If only non-valid path are found during execution of the algorithm, a "maybe" answer is returned at the end. Indeed, a "no" answer cannot be returned since valid paths might exist, but are missed in the convex-hull search.

- The inclusion abstraction computed is not necessarily optimal. Indeed, given two zones $S_1 \subseteq S_2$, we cannot tell in advance which one is going to be visited first: this depends on the DFS order, which is a-priori unknown. If S_1 is visited after S_2, it will be merged with the latter, otherwise, both zones will be present in the inclusion graph.

The yes/no reachability algorithm has been implemented in KRONOS (see chapter 11) and used in a number of case studies, like the ones presented in sections 12.3 and 12.4.

```
YesNoReach^α (S_1, Q̂) {
  Visit := {} ;
  c := c_max(A) ;
  return Reach^α (S_1) ;

  Reach^α (S) {
    if (discrete(S) ∈ Q̂) then return^α ;
    store^α (S) ;
    for each (e ∈ out(q)) do
      S' := post^α(e, S, c) ;
      if (S' ≠ ∅ and not is_visited^α(S')) then
        Reach^α (S') ;
    end for each
    return "No" ;
  }
}
```

$$\texttt{is\_visited}^{\alpha}((q,\zeta)) \stackrel{\text{def}}{=} \begin{cases} (q,\zeta) \in \mathit{Visit}, & \text{if } \alpha \in \{\mathbf{1}, \alpha_{act}\} \\ \exists (q,\zeta') \in \mathit{Visit} \,.\, \zeta \subseteq \zeta', & \text{otherwise} \end{cases}$$

$$\mathsf{post}^{\alpha} \stackrel{\text{def}}{=} \begin{cases} \alpha_{act} \circ \mathsf{post}, & \text{if } \alpha \in \{\alpha_{act}, \alpha_{act} \circ \alpha_{inc}, \alpha_{act} \circ \alpha_{ch}\} \\ \mathsf{post}, & \text{otherwise} \end{cases}$$

$$\texttt{store}^{\alpha}((q,\zeta)) \stackrel{\text{def}}{=} \begin{cases} \mathit{Visit} := \mathit{Visit} \setminus \{(q,\zeta')\} \\ \qquad\qquad \cup \{(q, \zeta \sqcup \zeta')\}, & \text{if } \alpha \in \{\alpha_{ch}, \alpha_{act} \circ \alpha_{ch}\} \\ \mathit{Visit} := \mathit{Visit} \cup \{(q,\zeta)\}, & \text{otherwise} \end{cases}$$

$$\texttt{return}^{\alpha} \stackrel{\text{def}}{=} \begin{cases} \texttt{return "Maybe"}, & \text{if } \alpha \in \{\alpha_{ch}, \alpha_{act} \circ \alpha_{ch}\} \\ \texttt{return "Yes"}, & \text{otherwise} \end{cases}$$

Figure 7.1: An abstraction-parameterized algorithm for Yes/No reachability.

Application: deadlock detection

Consider a TA $A = (\mathcal{X}, Q, q_0, E, \mathsf{invar})$. Checking that A is deadlock-free can be reduced to yes/no reachability in a new TA $A' = (\mathcal{X}, Q \cup \{q'\}, q_0, E \cup E', \mathsf{invar}')$, where $q' \notin Q$ and:

- For each $q \in Q$, and each $\zeta \in \mathsf{convex}(\overline{\mathsf{free}(q)})$, E' contains an edge $(q, \zeta, a, \emptyset, q')$.
- $\mathsf{invar}'(q) = \mathsf{invar}(q)$, if $q \in Q$, and $\mathsf{invar}'(q') = \mathsf{true}$.

Intuitively, the above construction adds a number of out-going edges to each discrete state q. These auxiliary edges serve as *escape* actions: whenever the automaton reaches a deadlock state (where in A it would normally block) it can take an escape transition and move to the error state q'. By lemma 3.4, it is easy to see that A is deadlock-free iff q' is reachable in A'.

7.1.2 Partial and total reachability

For these two problems, we cannot use convex-hull abstractions, since they only preserve reachable states conservatively. Thus, throughout this section, α denotes any abstraction except the convex-hull abstractions α_{ch} and $\alpha_{act} \circ \alpha_{ch}$.

Partial reachability by pre-stabilization

The yes/no reachability algorithm can be modified to solve partial reachability. Let S_1 be an initial zone and let $\texttt{Reach}^{\alpha}(S_1)$ be successful, finding a path $\pi = S_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{l-1}} S_l$ hitting the target states. We define the zones S'_i, for $i = 1, ..., l$:

$$\begin{aligned} S'_l &\stackrel{\text{def}}{=} S_l \\ S'_i &\stackrel{\text{def}}{=} S_i \cap \mathsf{pre}(e_i, S'_{i+1}), \qquad i = l \Leftrightarrow 1, ..., 1 \end{aligned}$$

By definition of $\mathsf{pre}()$, every state in S'_1 leads to some state in S_l. Moreover, S'_1 cannot be empty, due to post-stability of π. Thus, S'_1 is indeed an answer to partial reachability. The technique is called *pre-stabilization* and S'_1 is denoted $\mathsf{pre\text{-}stable\text{-}root}(\pi)$. We call $\texttt{Partial\_Reach}^{\alpha}(S_1, \hat{Q})$ the procedure obtained by modifying `Reach`, so that it returns $\mathsf{pre\text{-}stable\text{-}root}(\pi)$ if it finds a path π reaching the target states, and $\emptyset$ otherwise.

Total reachability

`Partial_Reach` does not solve the problem of total reachability: although no state $s \in S_1 \setminus S'_1$ can reach $\hat{Q}$ following a run $\xrightarrow{e_1} \cdots \xrightarrow{e_{l-1}}$, there might exist another run starting from s and reaching $\hat{Q}$. Consequently, in order to make sure that no states in $S_1 \setminus S'_1$ can reach $\hat{Q}$, we have to repeat the partial reachability procedure starting from $S'' = S_1 \setminus S'_1$. Since S'' is not necessarily a zone, it has to be "split" into a number of zones and `Partial_Reach` has to be called separately for each one of them.

The complete algorithm for total reachability is shown in figure 7.2. It takes as input an initial zone S_1 and a set of target discrete states $\hat{Q}$ and returns the set of states in S_1 which cannot reach $\hat{Q}$. This set (not necessarily convex) is represented as a set of zones $\mathcal{Z}$. For notational convenience, we write $\mathsf{convex}((q, \zeta))$ instead of $\{(q, \zeta') \mid \zeta' \in \mathsf{convex}(\zeta)\}$.

```
TotalReach^α (S_1, Q̂) {
  Z := {S_1} ;
  c' := c_max(A) ;
  while (∃S = (q, ζ) ∈ Z) do
    Visit := {} ;
    c := max{c', c_max(ζ)} ;
    S' := Partial_Reach^α(S, Q̂) ;
    Z := (Z \ {S}) ∪ convex(S \ S') ;
  end while
  return Z ;
}
```

Figure 7.2: An algorithm for total reachability.

Application: timelock detection

Timelocks can be detected using a nested reachability: at the outer level, a yes/no reachability is performed to generate the abstract graph node by node; for each node S, an inner-level total reachability is performed, to compute the subset of S where time is blocked. The algorithm is inspired from the fact that timelock-freedom is equivalent to model-checking the TCTL formula $\forall\Box\,\exists\Diamond_{\geq 1}\,\mathsf{true}$ (see section 6.2.4).

More precisely, consider a TA $A = (\mathcal{X}, Q, q_0, E, \mathsf{invar})$. Define the TA $A' = (\mathcal{X} \cup \{z\}, Q \cup \{q'\}, q_0, E \cup E', \mathsf{invar}')$, where $z \notin \mathcal{X}$, $q' \notin Q$ and:

- For each $q \in Q$, E' contains an edge $(q, z \geq 1, a, \emptyset, q')$.
- $\mathsf{invar}'(q) = \mathsf{invar}(q)$, if $q \in Q$, and $\mathsf{invar}'(q') = \mathsf{true}$.

In words, A' has an escape edge from each discrete state of A leading to an auxiliary sink state q', provided that the value of the auxiliary clock z is at least 1.

The algorithm for timelock-detection is shown in figure 7.3. The outer-most reachability procedure, `TimelockReach`, is identical to procedure `Reach` of figure 7.1, except that the set of target states $\mathcal{Z}$ is generated dynamically. $\mathcal{Z}$ is a set of zones $\{S_1, ..., S_k\}$, such that each S_i is a subset of the current zone S, and their union contains all states of S from which not even one time unit can elapse. $\mathcal{Z}$ is built using the inner-most total reachability. Notice that the outer-most reachability is performed on A, while the inner-most reachability is performed on A'.

7.2 Timed Büchi Automata Emptiness

By lemma 4.2, checking whether a TA A satisfies a TBA B comes down to checking language emptiness of $A \times B$. In this section we show how TBA emptiness can be checked on-the-fly on abstract graphs, based on the preservation results of section 5.2. In the algorithms presented in the sequel, we assume that the abstraction used is either the zone or activity graph, since they preserve linear properties in an exact manner (lemmas 5.9 and 5.10). The same algorithms can be used on the inclusion or convex-hull graphs in a conservative manner: if the abstract

```
TimelockDetect^α (S_1) {
  Visit := {} ;
  c := c_max(A) ;
  return TimelockReach^α (S_1) ;

  TimelockReach^α (S) {
    Z := TotalReach^α_{A'}(S ∩ z = 0, {q'}) ;
    if (Z ≠ ∅) then return "Timelock found" ;
    Visit := Visit ∪ {S} ;
    for each (e ∈ out(q)) do
      S' := post^α(e, S, c) ;
      if (S' ≠ ∅ and not is_visited^α(S')) then
        TimelockReach^α (S') ;
    end for each
    return "No timelocks" ;
  }
}
```

Figure 7.3: Nested reachability for timelock detection.

graph is empty, then the concrete TBA has an empty language, otherwise, no conclusion can be made.

We distinguish four cases altogether, depending on whether the system is strongly non-zeno and/or has trivial acceptance conditions. Strong non-zenoness dispenses us with the burden of checking time progress, thus, we are left with checking discrete acceptance conditions. This can be generally done using a maximal-SCC search, while in the case of trivial acceptance conditions, a simple DFS suffices.

In the general case of systems which are not strongly non-zeno, checking time-progress is reduced to a search for non-maximal SCCs, similar to the algorithms for strong fairness proposed in [EL85, LP85] [2]. This search can be expensive, thus, we also propose sound but generally incomplete algorithms based on a simple DF search.

In sections 7.2.1 and 7.2.2 we give the algorithms for TBA emptiness. These are essentially yes/no algorithms, which can be extended to provide diagnostics. In section 7.2.3 we prepare the ground for ETCTL$^*_\exists$ model-checking (presented in section 4.3). Using a pre-stabilization technique similar to the one for partial reachability, we extend the yes/no emptiness algorithm to compute all states leading to runs in the language of the TBA.

7.2.1 Special case: strongly non-zeno TBA

Consider a strongly non-zeno TBA B with set of repeating states F. Let G be the zone or activity graph of B. A cycle of G is called accepting if it contains a repeating node.

Lemma 7.1 *A strongly non-zeno TBA B has a non-empty language iff its simulation or activity graph G has an elementary accepting cycle.*

[2] Recently, more efficient algorithms have been proposed in [HT96].

```
StrongNonZenoness_TrivialAcceptance_Emptiness^α (S_1) {
  Visit := {false} ;
  Stack := {} ;
  c := c_max(B) ;
  return ReachCycle^α (S_1) ;

  ReachCycle^α (S) {
    Visit := Visit ∪ {S} ;
    for each (e ∈ out(discrete(S))) do
      S' := post^α(e, S, c) ;
      if (S' ∉ Visit) then
        Stack := Stack ∪ {S'} ;
        ReachCycle^α (S') ;
        Stack := Stack \ {S'} ;
      else if (S' ∈ Stack and ∀(q, ζ) ∈ Stack . q ∈ F) then
        return "B is non-empty" ;
      end if
    end for each
    return "B is empty" ;
  }
}
```

Figure 7.4: A DFS for strongly non-zeno, trivial-acceptance TBA emptiness.

Proof: By lemma 3.2, all runs of B are non-zeno, thus, by lemma 5.9, every infinite path in G is non-zeno. Since G has a finite number of nodes, B is non-empty iff G has an accepting cycle, from which we can always extract an elementary sub-cycle which is also accepting. ■

We present two algorithms for elementary accepting cycles, depending on whether the acceptance conditions are trivial or not. In the first case, a simple DFS suffices. Otherwise, a maximal-SCC search or the doubly-nested DFS of [CVWY92] has to be used. Although the three algorithms have the same worst-case complexity (linear in the size of G), the DFS algorithms are preferable, since they can be implemented with a lower memory cost and can usually provide an answer faster.

Trivial acceptance

In this case we are looking for an elementary cycle visiting nothing but repeating nodes. Such a cycle can be found using the DFS of figure 7.4. To see that no accepting cycle is missed, let $\lambda = S_1 \stackrel{e_1}{\rightarrow} \cdots S_l \stackrel{e_l}{\rightarrow} S_1$ be such a cycle. Without loss of generality, we assume that S_1 is the first node visited, with respect to depth-first order and that λ is elementary (if not, an elementary subcycle λ' of λ can be extracted and all nodes of λ' are also repeating). Then, if λ is missed, this is due to the DFS stopping upon an already visited node S_i for some $1 < i \leq l$. But this means that S_i is visited before S_1 which contradicts the hypothesis.

General acceptance

In this case we are looking for a cycle visiting at least one repeating node. Such a cycle cannot be generally found with a simple DFS. For example, consider the TBA B_1 of figure 7.5. The simulation graph of this automaton is isomorphic to its discrete structure, that is, has four zones $(i, x \leq 1)$, for $i = 1, 2, 3, 4$. Assume that the nodes are visited in this order during the DFS: $1, 2, 3, 4$ (the order of visit is arbitrary, depending on the input format). Then, the accepting cycle $1 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 1$ is missed, because when exploring the successor 2 of 4 the search stops since finding that 2 is already visited.

The solution is to use a search for a maximal SCC containing at least one repeating node, or the *double-DFS* of [CVWY92] looking for a repeating node that has a cycle back to itself. The complexity of both algorithms is linear in the size of the graph. In practice, the double-DFS algorithm can be implemented efficiently using an extra bit per stored state, to indicate whether the state has been visited only in the outer DFS or also the inner one [Hol91]. Thus, the "real" cost of the algorithm is in the worst case twice the cost of a simple DFS, in time and memory. The SCC algorithm is usually more costly in practice.

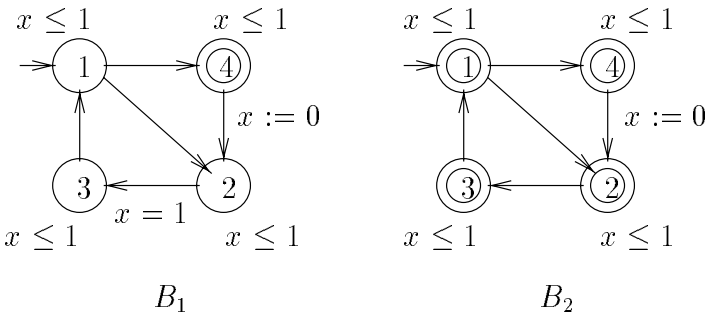


Figure 7.5: A DFS can sometimes miss an accepting cycle (B_1) or a non-zeno cycle (B_2).

7.2.2 General case

If the system is not strongly non-zeno, we have to find an infinite path π satisfying two progress requirements, namely, discrete acceptance and non-zenoness. Since G has a finite number of nodes, finding π is reduced to a search for an accepting non-zeno cycle in G [3]. Two questions must be answered. First, do elementary cycles suffice ? Second, if they do suffice, how can they be found ? Elementary cycles are important, since they are easier to find. However, as we shall see below, elementary cycles suffice only in the case of trivial acceptance. Moreover, even in that case, a simple DFS is not enough since it does not generally find all elementary cycles.

Incomplete algorithms

The idea is to modify the algorithms of the previous section, so that each time an accepting cycle is found, it is tested for non-zenoness, using the syntactic conditions of page 58. If the cycle is non-zeno then a counter-example to the emptiness of the TBA has been found, otherwise the search continues.

[3] Non-zeno cycles are defined syntactically, like non-zeno paths, since a cycle defines an infinite path.

Unfortunately, this gives sound but incomplete algorithms: if a non-zeno accepting cycle is found then the TBA is non-empty; if no accepting cycle is found then the TBA is empty; otherwise (i.e., if only zeno accepting cycles are found) no conclusion can be made. For example, assume a DFS on the TBA B_2 of figure 7.5 visits nodes in the order $1, 2, 3, 4$. Then, the single non-zeno cycle $1 \to 4 \to 2 \to 3 \to 1$ is missed during the search [4].

Although incomplete, these algorithms are still worth applying, since they are much less expensive than complete ones, as we shall see below.

Complete algorithms

Let B be a TBA and G be its zone or activity graph. A SCC G' of G is called accepting if it contains at least one repeating node. G' is called non-zeno if for each clock x of $A \times B$, either x is reset in some edge of G' or x is unbounded in all nodes of G'.

Lemma 7.2 *A (general) TBA B has a non-empty language iff its simulation or activity graph G has a non-zeno, accepting SCC.*

The proof of the above result is similar to the one of lemma 6.3, with an important difference, however: since pre-stability is not a property of G, we cannot ensure the existence of a maximal non-zeno SCC. Since the number of general (non-maximal) SCCs in a graph can be exponential, it is not efficient to check every one of them.

In the paragraphs that follow, we try to remedy this by giving three different sound and complete algorithms for emptiness:

- An adaptation of the *strong-fairness* model-checking algorithm of [EL85, LP85]: the algorithm finds all accepting maximal SCCs, and computes for each one of them the greatest (possibly empty) non-zeno sub-SCC.
- A *full* DFS: the algorithm finds all elementary accepting cycles, and checks each one of them for non-zenoness. The method works only for trivial acceptance.
- An algorithm based on *weak fairness*: it transforms B to a new TBA where every accepting run is guaranteed to be also non-zeno. Thus, the problem is reduced to the strong non-zenoness case.

A strong-fairness SCC algorithm. This is a straightforward application of lemma 7.2. Using the algorithm of [Tar72], we construct on-the-fly the maximal SCCs of G. For each maximal SCC $(V, \to)$, we call procedure `AcceptingNonZenoSCC` of figure 7.6, which returns the greatest accepting and non-zeno SCC contained in $(V, \to)$, if it exists, otherwise returns $\emptyset$.

The algorithm works recursively. First, it removes all nodes and edges which can "block" a clock x not reset by any edge in $\to$. If nothing is removed, this implies that x is unbounded in the SCC, thus, the latter is non-zeno. Otherwise, the maximal SCCs of the new graph are found and the procedure is called recursively. Since the complexity of finding maximal SCCs is linear in the size of the graph and at least one node is removed at each recursive search, the algorithm's complexity is quadratic in the size of the quotient graph.

[4] We should point out that, although in this example the double-DFS of [CVWY92] would find the non-zeno cycle, this is not the case in general.

```
AcceptingNonZenoSCC ((V, →)) {
  if (V does not contain any repeating node) then
    return ∅ ;
  else if (∃x . ∀ -e→ . x ∉ reset(e)) then
    V' := V \ {(q, ζ) ∈ V | ¬unbounded(x, ζ)} ;
    →' := → \{-e→| ¬unbounded(x, guard(e))} ;
    if (V' = V ∧ →'=→) then return (V, →) ;
    for each (maximal SCC G' of (V', →')) do
      G'' := AcceptingNonZenoSCC (G') ;
      if (G'' ≠ ∅) then
        return G'' ;
    end for each
    return ∅ ;
  end if
}
```

Figure 7.6: An algorithm to check whether a SCC is accepting and non-zeno.

A full DFS for trivial acceptance. The problem with the incomplete algorithms is that a DFS which stops whenever a visited node is encountered does not find all cycles. This can be remedied by using a DFS which stops only when a newly-created node is already in the stack. In this case, all elementary cycles are found during the search. The modified search is shown in figure 7.7 (we assume that the procedure is-non-zeno() performs the syntactic test for non-zenoness of a cycle).

The correctness of this algorithm is based on the lemma below (the proof is given in the appendix).

Lemma 7.3 *A graph has a non-zeno cycle iff it has an elementary non-zeno cycle.*

Lemma 7.4 *A (general) TBA B with trivial acceptance condition is non-empty iff its simulation or activity graph has an elementary non-zeno cycle visiting exclusively repeating nodes.*

Proof: A corollary of lemmas 7.2 and 7.3. ∎

Notice that lemma 7.4 does not hold for non-trivial acceptance. A counter-example is shown in figure 7.8. The figure presents a TBA (left) and its simulation graph (right). Neither of the two elementary cycles in the simulation graph is both accepting and non-zeno, although their combination is.

Since the full-DFS of figure 7.7 finds all elementary non-zeno cycles, it is sound and complete for trivial-acceptance TBA. The algorithm has a quite high time complexity: the number of elementary cycles is exponential in the number of nodes in a graph. On the other hand, the algorithm has a low memory cost: only a stack is used to store the path currently explored.

A weak-fairness solution. We reduce the problem to the strong-non-zenoness case, by transforming the TBA B to a TBA B' such that all accepting runs of B' are non-zeno [5]. More

[5] The construction of B' is inspired from the *fair product* construction of BA [Cho74, Hol91].

```
TrivialAcceptance_Emptiness^α (S_1) {
  Stack := {} ;
  c := c_max(B) ;
  return ReachCycle^α (S_1) ;

  ReachCycle^α (S) {
    for each (e ∈ out(discrete(S))) do
      S' := post^α(e, S, c) ;
      if (S' = false) then continue ;
      if (S' ∉ Stack) then
        Stack := Stack ∪ {S'} ;
        ReachCycle^α (S') ;
        Stack := Stack \ {S'} ;
      else if (∀(q, ζ) ∈ Stack . q ∈ F and is-non-zeno(Stack)) then
        return "B is non-empty" ;
      end if
    end for each
    return "B is empty" ;
  }
}
```

Figure 7.7: A full DFS for trivial-acceptance TBA emptiness.

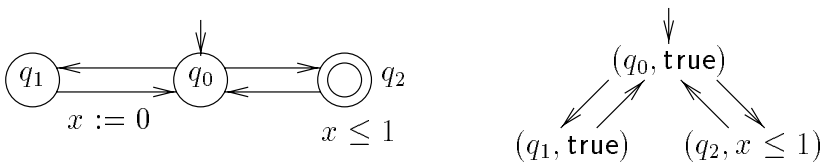


Figure 7.8: Elementary cycles do not suffice for non-trivial acceptance conditions.

precisely, let $B = (\mathcal{X}, Q, q_0, E, \mathsf{invar}, F)$. Then B' is defined to be the TBA $(\mathcal{X} \cup \{z\}, Q \times \{0,1\}, (q_0, 0), E', \mathsf{invar}', F \times \{1\})$, where z is an auxiliary clock not in $\mathcal{X}$ and:

- For each edge $(q, \zeta, a, X, q') \in E$, E' has an edge $((q,0), \zeta, a, X, (q',0))$. If $q \in F$, then E' has also the edges $((q,0), z \geq 1, \mathsf{tp}, \{z\}, (q,1))$ and $((q,1), \zeta, a, X, (q',0))$.
- $\mathsf{invar}'(q,i) = \mathsf{invar}(q)$, for $i = 0, 1$.

Intuitively, each discrete state is extended with a boolean flag to memorize whether time has progressed or not. When the system is on a repeating state $q \in F$, a *time-progress* transition (tp) can be issued, setting the flag. Any outgoing transition from q unsets the flag. The repeating states of B' are of the form $(q,1)$, where $q \in F$.

Then, any run visiting $(q,1)$ infinitely often must also execute tp-transitions infinitely often: since at least one time unit passes between two successive tp-transitions, the run is non-zeno. Also notice that any accepting non-zeno run of B can be transformed to an accepting run of B' by "inserting" tp-transitions every now and then. Thus, we have the following result.

Lemma 7.5 *A TBA B has a non-empty language iff there is an elementary accepting cycle in the simulation or activity graph of B'.*

The techniques of section 7.2.1 can be used to find accepting cycles in the abstract graph of B'.

7.2.3 Computing states leading to accepting non-zeno runs

On-the-fly ETCTL$^*_\exists$ model checking is based on an extension of the TBA-emptiness algorithms presented so far: instead of just checking whether a TBA B has an empty language, we compute states which lead to runs in the language of B. More precisely, given an initial zone S, we are interested in computing a subset (resp. the largest subset) $S' \subseteq S$, such that from each state $s \in S'$ there is a run in $Lang(B)$. We refer to the two problems as partial and total TBA-emptiness, respectively.

Similar to the case of partial and total reachability, the solution lies in the pre-stabilization technique, adapted for cycles. More precisely, let $\lambda = S_1 \xrightarrow{e_1} \cdots S_l \xrightarrow{e_l} S_1$ be a cycle in the zone or activity graph of B. We define the following system of equations, for $i = 1, ..., l$:

$$S'_i \quad = \quad S_i \cap \mathsf{pre}(e_i, S'_{i+1})$$

where addition is taken modulo l. Based on lemma 5.9, we can prove that the above system of equations has a greatest fix-point where none of the zones S'_i is empty.

This fix-point defines a sequence $\lambda' = S'_1 \xrightarrow{e_1} \cdots S'_l \xrightarrow{e_l} S'_1$ such that S'_i is both pre-stable with respect to S'_{i+1} and post-stable with respect to S'_{i-1}. Thus, from every state in S'_1 there exists an infinite run ρ inscribed in $(\lambda')^\omega$.

Then, procedure `Partial_Emptiness` works as follows:

- Apply one of the emptiness algorithms of sections 7.2.1 or 7.2.2 for finding non-zeno accepting cycles.
- Each time a path ending in a cycle is found, $\pi = \pi'\lambda$, pre-stabilize λ, then pre-stabilize π' with respect to the pre-stable root of λ.
- Return $\mathsf{pre\text{-}stable\text{-}root}(\pi)$.

Procedure `Total_Emptiness` is defined similarly to `Total_Reach`: given an initial zone S, it repeatedly calls `Partial_Emptiness` to compute $S' \subseteq S$, then $S'' \subseteq S \setminus S'$, and so on, until no more states in S can lead to accepted runs.

7.3 On-the-fly model checking of ETCTL$^*_\exists$

ETCTL$^*_\exists$ model checking is performed by a recursive procedure etctl-eval, which takes as input a TA A, an initial symbolic state S of A and an ETCTL$^*_\exists$ formula ϕ, and returns the greatest subset $S' \subseteq S$, such that all states in S' satisfy ϕ. The non-trivial case comes when ϕ is of the form $\exists B(\phi_1, ..., \phi_m)$, where an adaptation of the procedure for total TBA-emptiness is applied on the product $A \times B$. Compared to the `Total_Emptiness` procedure of section 7.2.3, the generation of the abstract graph of $A \times B$ differs in two points:

1. The post() operator is modified to take into account satisfaction of the sub-formulae $\phi_1, ..., \phi_m$ of B. More precisely, when computing a successor of S as $S' = \mathsf{post}(e, S, c)$, we keep only those states in S' which satisfy the sub-formula corresponding to the discrete state of B in S'. This is done by a recursive call to etctl-eval.

2. Due to point 1, the nodes of the abstract graph are no longer zones, but have the form (q, ζ), where ζ can be a non-convex polyhedron. This does not affect any of the definitions or preservation properties of abstract graphs.

In order to introduce the modified post() operator, we need a definition first. Given two symbolic states S, S' of a TA, we define the dual of the until operator:

$$\mathsf{since}(S', S) \stackrel{\text{def}}{=} \{s \in S \mid \exists \delta \in \mathsf{R} \,.\, s \Leftarrow\!\delta \in S' \wedge \forall \delta' < \delta \,.\, s \Leftarrow\!\delta' \in S' \cup S\}$$

Intuitively, $\mathsf{since}(S', S)$ contains all states of S which can be reached by S' by letting time pass, continuously staying in S'.

The model-checking algorithm is shown in figure 7.9. When a formula of the form $\exists B(\phi_1, ..., \phi_m)$ is to be model-checked, the abstract graph of $A \times B$, say G, is explored. If $\mathcal{X}_A$ and $\mathcal{X}_B$ are the sets of clocks of A and B, then the nodes of G have the form (q, q_i, ζ), where q and q' are discrete states of A and B, and ζ is a polyhedron on $\mathcal{X}_A \cup \mathcal{X}_B$.

S is a symbolic state of A. S_1 is the initial node of G, where all clocks of B are initialized to zero. The result of the search is also a symbolic state of $A \times B$, thus, it has to be projected on the state space of A before being returned.

The procedure `Total_Emptiness` uses as successor operator the function etctl-post(). Given a symbolic state S, etctl-post() computes the symbolic successors of S, S_2, as before. Now, S_2 has the form (q, q_i, ζ), for some $i = 1, ..., m$. Not all states in S_2 satisfy the sub-formula ϕ_i specified by B (ϕ_i is denoted sub-formula(S) in figure 7.9). To compute these states, etctl-eval is called recursively. Finally, the since() operator is called to eliminate all states in S_3 which cannot be continuously reached by an e-successor of S.

Relation to the literature

On-the-fly verification on the simulation graph has been introduced independently in [DOY94, TC96], These works consider only safety properties such as invariance and bounded response, which are reduced to reachability. To our knowledge, simulation graphs have not been previously used for deadlock or timelock detection, neither for checking emptiness. ETCTL$^*_\exists$ model-checking has been first presented in [BTY97], along with experimental results on the FDDI protocol [Jai94] comparing the fix-point TCTL model-checking algorithm of KRONOS with the on-the-fly algorithm. The formula verified was of the form $\forall\Diamond\, p$ ("inevitably p holds"). The results are shown in table 7.1 (n is the number of processes, $\perp$ stands for "out-of-memory" and

```
etctl-eval (S, φ) {
  case (φ)
    true :            return S ;
    p :               return {(q, v) ∈ S | q ∈ P(p)} ;
    ¬φ1 :             return etctl-eval(S, φ1) ;
    φ1 ∨ φ2 :         return etctl-eval(S, φ1) ∪ etctl-eval(S, φ2) ;
    ∃B(φ1, ..., φm) : Visit := {} ;
                      Stack := {} ;
                      c := max{c_max(A), c_max(B)} ;
                      S1 := S ∩ ⋂_{x∈X_B} x = 0 ;
                      (q, q', ζ) := Total_Emptiness(A × B, S1) ;
                      return (q, ζ⌋X_A) ;
  end case
}

etctl-post(e, S, c) {
  S1 := disc-succ(e, S) ;
  S2 := post(e, S, c) ;
  let S2 = (q, qi, ζ) ;
  S3 := etctl-eval((q, ζ⌋X_A), φi) ∩ S2 ;
  return since(S1, S3) ;
}
```

Figure 7.9: ETCTL$^*_\exists$ model checking.

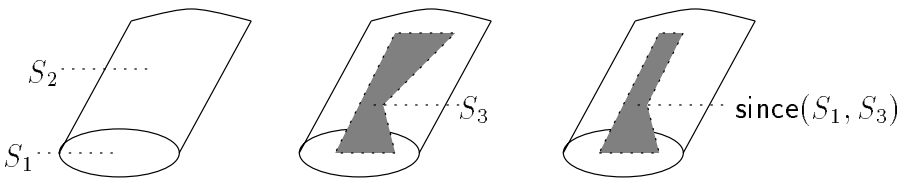


Figure 7.10: Illustration of the etctl-post() operator.

n	fix-point	on-the-fly		
		symbolic states	time (sec)	memory (Mbytes)
4	†	1786	53	2
5	†	5834	180	4
6	⊥	18476	1388	9
7	⊥	57538	4813	25.8

Table 7.1: Experimental results on FDDI.

† sign stands for "out-of-time-and-patience" – more than 1 hour of computation). Notice that the type of formula verified is the most expensive to compute using the fix-point method. On the other hand, the on-the-fly technique is also less favored in this case, where the property holds and the whole state-space is generated.

[SS95] propose a *local* model-checking algorithm for strongly non-zeno systems with respect to *timed μ-calculus*. The algorithm works by generating on-the-fly a symbolic graph, however, it is necessary to refine the nodes of the graph, which involves back-tracking. No experimental results are presented. An on-the-fly algorithm based on the region graph is given in [HKV96].

Chapter 8

Diagnostics

When checking a system against a property, a simple yes/no answer is often not satisfactory. *Diagnostics* are any kind of supplementary information (for instance, states, executions or sets of these) which helps the user understand why verification fails or succeeds. Diagnostics are important for the following reasons:

- Without them no confidence in the system's model can be gained. For instance, in case the property is not satisfied by the model, it might be that it is not the system which is wrong, but the modeling.
- Even if the model is correct, the fault of the system cannot be easily located without any guidance.

In the case of TA, there is a need for *timed* diagnostics, containing information both about the discrete state changes of the system, as well as the exact time delay between two discrete transitions. These delays can be essential to the understanding of a sample behavior of the system.

The algorithms presented in the previous chapters work on abstract graphs, therefore, they can only provide diagnostics in the form of symbolic paths. In this chapter we show how timed diagnostics can be computed for finite or infinite linear-time properties. Timed diagnostics are given in one of the equivalent forms of runs or *timed trails* (defined below). We treat the cases of finite and infinite diagnostics in separate sections, since the techniques for computing them are different. First, we define timed trails.

Timed trails. Consider a TA A and a (finite or infinite) run $\rho = s_0 \stackrel{\delta_0}{\rightarrow}\stackrel{e_1}{\rightarrow} s_1 \stackrel{\delta_1}{\rightarrow}\stackrel{e_1}{\rightarrow} \cdots$. The *timed trail* (trail, for short) corresponding to ρ is the (finite or infinite) sequence $(e_0, \delta_0)(e_1, \delta_0 + \delta_1)\cdots$. That is, the trail keeps count of the global time along the run and records at what time each discrete transition is taken.

The trail contains all the information necessary to reconstruct its run. A trail $(e_0, t_0)(e_1, t_1)\cdots$ corresponds to the run $(q_0, \mathbf{v}_0) \stackrel{\delta_0}{\rightarrow}\stackrel{e_1}{\rightarrow} (q_1, \mathbf{v}_1) \stackrel{\delta_1}{\rightarrow}\stackrel{e_1}{\rightarrow} \cdots$, where $q_0 = \mathsf{source}(e_0)$, $\mathbf{v}_0 = \mathbf{0}$, $\delta_0 = t_0$, and:

$$\begin{aligned} q_{i+1} &= \mathsf{target}(e_i) \\ \mathbf{v}_{i+1} &= (\mathbf{v}_i + \delta_i)[\mathsf{reset}(e_i)] \\ \delta_{i+1} &= t_{i+1} \Leftrightarrow t_i \end{aligned}$$

8.1 Finite runs and trails

In the finite case, timed diagnostics are *extracted* from symbolic paths. That is, our approach is in two steps. First, use a algorithm such as those presented in chapters 6 and 7 to generate a diagnostic path $\pi = S_1 \stackrel{e_1}{\to} \cdots \stackrel{e_{l-1}}{\to} S_l$, where for each $i = 1, ..., l$, S_i is a zone (q_i, ζ_i). Then, build a run $\rho = s_1 \stackrel{\delta_1}{\to}\stackrel{e_1}{\to} \cdots \stackrel{\delta_{l-1}}{\to}\stackrel{e_{l-1}}{\to}\stackrel{\delta_l}{\to} s_l$, such that ρ is inscribed in π. There are two cases to consider:

- If π is pre-stable (e.g., a path of a STa-quotient graph), then the run is built in a forward way: initially we choose $s_1 \in S_1$, then successively we find $\delta_1 \in \mathsf{R}, s_2 \in S_2$ such that $s_1 \stackrel{\delta_1}{\to}\stackrel{e_1}{\to} s_2$, and so on.
- If π is post-stable (e.g., a path of a simulation graph), then the run is built in two passes, first backwards and then forwards:
  - Backward pass: initially we choose $s_l \in S_l$ and then successively find $\delta_i \in \mathsf{R}$, $s_i \in S_i$, for $i = l \Leftrightarrow 1, ..., 1$, such that $s_i \stackrel{\delta_i}{\to}\stackrel{e_i}{\to} s'_{i+1}$, for some s'_{i+1} which is c-equivalent to s_{i+1}.
  - Forward pass: starting from $s_1 \in S_1$, we compute s'_i, for $i = 2, ..., l$, based on δ_i, e_i. The final run is $s_1 \stackrel{\delta_1}{\to}\stackrel{e_1}{\to} s'_2 \stackrel{\delta_2}{\to}\stackrel{e_2}{\to} \cdots s'_l$.

  Intuitively, the backward pass generates an invalid run which might contain some "jumps" among c-equivalent states. The forward pass corrects the run by "adjusting" the clocks which have grown greater than $c_{max}(A)$.

Before describing the forward and backward constructions in detail, we show how choosing a state in a zone (q, ζ) can be done effectively. In fact, this comes down to extracting a valuation $\mathbf{v} \in \zeta$. In the sequel, we assume that the set of clocks is $\mathcal{X} = \{x_1, ..., x_n\}$.

Extracting valuations from polyhedra

An *k-incomplete valuation* is a valuation $\mathbf{v}$ on $\{x_1, ..., x_k\}$. We say that $\mathbf{v}$ can be *completed* in ζ if there exists an $\mathcal{X}$-valuation $\mathbf{v}' \in \zeta$, such that $\mathbf{v}'(x_j) = \mathbf{v}(x_j)$, for all $j \leq k$. Completing $\mathbf{v}$ in ζ means finding such a $\mathbf{v}'$. Notice that we permit $k = 0$, so that completing a 0-incomplete valuation in ζ means extracting a valuation from ζ.

Lemma 8.1 *Given a convex $\mathcal{X}$-polyhedron ζ and a k-incomplete valuation $\mathbf{v}$, it takes $O(n^2)$ time to complete $\mathbf{v}$ in ζ, or find that this is not possible.*

The proof of the lemma is given in section 10.2, since the complexity of the operation generally depends on the data structure used to represent ζ.

Forward pass

It suffices to describe the construction for one step, that is, given a pre-stable edge $S_1 \to S_2$ and $s_1 \in S_1$, how to find $s_2 \in S_2$, such that $s_1 \to s_2$. The construction can then be repeated l times to yield the whole run. The type of the transition $s_1 \to s_2$ depends on the type of the edge $S_1 \to S_2$.

First, consider the case $S_1 \stackrel{e}{\to} S_2$, that is, $S_1 \subseteq \mathsf{disc\text{-}pred}(e, S_2)$. Let $S_i = (q_i, \zeta_i)$, for $i = 1, 2$. Let $\mathbf{v}_1 \in \zeta_1$. By pre-stability, $\mathbf{v}_1$ satisfies $\mathsf{guard}(e)$, thus, if we let $\mathbf{v}_2 = \mathbf{v}_1[\mathsf{reset}(e) := 0]$, we have $(q_1, \mathbf{v}_1) \stackrel{e}{\to} (q_2, \mathbf{v}_2)$.

Second, consider the case $S_1 \xrightarrow{\tau} S_2$, that is, $\mathsf{until}(S_1, S_2) = S_1$. Let $S_i = (q, \zeta_i)$, for $i = 1, 2$. For simplicity, we write $\mathsf{until}(\zeta_1, \zeta_2)$.

Lemma 8.2 *Consider two convex $\mathcal{X}$-polyhedra ζ_1, ζ_2 such that $\mathsf{until}(\zeta_1, \zeta_2) = \zeta_1$. For any $\mathbf{v}_1 \in \zeta_1$, we can find in time $O(n)$ some $\delta \in \mathsf{R}$ such that $\mathbf{v}_1 + \delta \in \zeta_2$.*

As before, the proof of the lemma is given in section 10.2. The construction follows immediately, since we have $(q, \mathbf{v}_1) \xrightarrow{\delta} (q, \mathbf{v}_2)$.

Regarding the complexity of building the whole run, observe that each step takes time either $O(n^2)$ or $O(n)$. Since the step is repeated l times, the whole run can be constructed in time $O(l \cdot n^2)$.

Backward-then-forward passes

Backward. As before, it suffices to show how the computation is done for a single step, say, $(q_1, \zeta_1) \xrightarrow{e} (q_2, \zeta_2)$. That is, given $\mathbf{v}_2 \in \zeta_2$, we shall show how to compute $\delta \in \mathsf{R}$ and $\mathbf{v}_1 \in \zeta_1$ such that $(q_1, \mathbf{v}_1) \xrightarrow{e} \xrightarrow{\delta} (q_2, \mathbf{v}_2')$, and $\mathbf{v}_2, \mathbf{v}_2'$ are c-equivalent.

Finding δ can be done by "pulling $\mathbf{v}_2$ backward in time", until some clock reset in e reaches 0. More precisely, if $\mathsf{reset}(e) = \emptyset$ then we let $\delta = 0$, otherwise we let $\delta = \mathbf{v}_2(x)$, for some $x \in \mathsf{reset}(e)$.

Now, let $\mathbf{v}_3 = \mathbf{v}_2 \Leftarrow \delta$. By definition, we have $\mathbf{v}_3 \in \zeta_2$ and $(q_2, \mathbf{v}_3) \xrightarrow{\delta} (q_2, \mathbf{v}_2)$. It remains to find $\mathbf{v}_1 \in \zeta_1$ such that $(q_1, \mathbf{v}_1) \xrightarrow{e} (q_2, \mathbf{v}_4)$ and $\mathbf{v}_4$ and $\mathbf{v}_3$ are c-equivalent, which implies that $\mathbf{v}_4 + \delta$ and $\mathbf{v}_2$ are also c-equivalent.

Without loss of generality, we assume that there exists $0 \leq k \leq n$ such that the clocks $x_1, ..., x_k$ are not reset in e and for each $j = 1, ..., k$, $\mathbf{v}_2(x_j) \leq c$.

First, $\mathbf{v}_1$ should satisfy $\mathsf{guard}(e)$. Moreover, since clocks $x_1, ..., x_k$ are not reset in e, they should have the same value in $\mathbf{v}_1$ and $\mathbf{v}_3$. Then, we let $\mathbf{v}$ be a k-incomplete valuation, such that $\mathbf{v}(x_i) = \mathbf{v}_3(x_i)$, for $i = 1, ..., k$. Using lemma 8.1, we can complete $\mathbf{v}$ in $\zeta_1 \cap \mathsf{guard}(e)$. This is always possible, by the second part of the post-stability property in the simulation graph.

Therefore, we define $\mathbf{v}_1$ to be the completed valuation. If we let $\mathbf{v}_4 = \mathbf{v}_1[\mathsf{reset}(e) := 0]$, we have:

- for $i = 1, ..., k$, $\mathbf{v}_4(x_i) = \mathbf{v}_3(x_i)$;
- for $i = k+1, ..., n$,
  - if $x_i \in \mathsf{reset}(e)$, then $\mathbf{v}_4(x_i) = \mathbf{v}_3(x_i) = 0$,
  - otherwise, $\mathbf{v}_4(x_i) > c$ and $\mathbf{v}_3(x_i) > c$.

That is, $\mathbf{v}_4$ and $\mathbf{v}_3$ are c-equivalent.

Regarding the complexity of the backward pass, observe that for each step, it takes $O(n)$ time to find the delay δ and $O(n^2)$ time to complete the valuation [1]. Therefore, the whole pass can be performed in time $O(l \cdot n^2)$.

[1] Completing a valuation in the intersection of more than one polyhedra, say, $\zeta_1 \cap \cdots \cap \zeta_m$, multiplies the complexity of the operation by only a constant factor m.

Forward pass. This pass is easy. We start from $s_1 = (q_1, \mathbf{v}_1)$, as computed in the backward pass. Then, for $i = 1, ..., l+1$, we compute $\mathbf{v}'_i$ by "adjusting" $\mathbf{v}_i$ as follows.

- $\mathbf{v}'_1 = \mathbf{v}_1$;
- for $i = 2, ..., l+1$, $\mathbf{v}'_i = (\mathbf{v}'_{i-1}[\mathsf{reset}(e_i) := 0]) + \delta_i$.

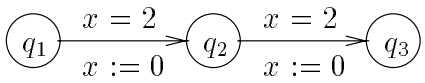
Using lemma 5.4 and induction on l, it is easy to prove that the resulting run is valid, that is, $(q_i, \mathbf{v}'_i) \xrightarrow{e_i} \xrightarrow{\delta_i} (q_{i+1}, \mathbf{v}'_{i+1})$, for all $i = 1, ..., l$.

The complexity of the forward pass is $O(l \cdot n)$. Therefore, the complexity of computing the whole run is $O(l \cdot n^2)$.

Example. Consider the simple TA shown in figure 8.1. We are interested in reachability of the target zone (q_3, true) from the initial zone $(q_1, x = y)$. Let e_1 be the edge from q_1 to q_2 and e_2 the edge from q_2 to q_3. The algorithm of figure 7.1 succeeds, returning the zone path $(q_1, x = y) \xrightarrow{e_1} (q_2, y = x + 2) \xrightarrow{e_2} (q_3, y > x + 2)$. Notice that for this example $c = 2$ and before applying $\mathsf{close}()$, the polyhedron associated to q_3 is $y = x + 4$.

For the backward pass, we start by choosing $\mathbf{v}_3 \in y > x + 2$, say, $\mathbf{v}_3 = (x = 0, y = 3)$. This gives $\delta_3 = 0$. Then, we must complete a 0-incomplete valuation in $y = x + 2 \wedge x = 2$, which gives us $\mathbf{v}_2 = (x = 2, y = 4)$. Since x is reset in e_1, we get $\delta_2 = 2$. Finally, we have to complete a 0-incomplete valuation in $y = x \wedge x = 2$, which gives us $\mathbf{v}_1 = (x = 2, y = 2)$. At the end of the backward pass, we have the sequence $(q_1, x = 2, y = 2) \xrightarrow{e_1} (q_2, x = 0, y = 2) \xrightarrow{2} (q_2, x = 2, y = 4) \xrightarrow{e_2} (q_3, x = 0, y = 3)$. This is not a valid run, since there is a "jump" of clock y on the e_2-transition.

The forward pass adjusts $\mathbf{v}_3$ to $\mathbf{v}'_3 = (x = 0, y = 4)$, yielding the final (valid) run: $(q_1, x = 2, y = 2) \xrightarrow{e_1} (q_2, x = 0, y = 2) \xrightarrow{2} (q_2, x = 2, y = 4) \xrightarrow{e_2} (q_3, x = 0, y = 4) \xrightarrow{0} (q_3, x = 0, y = 4)$.



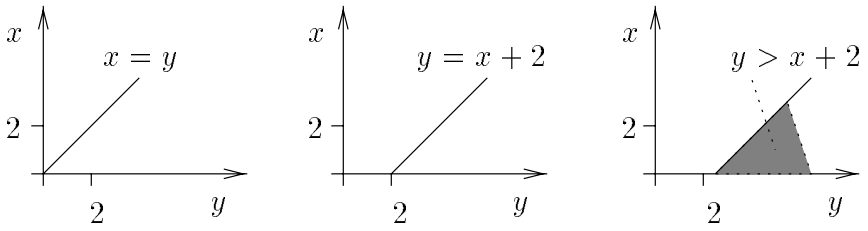


Figure 8.1: An example for finite diagnostics.

8.2 Infinite runs and trails

Infinite diagnostics must have a finite representation. This is possible if the infinite run (or trail) is *periodic*. This is not always the case, as can be seen in the example of figure 8.2: in all

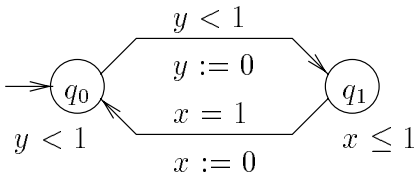


Figure 8.2: A TA having no periodic runs.

infinite runs of the TA in the figure, the difference $y \Leftrightarrow x$ at state q_0 keeps increasing, so that a periodic run cannot exist (notice that the system is not zeno).

The problem with the previous example lies on the presence of the strict constraint $y < 1$. As it turns out, if we restrict ourselves to TA with no such constraints, then periodic trails (thus, also runs) always exist, as we show below. For the rest of this section, we consider TA where guards and invariants are defined by atomic constraints of the form $x \leq c$ or $x \geq c$.

Our approach is as follows. We start from a zone path ending in a cycle:

$$\pi = S_0 \stackrel{e_0}{\rightarrow} \cdots \stackrel{e_{l-1}}{\rightarrow} (S_l \stackrel{e_l}{\rightarrow} \cdots \stackrel{e_{l+m}}{\rightarrow} S_l)^\omega$$

This path can be returned as symbolic diagnostic by one of the algorithms of section 7.2. Based on this path, we construct a trail

$$(e_0, t_0) \cdots (e_{l-1}, t_{l-1})(e_l, t_l^0) \cdots (e_{l+m}, t_{l+m}^0)(e_l, t_l^1) \cdots$$

which is periodic, that is, for each $k = 1, 2, ...$, $t_l^k \Leftrightarrow t_l^{k-1} = t_l^{k+1} \Leftrightarrow t_l^k$. The trail can be transformed to an infinite run as described in the beginning of the chapter.

The technique for building the periodic trail is called *constraint induction*. Intuitively, it works in two phases as follows. The first phase computes a fix-point of a set of constraints on variables corresponding to the time stamps $t_l^0, ..., t_l^1, ..., t_{l+m}^1$, that is, the first two "iterations" of the trail. At each iteration of the fix-point, the constraints induced by the second iteration are transposed to the first iteration. Computation stops when a stabilized set of constraints is obtained. During the second phase, a solution is computed for the first two iterations such that it can be extended to a periodic solution. We now formally describe the technique.

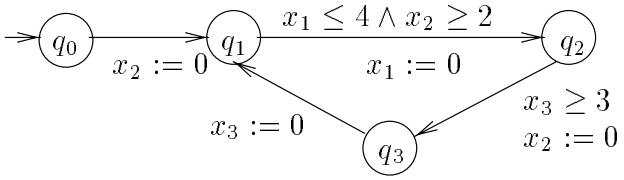


Figure 8.3: A cycle for constructing a periodic trail.

First, we associate real variables $z_0, ..., z_{l-1}$ to the time stamps $t_0, ..., t_{l-1}$, and $y_0, ..., y_{2m-1}$ to $t_l^0, ..., t_{l+m-1}^0, t_{l+m}^1, ..., t_{l+m-1}^1$. Let Y denote the set of variables $\{y_0, ..., y_{2m-1}\}$. Then, we build a system of linear constraints J on Y as follows:

- J has the constraints $y_0 \leq \cdots \leq y_{2m-1}$.
- If clock x is reset in e_{l+i} and $x \sim c$ is an atomic constraint of $\mathsf{guard}(e_{l+j})$, for $0 \leq i < j \leq m$, and x is not reset between i and j, then J has the constraint $y_j \Leftrightarrow y_i \sim k$.

- If clock x is reset in e_{l+i} and $x \sim k$ is an atomic constraint of $\mathsf{guard}(e_{l+j})$, for $0 \leq j \leq i \leq m$, and x is not reset between i and j (circularly), then J has the constraint $y_{j+m} \Leftrightarrow y_i \sim k$.

Thus, J is a set of constraints on the first two iterations of the cycle. For example, the loop of figure 8.3 yields the following system of constraints:

$$\begin{aligned} & y_0 \leq y_1 \leq y_2 \leq y_3 \leq y_4 \leq y_5, \\ J \quad : \quad & y_3 \Leftrightarrow y_0 \leq 4, \\ & y_3 \Leftrightarrow y_1 \geq 2, \\ & y_4 \Leftrightarrow y_2 \geq 3 \end{aligned}$$

Since J is a system of constraints on real variables, it can be viewed as a convex polyhedron on Y, defined by the intersection of atomic constraints like the ones above. Thus, in the sequel, we apply polyhedra operations on J. We also use the operation of *variable substitution*, $J[y := z]$, defined as usual.

The next step is to stabilize J to a system of constraints $\hat{J}$, such that $\hat{J}$ contains all solutions of Y which can be extended to an infinite set of time stamps defining an infinite trail. The idea is as follows. If there were an infinite number of variables $y_0, y_1, ...$, each one associated to one time stamp $t_l^0, t_{l+1}^0, ...$, then we would have an infinite system of constraints (see figure 8.4). Some of the constraints in this system can be combined to yield stronger constraints. For instance, from constraints $y_3 \Leftrightarrow y_0 \leq 4$ and $y_3 \Leftrightarrow y_1 \geq 2$ we can deduce

$$y_1 \Leftrightarrow y_0 \leq 2 \quad (*)$$

This is an extra piece of information on iteration 1, obtained using information on both iterations 1 and 2. Now, we can "shift" the "constraint window" from iterations 1 and 2 to iterations 2 and 3, and infer the constraint

$$y_4 \Leftrightarrow y_3 \leq 2 \quad (**)$$

which is identical to $(*)$, up to renaming. Notice that $(**)$ is not deduced explicitly but is obtained from $(*)$ simply by renaming. This is the idea of constraint induction, namely, to transpose the amount of information obtained for iteration 1 to iteration 2.

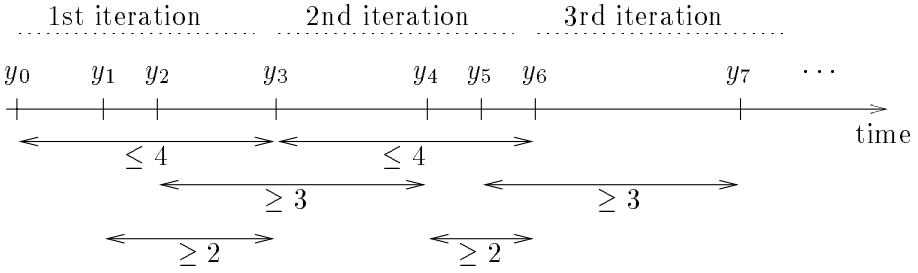


Figure 8.4: The loop of figure 8.3, developed.

Having obtained constraints such as $(**)$, they can be "re-injected" into J to strengthen it, permitting to deduce stricter constraints on iteration 1. The process is repeated until no stricter constraints can be deduced [2].

[2]Notice that J cannot become unsatisfiable, since this would violate the assumption that there exists an infinite path π. In a more general setting where we start directly from the structural loop of the TA, this would also be possible, meaning that there is no infinite trail corresponding to this cycle of edges.

Formally, J is strengthened iteratively as follows:

$$\begin{aligned} J^0 &= J \\ J^{i+1} &= J^i \cap ((J^i/_{\{y_0,...,y_{m-1}\}})[y_0 := y_m, ..., y_{m-1} := y_{2m-1}]) \end{aligned}$$

$J^i/_{\{y_0,...,y_{m-1}\}}$ corresponds to the amount of information obtained on iteration 1, and the substitution $[y_0 := y_m, ..., y_{m-1} := y_{2m-1}]$ corresponds to transposing this information to iteration 2.

The lemma below proves that the above process terminates. The proof is given in the appendix.

Lemma 8.3 *There exists i such that $J^{i+1} = J^i$.*

Let $\hat{J}$ denote the stabilized J^i. For the example above, the process terminates after two steps, yielding the following set of constraints (represented as a square matrix where the constant c of the constraint $y_j \Leftrightarrow y_k \leq c$ corresponds to element j, k):

	y_0	y_1	y_2	y_3	y_4	y_5
y_0	0	0	0	$\Leftrightarrow$2	$\Leftrightarrow$3	$\Leftrightarrow$3
y_1	2	0	0	$\Leftrightarrow$2	$\Leftrightarrow$3	$\Leftrightarrow$3
y_2	3	3	0	$\Leftrightarrow$1	$\Leftrightarrow$3	$\Leftrightarrow$3
y_3	4	4	4	0	0	0
y_4	6	6	6	2	0	0
y_5	7	7	7	3	3	0

The next step is to augment $\hat{J}$ with the constraints on the "tail" of the cycle, $S_0 \stackrel{e_0}{\rightarrow} \cdots \stackrel{e_{l-1}}{\rightarrow}$. More precisely, we build the system of constraints J^* such that:

- J^* has all the constraints of $\hat{J}$.
- J^* has the constraints $z_0 \leq z_1 \leq \cdots \leq z_{l-1} \leq y_0$.
- If clock x is reset in e_i, for $0 \leq i < l$ and $x \sim c$ is a constraint of $\mathsf{guard}(e_j)$, for $0 \leq j < l$ (resp. $l \leq j < l + m$), and x is not reset between i and j, then J^* has the constraint $z_j \Leftrightarrow z_i \sim c$ (resp. $y_{j-l} \Leftrightarrow z_i \sim c$).
- If $x \sim k$ is a constraint of $\mathsf{guard}(e_j)$, for $0 \leq j < l$ (resp. $l \leq j < l + m$), and clock x is not reset before j, then J^* has the constraint $z_j \sim c$ (resp. $y_{j-l} \sim c$). This is because clocks are assumed to be reset to zero initially.

For the example of figure 8.3, J^* has all the constraints of $\hat{J}$ plus $z_0 \leq y_0$ and $y_0 \Leftrightarrow z_0 \leq 2$.

Notice that J^* cannot be unsatisfiable since this would violate the assumption that there exists an infinite path π. The following lemma shows how to choose a solution of J^* which can be then extended to an infinite sequence of time stamps.

Lemma 8.4 *J^* admits a solution $\mathbf{v} : \{z_0, ..., z_{l-1}, y_0, ..., y_{2m-1}\} \mapsto \mathsf{R}$ such that for all $0 \leq i, j < m$, $\mathbf{v}(y_i) \Leftrightarrow \mathbf{v}(y_j) = \mathbf{v}(y_{m+i}) \Leftrightarrow \mathbf{v}(y_{m+j})$. The solution can be effectively computed in time $O((m + l)^2)$.*

Proof: We assume that $c_{i,j}$ is the bound of $y_i \Leftrightarrow y_j$ in J^*, that is, all constraints of J^* on variables in Y are of the form $y_i \Leftrightarrow y_j \leq c_{i,j}$. Similarly, we assume that constraints on one variable from Y and one variable from $\{z_0, ..., z_{l-1}\}$ are of the form $d_{i,j} \leq y_i \Leftrightarrow z_j \leq e_{i,j}$, for $0 \leq i < 2m, 0 \leq j < l$.

From the fact that J^* is satisfiable, we have the following set of hypotheses:

$$c_{i,j} \leq c_{i,l} + c_{l,j}, \quad \text{for all} \quad 0 \leq i, j, l < 2m \tag{8.1}$$

$$d_{i,j} \leq e_{i,j}, \quad \text{for all} \quad 0 \leq i < 2m,\ 0 \leq j < l \tag{8.2}$$

Now, let $Y_1 = \{y_0, ..., y_{m-1}\}$ and $Y_2 = \{y_l, ..., y_{2m-1}\}$, From the definition of J^*, it follows that $J^*/_{Y_2} \subseteq (J^*/_{Y_1})[Y_1 := Y_2]$, that is, every solution of $J^*/_{Y_2}$ is a solution of $(J^*/_{Y_1})[Y_1 := Y_2]$. Thus, we have a second set of hypotheses:

$$c_{i+m,j+m} \leq c_{i,j}, \quad \text{for all} \quad 0 \leq i, j < m \tag{8.3}$$

We are now ready to present the construction of $\hat{\mathbf{v}}$. For simplicity, we write $z_i = t$, $y_j = t'$, and so on, instead of $\hat{\mathbf{v}}(z_i) = t$, $\hat{\mathbf{v}}(y_j) = t'$.

First, we set $z_0 = 0$ and choose values for $z_1, ..., z_{l-1}$ (as shown by lemma 8.1) so that the projection $J^*\rfloor_{\{z_0,...,z_{l-1}\}}$ is satisfied.

Then, we choose y_0 such that:

$$\forall 0 \leq j < l \ .\ d_{0,j} + z_j \leq y_0 \leq e_{0,j} + z_j$$

Such a value can always be chosen according to hypotheses 8.2.

Next, we choose y_m such that:

$$\forall 0 \leq i < m \ .\ y_0 \Leftrightarrow c_{i,m+i} \leq y_m \leq y_0 + c_{m+i,i}$$

To see that such a value can always be chosen, let the minimum of $c_{m+i,i}$ be at i_0 and the maximum of $\Leftrightarrow c_{i,m+i}$ be at j_0. Then:

$$\begin{array}{rcll} c_{m+j_0,m+i_0} & \leq & c_{j_0,i_0} & \text{from hypotheses 8.3} \\ & \leq & c_{j_0,m+j_0} + c_{m+j_0,m+i_0} + c_{m+i_0,i_0} & \text{from hypotheses 8.2} \end{array}$$

which directly implies $\Leftrightarrow c_{j_0,m+j_0} \leq c_{m+i_0,i_0}$.

We continue by choosing, for each $i = 1, ..., m \Leftrightarrow 1$, a pair of values y_i and y_{m+i} such that $y_{m+i} \Leftrightarrow y_i = y_m \Leftrightarrow y_0$. Moreover, y_i and y_{m+i} are chosen so that they satisfy the constraints with any previously chosen y_j and y_{m+j}, for $0 \leq j < i$. Finally, y_i should satisfy the constraints with z_k, for $0 \leq k < l$. More precisely, for all $0 \leq k < l$, $0 \leq j < i$, we need:

$$\begin{array}{rcccl} d_{i,k} & \leq & y_i \Leftrightarrow z_k & \leq & e_{i,k} \\ \Leftrightarrow c_{j,i} & \leq & y_i \Leftrightarrow y_j & \leq & c_{i,j} \\ \Leftrightarrow c_{m+j,i} & \leq & y_i \Leftrightarrow y_{m+j} & \leq & c_{i,m+j} \\ \Leftrightarrow c_{j,m+i} & \leq & y_{m+i} \Leftrightarrow y_j & \leq & c_{m+i,j} \\ \Leftrightarrow c_{m+j,m+i} & \leq & y_{m+i} \Leftrightarrow y_{m+j} & \leq & c_{m+i,m+j} \end{array}$$

Replacing y_{m+i} and y_{m+j} by $y_i + (y_m \Leftrightarrow y_0)$ and $y_j + (y_m \Leftrightarrow y_0)$, respectively, we get:

$$\begin{array}{rcccl} z_k + d_{i,k} & \leq & y_i & \leq & z_k + e_{i,k} \\ y_j \Leftrightarrow c_{j,i} & \leq & y_i & \leq & y_j + c_{i,j} \\ y_j + (y_m \Leftrightarrow y_0) \Leftrightarrow c_{m+j,i} & \leq & y_i & \leq & y_j + (y_m \Leftrightarrow y_0) + c_{i,m+j} \\ y_j \Leftrightarrow (y_m \Leftrightarrow y_0) \Leftrightarrow c_{j,m+i} & \leq & y_i & \leq & y_j \Leftrightarrow (y_m \Leftrightarrow y_0) + c_{m+i,j} \\ y_j \Leftrightarrow c_{m+j,m+i} & \leq & y_i & \leq & y_j + c_{m+i,m+j} \end{array}$$

The second line can be omitted, because the fifth line imposes stricter bounds, since $c_{m+i,m+j} \leq c_{i,j}$ (hypotheses 8.3).

As previously, we have to ensure that such a value for y_i exists. We show that this is the case by induction on i. We only consider two of the possible cases for the minimum and maximum of right-hand and left-hand values in the constraints above, respectively. The rest of the cases are similar.

As a first case, suppose that the minimum of the right-hand values is $y_{j_1} + c_{m+i,m+j_1}$ and that the maximum of the left-hand values is $y_{j_2} \Leftrightarrow (y_m \Leftrightarrow y_0) \Leftrightarrow c_{j_2,m+i}$. We have to prove that $y_{j_2} \Leftrightarrow (y_m \Leftrightarrow y_0) \Leftrightarrow c_{j_2,m+i} \leq y_{j_1} + c_{m+i,m+j_1}$. The proof is as follows:

$$\begin{array}{lcll} y_{j_2} \Leftrightarrow y_{j_1} \Leftrightarrow (y_m \Leftrightarrow y_0) & = & y_{j_2} \Leftrightarrow y_{m+j_1} + y_{m+j_1} \Leftrightarrow y_{j_1} \Leftrightarrow (y_m \Leftrightarrow y_0) & \\ & = & y_{j_2} \Leftrightarrow y_{m+j_1} & \text{from the construction of } y_{m+j_1}, y_{j_1} \\ & \leq & c_{j_2,m+j_1} & \text{from the induction hyp.} \\ & \leq & c_{j_2,m+i} + c_{m+i,m+j_1} & \text{from hyp. 8.2} \end{array}$$

which directly implies the result.

As a second case, suppose that the minimum of the right-hand values is $y_{j_1} \Leftrightarrow (y_m \Leftrightarrow y_0) + c_{m+i,j_1}$ and that the maximum of the left-hand values is $y_{j_2} + (y_m \Leftrightarrow y_0) \Leftrightarrow c_{m+j_2,i}$. We have to prove that $y_{j_2} + (y_m \Leftrightarrow y_0) \Leftrightarrow c_{m+j_2,i} \leq y_{j_1} \Leftrightarrow (y_m \Leftrightarrow y_0) + c_{m+i,j_1}$. The proof is as follows:

$$\begin{array}{lcl} y_{j_2} + (y_m \Leftrightarrow y_0) \Leftrightarrow y_{j_1} + (y_m \Leftrightarrow y_0) & = & y_{j_2} \Leftrightarrow y_{m+j_2} + (y_m \Leftrightarrow y_0) + y_{m+j_2} \Leftrightarrow y_{j_1} + (y_m \Leftrightarrow y_0) \\ & = & y_{m+j_2} \Leftrightarrow y_{j_1} + (y_m \Leftrightarrow y_0) \\ & = & y_{m+j_2} \Leftrightarrow y_{j_1} + y_{m+i} \Leftrightarrow y_i \\ & = & (y_{m+j_2} \Leftrightarrow y_i) + (y_{m+i} \Leftrightarrow y_{j_1}) \\ & \leq & c_{m+j_2,i} + c_{m+i,j_1} \end{array}$$

which directly implies the result.

Regarding the time complexity of the construction of $\hat{\mathbf{v}}$, we have the following: first, choosing values for $z_0, ..., z_{l-1}$ can be done in $O(l^2)$; second, each time a value is chosen for some y_i, we need the minimum and maximum of at most $2 \cdot (m + l)$ values, which can be found in $O(m + l)$; this is done m times. Therefore, the whole solution can be assembled in time $O((m + l)^2)$. ∎

For the example of figure 8.3, we find the solution: $(z_0 : 0, y_0 : 2, y_1 : 4, y_2 : 5, y_3 : 6, y_4 : 8, y_5 : 9)$. Observe that $y_3 \Leftrightarrow y_0 = y_4 \Leftrightarrow y_1 = y_5 \Leftrightarrow y_2 = 4$.

From $\hat{\mathbf{v}}$, we can build the periodic trail such that:

$$\begin{array}{ll} t_i = \mathbf{v}(z_i), & \text{for } i = 0, ..., l \Leftrightarrow 1 \\ t_j^k = \mathbf{v}(y_j) + k \cdot (\mathbf{v}(y_{m+j}) \Leftrightarrow \mathbf{v}(y_j)), & \text{for } j = 0, ..., m \Leftrightarrow 1,\ k = 0, 1, ... \end{array}$$

In our example, this gives the infinite trail:

$$(e_0, 0)(e_1, 2)(e_2, 4)(e_3, 5)(e_1, 6)(e_2, 8)(e_3, 9)(e_1, 10)(e_2, 12)(e_3, 13) \cdots$$

yielding the infinite run:

$$\begin{array}{l} (q_0, x_1 : 0, x_2 : 0, x_3 : 0) \overset{e_0}{\Leftrightarrow\!\!\rightarrow} \\ (q_1, x_1 : 0, x_2 : 0, x_3 : 0) \overset{2}{\Leftrightarrow\!\!\rightarrow} \overset{e_1}{\Leftrightarrow\!\!\rightarrow} (q_2, x_1 : 0, x_2 : 2, x_3 : 2) \overset{2}{\Leftrightarrow\!\!\rightarrow} \overset{e_2}{\Leftrightarrow\!\!\rightarrow} (q_3, x_1 : 2, x_2 : 0, x_3 : 4) \overset{1}{\Leftrightarrow\!\!\rightarrow} \overset{e_3}{\Leftrightarrow\!\!\rightarrow} \\ (q_1, x_1 : 3, x_2 : 1, x_3 : 0) \overset{1}{\Leftrightarrow\!\!\rightarrow} \overset{e_1}{\Leftrightarrow\!\!\rightarrow} (q_2, x_1 : 0, x_2 : 2, x_3 : 1) \overset{2}{\Leftrightarrow\!\!\rightarrow} \overset{e_2}{\Leftrightarrow\!\!\rightarrow} (q_3, x_1 : 2, x_2 : 0, x_3 : 3) \overset{1}{\Leftrightarrow\!\!\rightarrow} \overset{e_3}{\Leftrightarrow\!\!\rightarrow} \\ (q_1, x_1 : 3, x_2 : 1, x_3 : 0) \overset{1}{\Leftrightarrow\!\!\rightarrow} \overset{e_1}{\Leftrightarrow\!\!\rightarrow} \cdots \end{array}$$

Relation to the literature

To our knowledge, the techniques presented in this chapter are new. Finite diagnostics have been considered independently in [LPY95], however, only the existence of a run inscribed in a symbolic path is stated and no method is given on how to extract the run. Moreover, the symbolic reachability of [LPY95] does not contain the c-closure operation. This makes the extraction of runs simpler, but without c-closure termination is not generally ensured.

Recently, [AKV98] have developed an algorithm which, given a sequence of edges, produces a corresponding run, if one exists. This algorithm has complexity $O(l \cdot n^2)$ as ours, and can also be used to extract a finite concrete diagnostic from a symbolic path.

The finite-diagnostic algorithm has been implemented in the tool `profounder` (section 11.4) and has been used to provide diagnostics in a number of case studies (see, for instance, section 12.3). The infinite-diagnostic algorithm is currently under implementation.

Chapter 9

Controller synthesis

Reactive systems are supposed to work in a certain environment which has to be part of the model to be analyzed. To avoid confusion, in this chapter we use the term *controller* to refer to the system without its environment, and the term *closed* system to refer to the system composed with its environment.

The environment can be *controllable*, that is, cooperative or deterministic, or *uncontrollable*, that is, unpredictable or even adversary. For example, when human interaction or physical measurements are involved, we have an uncontrollable environment. When the environment is simply another system with known behavior, it is controllable.

The controller, in turn, can be *complete* or *incomplete*. In the former case the specification is fixed: for example, when modeling an existing communication protocol or distributed algorithm [1]. In the latter case, the specification corresponds to an intermediate design phase where some parameters still remain to be fixed, for instance, how much time should an operation last, or in which order a set of tasks are to be scheduled.

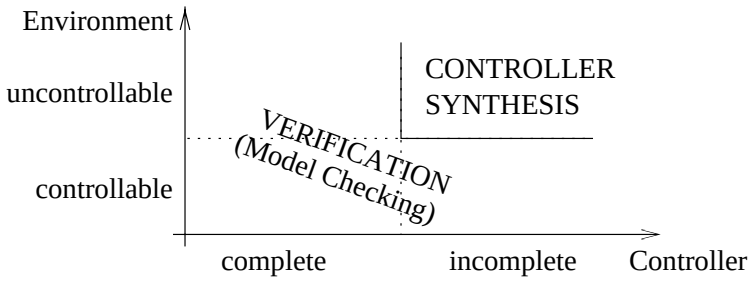


Figure 9.1: The problem space of verification and controller synthesis.

According to the above classification, the problem space is divided into four regions, as illustrated in figure 9.1. The verification techniques that we have presented thus far are sufficient to treat three of these regions. In particular:

- The TGC example of section 3.1 falls in the upper-left region of the classification: the environment consists of the train and the gate, which have unpredictable delays of staying in the crossing, rising and lowering. The Controller here is completely specified. Thus, all that remains to be done is to verify whether all behaviors of the closed system satisfy

[1] However, this does not imply that the specification is deterministic.

a given property. The quantification "all behaviors" implies that the property holds no matter what the environment does.

- The scheduling case study presented in section 12.5 falls in the lower-right region of the classification: the environment here is deterministic, since the periods and execution delays of the tasks are fixed. On the other hand, the controller (scheduler) is incomplete, since the order of execution of the tasks is not a-priori determined. Completing the scheduler means finding a sample execution scenario where all tasks are served, which can be expressed as a property of the form "there exists a behavior where ...". In general, completing an incomplete controller with controllable environment can be done using model checking to find a set of behaviors (or states) satisfying a given property.

There exists a class of problems which cannot be reduced to verification, namely, incomplete controllers with uncontrollable environment (upper-right region in the figure). Here, the controller's non-determinism corresponds to the possible design choices, where as the non-determinism of the environment corresponds to our uncertainty about the latter. To take into account this difference in semantics, the actions of the closed system are explicitly distinguished by labeling those which belong to the controller as controllable and those belonging to the environment as uncontrollable [2].

The problem of *controller synthesis* consists in restricting the controllable actions so that the closed system satisfies a given property, independently of the uncontrollable actions (these are not modified). This restriction of the controller's non-determinism is called a *strategy*.

In this chapter we study the controller-synthesis problem for timed systems, with respect to two kinds of properties, namely, invariance and reachability. After defining the problem formally (section 9.1), we present two different approaches to solve it:

- The first approach consists in computing symbolically a set of states, as a fix-point of a pre() operator, modified to take into account controllability. Then, the strategy is computed by restricting the controller to the above set of states. This approach is presented in section 9.2.

- In the second approach, the problem is reduced to the untimed case: first, we present an on-the-fly algorithm for controller synthesis on finite graphs; then, we adapt this algorithm for the STa-quotient of a timed system. This approach is presented in section 9.3.

The motivation of the second approach comes from the fact that the fix-point computation involves an expensive predecessor operator using complementation. Moreover, it computes the *maximal* strategy, while usually it suffices to exhibit just one strategy. The on-the-fly algorithm takes advantage of this fact, returning the first strategy (or counter-strategy) as soon as it is found.

9.1 Timed Controller Synthesis

9.1.1 Controllable Timed Automata

A *controllable timed automaton* (CTA) [MPS95] is a TA whose set of edges E is partitioned into two disjoint sets E^c and E^u. The latter correspond to the *controllable* and *uncontrollable*

[2] This assumes that the controller and the environment never synchronize. In section 9.1.2 we consider the general case, showing how to compose controllable and uncontrollable actions.

discrete actions of the closed system, respectively.

Consider again the TGC example of section 3.1. To make the example interesting for controller synthesis, we slightly modify it, as shown in figure 9.2: first, a delay of at least 1 time unit is put between the departure of a Train from the crossing and the arrival of the next; second, the Controller is made less deterministic, by allowing it to issue the "lower" command to the gate at any time between 0 and 3 time units after receiving the "approach" signal.

Regarding the Train and Gate automata as being the environment, we can represent the above system as a CTA, shown to the lower part of the figure. This CTA corresponds to the parallel composition of the three automata, where edges are marked either controllable or uncontrollable. The distinction is made depending on whether the edges correspond to commands issued by the Controller (e.g., "lower" or "raise") or actions/responses of the environment (e.g., "in" or "up"). In the figure, controllable edges are drawn by dashed lines.

Strategies

The semantics of a CTA A are given in terms of *strategies*. Intuitively, a strategy is a sub-graph of the semantic graph of A, obtained by restricting the choices of the controller while preserving the choices of the environment.

More formally, let G be the semantic graph of A and s be a node of G. A strategy of A starting from s is a sub-graph *Strat* of G satisfying the following conditions:

1. s is a node of *Strat*.
2. If $s \xrightarrow{\delta} s+\delta$ is an edge of *Strat* then for all $\delta' < \delta$, $s+\delta'$ is a node of *Strat* and $s \xrightarrow{\delta'} s+\delta'$ is an edge of *Strat*.
3. For each node s of *Strat*, if there exist $\delta \in \mathsf{R}$ and $e \in E^u$ such that $s \xrightarrow{\delta} \xrightarrow{e} s'$, then:
   - either $s \xrightarrow{\delta} s+\delta$ and $s+\delta \xrightarrow{e} s'$ are edges of *Strat*,
   - or there exist $\delta' < \delta$ and $e' \in E^c$, such that $s \xrightarrow{\delta'} s+\delta'$ and $s+\delta' \xrightarrow{e'} s''$ are edges of *Strat*.

Intuitively, condition 2 ensures continuity of time transitions. Condition 3 makes sure that the controller does not "cheat", that is, if the environment can make a move after some delay δ then this move must be included in the strategy, unless if the controller can make its move earlier, in some $\delta' < \delta$.

We can use TA notation to represent strategies. For example, a possible strategy of the CTA of figure 9.2 is obtained by replacing the constraint $z \leq 3$ in the invariant of discrete state 1 by $z \leq 2$. This corresponds to the Controller deciding to make its move sooner, in particular, at most 2 time units (instead of 3) after receiving the "approach" signal.

Winning strategies and controller-synthesis problem

We are interested in strategies preserving reachability and invariance. In the case of invariance, the controller tries to keep the closed system inside a set of "safe" states. In the case of reachability, the controller tries to lead the closed system to a set of "target" states. In the sequel, we make the simplifying assumption that all successors of the target states are also target states.

More formally, consider a strategy *Strat* and a set of states $\hat{S}$.

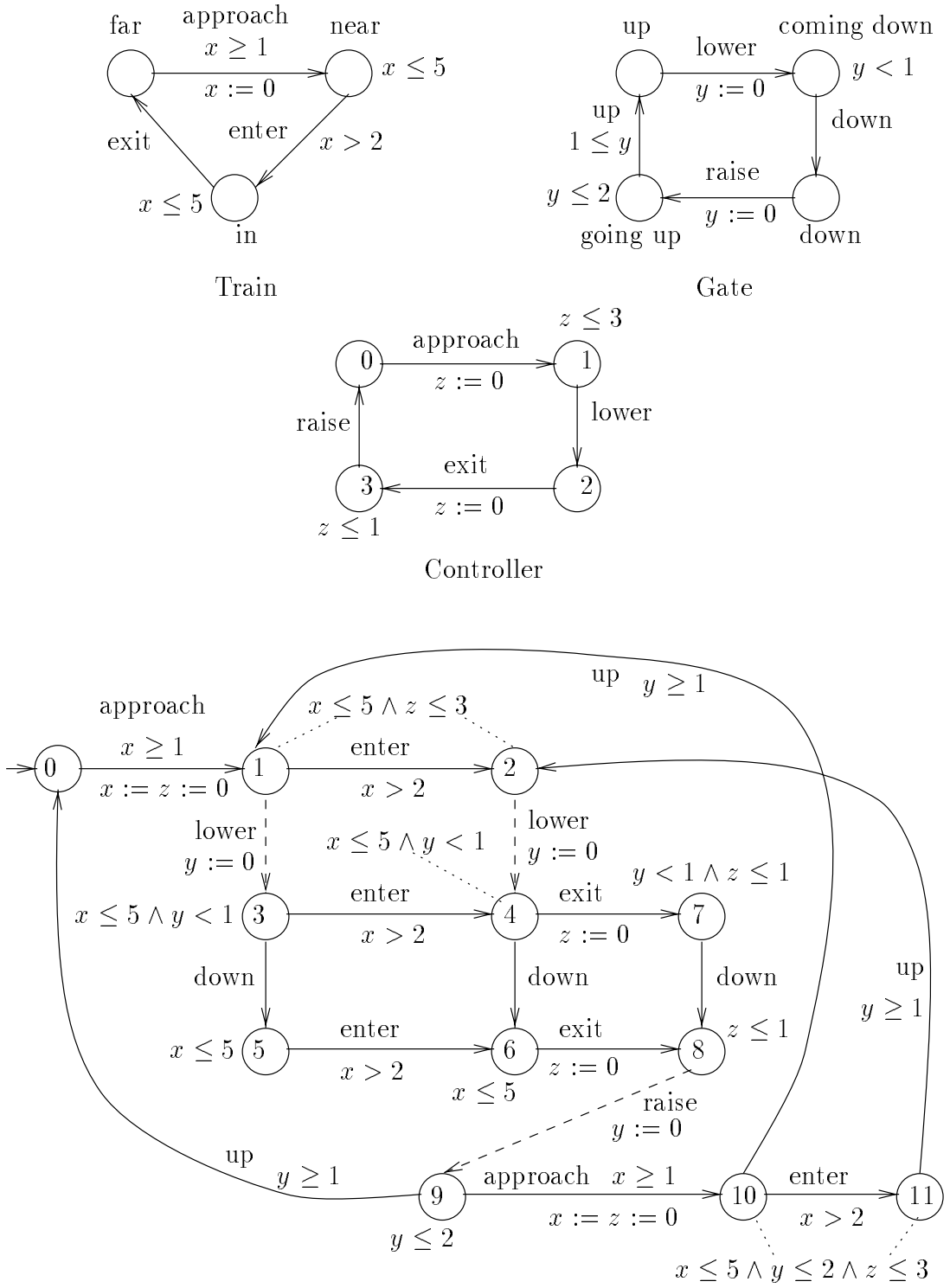


Figure 9.2: The composite CTA for the TGC system.

- *Strat* is said to be *winning* with respect to $\hat{S}$-invariance (or, *Strat stays in* $\hat{S}$) if all nodes of *Strat* belong to $\hat{S}$.
- *Strat* is said to be *winning* with respect to $\hat{S}$-reachability (or, *Strat leads to* $\hat{S}$) if for any path $s_0 \stackrel{\delta_0}{\rightarrow}\stackrel{e_0}{\rightarrow} s_1 \stackrel{\delta_1}{\rightarrow}\stackrel{e_1}{\rightarrow} \cdots$ in *Strat*, there exists some $i = 0, 1, ...$, such that $s_i \in \hat{S}$.

A state s is called winning with respect to $\hat{S}$-invariance (resp. $\hat{S}$-reachability) if there exists a strategy starting from s which leads to $\hat{S}$ (resp. leads to $\hat{S}$).

The *controller-synthesis* invariance (resp. reachability) problem for a CTA A and a set of states $\hat{S}$ is to find whether there exists a strategy of A starting from its initial state, which is winning with respect to $\hat{S}$-invariance (resp. $\hat{S}$-reachability). In case a winning strategy exists, we would also like to compute it. In case it does not exist, we would like a counter-strategy as diagnostics.

Consider again the TGC example. The strategy obtained by changing $z \leq 3$ to $z \leq 2$ is not winning with respect to the invariance property $\forall\Box$ (in $\Rightarrow$ down) (this property states that whenever the train is in the crossing the gate is down). The property is satisfied in all discrete states except 2, 4 and 11.

9.1.2 Parallel composition of CTA

As for TA, we shall define the parallel composition of two CTA. Before giving the formal definitions, let us explain intuitively what the meaning of parallel composition is, in particular, where does it differ with respect to the TA case.

Given the fact that we have two types of discrete edges, namely, controllable and uncontrollable, we have three possibilities regarding the controllability status of an edge obtained by two component-edges by synchronization:

- Either both component-edges are controllable: In this case, the synchronization is supposed to model either the *cooperation* of two actions of the controller, or a *command* of the controller to a component of its environment which waits passively to receive a message.
- Or one of the component-edges is controllable and the other uncontrollable: In this case, the synchronization is supposed to model an *immediate response* of the controller to an action of the environment.
- Or both component-edges are uncontrollable: This is considered as a meaningless synchronization, since it imposes a constraint on two actions which are triggered independently by the environment. Therefore, we disallow two uncontrollable edges to synchronize.

We have already applied the above rules informally, when composing the TGC system in the examples of the previous section. For instance, the synchronization of edges labeled "approach" between the Train and the Controller belongs to the second case above, thus, it has been marked uncontrollable. The synchronization of edges labeled "lower" between the Controller and the Gate belongs to the first case, thus, it has been marked controllable.

We now formalize the above rules. Consider two CTA $A_i = (\mathcal{X}_i, Q_i, q_i, E_i, \mathsf{invar}_i)$, $i = 1, 2$, such that $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$. Let $Labels_1, Labels_2 \subseteq Labels$ be the sets of labels used by A_1 and A_2 respectively. We assume that for any pair $e_1 \in E_1^u, e_2 \in E_2^u$, $\mathsf{label}(e_1) \neq \mathsf{label}(e_2)$.

Consider two strategies $Strat_1$ and $Strat_2$ of A_1 and A_2 respectively. The parallel composition of $Strat_1$ and $Strat_2$ is defined to be:

- The graph $G = Strat_1 \| Strat_2$, where $Strat_1$ and $Strat_2$ are viewed as semantic graphs, if the following conditions hold:

  1. For each node (s_1, s_2) of G, if $s_1 \xrightarrow{e_1} s_1'$ is an edge of $Strat_1$ and $e_1 \in E_1^u$ then $(s_1, s_2) \xrightarrow{e} (s_1', s_2)$ is an edge of G, where $e = e_1$ or $e = e_1 \| e_2$ for some $e_2 \in E_2$.
  2. For each node (s_1, s_2) of G, if $s_2 \xrightarrow{e_2} s_2'$ is an edge of $Strat_2$ and $e_2 \in E_2^u$ then $(s_1, s_2) \xrightarrow{e} (s_1, s_2')$ is an edge of G, where $e = e_2$ or $e = e_1 \| e_2$ for some $e_1 \in E_1$.

- Empty, otherwise.

Intuitively, conditions 1 and 2 require that no uncontrollable actions are missed during synchronization.

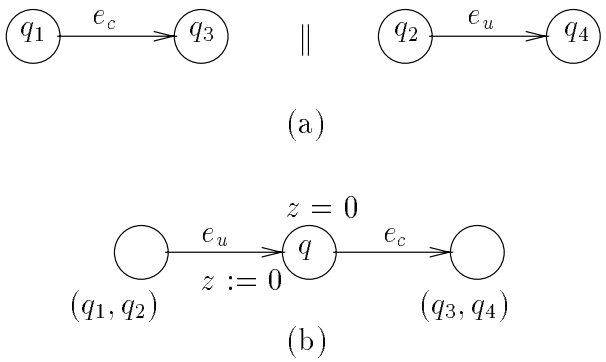


Figure 9.3: Syntactic parallel composition of CTA.

The definition of parallel composition of CTA at the semantic level (in terms of strategies), is not very useful for practical purposes. Therefore, we need to define its syntactic counterpart. This is not as straightforward as in the case of TA because of the different nature of controllable and uncontrollable edges. Indeed, if $e \in E^c$ and $e' \in E^u$, we cannot say that $e \| e'$ is either controllable or uncontrollable. Therefore, syntactic composition of a controllable and an uncontrollable edge does not make sense for CTA.

In order to deal with this problem, we shall introduce auxiliary discrete states in the composite CTA and simulate the immediate response of the controllable action to the uncontrollable one. This idea is illustrated in figure 9.3. Part (a) shows a controllable edge e_c and an uncontrollable edge e_u which have to be composed syntactically and part (b) shows the result. q is an auxiliary discrete state and z an auxiliary clock. e_u happens first, and has to be immediately followed by e_c. This is ensured by resetting z before entering q and having $z = 0$ as the invariant of q.

More formally, consider the two CTA A_1 and A_2 above. Their parallel composition, denoted $A_1 \| A_2$, is defined to be the TA $(\mathcal{X}, Q, (q_1, q_2), E^c \cup E^u, \mathsf{invar})$, such that:

- $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \{z\}$, for $z \notin \mathcal{X}_1 \cup \mathcal{X}_2$.
- $Q = Q_1 \times Q_2 \cup Q_1 \times Q_2' \cup Q_1' \times Q_2$, where Q_i' contains one member for each edge $e_i \in E_i^u$.
- E^c contains all composite edges $e_1 \| e_2$, $e_1 \| \perp$, $\perp \| e_2$, such that $e_1 \in E_1^c, e_2 \in E_2^c$.
- E^u contains all edges $e_1 \| \perp$, $\perp \| e_2$, such that $e_1 \in E_1^u, e_2 \in E_2^u$.

- If $e_i = (q_i, \zeta_i, a, X_i, q_{i+1}) \in E_i$, $i = 1, 2$, such that $e_1 \in E_1^c$ and $e_2 \in E_2^u$, then E^u contains the edge $((q_1, q_2), \zeta_2, a, X_2 \cup \{z\}, (q_1, q_2'))$ and E^c contains the edge $((q_1, q_2'), \zeta_1 \wedge z = 0, a, X_1, (q_3, q_4))$, where $q_2' \in Q_2'$.

- If $e_i = (q_i, \zeta_i, a, X_i, q_{i+1}) \in E_i$, $i = 1, 2$, such that $e_1 \in E_1^u$ and $e_2 \in E_2^c$, then E^u contains the edge $((q_1, q_2), \zeta_1, a, X_1 \cup \{z\}, (q_1', q_2))$ and E^c contains the edge $((q_1', q_2), \zeta_2 \wedge z = 0, a, X_2, (q_3, q_4))$, where $q_1' \in Q_1'$.

- If $q_i \in Q_i$, $q_i' \in Q_i'$, for $i = 1, 2$, then $\mathsf{invar}(q_1, q_2) = \mathsf{invar}_1(q_1) \cap \mathsf{invar}_2(q_2)$, $\mathsf{invar}(q_1, q_2') = \mathsf{invar}_1(q_1) \cap z = 0$ and $\mathsf{invar}(q_1', q_2) = \mathsf{invar}_2(q_2) \cap z = 0$.

Then, we can prove the following result [3].

Lemma 9.1 *$A_1 \| A_2$ has a strategy iff A_i have strategies $Strat_i$, $i = 1, 2$ and the parallel composition of $Strat_1$ and $Strat_2$ is not empty.*

9.2 A fixpoint solution to controller synthesis

In this section we present a first approach to solving controller synthesis, based in the results of the paper [MPS95]. The approach follows the spirit of model checking using fix-points of symbolic operators [Yov93, HNSY94]. The idea is to adapt the pre() operator to take into account controllability.

Controllable predecessors

Consider a CTA A with set of edges $E = E^u \cup E^c$. Given a symbolic state S of A, we define the following *controllable-predecessor* operators:

$$\begin{array}{lcll} \mathsf{controlled\text{-}pre}(S) & \stackrel{\text{def}}{=} & \{s \mid & \exists \delta \in \mathsf{R},\ e \in E,\ s' \in S \,.\, s \xrightarrow{\delta} \xrightarrow{e} s' \\ & & & \wedge \\ & & & \forall \delta_u \in \mathsf{R} \,. \quad \text{if } \exists e_u \in E^u,\ s_u \notin S \,.\, s \xrightarrow{\delta_u} \xrightarrow{e_u} s_u \\ & & & \qquad\qquad \text{then } \exists \delta_c < \delta_u,\ e_c \in E^c,\ s_c \in S \,.\, s \xrightarrow{\delta_c} \xrightarrow{e_c} s_c\} \end{array}$$

Intuitively, if a state s belongs in $\mathsf{controlled\text{-}pre}(S)$ then:

- s can reach S in one step (time passage plus discrete transition);

- if the environment, starting from s, can lead the system out of S after some time δ_u, then the controller can act earlier and lead the system to S.

In section 10.2 we show how controlled-pre() can be computed effectively, either by using a straightforward reduction to more basic operators, or by a special technique of quantifier elimination. Both ways are costly since they involve non-convex polyhedra.

[3] In fact, a much stronger result holds, namely: for each strategy *Strat* of $A_1 \| A_2$ there exist strategies $Strat_1$ and $Strat_2$ of A_1 and A_2 such that *Strat* and $Strat_1 \| Strat_2$ are equivalent, and vice versa. Equivalence is taken modulo elimination of the auxiliary clock z and all auxiliary discrete states $Q_1 \times Q_2' \cup Q_1' \times Q_2$ in the composite CTA $A_1 \| A_2$.

Fix-point characterization of winning states

The following lemma is a result of [MPS95].

Lemma 9.2 *A state is winning with respect to $\hat{S}$-invariance iff it belongs to the greatest fix-point:*

$$\mu S \ . \ \hat{S} \cap \mathsf{controlled\text{-}pre}(S)$$

A state is winning with respect to $\hat{S}$-reachability iff it belongs to the least fix-point:

$$\nu S \ . \ \hat{S} \cup \mathsf{controlled\text{-}pre}(S)$$

The operator controlled-pre() is *monotonic*, that is, $S_1 \subseteq S_2$ implies $\mathsf{controlled\text{-}pre}(S_1) \subseteq \mathsf{controlled\text{-}pre}(S_2)$. Thus, by Tarski's theorem [Tar55], the fix-points of lemma 9.2 always exist and can be computed using the algorithms shown below:

```
StaysInFixpointSynthesis (Ŝ)
{
  S_0 := Ŝ ;
  repeat
    S_{i+1} := S_i ∩ controlled-pre(S_i) ;
  until (S_{i+1} = S_i)
  return (S_i) ;
}
```

```
LeadsToFixpointSynthesis (Ŝ)
{
  S_0 := Ŝ ;
  repeat
    S_{i+1} := S_i ∪ controlled-pre(S_i) ;
  until (S_{i+1} = S_i)
  return (S_i) ;
}
```

The above fix-point algorithms have been implemented in the module `synth-kro` of KRONOS (see section 11.3). Running `synth-kro` on the TGC example of figure 9.2 with respect to the invariance property $\forall\Box$ (in $\Rightarrow$ down) gives the following set of winning states (for simplicity, some unreachable winning states are not shown):

$$\begin{array}{lllll}
 & (\text{far}, & \text{up}, & 0, & \mathsf{true}) \\
\vee & (\text{near}, & \text{up}, & 1, & x \leq 1 \wedge z =\leq 3) \\
\vee & (\text{near}, & \text{coming down}, & 2, & y < 1 \wedge x \leq y + 1) \\
\vee & (\text{near}, & \text{down}, & 2, & x \leq 5) \\
\vee & (\text{in}, & \text{down}, & 2, & x \leq 5) \\
\vee & (\text{far}, & \text{down}, & 3, & x = 0 \wedge z \leq 1) \\
\vee & (\text{far}, & \text{going up}, & 0, & y \leq 2 \wedge x \leq y) \\
\vee & (\text{near}, & \text{going up}, & 1, & y \leq 2 \wedge x + 1 \leq y \wedge z \leq y + 1)
\end{array}$$

Restricting the CTA to its winning states

Solving the controller-synthesis problem for a CTA A (independently of invariance or reachability) means computing the set of winning states, say, S^*, and then checking whether the initial state s_0 belongs to S^*. If $s_0 \in S^*$, then we can use S^* to exhibit a winning strategy, which will be represented as a new TA A_1. A_1 is the largest (in terms of states) sub-automaton of A, obtained by restricting A so that the "bad" choices of the controller are eliminated, while the choices of the environment are not affected at all. The set of states of A_1 is exactly S^*, thus, its semantic graph is the largest winning strategy.

More precisely, let $A = (\mathcal{X}, Q, q_0, E^c \cup E^u, \mathsf{invar})$, where $Q = \{q_1, ..., q_m\}$. S^* can be written as $(q_1, \zeta_1) \cup \cdots \cup (q_m, \zeta_m)$, where ζ_i is an $\mathcal{X}$-polyhedron, for $i = 1, ..., m$. Then, A_1 is defined as $(\mathcal{X}, Q, q_0, E_1^c \cup E_1^u, \mathsf{invar}_1)$, where:

- If $e = (q_i, \zeta, a, X, q_j) \in E^u$ and $\zeta \cap \zeta_i \neq \mathsf{false}$, then E_1^u contains the edge $(q_i, \zeta \cap \zeta_i, a, X, q_j)$.
- If $e = (q_i, \zeta, a, X, q_j) \in E^c$ and $\zeta \cap \zeta_i \cap \mathsf{pre}(e, S_j) \neq \mathsf{false}$, then E_1^c contains the edge $(q_i, \zeta \cap \zeta_i \cap \mathsf{pre}(e, S_j), a, X, q_j)$.
- For each $q_i \in Q$, $\mathsf{invar}_1(q_i)$ is defined to be $\mathsf{invar}(q_i) \cap S_i \cap \mathsf{time\text{-}pred}(\bigcup_{e \in E_1} \mathsf{guard}(e))$.

Back to the TGC example, the restricted CTA with respect to the previously computed set of winning states is shown in figure 9.4.

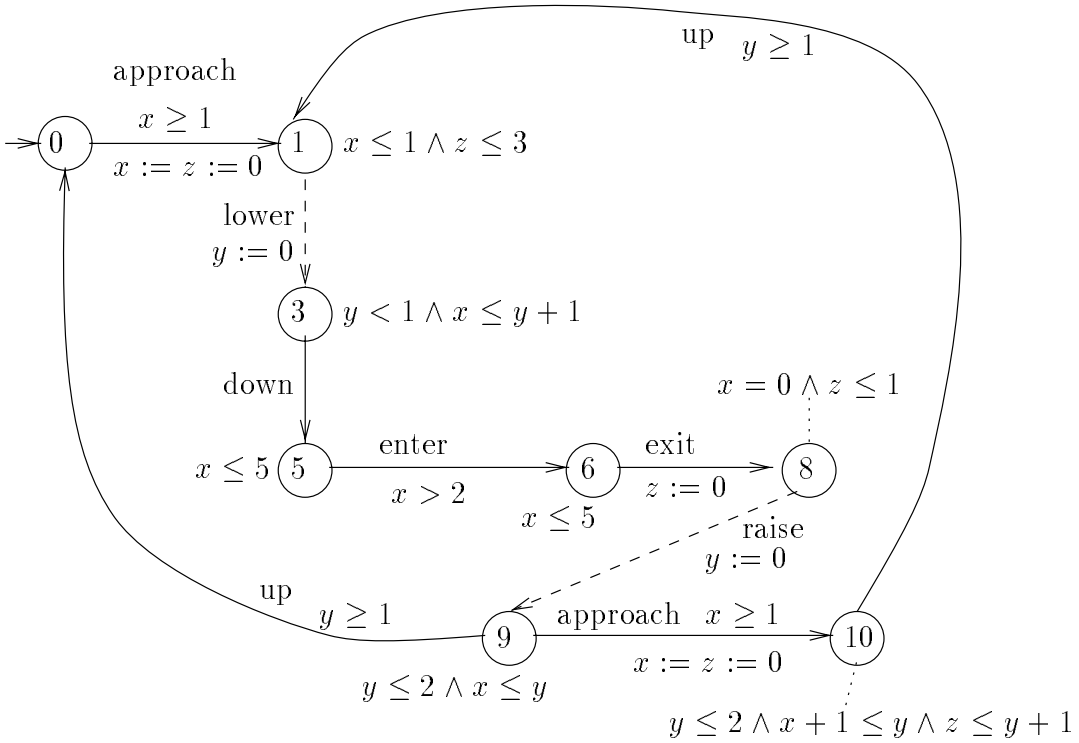


Figure 9.4: Restricting the CTA of figure 9.2 with respect to winning states.

9.3 On-the-fly controller synthesis

The high cost of the controlled-pre() operator used in the fix-point method presented above motivates the need for a less expensive approach. Here, we follow the philosophy of chapter 6 and reduce the problem to the untimed case. First, we present an on-the-fly algorithm for controller synthesis in untimed graphs. Then, we show how to use this algorithm on the STa-quotient of a CTA, to solve timed controller synthesis. Although the whole approach is only "half" on-the-fly, since the STa-quotient has to be generated a-priori, we believe it to be more practical, because it avoids costly operations as complementation altogether and can return a winning strategy as soon as it is found.

In the sequel, we first introduce the method in the context of untimed discrete-state systems and then present its extension to dense-time systems.

9.3.1 Untimed case

The method is based on two on-the-fly synthesis algorithms, for invariance and reachability. We have developed these algorithms for untimed systems, that is, graphs with controllable and uncontrollable transitions. The algorithms are based on a DFS similar to the one used for reachability analysis (see section 7.1). However, they differ in the following points:

- Instead of searching for paths in a graph, we are looking for sub-graphs: the reason is that a strategy has a branching instead of linear structure.
- A simple depth-first pass does not suffice, and back-tracking is necessary sometimes to update information about the controllability status of nodes.

Before presenting the algorithms, we introduce controller synthesis in the untimed setting and re-define the notions of strategy and winning strategy.

Untimed controller synthesis

We consider a finite graph $G = (V, \rightarrow)$, the edges of which are partitioned in two disjoint sets $\stackrel{c}{\rightarrow}$ and $\stackrel{u}{\rightarrow}$, representing the controllable and uncontrollable transitions, respectively. If $v \stackrel{c}{\rightarrow} w$ (resp. $v \stackrel{u}{\rightarrow} w$) then w is called a controllable (resp. uncontrollable) successor of v. We assume that G does not contain any sink nodes.

A strategy of G from an initial node v_0 is a subgraph *Strat* of G such that:

1. v_0 is a node of *Strat*.
2. For each node v of *Strat*, if $v \stackrel{u}{\rightarrow} w$ is an edge of G then w is a node of *Strat* and $v \stackrel{u}{\rightarrow} w$ is an edge of *Strat*.
3. For each node v of *Strat*, there is an edge $v \stackrel{c}{\rightarrow} w$ of *Strat*.

The definition differs from the one for the timed case, since it requires that the controller has a move from every node. As will become clear below, this requirement is technically convenient for adapting the method to the timed case. Moreover, we do not loose in generality. Indeed, consider a node v which has only uncontrollable successors. In our setting, there can be no strategy from v, whereas in a less restrictive setting there could be a strategy including all (uncontrollable) successors of v. However, we can do the following "trick": change the status of one (any) of the uncontrollable successors of v to controllable. Then, we can also obtain a strategy in the restrictive setting.

This transformation can be performed in general when a node v has only uncontrollable successors. One of them, say w, is chosen randomly and is marked controllable. This does not increase the choices of the controller, since, being in v, it can only chose w as its controllable successor. The choices of the environment are not restricted either, since a controllable successor *must* be chosen, thus, w will certainly be included in a strategy from v.

Now, consider a subset of nodes $\hat{V} \subseteq V$.

- *Strat* is winning with respect to $\hat{V}$-invariance (or, *Strat* stays in $\hat{V}$) if every node of *Strat* belongs to $\hat{V}$.
- *Strat* is winning with respect to $\hat{V}$-reachability (or, *Strat* leads to $\hat{V}$) if for any path $v_0 \rightarrow v_1 \rightarrow \cdots$ in *Strat*, there exists some $i = 0, 1, ...$, such that $v_i \in \hat{V}$.

A node v is called winning if there exists a winning strategy from v. In the sequel, we make the following simplifying hypothesis for a set $\hat{V}$ of target nodes for reachability: for every $v \in \hat{V}$, v has no uncontrollable successors and v is a controllable successor of itself.

Given a set of nodes $W \subseteq V$, define $\mathsf{preds}_c(W) = \{v \mid \exists w \in W \,.\, v \xrightarrow{c} w\}$ and $\mathsf{preds}_u(W) = \{v \mid \exists w \in W \,.\, v \xrightarrow{u} w\}$. The following result is the untimed version of lemma 9.2.

Lemma 9.3 *A node v is winning with respect to $\hat{V}$-invariance iff it belongs to the greatest fix-point:*

$$\mu W \,.\, \hat{V} \cap (\mathsf{preds}_c(W) \setminus \mathsf{preds}_u(\overline{W}))$$

A state is winning with respect to $\hat{V}$-reachability iff it belongs to the least fix-point:

$$\nu W \,.\, \hat{V} \cup (\mathsf{preds}(W) \setminus \mathsf{preds}_u(\overline{W}))$$

In words, the first part of the lemma says that v is winning iff $v \in \hat{V}$, every uncontrollable successor of v is winning, and at least one controllable successor of v is winning. The second part says that v is winning iff either $v \in \hat{V}$, or every uncontrollable successor of v is winning and at least one successor of v is winning.

The on-the-fly synthesis algorithms

Based on lemma 9.3 we derive two algorithms for computing a winning strategy (if one exists) with respect to invariance or reachability. The algorithms are shown in figures 9.5 and 9.6, respectively. Both of them are based on a DFS where nodes are marked with their controllability status while they are visited. Sets *Maybe*, *No* and *Yes* are used to store visited nodes and represent their marks. The algorithms also use a set of edges *Strat* representing the winning strategy. The set of edges *NegStrat* in the algorithm for invariance represents the counter-example "strategy" of the environment, in case a winning strategy for the controller does not exist. In the case of reachability, such a special structure is not needed, since the explored graph is also the counter-example.

In the algorithm for invariance, a node is initially marked *maybe*, until it is found that it cannot be winning, where-upon its mark is updated to *no*. Dually, in the algorithm for reachability, a node is initially marked *maybe*, until it is found that it is winning, where-upon its mark is updated to *yes*.

Intuitively, the invariance algorithm works as follows. Procedure `Reach` explores the graph in depth-first order. For each newly visited node v, the uncontrollable successors of v are explored first. If not all of them are winning then v cannot be winning either, and control moves to procedure `UndoMaybe` (line 1). Otherwise, its controllable successors are explored by procedure `CheckControllable` (line 2). If none of them is winning then again v cannot be winning. Procedure `UndoMaybe` updates a node v which was falsely assumed to be winning, as well as all predecessors of v, since their computed strategies are no longer valid. In particular, all uncontrollable predecessors of v are not winning. Also, if w is a controllable predecessor of v, then a new controllable successor should be found for w (line 10). This is done by procedure `CheckControllable`, which explores the remaining controllable successors of v.

At the end of the algorithm, and if the answer is not *no*, then the sub-graph represented by the set of edges *Strat* contains the winning strategy. If the answer is *no*, then *NegStrat* contains a counter-example, that is, a "counter-strategy" showing that the controller has no way to avoid the environment leading the system to a bad state.

```
OnTheFlyStrategyStaysIn (v, V̂) {
  No := Maybe := {} ;
  Strat := NegStrat := {} ;
  Controllable := {v1 -c-> v2 | v1, v2 ∈ V} ;
  if (Reach(v) = no) then return "No strategy exists" ;
  else return "Found strategy Strat" ;

  Reach(v) {
    if (v ∈ No or v ∉ V̂) then return no ;
    if (v ∈ Maybe) then return maybe ;
    Maybe := Maybe ∪ {v} ;    /* new node */
    for each (v -u-> w) do
      if (Reach(w) = no) then
        NegStrat := NegStrat ∪ {v -u-> w} ;
        goto FAIL ;                                          (1)
      end if
    if (CheckControllable(v) ≠ no) then
      Strat := Strat ∪ {v -u-> v' | v' ∈ V} ;                 (2)
      return maybe ;
    else
      NegStrat := NegStrat ∪ {v -c-> w} ;
    end if
  FAIL:
    UndoMaybe(v) ;
    return no ;
  }
  CheckControllable(v) {
    while (∃v -c-> w ∈ Controllable) do                      (3)
      Controllable := Controllable \ {v -c-> w} ;
      if (Reach(w) ≠ no) then                                (4)
        Strat := Strat ∪ {v -c-> w} ;                        (5)
        return maybe ;
      end if
    end while
    return no ;                                              (6)
  }
  UndoMaybe(v) {    /* Update from Maybe to No */
    Maybe := Maybe \ {v} ;                                   (7)
    No := No ∪ {v} ;                                         (8)
    while (∃w ∈ Maybe . w → v ∈ Strat) do
      Strat := Strat \ {w → v} ;                             (9)
      NegStrat := NegStrat ∪ {w → v} ;
      if (w -u-> v) then                                     (10)
        UndoMaybe(w) ;
      else if (CheckControllable(v) = no) then
        UndoMaybe(w) ;
    end while
  }
}
```

Figure 9.5: On-the-fly controller synthesis for invariance.

The algorithm for reachability works in a dual manner. A difference is that edges which are inserted in *Strat* are no longer removed. A node v is inserted in *Yes* (procedure `UndoMaybe`) only if all its uncontrollable successors are already in *Yes* and it has at least one successor. When v is inserted in *Yes*, its predecessors are also updated: if v was the "missing" successor for a node w to be winning, then procedure `UndoMaybe` is called recursively for w. Another difference from the algorithm for invariance is that here the counter-example strategy is not explicitly shown: this is because the explored graph itself is a counter-example.

Termination of both algorithms is guaranteed by the fact that the graph is finite. The complexity is linear in the size of the graph. For invariance, each node and edge is considered at most twice: one time when they are inserted in *Maybe* or *Strat* and possibly a second time to be removed. Reachability is similar, except that edges are considered at most once. Correctness of the algorithms is proven in lemmas 9.4 and 9.5.

Lemma 9.4 *The algorithm of figure 9.5 computes a winning strategy starting from a node v and staying in $\hat{V}$ iff such a strategy exists.*

Proof: First notice that for any visited node v, `Reach`(v) returns *no* iff at that point in the execution of the algorithm, $v \in No$.

We prove the following facts by induction on the number of nodes:

1. If $v \in No$ or $v \notin \hat{V}$ then at the end of the algorithm there is no edge in *Strat* having as target v.

2. If $v \in Maybe$ at the end of the algorithm then *Strat* contains a winning strategy from v.

3. If $v \in No$ then there exists no winning strategy from v.

For fact 1, observe that all edges $w \rightarrow v$ are removed from *Strat* whenever v is inserted in *No* (line 9). Also, if $v \notin \hat{V}$ then `Reach`(v) returns *no* and no edge leading to v is ever inserted in *Strat* (line 4).

For fact 2, let $v \in Maybe$ at the end of the algorithm. We shall prove that all uncontrollable edges from v and at least one controllable edge are in *Strat*. When v is visited by `Reach`(v), `CheckControllable`(v) returns *maybe* (line 2), otherwise v would be removed from *Maybe* (line 7). At this point all uncontrollable edges and at least one controllable edge from v are inserted in *Strat* (lines 2 and 5). During the algorithm, an edge is removed (line 9) only if a successor of v is inserted in *No*. This edge cannot be uncontrollable since v would also be inserted in *No* (line 10). The removed controllable edge is replaced by another controllable edge since the call to `CheckControllable`(w) returns *maybe*. Having shown that all uncontrollable edges from v and at least one controllable edge are in *Strat*, we can conclude by fact 1 that all successors by these edges are also in *Maybe*. By the induction hypothesis, all these successors have a winning strategy, and the result follows by lemma 9.3.

For fact 3, let $v \in No$. Notice that the only procedure inserting nodes in set *No* is `UndoMaybe` (line 8). This means that during `Reach`(v), control has reached point `FAIL`. Then, either there exists $v \xrightarrow{u} w$ such that $w \in No$ (line 1), or for all $v \xrightarrow{c} w$, $w \in No$ (lines 3 and 6). In both cases lemma 9.3 applies, showing that no winning strategy exists from v.

Facts 2 and 3 settle the "if" and "only if" parts of the proof, respectively. ∎

The following lemma proves correctness for the reachability algorithm. The proof is given in the appendix.

```
OnTheFlyStrategyLeadsTo (v, V̂) {
  Yes := Maybe := {} ;
  Strat := {} ;
  if (Reach(v) = yes) then return "Found strategy Strat" ;
  else return "No strategy exists" ;

  Reach(v) {
    if (v ∈ Yes or v ∈ V̂) then return yes ;
    if (v ∈ Maybe) then return maybe ;
    Maybe := Maybe ∪ {v} ;    /* new node */
    for each (v -u-> w) do
      if (Reach(w) ≠ yes) then return maybe ;                    (1)
    if (CheckControllable(v) = yes) then                         (2)
      UndoMaybe(v) ;                                             (3)
      return yes ;
    end if
    return maybe ;
  }

  CheckControllable(v) {
    for each (v -c-> w) do
      if (Reach(w) = yes) then
        Strat := Strat ∪ {v -c-> w} ;                            (4)
        return yes ;
      end if
    return maybe ;
  }

  UndoMaybe(v) {    /* Update from Maybe to Yes */
    Maybe := Maybe \ {v} ;
    Yes := Yes ∪ {v} ;
    Strat := Strat ∪ {v -u-> v' | v' ∈ V} ;                      (5)
    while (∃w ∈ Maybe . w → v ∧ ∀w -u-> v' . v' ∈ Yes) do
      if (w -c-> v) then                                         (6)
        Strat := Strat ∪ {w -c-> v} ;                            (7)
        UndoMaybe(w) ;
      else if (CheckControllable(w) = yes) then                  (8)
        UndoMaybe(w) ;
      end if
    end while
  }
}
```

Figure 9.6: On-the-fly controller synthesis for reachability.

Lemma 9.5 *The algorithm of figure 9.6 computes a winning strategy starting from v and leading to $\hat{V}$ iff such a strategy exists.*

An example showing that the algorithms are on-the-fly is given in the following section.

9.3.2 Timed case

We are now going to use the algorithms of the previous section to synthesize controllers for CTA. Consider a CTA A with set of edges $E = E^c \cup E^u$. Let $\approx$ be a STaB on A and let $G_\approx = (\mathcal{C}, \rightarrow_\approx)$ be the $\approx$-quotient of A. Recall that $\rightarrow_\approx$ is a labeled transition relation, so that if $C \rightarrow_\approx C'$ for two classes $C, C' \in V$, then either $C \xrightarrow{e}_\approx C'$ for some edge $e \in E$ (i.e., C' contains the e-successors of C), or $C \xrightarrow{\tau}_\approx C'$ (i.e., C' contains the immediate time successors of C).

From $G_\approx$ we build the graph $G = (\mathcal{C}, \xrightarrow{c} \cup \xrightarrow{u})$, where $\xrightarrow{c}$ and $\xrightarrow{u}$ are as follows:

- If $C \xrightarrow{e}_\approx C'$ for some edge $e \in E^c$ (resp. $e \in E^u$) then $C \xrightarrow{c} C'$ (resp. $C \xrightarrow{u} C'$) is an edge of G.
- If $C \xrightarrow{\tau}_\approx C'$ then $C \xrightarrow{c} C'$ is an edge of G.
- If, at the end of the above two steps, a node C has no out-going edge $C \xrightarrow{c} C'$, then some edge $C \xrightarrow{u} C''$ is removed arbitrarily, and $C \xrightarrow{c} C''$ is added.

In other words, discrete transitions do not change controllability status, while time transitions are considered controllable. The intuition behind this choice is the following:

- If a node C has no τ-successor, then the controller has no other choice but wait for the environment to make a move (recall that the system is assumed deadlock-free). If more than one uncontrollable moves are possible, marking arbitrarily one of them controllable, does not affect the existance of winning strategies, as explained above.
- If a node C has a τ-successor C' then
  - If C has no controllable successor, then it has no choice but wait, that is, move to C'.
  - If C has at least one controllable successor C'', then the controller can choose either to move to C'' or wait, which corresponds to choosing C' as its move.

We claim that the above construction is enough to reduce timed controller synthesis to the untimed case. Let $\hat{\mathcal{C}} \subseteq \mathcal{C}$ be a set of nodes of G and consider the set of states $\hat{S} = \cup\{C \mid C \in \hat{\mathcal{C}}\}$. Also let s be a state of A and C be the class of s in $\mathcal{C}$.

Lemma 9.6 *There is a winning strategy of A with respect to $\hat{S}$ starting from a state s iff there is a winning strategy of G with respect to $\hat{\mathcal{C}}$ starting from C, where C is the class of s.*

Proof: We prove the result for the invariance case. The proof for reachability is similar.

First assume that C is not winning. We shall prove by induction on the number of nodes in G that no state $s \in C$ has a winning strategy. By lemma 9.3, either $C \notin \hat{\mathcal{C}}$, or there exists $C \xrightarrow{u} C'$ and C' has no winning strategy, or for all $C \xrightarrow{c} C''$, C'' has no winning strategy. In the first case, $s \notin \hat{S}$. In the second case, by definition of G, $C \xrightarrow{e} C'$ for some $e \in E^u$. Thus,

$s \xrightarrow{e} s'$ for some $s' \in C'$ and from the induction hypothesis, s' has no winning strategy. In the third case, by definition of G, there is no C'' having a winning strategy and either $C \xrightarrow{\tau} C''$ or $C \xrightarrow{e} C''$ for some $e \in E^c$. Thus, there exist no $\delta \in \mathsf{R}$, $e \in E^c$ such that $s \xrightarrow{\delta}\xrightarrow{e} s'$ and s' has a winning strategy.

For the "only if" part of the proof, assume that there is a winning strategy $Strat_{\approx}$ starting from C and let $s \in C$. Consider the (possibly infinite) semantic graph *Strat* reachable by s and inscribed in $Strat_{\approx}$. It is easy to see that *Strat* satisfies the four conditions of the definition of a strategy. Moreover, all states in *Strat* belong to $\hat{S}$ since all classes in $Strat_{\approx}$ belong to $\hat{\mathcal{C}}$. Thus, *Strat* is a winning strategy. ■

The strategy of G corresponds to a strategy of A given in symbolic form. At each symbolic state (class) the controller chooses either to let time pass (τ-edge) or make a move (e-edge). In the latter case, the move can also be delayed (that is, the strategy is not time-deterministic) as long as the system remains in the same symbolic state.

Example. We illustrate the on-the-fly algorithm for invariance on the modified TGC example of figure 9.2. The graph G corresponding to this system is shown in figure 9.7. Edges labeled "C" correspond to τ-edges (self-loops are not shown, for simplicity). All edges except those labeled "C", "lower" or "raise" are uncontrollable.

As previously, we are interested in computing a controller so that the closed system satisfies the invariance $\forall\Box$ (in $\Rightarrow$ down). The implication (in $\Rightarrow$ down) holds at all nodes of G except nodes 5 and 9. Executing the algorithm of figure 9.5 on G, we obtain the strategy shown in figure 9.8. Assuming that in procedure `CheckControllable`, "C" labeled edges are explored last [4], then apart from the portion corresponding to the computed strategy, the rest of the graph is not explored at all, which shows that the algorithm is on-the-fly.

The nodes of the strategy of figure 9.8 are the zones shown below:

0:	(far,	up,	0,	$x < 1$)
1:	(far,	up,	0,	$x \geq 1$)
4:	(near,	up,	1,	$x \leq 1 \wedge z < x + 1$)
8:	(near,	coming down,	2,	$y < 1 \wedge x \leq y + 1 \wedge x < z + 2$)
10:	(near,	down,	2,	$2 < x \leq 5$)
11:	(near,	down,	2,	$x \leq 2$)
12:	(in,	down,	2,	$x \leq 5$)
16:	(far,	down,	3,	$x = 0 \wedge z \leq x$)
18:	(far,	going up,	0,	$x \geq 1 \wedge 1 \leq y \leq 2$)
20:	(far,	going up,	0,	$x < 1 \wedge x = y$)
24:	(near,	going up,	1,	$y \leq 2 \wedge x + 1 \leq y \wedge z < x + 1$)

Notice that the set of states defined by the zones above is contained in the set of winning states computed with the fix-point method of section 9.2. Consequently, we can restrict the original CTA with respect to the above states, and get a TA corresponding to the closed system.

[4] This a good heuristic, since "C"-edges come from τ-transitions, and choosing another controllable successor means attempting first the policy where the controller acts as soon as possible. Most of the times this is a good policy.

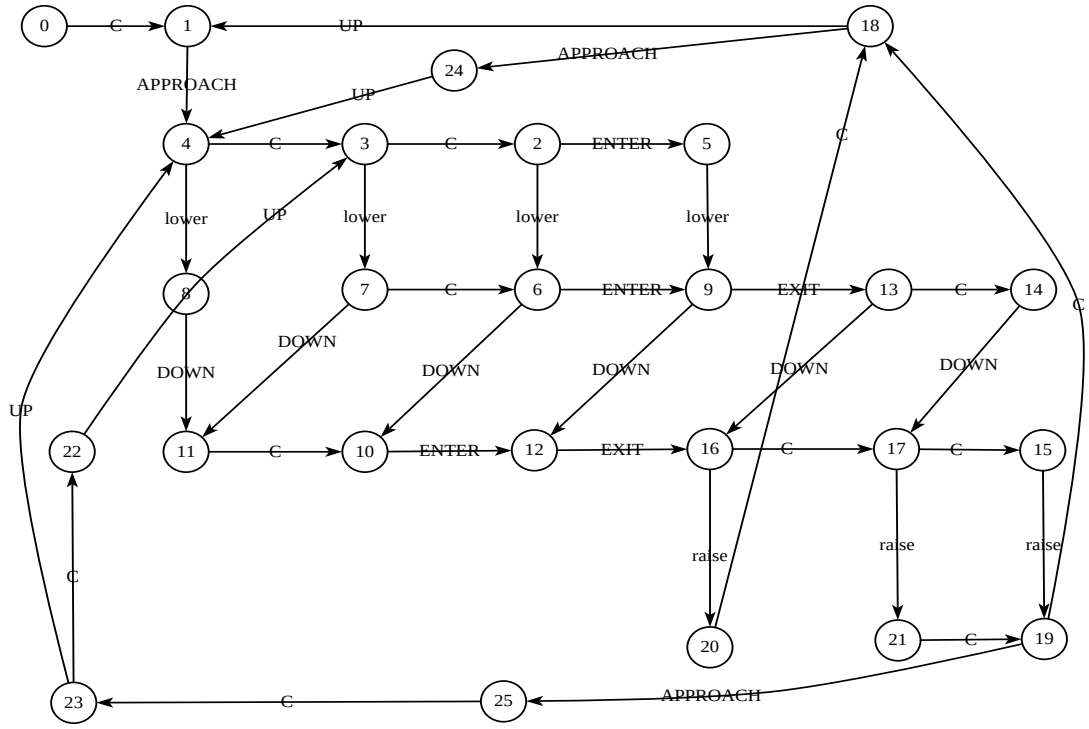


Figure 9.7: The STa-quotient of the CTA of figure 9.2, used for on-the-fly controller synthesis.

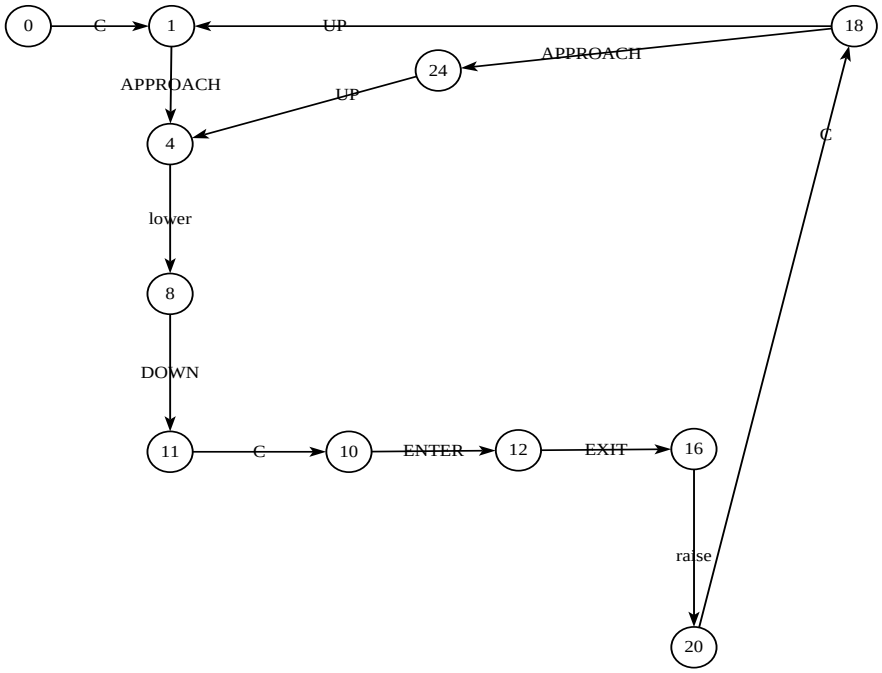


Figure 9.8: A strategy for the graph of figure 9.7.

Relation to the literature

Controller synthesis is close to the theory of *games*. In the domain of formal methods, pioneering have been the works of [RW87, PR89], who studied the problem in the untimed case. In the timed case, [HW91] use the framework of [RW87] to solve controller synthesis for deterministic TA and [MPS95] present a symbolic fix-point algorithm for general TA. To our knowledge, on-the-fly algorithms for controller-synthesis have not been proposed elsewhere. We are not aware of any definition of (semantic or syntactic) parallel composition in the presence of controllability either. A first version of the definition of strategies has been joint work with K. Altisen [Alt98].

Part III

Implementation and Tools

Chapter 10

Symbolic representation

The most important part in the implementation of the algorithms presented in the previous chapters is the representation of symbolic states and their operations. Any symbolic state can be written as a union of the form $(q_1, \zeta_1) \cup \cdots \cup (q_m, \zeta_m)$, where each q_i is a discrete state and each ζ_i is a polyhedron, for $i = 1, ..., m$. We choose a straightforward representation for discrete states by their index $i \in \{1, ..., m\}$. It remains to show how polyhedra can be represented.

In this thesis we have given emphasis on the usage of convex polyhedra, whenever possible. This is because they admit a space-efficient data structure, namely, a $(n+1) \times (n+1)$ square matrix, for n clocks, and time-efficient operations, of constant-time complexity to worst-case complexity $O(n^3)$. Whenever it is necessary to use non-convex polyhedra (for instance, in the synthesis algorithm of section 9.2), we represent them by lists of matrices (i.e., unions of convex polyhedra).

In the rest of this chapter, we consider a set of clocks $\mathcal{X} = \{x_1, ..., x_n\}$.

10.1 Difference Bound Matrices

Bounds

A *bound* is a pair $(c, \prec)$ where $c \in \mathsf{Z} \cup \{\infty\}$ and $\prec \in \{<, \leq\}$. The usual order $<$ on integers can be extended to members of $\mathsf{Z} \cup \{\infty\}$ so that $c < \infty$ for any $c \in \mathsf{Z}$. Then, a total order is defined on bounds, where $(c, \prec)$ is *stricter* than $(c', \prec')$ if either $c < c'$ or $c = c'$ and $\prec = <$, $\prec' = \leq$. We write $(c, \prec) < (c', \prec')$ to denote the fact that $(c, \prec)$ is stricter than $(c', \prec')$ and $(c, \prec) \leq (c', \prec')$ if either $(c, \prec) < (c', \prec')$ or the two bounds are identical. The two greatest bounds with respect to this order, namely, $(\infty, <)$ and $(\infty, \leq)$ are said to be *trivial*. We also define an order between real numbers and bounds. If δ is a (negative or positive) real and $(c, \prec)$ a bound, then we write $\delta \leq (c, \prec)$ if, either $\prec = <$ and $\delta < c$, or $\prec = \leq$ and $\delta \leq c$.

The minimum of two bounds $(c, \prec)$ and $(c', \prec')$, denoted $\min((c, \prec), (c', \prec'))$, is $(c, \prec)$ if $(c, \prec) \leq (c', \prec')$ and $(c', \prec')$ otherwise. The addition of $(c, \prec)$ and $(c', \prec')$, denoted $(c, \prec) + (c', \prec')$, is defined to be the bound $(c + c', \prec'')$, where $\prec''$ is $<$ if one of $\prec, \prec'$ is $<$ and $\leq$ otherwise. The *complement* of $(c, \prec)$ is the bound $(c', \prec')$ such that $(c, \prec) + (c', \prec') = (0, <)$. For instance, the complement of $(5, <)$ is $(\Leftrightarrow 5, \leq)$. The *rounding* of a bound $(c, <)$ (resp. $(c', \leq)$) is defined to be the bound $(c, \leq)$ (resp. $(c', \leq)$). Let $\mathsf{round}()$ denote the rounding operator on bounds.

DBMs and representation of convex polyhedra

A *difference bound matrix* (DBM) [Dil89] of dimension n is a $(n+1) \times (n+1)$ square matrix M, the elements of which are bounds. For $0 \leq i, j \leq n$, we write $M(i,j)$ for the element of M in row i and column j.

The idea behind the representation of a convex polyhedron by a DBM M is that each element $M(i,j)$ will encode the upper bound of the clock difference $x_i \Leftrightarrow x_j$. For example, the atomic constraint $x_i \Leftrightarrow x_j < 5$ will be encoded as $M(i,j) = (5, <)$, while $x_i \Leftrightarrow x_j \geq 3$ will be encoded as $M(j,i) = (\Leftrightarrow 3, \leq)$. Row and column 0 are used to encode bounds on single clocks, for instance, $x_i < 5$ is encoded as $M(i,0) = (5, <)$ and $x_i \geq 3$ as $M(0,i) = (\Leftrightarrow 3, \leq)$.

More formally, consider a convex $\mathcal{X}$-polyhedron $\zeta = \zeta_1 \cap \cdots \cap \zeta_m$, where $\zeta_1, ..., \zeta_m$ is a set of hyperplanes. ζ is represented by a DBM M of dimension n, such that:

- If some ζ_k is defined by an atomic constraint of the form $x_i \Leftrightarrow x_j < c$ (resp. $x_i \Leftrightarrow x_j \leq c$) then $M(i,j) = (c, <)$ (resp. $M(i,j) = (c, \leq)$).
- If some ζ_k is defined by an atomic constraint of the form $x_i \Leftrightarrow x_j > c$ (resp. $x_i \Leftrightarrow x_j \geq c$) then $M(j,i) = (\Leftrightarrow c, <)$ (resp. $M(j,i) = (\Leftrightarrow c, \leq)$).
- If some ζ_k is defined by an atomic constraint of the form $x_i < c$ (resp. $x_i \leq c$) then $M(i,0) = (c, <)$ (resp. $M(i,0) = (c, \leq)$).
- If some ζ_k is defined by an atomic constraint of the form $x_i > c$ (resp. $x_i \geq c$) then $M(0,i) = (\Leftrightarrow c, <)$ (resp. $M(0,i) = (\Leftrightarrow c, \leq)$).
- All other elements of M are set to $(\infty, <)$.

For example, if $n = 2$, the polyhedron $x_1 \leq 2 \wedge x_2 > 3$ can be represented by the DBM:

	x_0	x_1	x_2
x_0	$(0, \leq)$	$(\infty, <)$	$(\Leftrightarrow 3, <)$
x_1	$(2, \leq)$	$(0, \leq)$	$(\infty, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, \leq)$

Inversely, any DBM M of dimension n defines a convex polyhedron ζ on $\{x_1, ..., x_n\}$, such that

$$\zeta \quad = \quad \bigcap_{1 \leq i \neq j \leq n} x_i \Leftrightarrow x_j \leq M(i,j) \cap \bigcap_{1 \leq i \leq n} (x_i \leq M(i,0) \cap x_i \geq \Leftrightarrow M(0,i))$$

For example, the DBM

	x_0	x_1	x_2
x_0	$(0, \leq)$	$(\infty, <)$	$(\Leftrightarrow 3, <)$
x_1	$(2, \leq)$	$(0, \leq)$	$(\Leftrightarrow 1, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, \leq)$

represents the same polyhedron as above, namely, $x_1 \leq 2 \wedge x_2 > 3$.

On the other hand, the DBM

	x_0	x_1	x_2
x_0	$(0, \leq)$	$(\infty, <)$	$(\Leftrightarrow 3, <)$
x_1	$(2, \leq)$	$(0, \leq)$	$(\infty, <)$
x_2	$(\infty, <)$	$(0, \leq)$	$(0, \leq)$

represents the empty polyhedron, since the conjunction of constraints $x_1 \leq 2 \wedge x_2 > 3 \wedge x_2 \leq x_1$ is unsatisfiable.

Canonical representation

As it becomes obvious from the above examples, two or more different DBMs might represent the same polyhedron. In order to be able to reduce semantic operations on polyhedra to syntactic DBM transformations, but also for efficiency reasons, we are interested in a *canonical* representation, where two DBMs represent the same polyhedron iff they are identical.

For this, we define the following partial order on DBMs of the same dimension n:

$$M_1 \leq M_2 \quad \text{iff} \quad \forall 0 \leq i,j \leq n \;.\; M_1(i,j) \leq M_2(i,j)$$

Then, it can be easily shown that for any DBM M representing a non-empty polyhedron ζ, there exists a DBM M' representing ζ and such that for all M'' representing ζ, $M' \leq M''$. M' is called the *canonical form* (or minimal form) of M.

It remains to find a canonical form for DBMs representing the empty polyhedron (for short, empty DBMs). Since there are many empty DBMs, one of them can be chosen by convention. For instance, we can agree that the empty DBM of dimension n, denoted $M^{\emptyset}$, is the following: $M^{\emptyset}(i,j) = (0,<)$, for any $i,j = 0,...,n$. Then, it suffices to be able to identify, given a DBM, whether it is empty or not and, in case it is, to replace it by $M^{\emptyset}$. Detecting whether a DBM is empty can be done while computing its canonical form, as described in the section that follows.

10.2 Implementation of symbolic operations

Canonical form

This operation is used to check whether a DBM is empty, and if not, to transform it to its canonical counter-part.

The idea is to view a DBM M of dimension n as a flow graph with $n+1$ nodes numbered 0 to n and $(n+1)\cdot(n+1)$ edges, where the edge from i to j has cost $M(i,j)$. The cost of a path $i_1, i_2, ..., i_m$, denoted $\mathsf{cost}(i_1, i_2, ..., i_m)$, is the bound $M(i_1,i_2) + M(i_2,i_3) + \cdots + M(i_{m-1},i_m)$. Then, it can be shown that M is empty iff there exists a cycle $i = i_1, i_2, ..., i_m = i$ such that $\mathsf{cost}(i_1, i_2, ..., i_m) < (0, \leq)$. Otherwise, M is equal to its canonical form iff for all $0 \leq i,j \leq n$, there exists no path $i = i_1, i_2, ..., i_m = j$ such that $\mathsf{cost}(i_1, i_2, ..., i_m) < \mathsf{cost}(i,j)$.

Thus, in order to compute the canonical form of M, we can use a shortest-path algorithm, like Floyd-Warshall's all-pairs shortest-path algorithm (figure 10.1). The complexity of the operation is $O(n^3)$. In the sequel, we denote $\mathsf{cf}(M)$ the canonical form of M. Unless explicitly mentioned, all DBMs in the sequel are assumed to be in canonical form.

c-closure

Given a a DBM M representing a polyhedron ζ, the c-closure of ζ, $\mathsf{close}(\zeta, c)$, is represented by the DBM M', where, for $0 \leq i \neq j \leq n$:

- if $M(i,j) > (c, \leq)$ then $M'(i,j) = (\infty, <)$;
- if $M(i,j) + (c, \leq) < (0, \leq)$ then $M'(i,j) = (\Leftrightarrow c, <)$;
- otherwise $M'(i,j) = M(i,j)$.

```
canonical_form (M)
{
  for k = 0, ..., n do
    for i = 0, ..., n do
      for j = 0, ..., n do
        M(i,j) := min(M(i,j), M(i,k) + M(k,j)) ;
        if (M(i,i) < (0, ≤)) then return M^∅ ;
    end for
  return M ;
}
```

Figure 10.1: Floyd-Warshall's all-pairs shortest-path algorithm.

That is, an upper bound such as $x \leq c'$, where $c' > c$, is replaced by $x < \infty$ (first line of the definition). Also, a lower bound such as $x \geq c'$, where again $c' > c$, is replaced by $x > c$. All other bounds remain unchanged [1].

Intersection

Given two DBMs M_1 and M_2 of dimension n, we define $\min(M_1, M_2)$ to be the DBM M such that $M(i,j) = \min(M_1(i,j), M_2(i,j))$, for all $0 \leq i, j \leq n$. Then, if M_1 and M_2 represent the polyhedra ζ_1 and ζ_2, respectively, the polyhedron $\zeta_1 \cap \zeta_2$ is represented by the DBM $\mathsf{cf}(\min(M_1, M_2))$.

Test for inclusion

The polyhedron represented by M_1 is included in the polyhedron represented by M_2 iff $\forall 0 \leq i, j \leq n \;.\; M_1(i,j) \leq M_2(i,j)$.

Orthogonal projections

Let M be a DBM representing the $\mathcal{X}$-polyhedron ζ and let $X \subseteq \mathcal{X}$ be a set of clocks, $X = \{x_{i_1}, ..., x_{i_m}\}$. Then, $\zeta/_X$ is represented by the DBM $\mathsf{cf}(M_1)$, where:

$$M_1(i,j) = \begin{cases} (\infty, <), \text{ if } i \text{ or } j \in \{i_1, ..., i_m\} \\ M(i,j), \text{ otherwise} \end{cases}$$

The dimension-restricting projection $\zeta\rfloor_X$ is represented by a DBM of dimension m, where the indices are re-arranged to map to $i_1, ..., i_m$ instead of $1, ..., m$. This can be achieved by a bit of hacking: since the elements of the diagonal of a DBM are useless (they encode the trivial constraint $x_k \Leftrightarrow x_k \leq 0$, for $k = 1, ..., n$) they can be used to store the values of $i_1, ..., i_m$, so that no extra memory is required.

[1] We should note that in the actual version of KRONOS, the implementation of close() is more sophisticated: instead for a single constant c, closure can be computed with respect to a number of constants $c_{i,j}$, for each pair $0 \leq i \neq j \leq n$. This generalization does not affect the theoretical results of the previous chapters. Moreover, it results in more efficient analysis, since the bounds for each clock (or pair of clocks) are closed "as soon as possible", thus yielding a smaller number of polyhedra during generation of graphs with post().

Diagonal projections

If M is a DBM representing the $\mathcal{X}$-polyhedron ζ, then $\nearrow\zeta$ and $\swarrow\zeta$ are represented by DBMs M_1 and $\mathsf{cf}(M_2)$, respectively, where:

$$M_1(i,j) = \begin{cases} (\infty, <), \text{ if } j = 0 \\ M(i,j), \text{ otherwise} \end{cases}$$

$$M_2(i,j) = \begin{cases} (0, \leq), \text{ if } i = 0 \\ M(i,j), \text{ otherwise} \end{cases}$$

Continuous time successors and predecessors

We describe the implementation of $\mathsf{until}(S_1, S_2)$ in the special case where S_i are zones, that is, $S_i = (q, \zeta_i)$, for $i = 1, 2$, where ζ_i is a convex polyhedron [2]. This special case of $\mathsf{until}()$ is used in the convex TA-MMGA (section 6.1.2). For simplicity, we write $\mathsf{until}(\zeta_1, \zeta_2)$.

Intuitively, there are three cases to consider:

1. Either $\zeta_1 \cap \zeta_2 \neq \emptyset$, in which case $\mathsf{until}(\zeta_1, \zeta_2) = \mathsf{time\text{-}pred}(\zeta_1 \cap \zeta_2) \cap \zeta_1$ (figure 10.2(a)).

2. Or $\zeta_1 \cap \zeta_2 = \emptyset$ and for every clock, either its upper bound in ζ_1 is greater than its lower bound in ζ_2 (e.g., $x \leq 4$ in ζ_1 and $x \geq 3$ in ζ_2), or these bounds are complementary (e.g., $y \leq 4$ in ζ_1 and $y > 4$ in ζ_2) This case, illustrated in figure 10.2(b), can be reduced to the first case above, by rounding strict bounds and then taking the intersection and time predecessors as before. For the above example, rounding $y > 5$ and taking the intersection with ζ_1 yields $3 \leq x \leq 4 \wedge y = 5$.

3. Or none of the above, in which case $\mathsf{until}(\zeta_1, \zeta_2) = \emptyset$ (figure 10.2(c)).

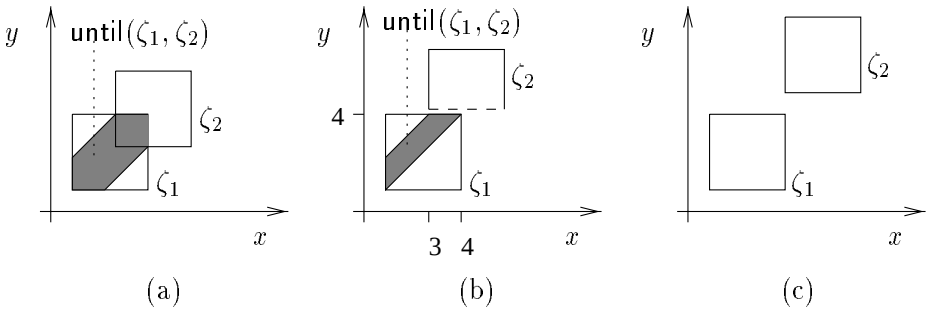


Figure 10.2: Three examples of the $\mathsf{until}()$ operator.

More precisely, if M_1 and M_2 are the DBMs representing ζ_1 and ζ_2, the DBM of $\mathsf{until}(\zeta_1, \zeta_2)$ is computed by the algorithm shown in figure 10.3.

Checking whether a clock is unbounded in a convex polyhedron

Given a DBM M representing the $\mathcal{X}$-polyhedron ζ and a clock $x_i \in \mathcal{X}$ the predicate $\mathsf{unbounded}(x_i, (q, \zeta))$ is implemented by the test $M(i, 0) = (\infty, <)$.

[2] The case where S_1 and S_2 are not associated with the same discrete state is trivial, since the result of $\mathsf{until}()$ is empty.

```
until (M_1, M_2)
{
  r := 0 ;
  for i = 1, ..., n do
    if (M_1(i,0) + M_2(0,i) < (0, ≤)) then
      if (M_1(i,0) + round(M_2(0,i)) = (0, ≤) ∧ r ≠ 1) then
        M_2(0,i) := round(M_2(0,i)) ;
        r := 2 ;
      else if (round(M_1(i,0) + M_2(0,i) = (0, ≤) ∧ r ≠ 2) then
        M_1(i,0) := round(M_1(i,0)) ;
        r := 1 ;
      else return M^∅ ;
  end for
  M := time-pred(cf(min(M_1, M_2))) ;
  return cf(min(M, M_1)) ;
}
```

Figure 10.3: The algorithm for computing until() on DBMs.

Extracting valuations from convex polyhedra (proof of lemma 8.1)

We recall lemma 8.1, used in building diagnostics (chapter 8).

> Given a convex $\mathcal{X}$-polyhedron ζ and a k-incomplete valuation $\mathbf{v}$, it takes $O(n^2)$ time to complete $\mathbf{v}$ in ζ, or find that this is not possible.

We now give the proof.

Let M be the DBM representing ζ and let $M(i,j) = (c_{i,j}, \prec_{i,j}$, for $0 \leq i,j \leq n$. For $i = 0, ..., k$, we define:

$$\delta_i \quad = \quad \begin{cases} 0, & \text{if } i = 0 \\ \mathbf{v}(x_i), & \text{if } 1 \leq i \leq k \end{cases}$$

Then, for $i = k+1, ..., n$, we choose δ_i such that:

$$\forall 0 \leq j < i \quad . \quad \Leftrightarrow c_{j,i} \prec_{j,i} \delta_i \prec_{i,j} c_{i,j}$$

If such a δ_i cannot be chosen for some i, then $\mathbf{v}$ cannot be completed. Otherwise, we let $\mathbf{v}'(x_i) = \delta_i$, for $i = 1, ..., n$. It is easy to see that $\mathbf{v}' \in \zeta$.

Regarding complexity, in the worst case we have $i = 0$, meaning that we have to perform $n \cdot (n \Leftrightarrow 1) + n$ comparisons and additions of bounds.

Finding time successors (proof of lemma 8.2)

We recall lemma 8.2, used in building diagnostics (chapter 8).

> Consider two convex $\mathcal{X}$-polyhedra ζ_1, ζ_2 such that $\mathsf{until}(\zeta_1, \zeta_2) = \zeta_1$. For any $\mathbf{v}_1 \in \zeta_1$, we can find in time $O(n)$ some $\delta \in \mathsf{R}$ such that $\mathbf{v}_1 + \delta \in \zeta_2$.

We now give the proof.

Let M_i be the DBM representing ζ_i, $i = 1, 2$. For $i = 1, ..., n$, let c_i (resp. d_i) be the constant of the bound $M_1(i, 0)$ (resp. $M_2(i, 0)$). Define δ_1 to be the minimum of $c_i \Leftrightarrow \mathbf{v}_1(x_i)$, for $i = 1, ..., n$. Intuitively, δ_1 is the minimum delay necessary for $\mathbf{v}_1$ to reach the "border" of ζ_1.

Now, if $\mathbf{v}_1 + \delta_1 \notin \zeta_1$ (this implies that the bound defining δ_1 was strict), then by definition of $\mathsf{until}()$, $\mathbf{v}_1 + \delta_1 \in \zeta_2$ and we are done.

If $\mathbf{v}_1 + \delta_1 \in \zeta_1$ (this implies that the bound defining δ_1 was $\leq$), then define $\delta_2 = \min_{i=1,...,n}\{\frac{1}{2} \cdot (d_i \Leftrightarrow \mathbf{v}_1(x_i) \Leftrightarrow \delta_1)\}$. Intuitively, δ_1 is the minimum delay necessary for $\mathbf{v}_1$ to reach the "border" of ζ_2. It is easy to show that $\mathbf{v}_1 + \delta_1 + \delta_2 \in \zeta_2$, thus, we can let $\delta = \delta_1 + \delta_2$.

10.3 Representation of non-convex polyhedra using lists of DBMs

Non-convex polyhedra do not admit an obvious efficient canonical representation. Instead, we present here a non-canonical representation (the one currently implemented in KRONOS), where a non-convex polyhedron is represented by a list of DBMs.

More precisely, if ζ is a polyhedron, then it can be written as a union of convex polyhedra $\zeta = \zeta_1 \cup \cdots \cup \zeta_k$. Now, if M_i is the DBM representing ζ_i, ζ is represented by the list $\{M_1, ..., M_k\}$.

This representation is not canonical, since there exist many possible ways to decompose ζ (notice, however, that each DBM M_i, $i = 1, ..., k$, is in canonical form). Moreover, there exists no obvious order with respect to which these decompositions can be compared, so that the "best" can be chosen. For instance, the polyhedron shown in figure 10.4 can be decomposed in at least three possible ways, and it is not clear which is the best, if one exists. Finally, even if such an order is defined, say, by convention [3], the canonicalization procedure will almost surely be at least as expensive as the algorithms we present below.

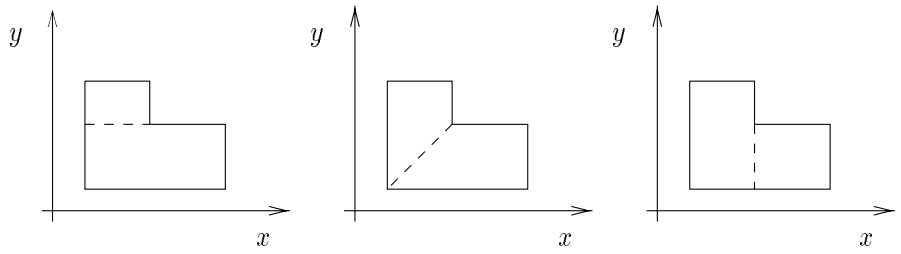


Figure 10.4: A non-convex polyhedron and three possible decompositions into convex polyhedra.

In the following paragraphs, we show how operations on symbolic states can be reduced to operations on non-convex polyhedra, which are in turn implemented as transformations of DBM lists.

Complementation

This is the only operation which does not preserve convexity, i.e., that can create a non-convex polyhedron from a convex one. Given a non-convex polyhedron $\zeta = \zeta_1 \cup \zeta_2$, we have $\overline{\zeta} = \overline{\zeta_1} \cap \overline{\zeta_2}$,

[3] This can always be done since we are talking about finite systems of linear constraints.

therefore, it suffices to consider complementation of convex polyhedra.

Assume that ζ is convex and let $\zeta = \zeta_1 \cap \cdots \cap \zeta_k$, where ζ_i is an $\mathcal{X}$-hyperplane, for $i = 1, ..., k$. We have: $\overline{\zeta} = \overline{\zeta_1} \cup \cdots \cup \overline{\zeta_k}$. Now, notice that the complement of an $\mathcal{X}$-hyperplane is an $\mathcal{X}$-hyperplane: for instance, the complement of $x \leq 4$ is $x > 4$, and the complement of $x \Leftrightarrow y \leq 4$ is $y \Leftrightarrow x < \Leftrightarrow 4$. Since an $\mathcal{X}$-hyperplane is a convex polyhedron, $\overline{\zeta}$ is expressed as a finite union of convex polyhedra, therefore, it can be represented by a list of DBMs $\{M_1, ..., M_k\}$, where M_i represents $\overline{\zeta_i}$.

An example is shown in figure 10.5: ζ is a convex polyhedron on $\{x, y\}$ and $\overline{\zeta}$ is the union of six convex polyhedra, one for each of the constraints $x \prec c_1$, $y \prec c_2$, $c_3 \prec x$, $c_4 \prec y$, $x \Leftrightarrow y \prec c_5$, $y \Leftrightarrow x \prec c_6$, where $\prec \in \{\leq, <\}$ and $c_1, ..., c_6$ are integer constants.

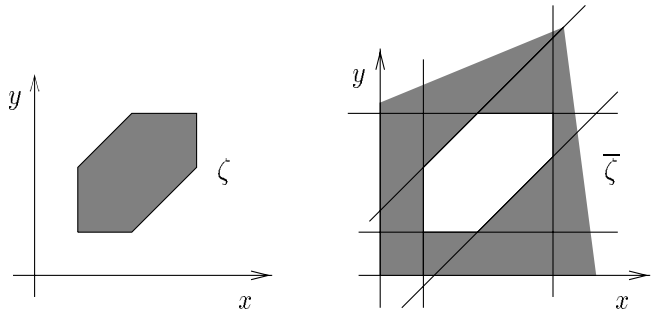


Figure 10.5: Complementation of convex polyhedra.

In order to compute $M_1, ..., M_k$ from the DBM M corresponding to ζ, we have to complement all non-trivial bounds of M. More precisely, let $\mathcal{X} = \{x_1, ..., x_n\}$ be the set of clocks and let $(c_1, \prec_1), ..., (c_k, \prec_k)$ be the set of non-trivial bounds of M, that is, for each $l = 1, ..., k$, there exist $0 \leq i_l, j_l \leq n$ such that $(c_l, \prec_l) = M(i_l, j_l)$. Notice that k is at most $n(n+1)$. Now, let $(c'_l, \prec'_l)$ be the complement of $(c_l, \prec_l)$. Then, for $l = 1, ..., k$, M_l is the canonical form of the DBM M'_l, such that $M'_l(i_l, j_l) = (c'_l, \prec'_l)$ and $M'_l(i, j) = (\infty, <)$ if $i \neq i_l$ or $j \neq j_l$.

It is worth noticing that with this construction some redundant DBMs might be created. For instance, if $\zeta = x \leq 2 \wedge y \leq 1$, then, using the above technique gives four polyhedra for $\overline{\zeta}$, namely, $x > 2$, $y > 1$, $x \Leftrightarrow y > 2$ and $y \Leftrightarrow x > 1$. However, the last two are redundant, since $x \Leftrightarrow y > 2 \subseteq x > 2$ and $y \Leftrightarrow x > 1 \subseteq y > 1$. Then a simple improvement of the above method is to test, for each newly created DBM M_l, whether it is included in any of $M_1, ..., M_{l-1}$. This method requires $n(n \Leftrightarrow 1)$ tests for inclusion (which can be done more efficiently than in $O(n^2)$ time, however, given the special form of $M_1, ..., M_k$).

Intersection

Consider two polyhedra $\zeta = \zeta_1 \cup \cdots \cup \zeta_k$ and $\zeta' = \zeta'_1 \cup \cdots \cup \zeta'_l$, represented by $\{M_1, ..., M_k\}$ and $\{M'_1, ..., M'_l\}$, respectively. Their intersection is equal to

$$\zeta \cap \zeta' = \bigcup_{1 \leq i \leq k, 1 \leq j \leq l} \zeta_i \cap \zeta'_j$$

Then, $\zeta \cap \zeta'$ can be represented by a list of at most $k \cdot l$ DBMs, one for each convex polyhedron $\zeta_i \cap \zeta'_j$.

Test for inclusion

Inclusion is reduced to intersection and complementation. Testing $\zeta \subseteq \zeta'$ comes down to checking $\zeta \cap \overline{\zeta'} = \emptyset$. This is done by computing the list of DBMs representing $\zeta \cap \overline{\zeta'}$ and checking that the canonical form of each one of them is $M^{\emptyset}$.

Projections

From the fact that existential quantification distributes over union, taking any type of projection of a polyhedron $\zeta = \zeta_1 \cup \cdots \cup \zeta_k$ comes down to computing the projection of each of $\zeta_1, ..., \zeta_k$, and then taking their union.

Continuous time successors and predecessors

We show how to compute $\mathsf{until}(\zeta_1, \zeta_2)$ and $\mathsf{since}(\zeta_1, \zeta_2)$ in the general case, where ζ_1, ζ_2 can be non-convex [4]. By definition:

$$\begin{aligned} \mathsf{until}(\zeta_1, \zeta_2) &= \{\mathbf{v} \in \zeta_1 \mid \exists \delta \,.\, \mathbf{v} + \delta \in \zeta_2 \wedge \forall \delta' < \delta \,.\, \mathbf{v} + \delta \in \zeta_1 \cup \zeta_2\} \\ \mathsf{since}(\zeta_1, \zeta_2) &= \{\mathbf{v} \in \zeta_2 \mid \exists \delta \,.\, \mathbf{v} \Leftrightarrow \delta \in \zeta_1 \wedge \forall \delta' < \delta \,.\, \mathbf{v} \Leftrightarrow \delta \in \zeta_1 \cup \zeta_2\} \end{aligned}$$

By the distributivity of existential quantification over union, we can easily prove that $\mathsf{until}()$ is *union-distributive* on its second argument [Yov93, Oli94, Alt98]:

$$\mathsf{until}(\zeta_1, \zeta_2 \cup \zeta_2') = \mathsf{until}(\zeta_1, \zeta_2) \cup \mathsf{until}(\zeta_1, \zeta_2')$$

Thus, we can assume that ζ_2 is convex. [Alt98, Oli94] prove that:

$$\mathsf{until}(\zeta_1, \zeta_2) = \zeta_1 \cap (\swarrow \zeta_2) \setminus \Big(\swarrow \big((\swarrow \zeta_2) \setminus (\zeta_1 \cup \zeta_2) \big) \Big)$$

For $\mathsf{since}()$, we can similarly prove the following:

$$\mathsf{since}(\zeta_1, \zeta_2) = \zeta_2 \cap (\nearrow \zeta_1) \setminus \Big(\nearrow \big((\nearrow \zeta_1) \setminus (\zeta_1 \cup \zeta_2) \big) \Big)$$

This means that the two operators can be computed using more basic operations. Notice, however, that two complementations are required in each case, in order to compute the polyhedra differences.

Controllable predecessors

The following lemma shows how $\mathsf{controlled\text{-}pre}()$ can be reduced to more basic operators.

Lemma 10.1 *Consider a symbolic state S and let:*

$$\begin{aligned} S_c &= \bigcup_{e \in E^c} \mathsf{pre}(e, S) \\ S_u &= \bigcup_{e \in E^u} \mathsf{pre}(e, S) \\ U &= \bigcup_{e \in E^u} \mathsf{pre}(e, \overline{S}) \end{aligned}$$

[4] For simplicity, we write $\mathsf{until}(\zeta_1, \zeta_2)$ instead of $\mathsf{until}((q, \zeta_1), (q, \zeta_2))$ and similarly for $\mathsf{since}()$.

Then:

$$\mathsf{controlled\text{-}pre}(S) \;=\; (\overline{U} \cap S_c) \;\cup\; \mathsf{until}(\overline{U}, \overline{U} \cap S_c) \;\cup\; \left(\overline{\mathsf{time\text{-}pred}(U)} \cap \mathsf{time\text{-}pred}(S_u)\right)$$

Proof: Intuitively:

- S_c represents the states which can lead to S by a controllable discrete transition.
- S_u represents the states which can lead to S by an uncontrollable discrete transition.
- U represents the states which can lead out of S by an uncontrollable discrete transition. Call these the *immediately bad states*.
- $\overline{U}$ represents the *temporarily safe states*, from which the environment does not have an immediate bad move.
- $\overline{\mathsf{time\text{-}pred}(U)}$ represents the *safe states*, from which the environment does not have a bad move, neither immediately, nor after letting any time pass.

Then, the controllable predecessors of S can be computed as the union of three sets:

- $\overline{U} \cap S_c$ represents the *immediately good states*, that is, temporarily safe states from which the controller can lead the system to S immediately.
- $\mathsf{until}(\overline{U}, \overline{U} \cap S_c)$ represents all states which can let time pass and reach an immediately good state, while continuously passing from temporarily safe states.
- $\overline{\mathsf{time\text{-}pred}(U)} \cap \mathsf{time\text{-}pred}(S_u)$ represents all safe states which have some predecessor in S.

Special thanks go to Karine Altisen who had the patience to finish the proof. ■

The implementation of controlled-pre() suggested by the above lemma is not the best from the point of view of efficiency, since it involves many complementations, namely, three complementations for $\overline{S}$, $\overline{U}$ and $\overline{\mathsf{time\text{-}pred}(U)}$, plus two more to compute until() as shown above.

A more efficient way to implement controlled-pre() would be to *eliminate quantifiers directly*, according to the definition of the operator (section 9.2). This idea has been exploited in [Alt98] to develop an alternative algorithm for computing controlled-pre(). Here, we only present the intuition behind the method through an example. The reader is referred to [Alt98] for more details.

We first define the following binary operator on symbolic states:

$$\begin{aligned} \mathsf{while\text{-}not}(S_1, S_2) &\stackrel{\text{def}}{=} \{s \mid \exists \delta \,.\, s+\delta \in S_2 \wedge \forall \delta' < \delta \,.\, s+\delta' \notin S_1\} \\ &= \{s \mid \exists \delta \,.\, s+\delta \in S_2 \wedge \nexists \delta' < \delta \,.\, s+\delta' \in S_1\} \end{aligned}$$

Then, computing $\mathsf{controlled\text{-}pre}(S)$ can be reduced to computing

Lemma 10.2 *Consider a symbolic state S and let:*

$$\begin{aligned} S_c &= \bigcup_{e \in E^c} \mathsf{pre}(e, S) \\ S_u &= \bigcup_{e \in E^u} \mathsf{pre}(e, S) \\ U &= \bigcup_{e \in E^u} \mathsf{pre}(e, \overline{S}) \end{aligned}$$

Then:

$$\mathsf{controlled\text{-}pre}(S) \;=\; (S_c \cup S_u) \setminus \mathsf{while\text{-}not}(S_c, U)$$

Proof: From the definition of controlled-pre() (section 9.2):

$$\begin{aligned}
\mathsf{controlled\text{-}pre}(S) = \;& \{s \mid \exists \delta \in \mathsf{R},\ e \in E,\ s' \in S \;.\; s \xrightarrow{\delta}\xrightarrow{e} s'\} \\
& \cap \\
& \{s \mid \forall \delta_u \in \mathsf{R} \;.\; (\exists e_u \in E^u,\ s_u \notin S \;.\; s \xrightarrow{\delta_u}\xrightarrow{e_u} s_u) \Rightarrow \\
& \qquad\qquad (\exists \delta_c < \delta_u,\ e_c \in E^c,\ s_c \in S \;.\; s \xrightarrow{\delta_c}\xrightarrow{e_c} s_c)\} \\
= \;& (S_c \cup S_u) \\
& \cap \\
& \{s \mid \forall \delta_u \in \mathsf{R} \;.\; (\exists e_u \in E^u,\ s_u \notin S \;.\; s \xrightarrow{\delta_u}\xrightarrow{e_u} s_u) \Rightarrow \\
& \qquad\qquad (\exists \delta_c < \delta_u,\ e_c \in E^c,\ s_c \in S \;.\; s \xrightarrow{\delta_c}\xrightarrow{e_c} s_c)\} \\
= \;& (S_c \cup S_u) \\
& \setminus \\
& \{s \mid \exists \delta_u \in \mathsf{R} \;.\; (\exists e_u \in E^u,\ s_u \notin S \;.\; s \xrightarrow{\delta_u}\xrightarrow{e_u} s_u) \wedge \\
& \qquad\qquad (\nexists \delta_c < \delta_u,\ e_c \in E^c,\ s_c \in S \;.\; s \xrightarrow{\delta_c}\xrightarrow{e_c} s_c)\} \\
= \;& (S_c \cup S_u) \setminus \mathsf{while\text{-}not}(S_c, U)
\end{aligned}$$

■

The above implementation of controlled-pre() is preferable to the one of lemma 10.1, since it uses only two complementations.

Example of direct quantifier elimination. We now present a simple example showing how while-not(ζ_1, ζ_2) is computed by direct elimination of quantifiers. Let $\zeta_1 = 3 \le x \le 5 \wedge 2 \le y \le 4$ and $\zeta_2 = 2 \le x \le 4 \wedge 3 \le y \le 5$. We use the notation $\zeta_i(x, y)$ to denote the set of linear constraints on x and y associated with ζ_i, for $i = 1, 2$. For example, $\zeta_2(x, y)$ is the set of constraints:

$$\begin{array}{rcccl}
2 & \le & x & \le & 4 \\
3 & \le & y & \le & 5 \\
\Leftrightarrow 3 & \le & x \Leftrightarrow y & \le & 1
\end{array}$$

whereas $\zeta_1(x + \delta, y + \delta)$ is the set of constraints:

$$\begin{array}{rcccl}
3 & \le & x + \delta & \le & 5 \\
2 & \le & y + \delta & \le & 4 \\
\Leftrightarrow 1 & \le & x \Leftrightarrow y & \le & 3
\end{array}$$

Now, while-not(ζ_1, ζ_2) is defined as:

$$\mathsf{while\text{-}not}(\zeta_1, \zeta_2) \;=\; \exists \delta \ge 0 \;.\; \zeta_2(x + \delta, y + \delta) \wedge \nexists 0 \le \delta' \le \delta \;.\; \zeta_1(x + \delta', y + \delta')$$

Eliminating the variable δ' from the set of constraints $\zeta_1(x + \delta', y + \delta') \wedge 0 \le \delta' \le \delta$, we obtain the following set of constraints:

$$\begin{array}{rcccl}
3 \Leftrightarrow \delta & \le & x & \le & 5 \\
2 \Leftrightarrow \delta & \le & y & \le & 4 \\
\Leftrightarrow 1 & \le & x \Leftrightarrow y & \le & 3
\end{array}$$

Taking the negation of the above set, together with the constraints $\zeta_2(x+\delta, y+\delta)$ and $\delta \geq 0$, we obtain the disjunction of two different sets of constraints (the rest are trivially unsatisfiable):

$$\begin{array}{rcccl} 2 & \leq & x+\delta & \leq & 4 \\ 3 & \leq & y+\delta & \leq & 5 \\ \Leftrightarrow 3 & \leq & x \Leftrightarrow y & \leq & 1 \\ 3 \Leftrightarrow \delta & < & x & & \\ 0 & \leq & \delta & \leq & \infty \end{array}$$

or

$$\begin{array}{rcccl} 2 & \leq & x+\delta & \leq & 4 \\ 3 & \leq & y+\delta & \leq & 5 \\ \Leftrightarrow 3 & \leq & x \Leftrightarrow y & \leq & 1 \\ 4 & < & y & & \\ 0 & \leq & \delta & \leq & \infty \end{array}$$

After elimination of δ in each of the above sets of constraints, we obtain:

$$\begin{array}{rcccl} & & x & < & 3 \\ & & y & \leq & 5 \\ \Leftrightarrow 3 & \leq & x \Leftrightarrow y & \leq & 0 \end{array}$$

or

$$\begin{array}{rcccl} 1 & < & x & \leq & 4 \\ 4 & < & y & \leq & 5 \\ \Leftrightarrow 3 & \leq & x \Leftrightarrow y & \leq & 0 \end{array}$$

Thus, the result of $\mathsf{while\text{-}not}(\zeta_1, \zeta_2)$ is the union of two convex polyhedra ζ_3 and ζ_4, corresponding to each of the above sets of constraints. The example is illustrated in figure 10.6.

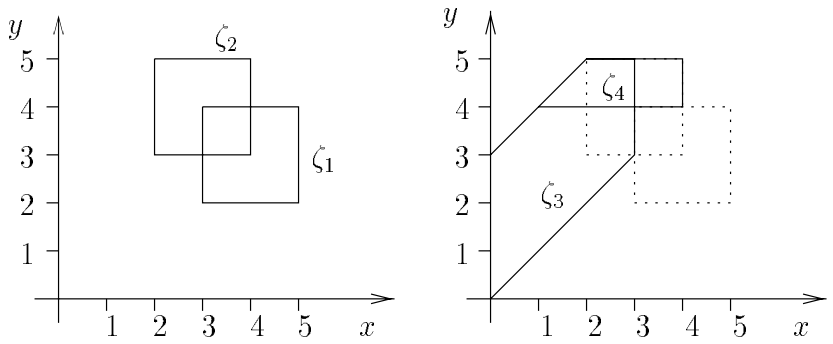


Figure 10.6: Direct quantifier elimination example.

Relation to the literature

DBMs have been introduced in [Dil89], along with the basic operations of canonicalization, intersection and projection. Most of the rest of the DBM operations described in this chapter, as well as the DBM-list polyhedra implementation, have been implemented by S.Yovine [Yov93], A.Olivero [Oli94] and C.Daws [Daw98]. The technique for direct quantifier elimination has been implemented by K.Altisen [Alt98].

Regarding optimizations, in the current implementation of KRONOS, auxiliary information is stored in each DBM indicating which bounds are *redundant*, in the sense that they can be deduced by the rest (non-redundant bounds). This permits faster complementation [Daw98]. Along this direction, [LLPY97] compute a minimal form of non-redundant bounds in a polyhedron, and use a list of non-redundant constraints for its representation.

Chapter 11

Tools

KRONOS [1] is a tool suite for the analysis of real-time systems. It has been developed at VERIMAG since 1992 [Yov93, DOY94, DOTY96, Yov97, BDM+98]. The current state of the tool is illustrated in figure 11.1. The upper part of the figure (enclosed in dotted box) represents what can be called the "first generation" of KRONOS, consisting in a collection of modules acting as *interpreters*, that is, performing the analysis directly on their input model. As of today, the modules of KRONOS are the following:

- `ptg` computes the parallel composition of a set of TA syntactically;
- `kronos` performs TCTL, TBA and reachability model checking (see section 11.1 below);
- `minim` computes the quotient graph of a TA with respect to the strong time-abstracting bisimulation;
- `synth-kro` performs controller synthesis for invariance and reachability, using the fixpoint method of section 9.2;
- `optikron` computes the set of active clocks per discrete state of a TA (i.e., the function `act`() of section 5.2.3), and accordingly optimizes the number of clocks using renaming and cross-clock assignments (see [Daw98] for more details).

The input language of these modules is the basic TA model, that is, finite-state automata with clocks, communicating by label synchronization.

The C-code generator module, called `kronos-open`, represents the next generation of KRONOS. It is based on the *compiler* philosophy of SPIN [Hol91], followed by OPEN-CAESAR [Gar98]. Given an input model, `kronos-open` produces C-code which can be in turn compiled to various executables, which perform the analysis for the specific input model. The interest behind this approach is that it permits to take advantage of the particularities of each input model in order to generate optimized code. Another difference from the first-generation tools is that `kronos-open` accepts a richer input language, namely, TA extended with bounded discrete variables and shared-variable or message-passing communication.

As part of the work for this thesis, we have contributed to the development of KRONOS by:

- extending `kronos` with a module computing the parallel composition of a set of TA on-the-fly;

[1] Named after the Titan of ancient Greek mythology, often indiscernible with *chronos*, which in Greek means "time".

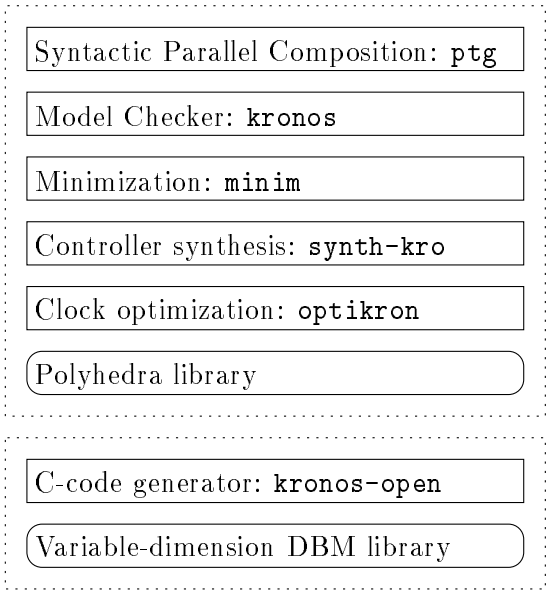


Figure 11.1: The modules of the KRONOS tool suite.

- extending `kronos` with a module for TBA model checking and reachability based on abstractions, which uses the on-the-fly parallel-composition module above;
- implementing the module `minim`;
- implementing the prototype version of `synth-kro`;
- implementing a library of variable-dimension DBMs, to be used when activity abstraction is applied during the analysis;
- implementing `kronos-open`: to date, it produces code for TBA model checking and reachability and uses the variable-dimension DBM module above.

For the extensions of `kronos` and the implementation of `minim` and `synth-kro`, we have used the parser and DBM library of KRONOS, developed by S.Yovine, A.Olivero and C.Daws. For the implementation of `kronos-open`, we have used the parser of SMI, developed by M.Bozga [Boz97]. The implementation of `synth-kro` has been completed by K.Altisen.

In the following sections, we present `kronos`, `minim`, `synth-kro` and `kronos-open`.

11.1 The model checker `kronos`

The functionalities of `kronos` are shown in figure 11.2. The tool operates in one of following basic modes:

1. Full-TCTL model checking (top of figure): the system to be verified is given as a TA A (file `.tg`) and the property as a TCTL formula ϕ (file `.tctl`). The tool computes the set of states of A satisfying ϕ (i.e., the characteristic set of ϕ). The output is given in terms of a symbolic state. Since the input is given as a single TA, in case of a system consisting of more than one components, they should be statically composed before the analysis.

2. Safety-TCTL model checking (second from top in the figure): using forward reachability analysis based on simulation graphs, this mode can treat a sub-class of TCTL formulae, such as invariance ($\forall\Box\, p$) and bounded response ($\forall\Box\,(p_1 \Rightarrow \forall\Diamond_{\leq c}\, p_2)$). The input system is given as a single TA, as in the previous mode. The output is a yes/no answer possibly accompanied by a symbolic diagnostic trace.

3. Reachability-TCTL model checking (third from top in the figure): this mode is used to check reachability of discrete states [2] using abstractions and on-the-fly parallel composition of the input system, which is given as a collection of TA. The property is given as a *state formula*, that is, a boolean expression of atomic propositions. As previously, a yes/no answer is returned, plus a symbolic diagnostic trace whenever reachability holds.

4. TBA model checking (bottom of figure): the property here is given in terms of a TBA, the discrete states of which are labeled with boolean expressions of atomic propositions on the system. As in previous mode, parallel composition is computed on-the-fly. Diagnostics are reported in terms of symbolic paths ending in a cycle. The implemented technique is based on the double-DFS algorithm of [CVWY92] to find non-zeno, accepting cycles. As explained in section 7.2 this technique is sound but generally incomplete.

Function modes 3 and 4 have been implemented as part of this thesis. We now give examples of their usage, for the verification of the TGC system of section 3.1. The input `.tg` files (text) for the three automata are shown below:

```
/* Train.tg */
#states 3
#trans 3
#clocks 1 X

state: 0
prop: far
invar: true
trans:
true => approach; reset{X}; goto 1

state: 1
prop: near
invar: X<=5
trans:
X>2 => in; reset{}; goto 2

state: 2
prop: in
invar: X<=5
trans:
X<=5 => exit; reset{}; goto 0
```

[2] Any safety property can be reduced to (negation of) reachability. For this, it is sometimes necessary to use an auxiliary automaton to *monitor* the system and move to an error state whenever the property is violated.

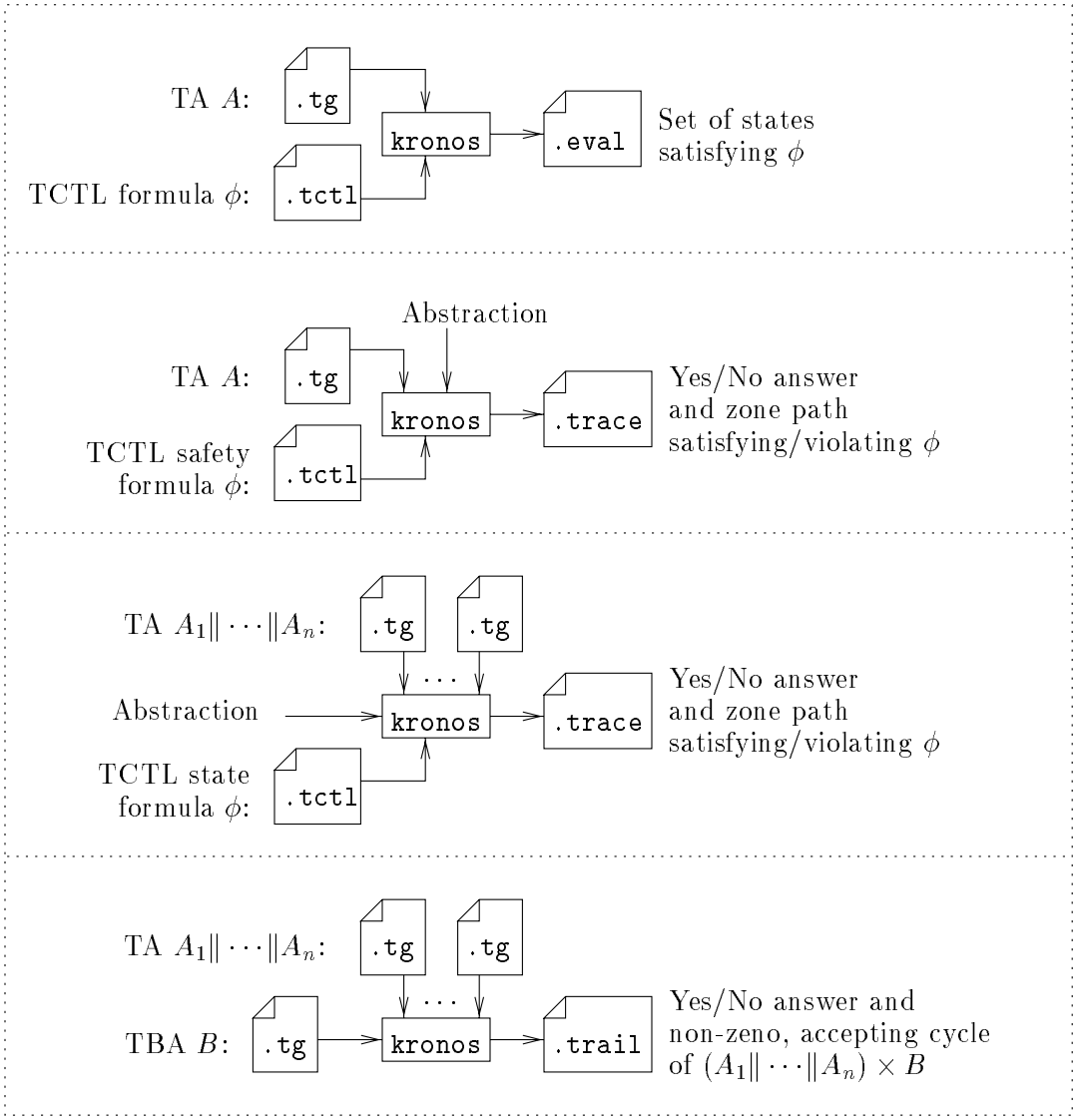


Figure 11.2: The function modes of the model checker `kronos`.

```
/* Gate.tg */
#states 4
#trans 4
#clocks 1 Y

state: 0
prop: up
invar: true
trans:
true => lower; reset{Y}; goto 1

state: 1
prop: coming_down
invar: Y<1
trans:
Y<1 => down; reset{}; goto 2

state: 2
prop: down
invar: true
trans:
true => raise; reset{Y}; goto 3

state: 3
prop: going_up
invar: Y<=2
trans:
Y>=1 => up; reset{}; goto 0

/* Controller.tg */
#states 4
#trans 4
#clocks 1 Z

state: 0
prop: c0
invar: true
trans:
true => approach; reset{Z}; goto 1

state: 1
prop: c1
invar: Z<=1
trans:
Z=1 => lower; reset{}; goto 2

state: 2
```

```
prop: c2
invar: true
trans:
true => exit; reset{Z}; goto 3

state: 3
prop: c3
invar: Z<=1
trans:
Z<=1 => raise; reset{}; goto 0
```

We would like to check the formula $\forall\Box$ (in $\Rightarrow$ down), stating that whenever the train is in the crossing, the gate is down. For this, it suffices to check that the state formula (in $\wedge \neg$down) is not reachable. The property can be verified by running `kronos` as follows:

```
kronos -R "in and not down" Train.tg Gate.tg Controller.tg
...
Building synchronization tables...
Using breadth-first search with max symbolic-states set size: 1000
Reachability failed.
Full state space explored: 11 states. Max depth reached: 9
```

The tool reports that reachability has failed, meaning that invariance holds.

As a second example, consider the bounded-response property "whenever the gate is down, it comes up at most 6 time units later". We can verify this property using a TBA encoding the negation of the property, similar to TBA B_2 of figure 4.1. The `.tg` file for this automaton (`buchi_bounded.tg`) is shown below:

```
/* Timed Buchi automaton testing bounded response */
#states 4
#trans 4
#clocks 1 W

state: 0
invar: true
trans:
true => go; reset{ W }; goto 1

state: 1
prop: down
invar: true
trans:
true => wait; reset{}; goto 2

state: 2
prop: not up
invar: true
trans:
```

```
W>6 => error; reset{}; goto 3

state: 3
prop: accept
invar: true
trans:
true => accept; reset{}; goto 3
```

Then, we can run `kronos` as follows:

```
kronos train.tg gate.tg control.tg buchi_bounded.tg
...
Building synchronization tables...
On-the-fly model checking by Buchi acceptance.
Using depth-first search with max stack depth: 1000
Search for acceptance cycles successful.
Sample scenario dumped in file: buchi_bounded.trail
State space explored: 19 states. Max depth reached: 18
```

The tool reports a counter-example of length 17, meaning that the property does not hold. In fact, we can get a shorter counter-example by limiting the size of the DFS stack to 12:

```
kronos train.tg gate.tg control.tg buchi_bounded.tg -STACK 12
...
Building synchronization tables...
On-the-fly model checking by Buchi acceptance.
Using depth-first search with max stack depth: 12
Search for acceptance cycles successful.
Sample scenario dumped in file: buchi_bounded.trail
State space explored: 13 states. Max depth reached: 11
```

The `buchi_bounded.trail` file is shown below:

```
                  Path reaching cycle (length: 10)

0: < 0, 0, 0, 0,    X=Y and X=Z and X=W >
  ---  APPROACH --->
1: < 1, 0, 1, 0,    X=1 and Z=1 and 1<=W and X<=Y and Y=W >
   ---  LOWER   --->
2: < 1, 1, 2, 0,    1<=X and X<2 and X=Y+1 and X=Z and X<=W >
   ---  DOWN    --->
3: < 1, 2, 2, 0,    1<=X and X<2 and X=Y+1 and X=Z and X<=W >
   ---  GO      --->
4: < 1, 2, 2, 1,    2<X and X<=5 and X=Y+1 and X=Z and X<W+2 and W+1<=X >
   ---  WAIT    --->
5: < 1, 2, 2, 2,    2<X and X<=5 and X=Y+1 and X=Z and X<W+2 and W+1<=X >
   ---  IN      --->
6: < 2, 2, 2, 2,    X<=5 and 3<W and X=Y+1 and X=Z and W+1<=X >
```

```
   ---  EXIT    --->
7: < 0, 2, 3, 2,    Z<=1 and 4<W and X=Y+1 and X<=Z+5 and Z+4<X and W+1<=X >
   ---  RAISE   --->
8: < 0, 3, 0, 2,    Y<=2 and 6<W and X<=Z+5 and W+1<=X and Y<Z and Z<=Y+1
                         and Z+3<W >
   ---  ERROR   --->
9: < 0, 3, 0, 3,    1<Y and Y<=2 and 6<=W and X<=Z+5 and W+1<=X and Z<=Y+1
                        and Y+4<W >
   ---  UP      --->

                  Cycle (length: 0)

10: < 0, 0, 0, 3,   1<Y and 6<=W and X<=Z+5 and W+1<=X and Z<=Y+1 and Y+4<W >
   ---  ACCEPT  --->
               ... back to node 10 ...
```

11.2 The minimization module `minim`

Figure 11.3 illustrates the usage of the module `minim`. The tool takes as input a TA [3] and outputs its STa-quotient. Optionally, the initial partition can also be given as input. By default the initial partition consists in a set of zones $(q, \zeta_1), ..., (q, \zeta_m)$ for each discrete state q, where $\zeta_1, ..., \zeta_m$ is the *canonical decomposition* of the guards of edges leaving q: for each zone ζ_i and each guard, ζ_i is either included in the guard or has an empty intersection with it. For example, if $x \leq 1$ and $y > 2$ are the guards, we would obtain four zones, namely, $x \leq 1 \wedge y > 2$, $x \leq 1 \wedge y \leq 2$, $x < 1 \wedge y > 2$ and $x < 1 \wedge y \leq 2$.

The output comes in a various set of formats, including the (untimed) labeled graph format `.aut` of CADP and an extended TA format `.mtg` to represent τ-transitions. Typically, the `.aut` graphs produced by `minim` are given as input to `aldebaran`, in order to be re-minimized or compared with respect to various (untimed) bisimulations or simulations. They can also be visualized (when they are reasonably small) using the module `bcg_edit`, or model checked against μ-calculus formulae using the module `evaluator`.

For example, the STa-quotient of the TGC system (figure 5.3) can be minimized with respect to the observational equivalence, yielding the following graph (in `.aut` format):

```
des (0, 9, 8)

(0, APPROACH,2)
(1, APPROACH,7)
(1, UP,0)
(2, LOWER,3)
(3, DOWN,4)
(4, IN,5)
(5, EXIT,6)
(6, RAISE,1)
(7, UP,2)
```

[3] Actually, `minim` accepts as input the parsed `.tg` file. The parsing is done by `kronos`.

According to lemma 5.2, this graph is the Tao-quotient of the TGC system. We can use `bcg_edit` to visualize and transform the graph, which can then be output in postscript format, shown in figure 5.5.

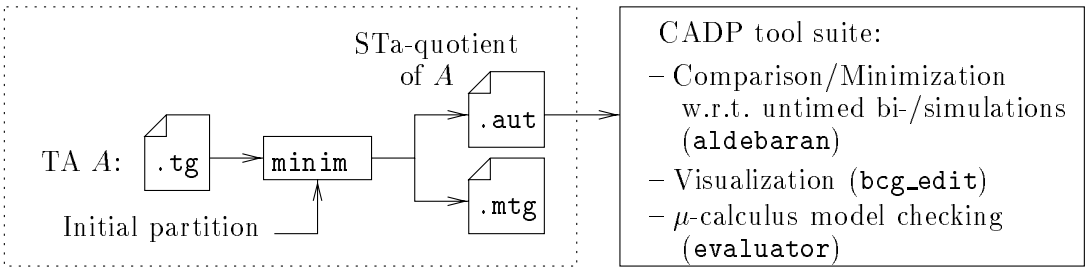


Figure 11.3: The minimization module `minim`.

11.3 The controller-synthesis module `synth-kro`

This module is presented in figure 11.4. It takes as input:

- a TA in `.tg` format (the special label `U_` is used to mark uncontrollable edges);
- a TCTL formula in `.tctl` format, of the form $\forall\Box\,\phi$ (invariance) or $\exists\Diamond\,\phi$ (reachability), where ϕ is a boolean expression on atomic propositions.

The tool produces two output files:

- a `.eval` file containing the set of winning states;
- (if the above set is non-empty) a `.tg` file specifying the restriction of the input TA to the set of winning states, as described in section 9.2.

The TGC example of figure 9.2 is specified by the TA shown below:

```
/* Timed graph generated for the parallel composition of:
        automaton 0: train.tg
        automaton 1: gate.tg
        automaton 2: control.tg
*/
#states 12
#trans  17
```

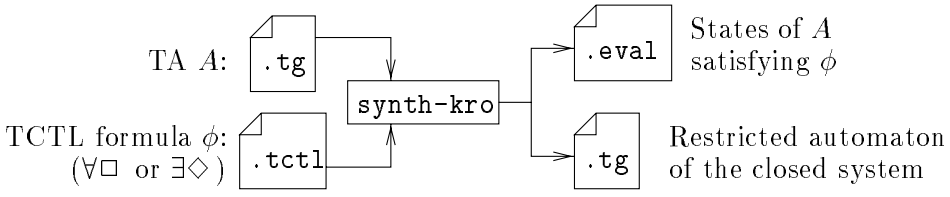


Figure 11.4: The controller-synthesis module `synth-kro`.

```
#clocks  3
        X /* train */
        Y /* gate */
        Z /* control */


state: 0        /* vector state: < 0, 0, 0 > */
prop: FAR UP C0
invar: TRUE
trans:
1<=X => U__ APPROACH ; RESET{ X Z }; goto 1

state: 1        /* vector state: < 1, 0, 1 > */
prop: NEAR UP C1
invar: X<=5 and Z<=3
trans:
2<X and X<=5 => U__ IN ; reset{}; goto 2
Z<=3 => LOWER ; RESET{ Y }; goto 3

state: 2        /* vector state: < 2, 0, 1 > */
prop: IN UP C1
invar: X<=5 and Z<=3
trans:
Z<=3 => LOWER ; RESET{ Y }; goto 4

state: 3        /* vector state: < 1, 1, 2 > */
prop: NEAR COMING_DOWN C2
invar: X<=5 and Y<1
trans:
2<X and X<=5 => U__ IN ; reset{}; goto 4
Y<1 => U__ DOWN ; reset{}; goto 5

state: 4        /* vector state: < 2, 1, 2 > */
prop: IN COMING_DOWN C2
invar: X<=5 and Y<1
trans:
Y<1 => U__ DOWN ; reset{}; goto 6
X<=5 => U__ EXIT ; RESET{ X Z }; goto 7

state: 5        /* vector state: < 1, 2, 2 > */
prop: NEAR DOWN C2
invar: X<=5
trans:
2<X and X<=5 => U__ IN ; reset{}; goto 6

state: 6        /* vector state: < 2, 2, 2 > */
prop: IN DOWN C2
```

```
invar: X<=5
trans:
X<=5 => U__ EXIT ; RESET{ X Z }; goto 8

state: 7        /* vector state: < 0, 1, 3 > */
prop: FAR COMING_DOWN C3
invar: Y<1 and Z<=1
trans:
Y<1 => U__ DOWN ; reset{}; goto 8

state: 8        /* vector state: < 0, 2, 3 > */
prop: FAR DOWN C3
invar: Z<=1
trans:
Z<=1 => RAISE ; RESET{ Y }; goto 9

state: 9        /* vector state: < 0, 3, 0 > */
prop: FAR GOING_UP C0
invar: Y<=2
trans:
1<=Y and Y<=2 => U__ UP ; reset{}; goto 0
1<=X => U__ APPROACH ; RESET{ X Z }; goto 10

state: 10       /* vector state: < 1, 3, 1 > */
prop: NEAR GOING_UP C1
invar: X<=5 and Y<=2 and Z<=3
trans:
2<X and X<=5 => U__ IN ; reset{}; goto 11
1<=Y and Y<=2 => U__ UP ; reset{}; goto 1

state: 11       /* vector state: < 2, 3, 1 > */
prop: IN GOING_UP C1
invar: X<=5 and Y<=2 and Z<=3
trans:
1<=Y and Y<=2 => U__ UP ; reset{}; goto 2
```

Running `synth-kro` on the above TA and the TCTL formula $\forall\Box$ (in $\Rightarrow$ down), yields the following restricted TA:

```
/* closed system for AB(IN impl DOWN) */

#states 12
#trans 17
#clocks 3 X Y Z

state: 0
prop: C0 UP FAR
```

```
invar: TRUE
trans:
1<=X => APPROACH U__; RESET{ X Z } ; goto 1

state: 1
prop: UP C1 NEAR
invar: X<=1 and Z<=3
trans:
2<X and X<=5 => U__ IN; RESET{} ; goto 2
X<=1 and Z<=3 => LOWER; RESET{ Y } ; goto 3

state: 2
prop: UP C1 IN
invar: false
trans:
false => LOWER; RESET{ Y } ; goto 4

state: 3
prop: NEAR C2 COMING_DOWN
invar: X<=5 and Y=1 or Y<=1 and X<=Y+1
trans:
2<X and X<=5 => U__ IN; RESET{} ; goto 4
Y<=1 => U__ DOWN; RESET{} ; goto 5

state: 4
prop: IN C2 COMING_DOWN
invar: false
trans:
Y<1 => U__ DOWN; RESET{} ; goto 6
X<=5 => U__ EXIT; RESET{ X Z } ; goto 7

state: 5
prop: NEAR C2 DOWN
invar: X<=5
trans:
2<X and X<=5 => U__ IN; RESET{} ; goto 6

state: 6
prop: IN C2 DOWN
invar: X<=5
trans:
X<=5 => U__ EXIT; RESET{ X Z } ; goto 8

state: 7
prop: FAR COMING_DOWN C3
invar: false
trans:
```

```
Y<1 => U__ DOWN; RESET{} ; goto 8

state: 8
prop: FAR DOWN C3
invar: X=0 and Z<=1
trans:
X=0 and Z<=1 => RAISE; RESET{ Y } ; goto 9

state: 9
prop: C0 FAR GOING_UP
invar: Y<=2 and X<=Y or 1<=Y and Y<=2 and Y<=X+1 or 1<=Y and Y<=2 and X<=Y+8
trans:
1<=Y and Y<=2 => U__ UP; RESET{} ; goto 0
1<=X => APPROACH U__; RESET{ X Z } ; goto 10

state: 10
prop: C1 NEAR GOING_UP
invar: Y<=2 and X+1<=Y and Z<=Y+1 or Z<=3 and X+2<=Z and Y+1<=Z and Z<=Y+2
trans:
2<X and X<=5 => U__ IN; RESET{} ; goto 11
1<=Y and Y<=2 => U__ UP; RESET{} ; goto 1

state: 11
prop: C1 IN GOING_UP
invar: false
trans:
1<=Y and Y<=2 => U__ UP; RESET{} ; goto 2
```

Notice that only controllable edges have restricted guards, for instance, the edge `LOWER` of state 1 finds its guard restricted from $z \leq 3$ to $x \leq 1 \wedge z \leq 3$. Also notice that the discrete states which are eliminated by the synthesis algorithm have invariant `false`.

11.4 The connection of KRONOS to OPEN-CAESAR

In this section we describe the code-generator `kronos-open` which interfaces KRONOS to the verification platform OPEN-CAESAR. The steps of the verification process using `kronos-open` are illustrated in figure 11.5 and explained in the following paragraphs.

Input

The input is given as a Lotos-like expression specifying the system components and their channel connections (file `.exp`). For example, the `.exp` file for the BANG&OLUFSEN case study (section 12.3) is shown below:

```
Lotos-Behavior

  (
```

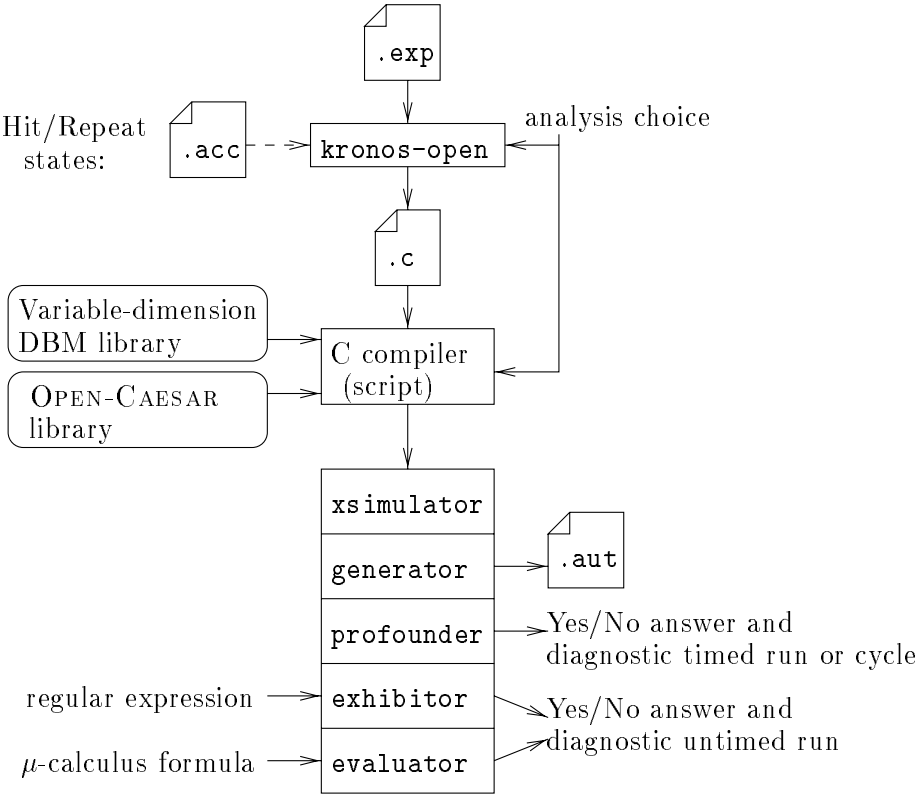


Figure 11.5: The usage of the module `kronos-open`.

```
Bus
|[ zero, one ]|
(
 (
  ( Sender_A |[ a_check ]| Detector_A )
  |[ a_frame, a_new_pn, a_reset ]|
    FrameGen_A
 )
 |||
 (
  ( Sender_B |[ b_check ]| Detector_B )
  |[ b_frame, b_new_pn, b_reset ]|
    FrameGen_B
 )
)
)
|||
Observer
```

In the above expression, names such as `Bus`, `Sender_A`, etc, denote the TA of the system. For each of these names there is a `.aut` file containing the description of the TA. Names such as `zero, one`, etc, are *channels*, used for communication between different components. Communication takes place through synchronization of two or more components. To specify which

components synchronize on which channels, the notation `|[ ... ]|` is used, for instance, the `Bus` synchronizes with the rest of the system on channels `|[ zero, one ]|`.

Apart from clocks, the TA can have boolean, bounded-integer, and enumerative-type variables. There is an associated `.types` file containing type definitions, such as:

```
enum msg {m0, m1, m2, m3, m4, m5, m6, m7, m8, m9, m10}
```

The variables and the structure of each TA are specified in the corresponding `.aut` file. Part of this file for the TA `Sender_A` is shown below:

```
/* variable declaration */

a_c     : clock
a_pf    : bool
a_pn    : bool
a_s1    : bool
a_s2    : bool
a_msg   : msg
...
atomic  : bool
\index{states!atomic}

/* the TA */

des(0,34,19)

/* start */
(0, [~atomic] send a_go a_c:=0, 1)

/* idle */
(1, [(~atomic) and (a_c=781)] send a_start_frame atomic:=true a_c:=0, 2)

/* atomic:ex_start */
(2, [(a_c<=0) and (~b_start)] send b_silent atomic:=false, 3)
(2, [(a_c<=0) and  b_start] send b_sends atomic:=false, 4)

/* ex_silence1 */
(3, [(~atomic) and (a_c=2343)] receive a_zero a_c:=0, 1)
(3, [(~atomic) and (a_c=2343)] receive a_one a_c:=0, 6)

/* other_started */
(4, [(~atomic) and  b_start and (a_c=3124)] send b_sends a_c:=0, 4)
(4, [(~atomic) and (~b_start) and (a_c=3124)] send b_silent a_c:=0, 3)

/* atomic:goto_idle (removed) */

/* ex_silence2 */
(6, [(~atomic) and (a_c=781)] receive a_zero a_c:=0, 1)
```

```
(6, [(~atomic) and (a_c=781)] receive a_one a_c:=0, 7)

/* transmit */
(7, [(~atomic) and (a_c=781)] send a_frame a_err:=e0 a_diff:=false
                                  a_pf:=true atomic:=true a_c:=0, 8)

...

/* until_silence */
(16, [(~atomic) and (a_c=781)] receive a_zero a_c:=0, 16)
(16, [(~atomic) and (a_c=781)] receive a_one a_c:=0, 17)

/* hold */
(17, [(~atomic) and (a_c=28116)] send a_hold a_res:=r0 a_c:=0, 1)

/* jam */
(18, [(~atomic) and (a_c=25000)] send a_jam a_pn:=true
                                  a_start:=false a_res:=r0 a_c:=0, 7)

/* invariants */

[1, a_c<=781]
[2, ~atomic]
[3, a_c<=2343]
[4, a_c<=3124]
...
[17, a_c<=28116]
[18, a_c<=25000]
```

Code generation

`kronos-open` creates a `.c` file which implements the on-the-fly generation of the simulation graph of the input model. The core of the `.c` file consists of the data structures to represent symbolic states (zones) and edges, and the implementation of post(). More precisely:

- A record-like data structure is used to represent zones. The structure has a separate field for each discrete variable, plus an additional field for the convex polyhedron. The size of each discrete-variable field is the number of bits necessary to encode the type of the variable. The field for the polyhedron is a pointer to a variable-dimension DBM. The implementation for the latter is parameterized by the maximal number of clocks (depending on the input model) and is contained in a separate library.
- There is a C function to produce the initial zone.
- post() is implemented by a set of C-functions:
  - Two functions for each edge e: the first one takes as input a zone and returns its intersection with the guard of e; the second function performs the assignments on

the discrete variables and applies the clock-reset and time-passage operators to the DBM.

– An iterator function which takes as input a zone and generates its successors one by one. The out-going edges of the zone are computed on-the-fly, based on information stored about the possible channel synchronizations of the input model.

It is worth noticing the main benefit of the compiler approach, compared to the interpreter one: guards, asssignments and clock resets are transformed directly to C code, which results in more efficient execution, than having generic functions for the above operations. In particular, when these operations are trivial (e.g., `true` guard, no assignment) they can be completely skipped.

On the other hand, the approach has the potential disadvantage of explosion of the size of the `.c` file generated, in case there is a very large number of transitions in the input model. Luckily, this is rarely the case, since these are high-level transitions: at the TA level, not at the graph level.

Final output

After the `.c` file has been generated, it is compiled and linked to the OPEN-CAESAR and DBM libraries using a script. As a result, we obtain an executable program performing a certain type of analysis. An option given to the script tells it which type of analysis is to be performed, that is, which type of executable is to be generated. Currently, the following types of executables are available:

- `xsimulator` performs user-guided simulation in a window-based environment.
- `generator` builds the simulation graph of the system in untimed labeled graph `.aut` format.
- `profounder` performs reachability analysis or TBA emptiness. It takes as supplementary input a `.acc` file specifying the discrete states to be reached (*hit* states) or the repeating states (*repeat* states). In case of reachability, `profounder` can generate timed diagnostics using the method described in section 8.1.
- `exhibitor` searches for a finite untimed trail matching an input regular expression.
- `evaluator` performs μ-calculus model checking.

Relation to the literature

Apart from KRONOS, perhaps the most successful tool for dense-time verification is UPPAAL [LPY97]. To our knowledge, `synth-kro` is currently the only tool for dense-time controller synthesis.

Part IV

Case studies and Conclusions

Chapter 12

Case studies

We have used our models and tools to treat a number of case studies. In this chapter, we present five of them. The criteria for choosing them have been related to their illustrative power, but also their proper interest. More precisely:

- Fischer's mutual-exclusion protocol is a well-known benchmark, demonstrating the performance capabilities of the tools, but also serving as a good illustration of the analysis techniques introduced in the previous.
- Two case studies have to do with modeling and verification of real-world protocols, namely, a collision-detection protocol of BANG&OLUFSEN and a bandwidth-reservation protocol of CNET. The results we obtain have revealed inconsistencies in both protocols.
- The last two case studies of real-time scheduling and multimedia-document controllers show how formal techniques can be used not only to check correctness of such systems (i.e., schedulability), but also compute solutions (i.e., schedulers). In the real-time scheduling example, computing a scheduler can be done using model checking. In the case of multimedia documents, we use controller synthesis. This case study is also a real-world one.

The general conclusions from our experiments can be summarized as follows:

- TA are a useful and powerful model, able to express systems of quite general type, including a number of real-world systems, within a satisfactory level of abstraction.
- The logical formalisms are also general enough to express a variety of problems, from classical verification, to the automatic generation of schedulers. Diagnostics are also essential from this point of view.
- The tools, although prototypes and developed in a non-professional manner, are mature enough to handle models of realistic size (e.g., systems with more than 10 clocks, symbolic graphs of size 10^6 to 10^7 nodes), in a reasonable amount of time (a few hours).

Experimentation is not only a way to evaluate models, techniques and tools, but also an important source of feedback for improving them. In fact, many of our analysis techniques and tool features have been motivated by the case studies.

One of the lessons learned from our experience is that TA is a low-level model: realistic systems are usually more complex, both in their discrete structure and their interaction mechanisms. In particular, they have richer data types and control-flow commands resembling those

of a programming language; they also communicate using higher-level primitives like broadcast, one-to-one handshake, interrupts, or waiting. These features often constitute the largest part of the system (in terms of lines of specification, or any similar measure), while the timing constraints, although essential to the functioning of the system, are the smallest part.

Consequently, in order to enhance the usability of tools, the basic TA model has to be extended with higher-level features. The `kronos-open` module follows this direction, accepting as input TA extended with discrete variables and a variety of communication mechanisms such as shared variables and one-to-many message passing. The theoretical results of the previous chapters are not affected by these extensions which can be seen as "syntactic sugar", directly encoded in the discrete structure (discrete states and edges) of the basic formal model. The details can be found in appendix A.

12.1 Fischer's Mutual-Exclusion Protocol

This is a well-known example in the literature of real-time verification, introduced in [AL91]. The protocol is a good benchmark for testing the capacity of tools, since it is parameterized by the number of processes involved and can be easily expanded to generate models of very large size. It is also a good example for illustrating many of the analysis techniques presented in the previous chapters, namely, minimization, CTL model checking on the quotient graph, usage of untimed bisimulations, on-the-fly reachability on abstract graphs and timed diagnostics. In the rest of this section, we first present the protocol, then describe the techniques used and finally show performance results obtained using KRONOS.

Description. The system is composed by a set of n processes (identical up to renaming), plus a shared variable last ranging from 0 to n. Process i behaves as follows: after remaining idle for some time, it checks whether the common resource is free (test last $= 0$) and if so, sets last to i. Then it waits for some time and, making sure that last is still equal to i, enters the critical section. If last is not equal to i (meaning that some other process has requested access meanwhile) then process i has to retry later.

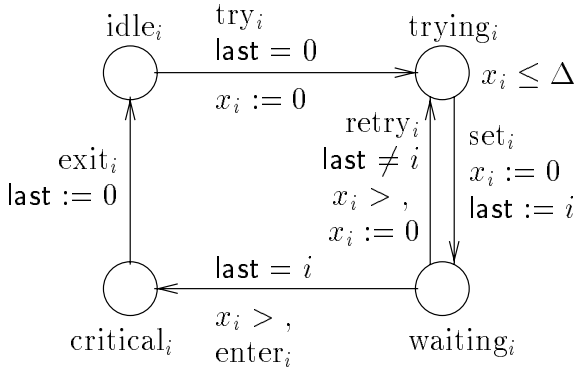


Figure 12.1: Fischer's Mutual-Exclusion Protocol.

The TA for process i is shown in figure 12.1. The timing assumptions of the protocol are two: first, each process sets last to its id no later than Δ time units after it has tested last $= 0$; second, each process waits at least , time units before checking that last equals its own id, and

entering the critical section. To model these constraints, we use one clock per process. Notice that the clock is active only at states "trying" and "waiting".

, and Δ are parameters of the system, instantiated to constants when doing verification. The correctness of the protocol depends on the values, in fact, it can be shown that mutual exclusion holds iff , $\geq \Delta$.

Verification. There is a number of properties that our model must satisfy.

First, we must test for absence of deadlocks and timelocks (there is no problem of critical races since no constraint of the form $x_i = 0$ appears in the system). Observe that each structural loop in the TA modeling process i (there are two such loops, one from "idle$_i$" back to itself and one between "trying$_i$" and "waiting$_i$") satisfies the conditions of strong non-zenoness (30). By lemma 3.2, the system is strongly non-zeno and it suffices to guarantee deadlock-freedom to deduce timelock-freedom. In fact, we prove a stronger property, namely, absence of *live-locks*, that is, states where at least one process is blocked while the rest can possibly continue execution. This can be expressed by the following CTL formula:

$$\forall\Box\,\exists\Diamond\,(\mathsf{last} = 0 \wedge \bigwedge_{1\leq i\leq n} \mathrm{idle}_i)$$

The formula states that it is always possible for the system to reach its initial state. It is easy to see that in the initial state all processes can execute, which, together with the above property, implies absence of timelocks.

To verify the formula, we first apply minimization to generate the STa-quotient of the system (section 6.1) and then model-checking to verify the formula on the quotient (section 6.2.2). The first step is performed by the module `minim`, and the second step by the tool `evaluator`. The STa-quotient (for $n = 2$, $\Delta = 1$, , $= 2$) is shown in figure 12.2. It is easy to see that the system is livelock-free.

The second property to verify is mutual exclusion, that is, the fact that at most one process is in its critical section at any time. That is, we are checking the reachability of

$$\bigvee_{1\leq i\neq j\leq n} \mathrm{critical}_i \wedge \mathrm{critical}_j$$

For this property we apply the on-the-fly reachability analysis on simulation graphs, as proposed in section 7.1.

Using `kronos`, we have experimented with values of n up to 9 processes, and various values of , and Δ. Whenever , $\geq \Delta$, mutual exclusion was proven to hold, independently of the abstraction used. That is, the above set of states has been found unreachable, even in approximative analysis using the convex-hull abstraction.

Whenever , $< \Delta$, mutual exclusion is violated. To get precise diagnostics, we use the executable `profounder` (generated by `kronos-open`, see section 11.4). For instance, when $n = 2$, , $= 7, \Delta = 11$, we get the counter-example run:

```
<0, 0, last=0> - 0 - <0, 0, last=0> -- x2:=0 "try2" -->
<0, 1, last=0, x2:0> - 0 - <0, 1, last=0, x2:0> -- x1:=0 "try1" -->
<1, 1, last=0, x2:0, x1:0> - 0 - <1, 1, last=0, x2:0, x1:0> -- x2:=0 "set2" -->
<1, 2, last=2, x2:0, x1:0> - 8 - <1, 2, last=2, x2:8, x1:8> -- "enter2" -->
<1, 2, last=2, x1:8> - 0 - <1, 2, last=2, x1:8> -- x1:=0 "set1" -->
<2, 3, last=1, x1:8> - 8 - <2, 3, last=1, x1:16> -- "enter1" -->
<3, 3, last=1>
```

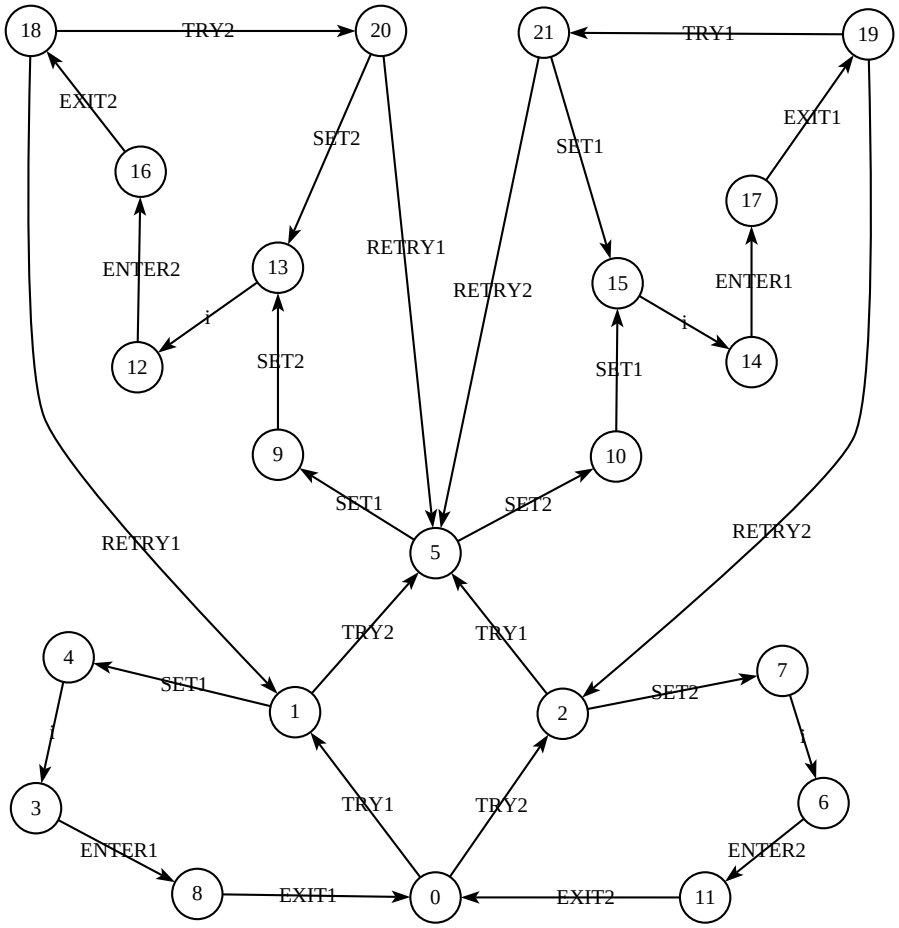


Figure 12.2: The STa-quotient of Fischer's protocol ($n = 2, \Delta = 1, , \ = 2$).

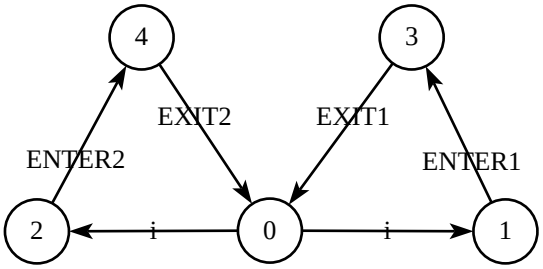


Figure 12.3: The observational-quotient of the graph of figure 12.2 (after hiding).

n	Size of TA		Size of quotient		time (sec)
	states	edges	classes	edges	
4	752	2240	629	2146	3
5	3552	12640	3501	15705	51
6	16320	67200	22085	122804	1000
7	73620	321000	too large input TA		

Table 12.1: Minimization results for Fischer's protocol.

The output is shown verbatim as produced by `profounder`: each line shows a state taking a time transition followed by a discrete one. Each state contains the values of the control locations of each process (from 0 for "idle", 1 for "trying", 2 for "waiting" and 3 for "critical"), the value of last and the clock valuation. Notice that valuations vary in dimension, according to which clocks are active at each state, for instance, in the initial and final state no clocks are active, in the second

Mutual exclusion can be verified also by combining TaBs with untimed bisimulations, as proposed in section 6.2.5. Having obtained the STa-quotient of the system (figure 12.2), we "hide" (i.e., replace by τ) all labels but "enter" and "exit" and then re-minimize the graph with respect to the observational bisimulation. This step is performed by the tool `aldebaran`. The resulting graph is shown in figure 12.3. Mutual exclusion can be deduced by merely observing this graph.

Results and performance. The minimization results are shown in table 12.1. `minim` has been able to generate the STa-quotient for up to 6 processes. For 7 processes the product automaton was too big to be parsed by `kronos`. Notice that the number of nodes and edges in the quotient increases with a greater exponential rate than in the product TA. The last column of the table shows the CPU time for minimization (excluding syntactic composition and parsing, which took about 20 seconds for 6 processes). The memory consumption was about 100 Mbytes for 6 processes.

The results of reachability are shown in figure 12.4. The diagrams display the size of simulation graphs [1], in linear (a) and logarithmic scale (b). Notice that α_{act} and $\alpha_{act} \circ \alpha_{inc}$ yield the same state space. Some conclusions that can be made from these results are the

[1]Only the number of nodes is shown, since the edges are not stored, apart from the edges of the current DFS path.

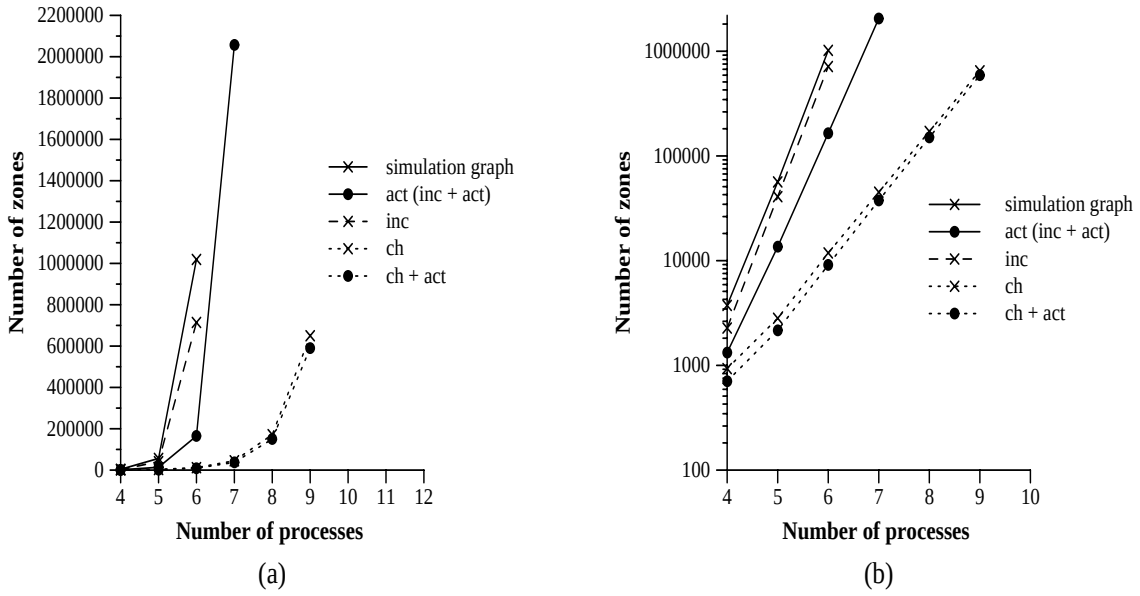


Figure 12.4: Number of zones in simulation graphs for Fischer's protocol.

following:

1. Combination of abstractions is definitely useful in absolute terms. For instance, using activity reduces the size of the graph from 10^6 to less than $2 \cdot 10^5$ nodes, for 6 processes.

2. The convex-hull abstraction radically reduces the state space, permitting to handle up to 9 processes.

3. The size of the graphs grows exponentially, even when convex hull is used.

A remark needs to be made concerning clock activity. Notice that in each process i, the clock is active in two out of four discrete states, namely, states "trying$_i$" and "waiting$_i$". Also, a system of n processes has n clocks in total. During our analysis, we have measured the distribution of active clocks in the reachable state space, that is, the percentage of reachable zones for any number of active clocks from 0 to n. The results show that only in about 50% of the state space all clocks were active. About 40% of the reachable nodes had $n \Leftrightarrow 1$ active clocks, and the remaining 10% had less than $n \Leftrightarrow 1$ active clocks. These results are not impressive, compared to the ones for the STARI circuit, presented in section 12.2.

Regarding performance of reachability, the largest graph contained more than 10^6 zones. It has been generated on a Sparc-station 20 with 224 Mbytes of main memory in 2 hours of CPU time, consuming 180 Mbytes.

Comparison with other results in the literature. Fischer's protocol has been a popular benchmark in the literature of timed verification. [ACD+92] apply two TA minimization algorithms to the protocol. The results of their experiments show that `minim` performs much better: they manage to generate the quotient graph for only up to 4 processes in about 500 seconds (compared to only 3 seconds for `minim`). We have also used version 2.17 of UPPAAL (dating March 1998) to check reachability of error states. The results show that KRONOS performs much better: for 5 processes, UPPAAL takes 30 minutes to explore the whole state

space, with all the reduction options enabled. KRONOS needs only 37 seconds for exactly the same input model. Recently, [STA98] use a novel technique to represent partitions as trees of atomic constraints, and apply it for model checking. They present results on Fischer's protocol which are better than KRONOS (e.g., 100 seconds for 6 processes, instead of 1000 seconds with KRONOS).

Due to its symmetric structure, Fischer's protocol can admit even more efficient or general solutions. In particular, [KLL+97] have verified a simple acyclic version of the protocol for up to 50 processes using a *formula-quotienting technique*. [AJ98] have verified an abstraction of the protocol parameterized by the number n of processes, for any n. These techniques cannot be directly compared to ours, since they exploit the protocol's symmetry and use simplified input models.

12.2 The STARI circuit

STARI [Gre97] is a circuit for synchronous communication between a transmitter and a receiver through an asynchronous FIFO buffer. The buffer makes the system tolerant to time-varying skew between the transmitter and receiver clocks. An internal handshake protocol using acknowledgments prevents data loss or duplication inside the queue.

Description. The functioning of STARI is based on a rather simple idea. The buffer must be initialized to be (approximately) half-full. During each global period one value is inserted to the buffer by the transmitter and one value is removed by the receiver. Due the complementary nature of these actions no control is required to prevent queue underflow or overflow. Short-term fluctuations in the clock rates of the transmitter and the receiver are handled by inserting or removing more items from the queue.

Following the STARI model proposed in [TB97], we represent the boolean values *true* and *false* by *dual rail* encoding (figure 12.5). An auxiliary *empty* value is needed to distinguish between the case of two consecutive identical values and the case of one value maintained during more than one clock cycle. The transmitter is constrained to send sequences of *true* and *false* where each two occurrences of these values are separated by an occurrence of *empty*. STARI consists of a linear array of n identical stages, each capable of storing a data value X.

X	*true*	*false*	*empty*
X.t	1	0	0
X.f	0	1	0

Figure 12.5: Dual rail encoding.

The operation principle of a stage k can be summarized as follows: *it may copy its predecessor value ($X_k := X_{k-1}$) when its successor has already copied (and acknowledged) its current value ($X_k = X_{k+1}$)*. Using the dual rail encoding of data values, such a behavior can be achieved using two Muller C-elements that hold the $X.t$ and $X.f$ components, and one NOR gate for computing the acknowledgment (figure 12.6).

A Muller C-element works as follows: when the two inputs become identical, after some delay the output takes on their value, otherwise the output maintains its previous value. Consider, for example, a situation where stages k and $k+1$ hold the *empty* value, stage $k \Leftrightarrow 1$ the *true*

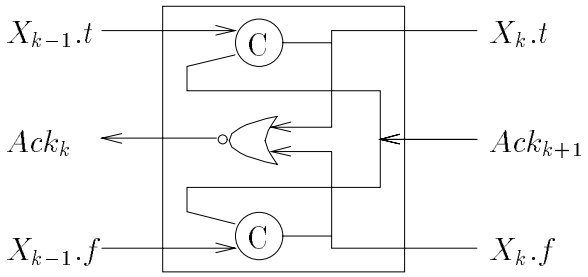


Figure 12.6: Stage k of STARI.

value and $Ack_{k+1} = 0$. When Ack_{k+1} becomes 1, the C-element for $X_k.f$ remains unchanged at 0 because its inputs are different (i.e. $Ack_{k+1} = 1$, $X_{k-1}.f = 0$). However, both the inputs of the C-element for $X_k.t$ are equal to 1 ($Ack_{k+1} = X_{k-1}.t = 1$), and after some delay, it will switch to 1. This way the *true* value has been copied from stage $k \Leftrightarrow 1$ to stage k.

Modeling. The correct functioning of STARI depends on the timing characteristics of the gates (the time it takes, say, for a C-element to switch) and its relation with the central clock period and the skew between the receiver and transmitter. We model the uncertainty concerning the delay associated with gates using the bi-bounded delay model, that is, we associate with every gate an interval $[l, u]$ indicating the lower and upper bounds for its switching delay (see [Lew89, BS94, MP95a, AMP98] for the exact definitions).

Following [MP95a] we can model any logical gate with a delay $[l, u]$ using a timed automaton with 4 states (0-stable, 0-excited, 1-stable and 1-excited) and one clock. In particular, each stage of STARI is modeled by the three timed automata of figures 12.7, 12.8 and 12.9.

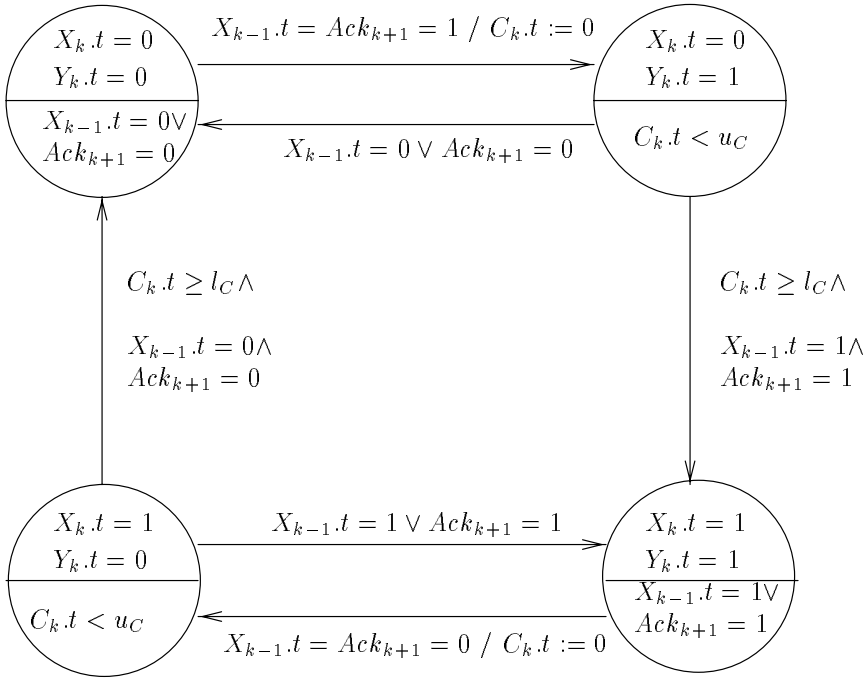


Figure 12.7: The timed automaton for the C-element $X_k.t$

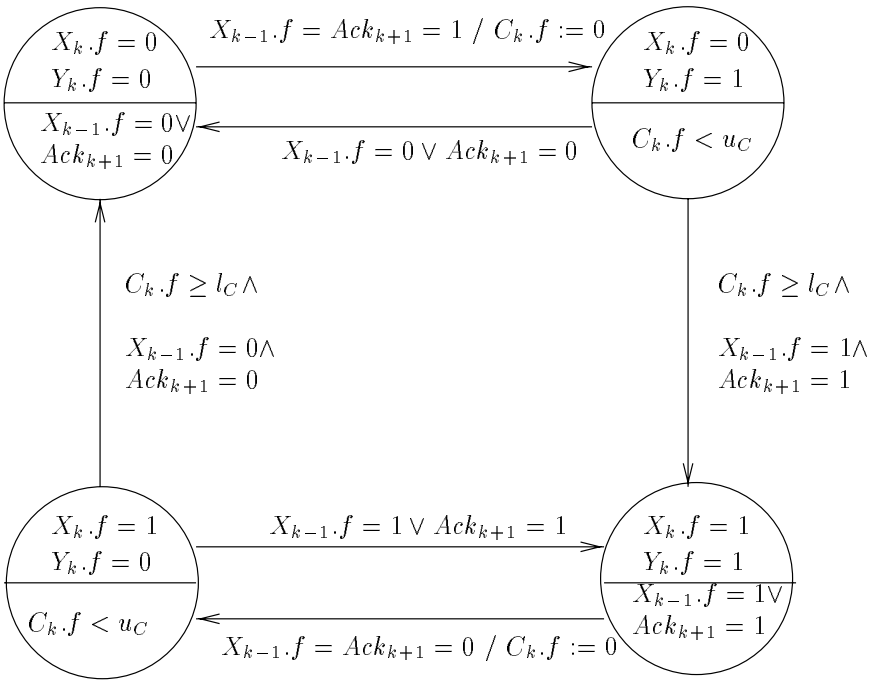


Figure 12.8: The timed automaton for the C-element $X_k.f$

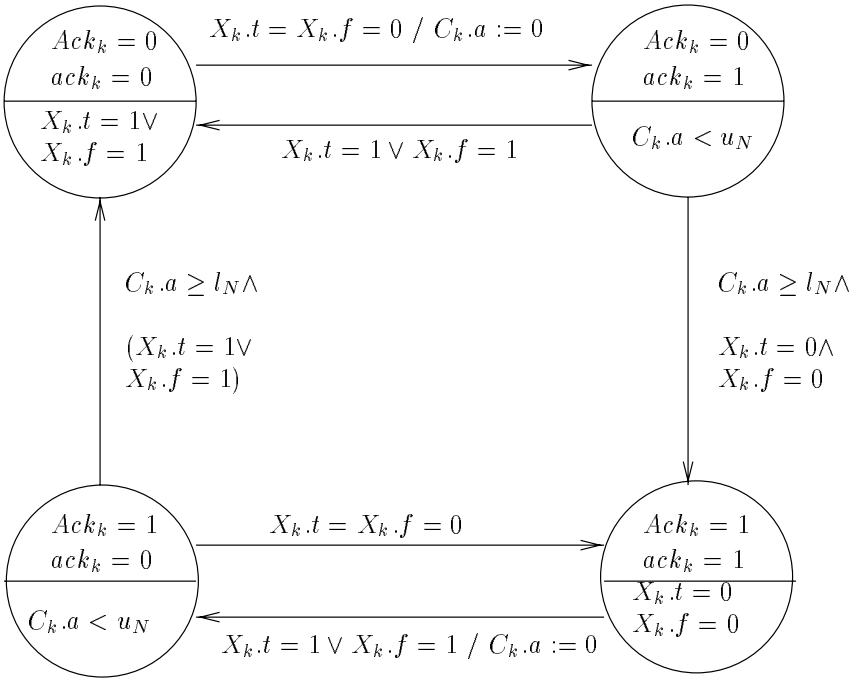
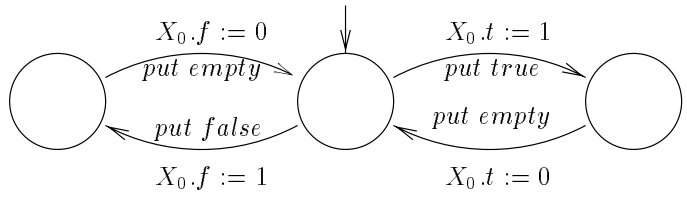


Figure 12.9: The timed automaton for the NOR gate Ack_k

Let us look at the automaton of figure 12.7 which models the $X.t$ component of the k^{th} stage. Its state is characterized by two boolean variables $X_k.t$, $Y_k.t$, the former stores the gate output and the latter stores the gate internal value, i.e. the value to which the gate "wants" to go after the delay. The stable states are those in which $X_k.t = Y_k.t$. The conditions for staying and leaving stable states are complementary and do not depend on clock values: for example, the automaton leaves state $(0,0)$ and goes to the unstable state $(0,1)$ exactly when both its inputs are 1. During this transition the clock variable $C_k.t$ is reset to zero. The automaton can stay at $(0,1)$ as long as $C_k.t < u_C$ and can change its output and stabilize in $(1,1)$ as soon as $C_k.t \geq l_C$, where $[l_C, u_C]$ is the delay interval associated with a C-element. The automaton for the $X.f$ component (figure 12.8) is exactly the same (with different inputs) and the automaton for the NOR gate (figure 12.9) is similarly characterized by two boolean variables Ack_k, ack_k, a clock variable $C_k.a$ and a delay bounded by $[l_N, u_N]$.

In addition to the automata for modeling the stages, we need two more automata for the transmitter and the receiver. The transmitter is modeled as a 3-state automaton (figure 12.10). At each clock cycle it puts a value at the input ports of the first stage ($X_0.t$ and $X_0.f$), according to the convention that every pair of data items is separated by an *empty* item. Moreover, the transmission can be done with some skew with respect to the global (perfect) period, bounded by the s_T constant, that is, the actual time of transmission can be anywhere in the interval $[p \Leftrightarrow s_T, p + s_T]$.



Clock invariant at all states: $x < p + s_T$
Clock guard at all edges: $x \geq p - s_T$

Assignments at all edges:
$x := 0$
`if` $(skew > 0)$ `then` $y := 0$
$skew := skew + 1$

Figure 12.10: The transmitter.

The *receiver* is a 1-state automaton (figure 12.11) which reads the current output value (i.e. $X_n.t$ and $X_n.f$) and acknowledges the reception by modifying Ack_{n+1} according to whether or not X_n is empty. As in the transmitter, a skew bounded by s_R is allowed.

In order to forbid the receiver and transmitter skews to accumulate during successive cycles, we use a discrete variable $skew \in \{\Leftrightarrow 1, 0, 1\}$. Whenever either the receiver or the transmitter lag too far behind one another (i.e., $skew \neq 0$), they re-synchronize by resetting both their clocks to zero. Notice, however, that their relative skew can vary non-deterministically from one cycle to another. This is more general than assuming a fixed skew given in advance, or a fixed skew chosen at start-up from a given interval.

The transitions of the automata are annotated by action names such as *put* and *get* whose role is explanatory – they have no effect on the functioning of the system.

A final remark needs to be made concerning the activity of clocks in the model. First notice that an n-stage STARI requires $3n + 1$ TA for the timed gates (i.e., C-elements or NOR gates),

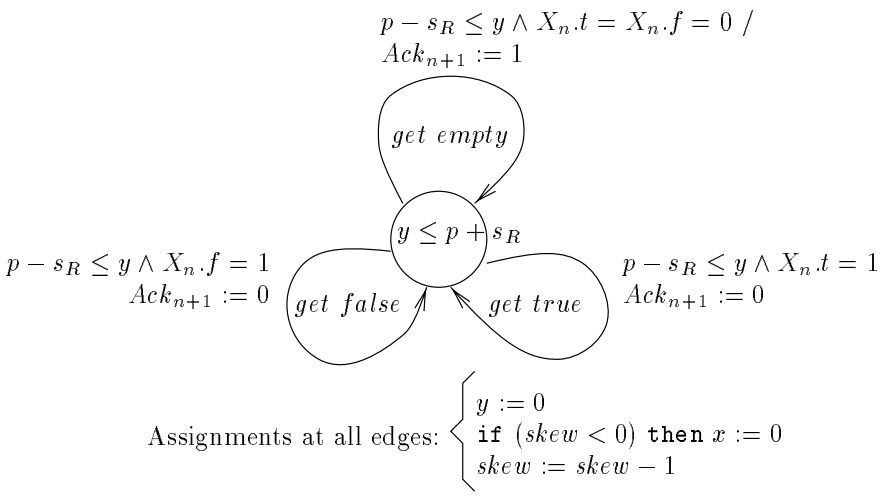


Figure 12.11: The receiver.

plus 2 TA for the transmitter and receiver. This totals $3(n+1)$ clocks and $6n+2$ boolean variables. Now, the basic building block used to model a timed gate is a four-state TA with one clock (figures 12.7, 12.8 and 12.9). Observe that the clock is active in only two of the four states, namely, in the unstable states. This information is crucial for the success of verification, as we shall see below.

Verification. The following two properties need to be proved to ensure the correct operation of the STARI circuit:

- Each data value output by the transmitter must be inserted in the buffer before the next one.
- A new value must be output by the buffer before each acknowledgment from the receiver.

These are finite-execution (i.e., safety) properties, meaning that they are preserved in the activity graph. Then, to verify the model we first generate its activity graph, and then minimize the latter with respect to the untimed observational bisimulation, after hiding all actions except *put* and *get*. As an example, a model of 3 stages generates the minimized graph shown in figure 12.12. It is easy to see that this graph corresponds to the ideal buffer implementation, satisfying the above two properties.

Results and performance. We have managed to verify STARI for up to 8 stages, with the following parameters: $l_C = l_N = 2$, $u_C = u_N = 4$, $p = 12$, $s_T = s_R = 1$. Figure 12.13 shows the time performance and the number of symbolic states generated for each number of stages.

The most interesting thing in this example concerns the activity of clocks. Indeed, a straightforward analysis without taking into account this information is doomed to state explosion, due to the fact that the number of clocks grows quite quickly with the number of stages (for 8 stages, there is a total of 27 clocks). We have measured the distribution of active clocks in the state space, that is, for each number of clocks k, the number of the number of DBMs of dimension k. The results confirm our expectations that only a small fraction of clocks are necessary at any time. For instance, in the case of 8 stages, at most 8 clocks were active, and this in less than 4% of the total number of DBMs generated (figure 12.14). In more than 85% of the symbolic

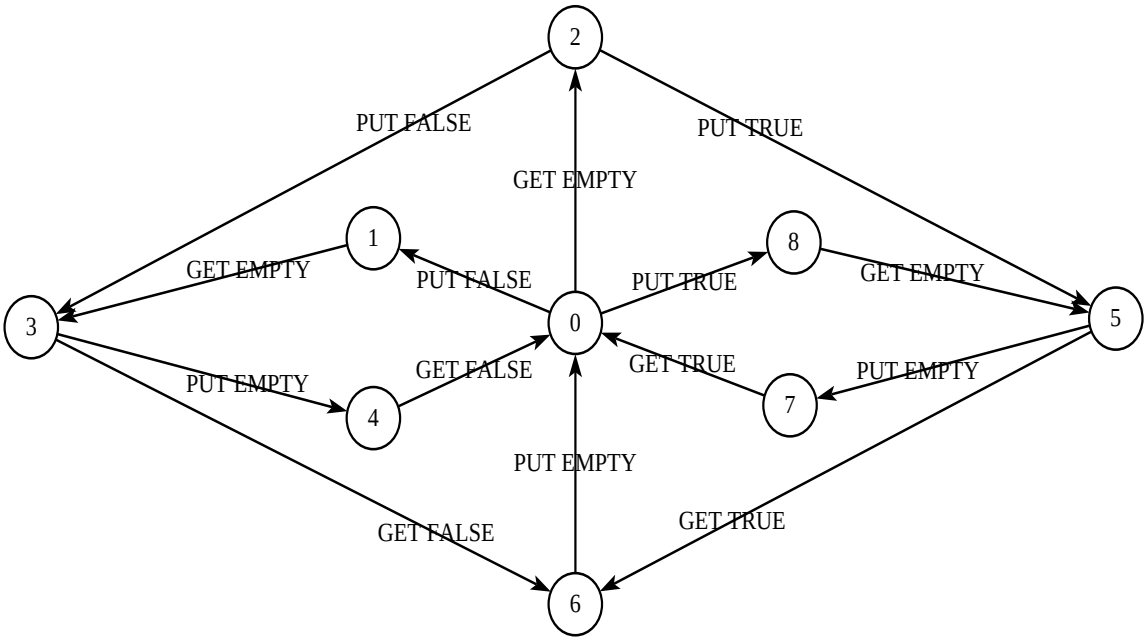


Figure 12.12: 3-stage Stari.

states, only 6 to 8 clocks were necessary. The shape of the distribution is the same for any other number of stages and also if we measure it with respect to the number of symbolic states instead of number of DBMs.

Relation to the literature. [Gre97] gave a deductive proof of STARI's correctness. This proof, although aided by a theorem prover, requires a lot of intervention and creativity from a user who understands why STARI works. [HBAB93] provided an automatic proof using a different methodology based on timed Petri nets, for which they developed an algorithm to calculate the time separation between events. [TB97] model the stages as TA and then prove, using techniques developed in [TAKB96], that every stage can be abstracted into a single 5-state automaton with one clock. Using this abstract model and the tool Timed-COSPANthey were able to verify an 8-stage STARI. Using the detailed model they could not verify more than 3 stages.

The same model as the one presented here, but interpreted in discrete time, is treated in [BMS99]. Based on the results of [AMP98] the discrete-time model is shown to be a conservative approximation of the dense-time one and a discrete-time version of KRONOS based on BDDs is used to verify the circuit for up to 18 stages. This large gap in performance with respect to the DBM-based results presented above is probably due to the huge size of the discrete state space (2^{24}): using DBMs, discrete variables are enumerated, whereas using BDDs all variables (including clocks) are handled uniformly, which results in more compact representation.

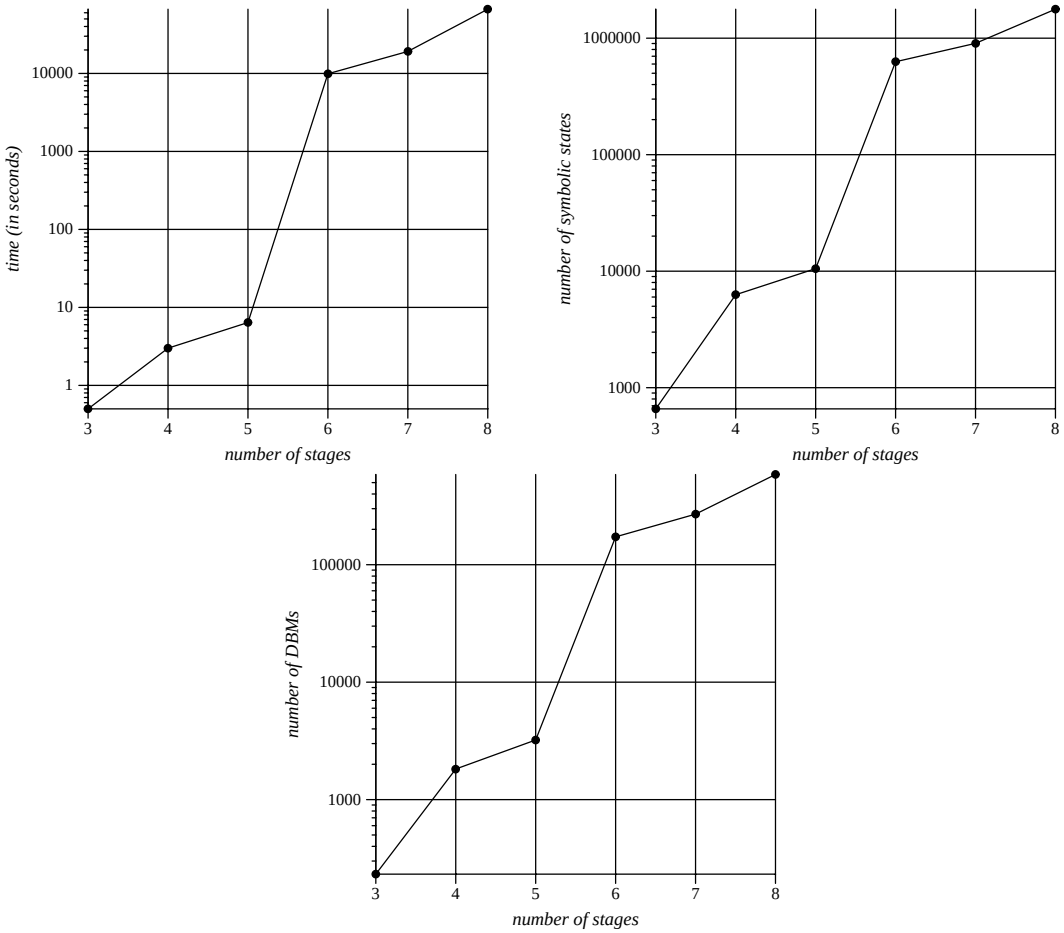


Figure 12.13: Experimental results for STARI.

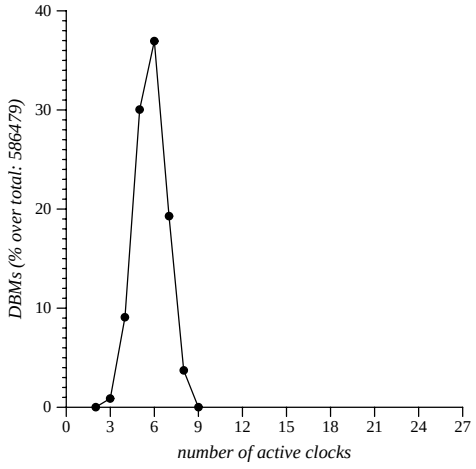


Figure 12.14: The distribution of active clocks in an 8-stage STARI.

12.3 BANG&OLUFSEN's Collision-Detection Protocol

This is an industrial case study, involving a protocol developed by BANG&OLUFSEN. It has been first analyzed using the verification-tool UPPAAL. The results of this analysis have been presented in [HSLL97].

We have treated the same case study with `kronos-open`, using the detailed description of the protocol contained in [HSLL97]. The TA models of `kronos-open` and UPPAAL are essentially the same, so that translating the specification of [HSLL97] to `kronos-open` format was almost straightforward.

The motivation for re-treating the same example has been twofold. First, being an industrial case study, it represents a good benchmark for any formal-analysis tool aspiring to be used in practice. Moreover, the results of our analysis have been unexpected, since we have found an error not reported in [HSLL97]. Second, from a methodological point of view, the protocol demonstrates the advantage of polyhedra-based techniques for the analysis of TA, as opposed to discretization techniques such as [AMP98]. (This point is discussed in more detail in what follows.)

We shall only give a brief description of the protocol here, rather insisting on the modeling issues involved. For a detailed description, the reader is referred to [HSLL97].

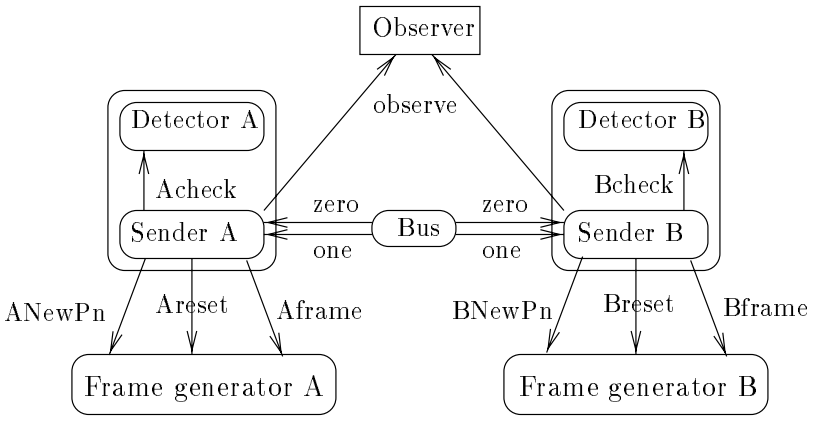


Figure 12.15: BANG&OLUFSEN's protocol: general architecture.

Brief description and modeling. The role of the protocol is to ensure collision detection in a distributed environment of components exchanging messages through a common multiple-access bus. The system modeled has two transmission components A and B (identical up to renaming) and the bus. Since we are interested only in the collision-detection protocol, the reception components are not modeled. A and B consist each of 3 sub-components, namely, the sender, the detector and the frame generator. The sender handles transmission of messages, which are grouped in *frames*. The latter are generated by the frame generator. The detector is responsible for collision detection.

The components along with their communication channels are shown in figure 12.15 (the observer is not part of the system itself, but is added to monitor the system for possible errors, as we explain below). Communication is done by *binary rendez-vous*, that is, a component

sending on a channel C synchronizes with a component receiving on C (in this case the rendez-vous serves only for synchronization, that is, no particular message is exchanged). For instance, the Bus component synchronizes on channel "zero" with either Senders A or B (only one at the time), which models the sender polling the value 0 from the bus.

The timing constraints of the system concern the frequency of senders' polling on the bus, the encoding of messages and the waiting delay required before retransmitting after a collision. For instance, a sender samples the value of the bus (1 for high voltage, 0 for low voltage) twice every 781 micro-seconds. Also, there are 5 different types of messages and the i-th message is encoded by the presence of a 1 on the bus, for $2 \cdot 1562 \cdot i$ micro-seconds. Finally, the *jamming signal*, after a collision, is a continuous 1 on the bus for 25 milli-seconds.

Each component is modeled as an automaton: senders A and B are modeled by timed automata whereas the rest of the automata are untimed. Figure 12.16 shows the TA for sender A. It is roughly divided in three parts, enclosed in dashed boxes: the upper part takes care of initialization when the sender attempts to transmit; the middle part models normal transmission; the lower part models the actions taken upon collision detection.

The figure is only intended to give an impression of the complexity of the case study and the modeling issues involved. More precisely, the UPPAAL model uses:

- 18 boolean variables, 4 bounded-integer variables and 4 enumerative-type variables, ranging in sets of up to twenty different values. For example, variable A_Pf (figure 12.16) is boolean while A_err ranges in $\{0, 1, 2\}$.
- Binary-rendez-vous communication.
- *Atomic* control states (drawn in dashed lines in figure 12.16). These states are transient, that is, if an automaton is in an atomic state, then no time passes. Moreover, when an automaton enters an atomic state, it has to exit before any other automaton can take a discrete step.

`kronos-open` supports discrete variables and rendez-vous communication directly. Atomic states are not supported by `kronos-open` directly. They can be modeled using an auxiliary boolean variable and a clock, as described in appendix A.2.

Perhaps the most interesting part of the protocol is its timing constraints. Duration constants vary from 40 micro-seconds to 0.5 seconds and have no common divisor (figure 12.16). This implies a very small time quantum of one micro-second, which results in very large constants in guards and invariants. Consequently, enumerative approaches based on discretization can lead to state explosion, since time units have to be counted one-by-one.

Verification. The protocol must ensure collision detection, that is, if a frame sent by a sender is destroyed by the other sender (collision), then both senders shall detect this. In order to state this requirement formally, we have to make clear what does it mean for a frame to be destroyed and when is a collision detected. For simplicity, we formalize the requirement just for sender A. The case of B is symmetric.

According to [HSLL97], collision happens when the boolean expression

$$\phi_{col} \stackrel{\text{def}}{=} \neg(A\_Pf \Leftrightarrow A\_S_1 \wedge A\_Pn \Leftrightarrow A\_S_2)$$

evaluates to false at the moment A_S_2 is assigned (transition from control state 11 to 12 in figure 12.16). A collision is detected when the result of the detector automaton (called by signal "Acheck !") is $A\_res = 1$ or $A\_res = 2$, whereupon the sender emits an "Areset !" signal.

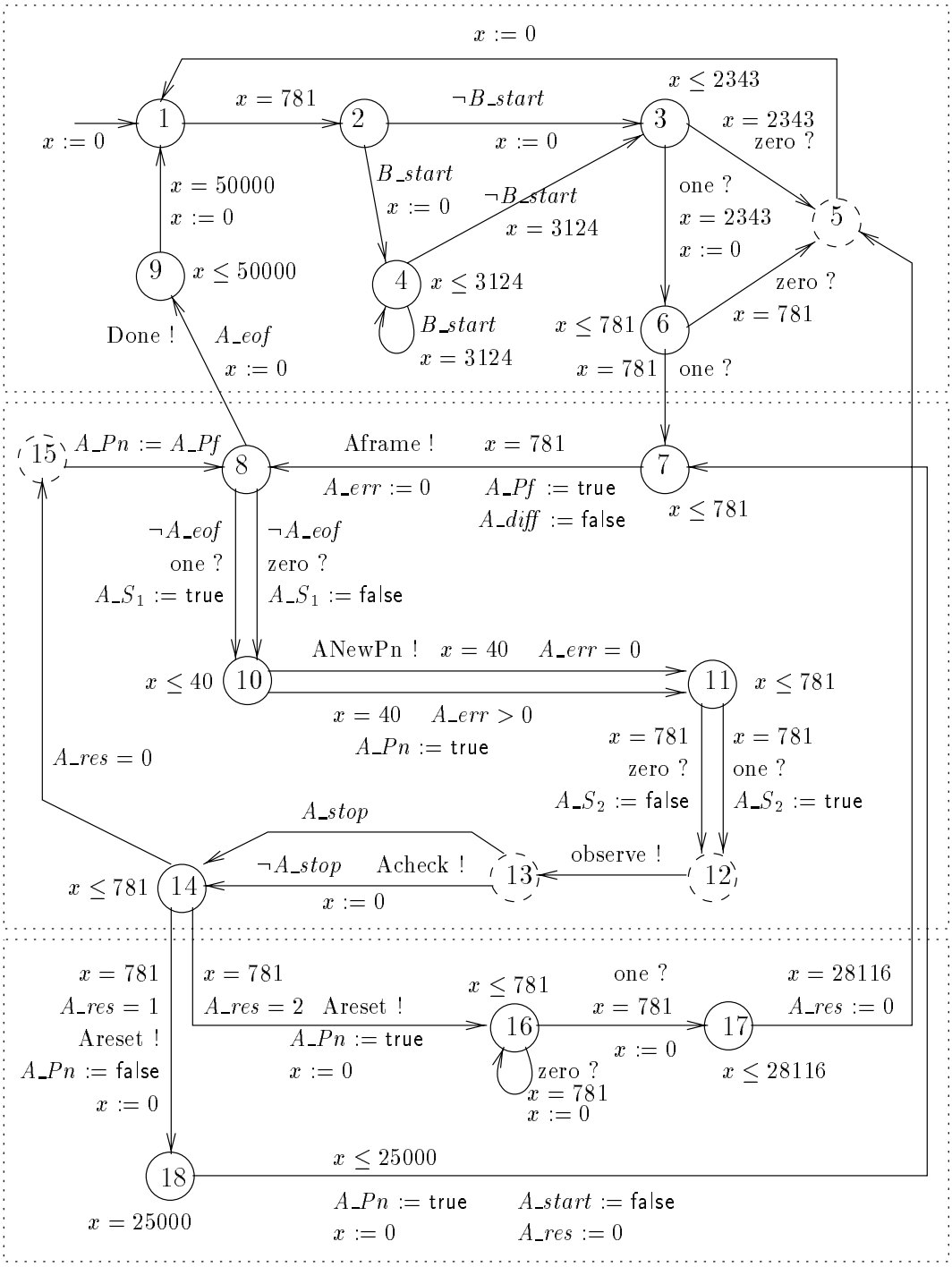


Figure 12.16: BANG&OLUFSEN's example: the TA for sender A.

Now we can model the requirement in terms or reachability of the "error" state of the observer automaton shown in figure 12.17. The observer starts at its left-most state and moves to its middle state when a collision happens. If the collision is detected before the the sender finishes transmitting (modeled by signal "Done !") then the observer returns to its initial state, otherwise it goes to the error state.

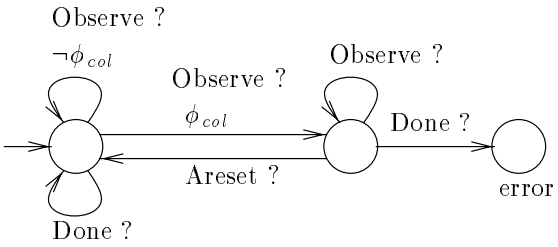


Figure 12.17: BANG&OLUFSEN's example: the observer automaton.

Results and performance. [HSLL97] present two versions of the protocol: the initial version contains an error (UPPAAL provides an abstract counter-example); then, the frame-generator automaton is slightly modified and the authors of [HSLL97] claim this version to be correct. However, we have found a counter-example in both versions, using the `profounder` executable generated by `kronos-open` (see section 11.4).

As in [HSLL97], in order to obtain a fast answer, we have used a simplified model of the protocol, where not the whole variety of messages could be generated. This does not affect the verification results in any way, since the model has the *data-independence* property [Wol86]. That is, the behaviors of the protocol do not depend, for a certain type of message, on the values of this messages. Therefore, although the message can have more than one values, we can assume that it always has the same value. In any case, the behaviors of the simplified model are a subset of the behaviors of the complete model, therefore, any counter-example produced in the former will also be valid in the latter.

The complete diagnostic run for the simplified "corrected" model is 1951 discrete/timed steps long. Here, we show only its head and its tail [2]:

```
<> - 0 - <>                                          -- b_c:=0 "b_go" -->
<b_c:0> - 40 - <b_c:40>                              -- a_c:=0 "a_go" -->
<a_c:0, b_c:40> - 741 - <a_c:741, b_c:781>           -- b_c:=0 "b_start_frame" -->
<a_c:741, b_c:781> - 40 - <a_c:781, b_c:821>         -- "a_silent" -->
<a_c:781, b_c:40> - 0 - <a_c:781, b_c:40>            -- a_c:=0 "a_start_frame" -->
<a_c:781, b_c:40> - 2303 - <a_c:3084, b_c:2343>  -- "b_silent" -->
<a_c:2303, b_c:2343> - 0 - <a_c:2303, b_c:2343>  -- b_c:=0 "b_one" -->
<a_c:2303, b_c:2343> - 40 - <a_c:2343, b_c:2383> -- a_c:=0 "a_one" -->
<a_c:2343, b_c:40> - 741 - <a_c:3084, b_c:781> -- b_c:=0 "b_one" -->
<a_c:741, b_c:781> - 40 - <a_c:781, b_c:821>   -- a_c:=0 "a_one" -->
<a_c:781, b_c:40> - 741 - <a_c:1522, b_c:781>  -- b_c:=0 "b_frame" -->
```

[2] There are too many discrete variables, thus, only the clock valuation is shown for each state. Clocks `a_c` and `b_c` correspond to senders A and B, respectively. The initial valuation is trivial since no clocks are initially active. In the second valuation, only `b_c` is active.

```
...
 <a_c:741, b_c:781> - 40 - <a_c:781, b_c:821>   -- "b_observe_ok" -->
 <a_c:781, b_c:40> - 0 - <a_c:781, b_c:40>      -- "b_stopped" -->
 <a_c:781, b_c:40> - 0 - <a_c:781, b_c:40>      -- a_c:=0 "a_zero" -->
 <a_c:781, b_c:40> - 741 - <a_c:1522, b_c:781>  -- "a_diff_pf_s1" -->
 <a_c:741, b_c:781> - 0 - <a_c:741, b_c:781>    -- "a_stopped" -->
 <a_c:741, b_c:781> - 0 - <a_c:741, b_c:781>    -- b_c:=0 "b_nocol" -->
 <a_c:741, b_c:781> - 40 - <a_c:781, b_c:821>   -- "b_pf0" -->
 <a_c:781, b_c:40> - 0 - <a_c:781, b_c:40>      -- "b_zero" -->
 <a_c:781, b_c:40> - 0 - <a_c:781, b_c:40>      -- "b_new_pn" -->
 <a_c:781, b_c:40> - 0 - <a_c:781, b_c:40>      -- a_c:=0 "a_nocol" -->
 <a_c:781, b_c:40> - 40 - <a_c:821, b_c:80>     -- "a_pf0" -->
 <a_c:40, b_c:80> - 0 - <a_c:40, b_c:80>        -- "a_zero" -->
 <a_c:40, b_c:80> - 0 - <a_c:40, b_c:80>        -- "a_new_pn" -->
```

Intuitively, the error seems to be due to the following reasons:

1. The two senders start transmitting with a difference of *exactly* 40 μ-seconds. Due to this fact and the way the sampling of the bus is performed, collision remains undetected until the last message of the frame is sent.

2. In the last message of the frame (a message signaling *end-of-frame*) the collision detection procedure is disarmed. This can be seen in the tail of the diagnostic run above: instead of the action `a_check` calling the collision detection procedure, we see the action `a_stopped`, which means that boolean variable *A_stop* is set. Therefore, collision is not detected by A. The situation is the same for sender B.

Regarding performance, the counter-example for the simplified model is produced by `profounder` in 25 seconds on a Sparc 20. To further test the capacity of the tool, we have built the entire simulation graph of the complete model, using `generator` (section 11.4). The graph has 9195634 nodes and 9509928 edges and was generated on a PC at 166 MHz with 512 Mbytes of main memory, consuming 15 minutes of CPU time and 300 Mbytes. The results obtained in [HSLL97] using UPPAAL were similar, for instance, 30 minutes of CPU and 90 Mbytes were necessary for the generation of the simulation graph of the complete model.

12.4 CNET's Fast-Reservation Protocol

This is an industrial case study concerning a protocol for bandwidth reservation in ATM (asynchronous transfer mode) networks. The protocol is called *Fast-Reservation Protocol with Delayed Transmission* (FRP-DT) and it has been developed by CNET at Lannion in the beginning of the '90s [BT92, Tra93]. Since then, the protocol has become a standard [ABT97].

Our modeling is based on the protocol specification in SDL (system description language) and discussions with the designers during a visit of one week in CNET. Initially, the protocol had been modeled in the basic TA language and verified with `kronos` (results published in [TY98]). Recently, we have treated a more general and detailed model of the protocol, using `kronos-open`. This is the model we present here.

The results of the original analysis showed some inconsistency in the protocol, claimed to be already known by the engineers of CNET. The recent analysis confirmed the previous results and provided more precise diagnostics.

Description. In ATM networks, clients negotiate with the network manager on the so-called *quality-of-service* parameters, in particular, on bandwidth allocation. If the latter has to be fixed upon connection establishment, then often the peak transmission rate has to be assumed for the client, resulting in poor resource management. FRP-DT offers an alternative, permitting dynamic modification of the transmission rate within a small amount of time (essentially the end-to-end round-trip delay).

The general scheme of an end-to-end connection managed by FRP-DT is shown in figure 12.18. It consists of two clients, two FRP-DT *units* (FRPU) and a number of *switching elements* (SE). Control messages flow in both directions, while data flow in one direction. Here we assume that data flow from left to right, thus, the left client is the sender and the right client the receiver.

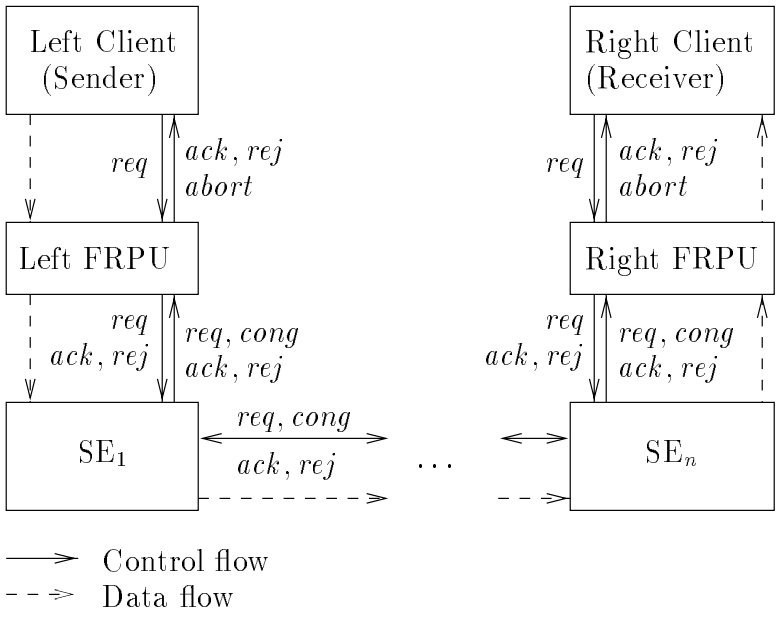


Figure 12.18: General scheme of an end-to-end connection managed by FRP-DT.

In our study we consider mainly control flow, for simplicity and tractability. Concerning data flow, we only consider the data transmission rate (rate, for short) which can be either low, medium, or high. Control messages are parameterized by the rate and are of five types, namely, *req* (to request a rate modification), *ack*, *rej* (to acknowledge or reject a request), *cong* (to signal *congestion*, that is, inability to accept request) and *abort* (to abort a request). Figure 12.18 shows which types of messages are exchanged between which components.

The connection starts in a *stable* state, when all components agree on the same rate r. When a client wishes to change the rate to, say r', it sends a $req(r')$ message to its FRPU [3]. The latter forwards the message to the network, which propagates it through the SEs along the connection. If the request is for an increase of the current rate and all the SEs have the capacity to respond, the request arrives unaltered to the peer FRPU, which informs its client and sends back an acknowledgment $ack(r')$. The acknowledgment passes through all SEs confirming the change of rate from r to r'. When the client who has originated the request receives $ack(r')$ it also changes from r to r' and the connection is back to a stable state.

[3]It may seem strange that the receiver can also request an increase of the rate. This is possible in some applications where the destination can inform the source that it is now able to accept more data.

Two things can go wrong. Either a message can be lost. Or a SE cannot accept the request due to lack of capacity. This is possible only if there is a request for a rate r' strictly higher than r. In this case, the SE forwards a $cong(r')$ request. When the peer FRPU receives $cong(r')$ it sends back $rej(r')$. The latter passes through all SEs canceling the rate change.

Lost messages are handled by the FRPU using timeouts, where-upon the request is retransmitted. If no reply is received after a number of attempts, an $abort(r')$ message is returned to the client.

One last remark needs to be made. The right direction has a higher priority than the left one, in the following sense: if a right-request is received while a left-request is pending, the latter is abandoned; also, if a left-request is received while a right-request is pending, the former is ignored. This asymmetry has nothing to do with politics, but mirrors the fact that the client that receives the data should have the command over their transmission rate.

Modeling. If n is the number of SEs in the connection, then $2n + 3$ automata are required to model the system: an untimed automaton for each SE, two untimed automata for the left and right clients, two TA the left and right FRPU, and one TA for each physical link between two adjacent SEs. The sender automaton is shown in figure 12.19. The receiver automaton is almost the same and it is not shown. Figure 12.20 shows the TA for the left FRPU. The TA for the right FRPU or for a SE are almost the same and are not shown. Figure 12.21 shows the TA modeling the physical link.

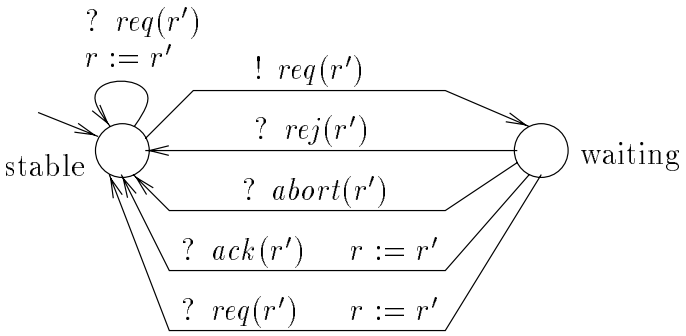


Figure 12.19: The sender client of the FRP-DT example.

In the figures, variables msg, r, r' and r'' are local in the automata they appear. msg ranges in $\{req, ack, rej, cong\}$ and r, r' and r'' range in {low, medium, high}. The value of r is the current rate of the connection, while r' stores the value of the rate requested by a pending request. In the case of the left FRPU (figure 12.20), the variable r'' is used to store the response of the adjacent SE. The rate carried by such a response can be different than the requested rate r', for instance, due to previous requests arriving from the peer, or due to message loss. In case the request is acknowledged or the right side requests a new rate, r'' is assigned to r.

In general, each automaton is connected to four rendez-vous channels, for each side (left or right) and each direction (input or output). The sender and receiver automata are connected to only two channels each. '!' and '?' denote transmission and reception, respectively. For example, "ToSE ! $req(r)$" means "send message $req(r)$ to channel ToSE", while "ToSE ? $req(r')$" means "receive from channel ToSE a message of type $req()$ and store its rate value to variable r'". Since we are talking about rendez-vous channels, the above two actions synchronize, resulting in the value of variable r being affected to r'.

Notice that in the sender automaton, channel names are omitted since there is no ambiguity. Also, in the TA of figure 12.21, '? *msg*' stands for reception of any message from any input channel, modeling loss.

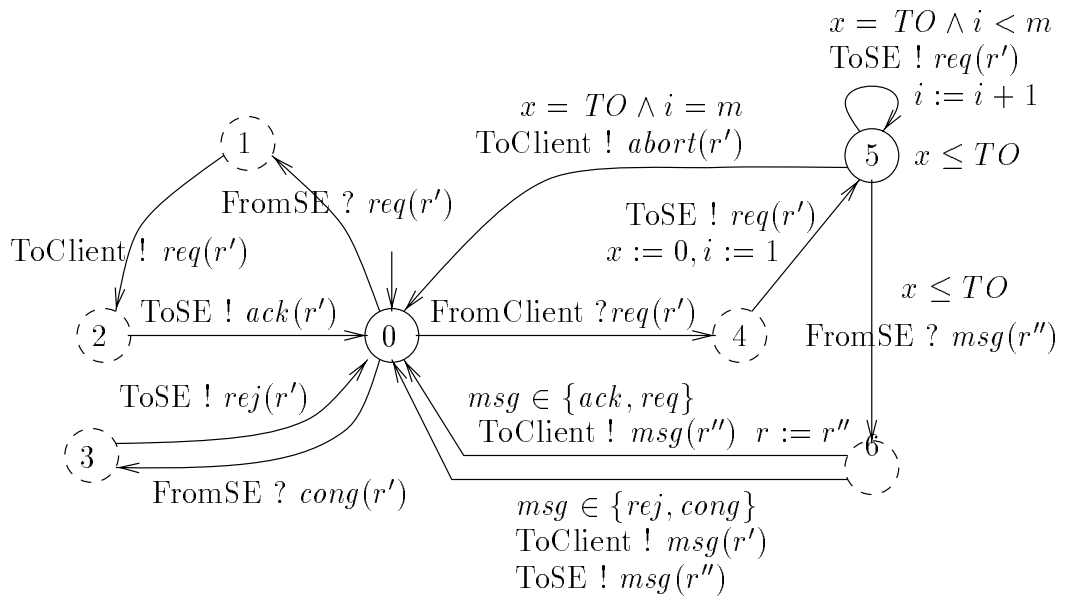


Figure 12.20: The left FRPU of the FRP-DT example.

The parameter m is the number of attempts to re-transmit before aborting. TO and Δ are also parameters modeling the timeout and link-propagation delays, respectively.

As in the case study of the previous section, we use atomic states to model operations that are assumed to consume no time. In figure 12.20 atomic states are drawn in dashed lines. For instance, when the FRPU receives a $req(r')$ message from the left-most SE (edge from state 0 to 1), it propagates the request to the client (edge from 1 to 2) and acknowledges to the SE (edge from 2 to 0) in zero time.

We also assume that the SE have no buffers, that is, they forward any message they receive right away. If a message is already present in the link, the new message is lost (notice, however, that a message might be lost even if the link is free).

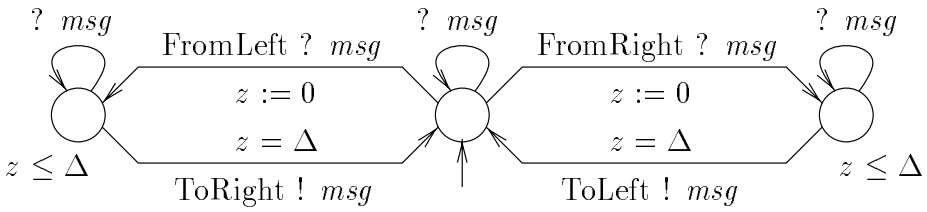


Figure 12.21: A physical link in the FRP-DT example.

Verification. Since the modeling is quite complex, the first property to be checked is deadlock-freedom. (Timelock-freedom is ensured by the fact that structural loops consume a strictly positive amount of time, either the round-trip or the timeout delay, thus, the system is strongly non-zeno.)

The main property to be checked is *consistency*, informally stated as follows:

> If the connection is in a stable state, then the sender and the receiver agree on the current transmission rate.

Consistency is a safety property, thus, it can reduced to checking reachability or error states. An error state is such that all automata are in their initial control states, and the variable r of the sender differs from the one of the receiver.

Results and performance. We have used the `profounder` option of `kronos-open` to perform reachability of error states. In `profounder` format, target states are specified by a boolean expression in a `.acc` file (see section 11.4). In our case this file is as follows:

```
hit : (rate_s<>rate_rcv) /\ (Sender:0=0) /\ (Receiver:0=0)
      /\ (FRPUleft:0=0) /\ (FRPUright:0=0) /\ (SE1:0=0) /\ (SE2:0=0) ;
```

Running `profounder`, we find that the FRP-DT is inconsistent. On a model of 2 SEs, with $TO = 10$, $\Delta = 2$ and $m = 1$, the tool generates a counter-example of 22 steps in 5 seconds.

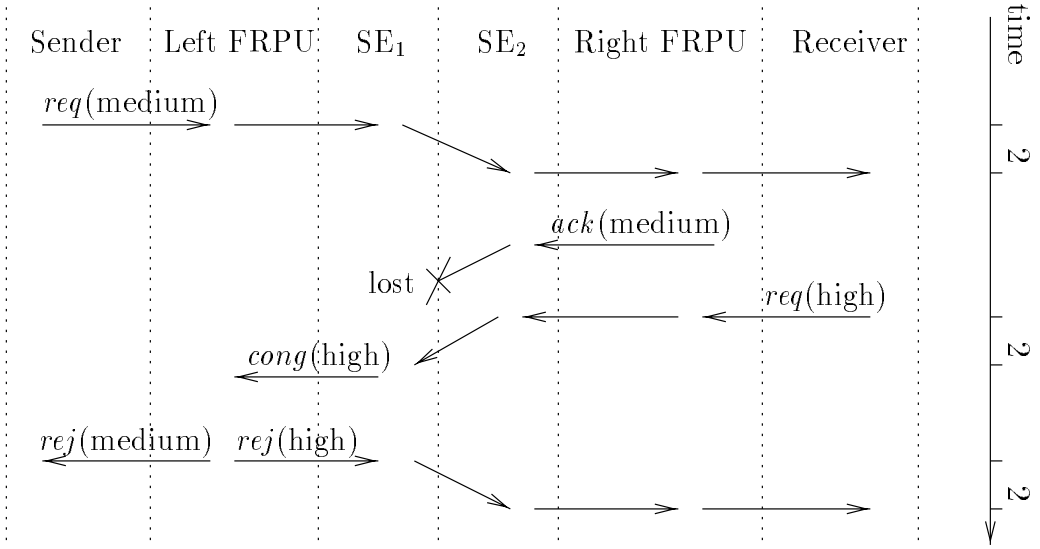


Figure 12.22: The message chart of the counter-example trail proving FRP-DT inconsistency.

The diagnostic run is shown in the form of a *message-chart* in figure 12.22. The vertical direction (top to bottom) represents the passage of time. The components of the system are listed horizontally. The arrows represent the propagation of messages among the components, with the tail of the arrow representing the moment of transmission and the head of the arrow the moment of reception (thus, a horizontal arrow means that the transmission takes no time).

The error lies in the fact that the acknowledgment to a request might be lost somewhere in the middle of the connection, so that one part of the SEs consider the request acknowledged, whereas the rest consider it still pending. Then, a new request which fails might reset the last part of the SEs to the old rate value, whereas the SEs which have acknowledged the request remain on the new rate value.

In the particular counter-example, the initial rate is low. Then, a request for medium rate is propagated from left to right and is acknowledged, however, the *ack*() message is lost in the

transmission between SE_2 and SE_1. At this point, SE_2 has updated its current rate to medium, while SE_1 has current rate low and pending rate medium. Then, a new request originates from the right for high rate, but is not accepted by SE_1, which propagates a congestion message and goes back to its initial state. The left FRPU receives the congestion message and informs both its client and the peer FRPU of the rejection of their requests. At this point, SE_2 and the right FRPU and client have their rates set to medium, whereas SE_1 and the left FRPU and client have their rates set to low.

The engineers of CNETwere already aware of the above behavior. According to them, it does not represent a problem, since first, message losses in ATM networks occur rarely, and second, a monitoring procedure periodically "resets" the connection to a consistent rate.

To test further the capacity of `kronos-open`, we have experimented generating the entire simulation graph for various values of the parameters. The largest case was for $n = 3$ (3 SEs and 2 links), $TO = 10$, $\Delta = 2$ and number of attempts $m = 1$: the complete graph comported 15488084 nodes and 23961733 edges and was generated in about 1 hour of CPU time.

It is worth noting that there is an impressive state-explosion in this example due to the number of different rate values. If we assume that only 2 such values are possible, say "low" and "high", then for the previous parameters, the size of the graph drops to 113566 nodes and 167707 edges, a reduction of more than 130 times ! It is worth noting, however, that this might be a too strong abstraction, since the protocol does not satisfy the criteria of data independence of [Wol86], since the behavior of the SEs actually depends on the rate values: the SEs do not refuse a request for a lower rate, although they can refuse a request for a higher rate. It is not clear, either, that three different rate values suffice (although our intuition says so). However, since adding more possible values only increases the number of possible behaviors, it is certain that any incorrect behaviors contained in the model with three values will also be contained in richer models.

12.5 Real-time scheduling

This case study concerns the problem of computing statically a scheduler for a set of periodic tasks with "hard" real-time constraints. In short, checking schedulability of a set of periodic tasks is reduced to checking non-emptiness of a TBA (section 7.2). Computing the scheduler means finding a non-zeno accepting cycle.

The case study is interesting for two reasons. First, it shows how model-checking can be used instead of synthesis, in a simple setting where the environment is deterministic. Second, it illustrates the use of `kronos` for TBA model checking.

Description. We consider a finite number of tasks $T_1, ..., T_n$. Task T_i has *period* π_i, that is, it becomes ready for execution every π_i time units. T_i also has a *deadline* ξ_i, that is, it must complete its execution at most ξ_i after the moment it becomes ready. The execution delay of T_i is η_i time units. We assume that $\eta_i \leq \xi_i \leq \pi_i$, for each $i = 1, ..., n$. The behavior of T_i is shown in figure 12.23.

All tasks are executed in a single processor which serves one task at a time and does not allow *preemption*, that is, the execution of a task cannot be interrupted and resumed later. We also assume that all tasks become ready simultaneously the first time. Of the above simplifying assumptions, the only significant one is preemption: indeed, modeling preemption a-priori

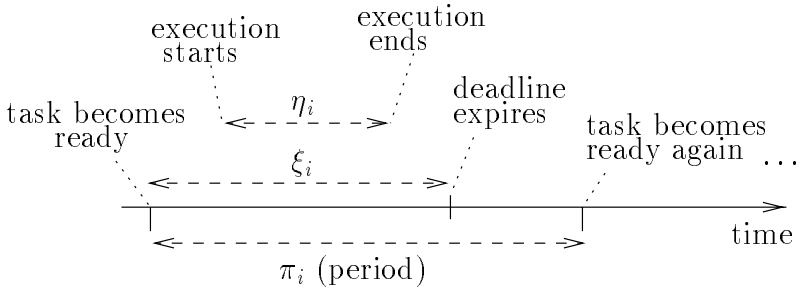


Figure 12.23: Behavior of a periodic task with deadline.

requires a model where clocks can be "frozen" and resumed, which is not in the class of TA [4]. The other assumptions are not essential: their omission is discussed at the end of the section.

The problem is to check whether the tasks are *schedulable*, that is, whether there exists an (infinite) order of execution where all tasks are executed infinitely often and all timing constraints are respected. Such an order of execution is called a *scheduler*. If the tasks are schedulable, we are interested in computing a scheduler.

Modeling. The system is modeled by n TA, one for each task, plus a global boolean variable free modeling the fact that the processor is free or occupied. The TA for T_i is shown in figure 12.24. The automaton has three control states labeled "wait$_i$" (the task has become ready and waits to be served), "use$_i$" (the task is using the processor) and "sleep$_i$" (the task has finished execution and hasn't become ready yet). Clock y_i counts the computation delay. Clock x_i counts the period and also makes sure the deadline is not violated.

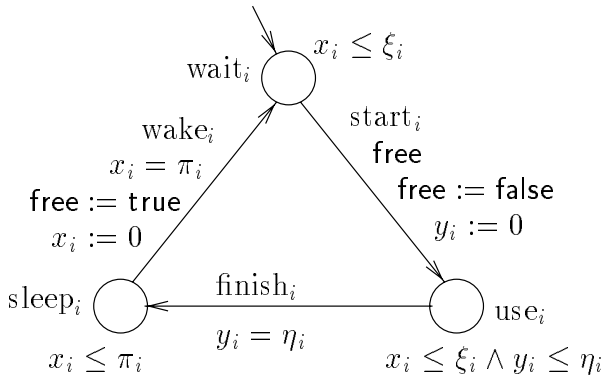


Figure 12.24: Modeling a periodic task with deadline as a TA.

Checking schedulability and computing schedulers. We solve the problem by checking whether the system satisfies an (untimed) BA. In fact, we present two solutions, which differ in the specification of the BA. The first one uses a BA with general type of acceptance conditions. Based on an observation about the structure of our system, the second solution uses a trivial BA with a single state and edge, thus, is more efficient.

[4] In fact, such a model is not generally decidable.

As a straightforward solution, we can use a cyclic BA like the one shown in figure 12.25, which specifies all acceptable schedulers for two tasks T_1 and T_2. An accepting run of the protocol passes infinitely often from both states labeled use_1 and use_2, meaning that both tasks are executed infinitely often. Notice that, due to the two intermediate states labeled **true**, there is not implicit requirement on the number of times that T_1 is served relatively to T_2, for instance, the scheduler which serves T_2 twice as much as T_1 is allowed by the BA. In general, for n tasks, we have to use a similar cyclic BA with $2n$ discrete states and $2n$ transitions.

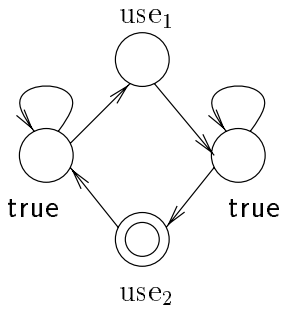


Figure 12.25: A BA specifying schedulability of two tasks.

The above solution is not very satisfactory, since it reduces the problem to checking emptiness of a TBA with general acceptance conditions, and this can be expensive, as discussed in section 7.2. Observe, however, that the system is strongly non-zeno: indeed, the TA for each task is strongly non-zeno (exactly π_i time units pass at each loop) and so is their parallel composition (lemma 3.2). By the same lemma, every infinite run of the system is non-zeno. Moreover, observe that in every infinite run, all tasks are served infinitely often. Indeed, if the TA for a task i was blocked in a discrete state then time could not progress because of the invariant, and the run would be zeno.

Figure 12.26: A trivial BA for finding non-zeno executions.

The above observations imply that the tasks are schedulable iff there exists an infinite run. Existence of such a run is specified by the trivial BA shown in figure 12.26. To check whether the system satisfies this BA we can check emptiness of their synchronous product using the technique of section 7.2.1 for trivial acceptance. In fact, it is even unnecessary to compute the synchronous product of the system's TA A and the above BA: their product is A itself, viewed as a TBA with all its discrete states marked repeating.

In case the tasks are schedulable, a sample infinite run is a scheduler. The scheduler must of course have a finite representation, so the run must be periodic. According to the results of section 8.2 such a run exists in this case (since there are no strict bounds) and can be computed using the constraint-induction technique. This is not necessary, however, because the period and execution delay of each task are fixed. Therefore, the schedule can be completely derived

by the execution order of the tasks, which corresponds to the cycle found by the emptiness algorithm.

Results and performance. Using the TBA-emptiness option of `kronos`, we have experimented with different values for n, π_i, ξ_i, η_i. For instance:

- For $n = 2$ and

i	π_i	ξ_i	η_i
1	3	2	1
2	6	3	1

  the tasks are schedulable and `kronos` returns almost instantaneously the sample scheduler shown in figure 12.27.

- For $n = 6$ and

i	π_i	ξ_i	η_i
1	10	5	1
2	20	10	2
3	15	10	5
4	20	15	10
5	12	11	7
6	17	9	1

  the tasks are not schedulable. `kronos` explores a graph of 484 nodes in 23 seconds of CPU time consuming 4 Mbytes of memory in a Sparc-station 20.

Generally, the performances are good, however, we should note that in real-world scheduling problems the number of tasks is often greater by an order of magnitude or more, and the same is true concerning the parameters π, ξ and η. In order for the tool to be used in such an environment, specific methods have to be developed such a heuristics for *best-first* search, or a pre-processing of the model to make periods more homogeneous.

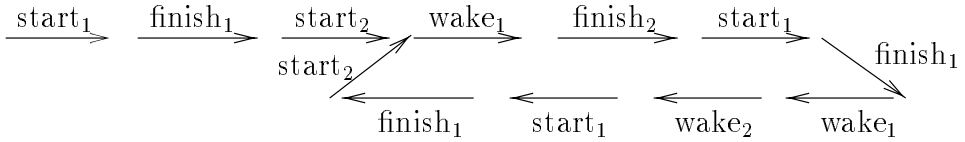


Figure 12.27: A scheduler for two tasks.

Possible extensions of the model. Dealing with more than one processors is trivial: if we have m processors at our disposal, we just have to replace the boolean variable free by a counter initially set to m, decremented each time a process starts execution and incremented each time a process finishes. Then, a process can only start when the counter is strictly positive.

The assumption of tasks becoming ready synchronously the first time can also be easily removed by adding an extra initial control location to the TA of each task i, and a starting edge which goes to state "wait$_i$" after resetting x_i to zero.

12.6 Controllers for multimedia documents

This is another real-world application, from the domain of multimedia. It concerns the language MADEUS for the specification of multimedia documents, developed as part of the INRIA project OPERA [JLSIR97].

A multimedia document consists of a set of *media objects* and a set of constraints specifying the temporal and spatial relations between objects. (In this work, we restrict our attention to temporal relations). Objects are either *basic* (video clip, audio clip, image, text, button, etc) or *composite*. Basic objects are either *controllable*, meaning that they start and finish according to commands of a controller, or *uncontrollable*, meaning that they are started by the controller, but stop according to commands of the environment. The controller in this case may be the program executing the document. The environment can be the viewer of the document (a human user), the media source which offers unpredictable delays of deliverability, and so on.

Given a MADEUS specification of a document, the problem consist in checking whether it is executable and if so, in computing a controller. The application is interesting for the following reasons:

- It represents a real problem, which could not be handled by people of the field of multimedia using classical methods such as linear programming [5]. Instead, the problem can be formulated as a reachability synthesis problem for CTA.
- It required a considerable amount of effort to develop a framework for modeling multimedia documents. In particular:
  - To capture the hierarchical nature and parallel execution of documents in a convenient way, we have modeled them as *Petri nets with deadlines* (PND) [BST98]. PND serve only as an intermediate description language, since they can be directly translated to TA. The details of the translation can be found in appendix A.3. In this section, PND are only presented informally.
  - To capture some of the MADEUS operators for building composite objects, we have introduced two new synchronization schemes, based on *interrupts* and *waiting*.

Description. We consider a finite set of basic objects, each having a beginning, an end and a *duration*. The latter represents the amount of time the corresponding medium is active, for example, the amount of time a video clip is displayed on the screen, or the time elapse from the point a button is shown one the screen until the moment it is pressed. Durations can be non-fixed, belonging to a *duration interval* of the form $[low, up]$ or $[low, \infty)$, where $low, up \in \mathbb{N}$ and $low \leq up$,

The duration of an uncontrollable object is determined by the environment, while the duration of a controllable object is determined by the controller. For example, a button is naturally uncontrollable: its duration is determined by the user clicking on the button. On the other hand, a still image is usually controllable: its duration can be adjusted so that the execution of the rest of the document is done smoothly.

[5] Indeed, in the presence of uncontrollability, the problem cannot be reduced to solving a system of linear constraints. This is because, variables need to be quantified existentially or universally, depending on whether they correspond to controllable or uncontrollable objects. However, the order of the quantifiers depends on the execution sequence chosen, and cannot be fixed a-priori in a single system of constraints.

Composite objects are specified hierarchically using a set of binary operators. Intuitively, if o_1, o_2 are objects, then:

- o_1 `meets` o_2 is the composite object which begins by launching o_1, launches o_2 as soon as o_1 terminates and ends when o_2 terminates.
- o_1 `equals` o_2 is the composite object which begins by launching o_1 and o_2 simultaneously and terminates when both o_1 and o_2 terminate. The two objects must be able to terminate simultaneously, otherwise the specification is inconsistent.
- o_1 `parmin` o_2 is the composite object which begins by launching o_1 and o_2 simultaneously and terminates as soon as one of them terminates.
- o_1 `parmax` o_2 is the composite object which begins by launching o_1 and o_2 simultaneously and terminates when the last one of them terminates.
- o_1 `master` o_2 is the composite object which begins by launching o_1 and o_2 simultaneously and terminates when o_1 (the *master*) terminates.

Notice that the `equals` operator, apart from building a composite object, also imposes a temporal constraint on its arguments, namely, that they finish at the same time.

12.6.1 Petri Nets with Deadlines: informal presentation

For readers who are familiar with Petri nets, a PND can be viewed as a *1-safe Petri net* extended with a finite number of clocks (like a TA is a finite automaton extended with clocks). Each transition of the PND has a guard and a set of clocks to be reset. The semantics are similar to those of TA: a state of a PND is a pair of a marking and a clock valuation; a discrete transition consists in firing a transition of the net, consuming and producing the corresponding tokens, and resetting some clocks; finally, time can pass in a marking as long as it can pass in every place having a token.

For readers not familiar with Petri nets, a PND is similar to a TA, with the difference that more than one discrete states (called *places*) can be active at the same time. An active place is said to have a *token*. A set of active places is called a *marking*.

As an example, consider the simple PND shown in the top of figure 12.28. In the initial marking the active places are q_1 and q_2. Firing transition e_1 requires two tokens, one in each place q_1 and q_2. Upon firing transition e_1, the tokens of places q_1 and q_2 are consumed, a token is produced in place q_3, and clock x is reset to zero. An amount of time passes at this marking (q_3 is active), between two and five time units. Then, transition e_2 is fired and in the new marking the active places are q_4 and q_5.

In our case the number of tokens remains bounded in any execution of the PND, so that the latter can be translated to a TA. The discrete structure of the TA corresponds to the *marking graph* of the PND, that is, all possible markings and transitions between them. The TA corresponding to the PND of the above example is shown in the bottom of figure 12.28.

A PND can also be extended for controllability, by simply dividing its transitions into controllable and uncontrollable. The TA corresponding to such a PND is a CTA.

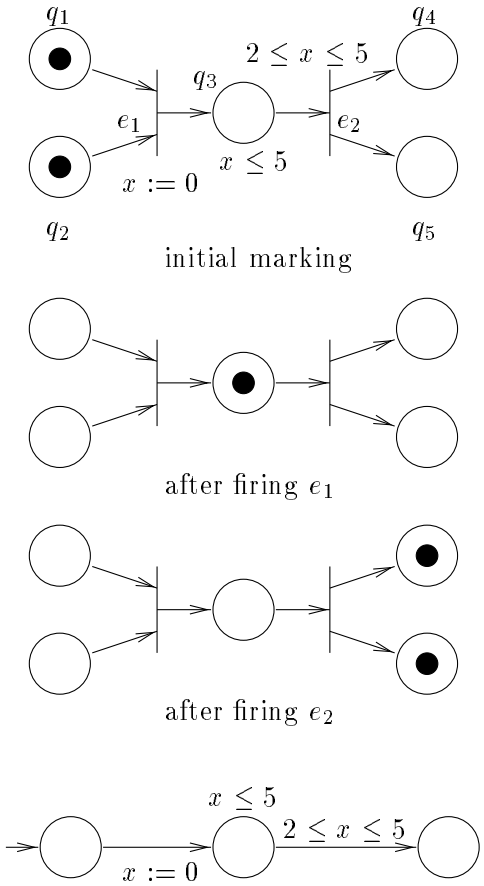


Figure 12.28: A simple PND (top), firing transitions (middle) and the corresponding TA (bottom)

12.6.2 Using Petri nets with deadlines to model multimedia documents

For a given a document o, we construct a PND which is supposed to capture all possible executions of o, if any exist. The construction is recursive on the syntax of o, that is, if o is made up of two sub-documents o_1 and o_2, we first construct the PND for o_1 and o_2, and then combine them to form the PND for o. To take controllability into account, each transition of the PND is marked as either controllable or uncontrollable.

The PND for a basic object o with duration interval $[low, up]$ is shown in figure 12.29(a). It has three places, two transitions, and a clock x counting its duration (we assume a different clock for each basic object). The transition labeled "start" is controllable, whereas the one labeled "finish" is controllable if o is controllable, and uncontrollable otherwise.

The PND for a sub-document o_i

- has a single initial place with one token;
- has a single final place;
- has a controllable initial transition, labeled start$_i$, which resets a set of clocks X_i and has trivial guard;
- has a final (controllable or uncontrollable) transition, labeled finish$_i$, with guard ζ_i, resetting no clocks;
- has a "body" of places and (controllable or uncontrollable) transitions;
- is *acyclic*, that is, has no structural loop.

The general form of this PND is shown in figure 12.29(b). Notice that the PND for a basic object matches the above general pattern.

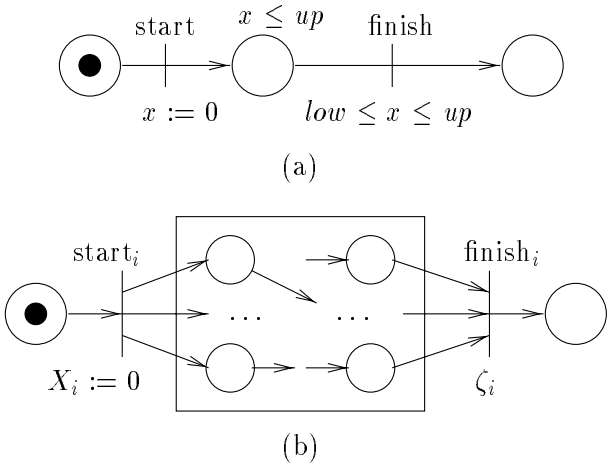


Figure 12.29: Modeling multimedia documents as PND: basic object (a) and general form of composite object (b).

Assuming that the PND for objects o_1 and o_2 have this form, the PND for o_1 `meets` o_2 is constructed as shown in figure 12.30. The construction consists simply in merging the final

transition of the PND for o_1 with the initial transition of the PND for o_2. The merged transition inherits the controllability status of the final transition of the PND for o_1. All other transitions keep their controllability status.

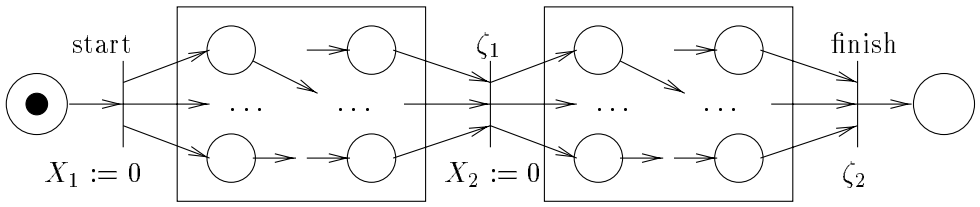


Figure 12.30: The PND for `meets`.

The PND for o_1 `equals` o_2 is shown in figure 12.31. Its initial transition is built by merging the initial transitions of the two sub-PND for o_1 and o_2. To make sure that o_1 and o_2 finish at the same time, we introduce two auxiliary clocks z_1 and z_2 which are reset in the final transitions of the two sub-PND and and tested to zero in the final transition of the composite PND. The initial and final transitions of the composite PND are controllable [6], whereas all other transitions keep their controllability status.

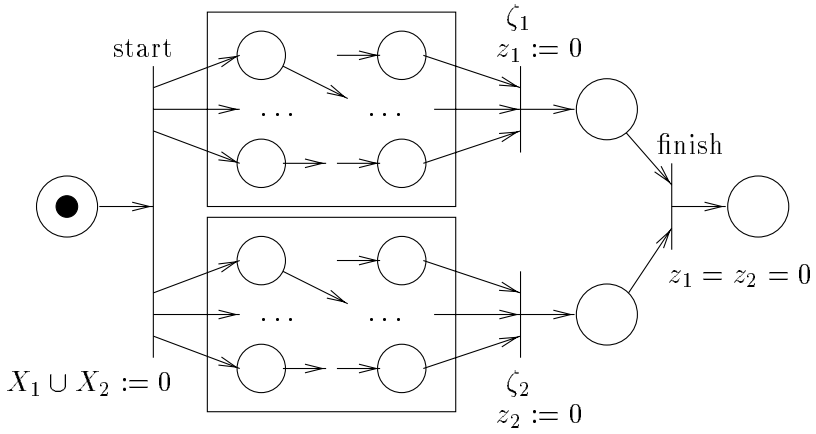


Figure 12.31: The PND for `equals`.

The PND for o_1 `parmin` o_2 is shown in figure 12.32. To model interruption, we employ a *macro-notation à la Statecharts* []: a transition connected to the body of a PND is called a *macro-transition*. It representes a set of transitions, one for each possible marking of the PND's body. In that way, when the macro-transition is fired, the body of the PND is "emptied", modeling interruption. Each macro-transition inherits the controllability status of the final transition of the corresponding sub-PND. The initial and final transitions of the composite PND are controllable. z is an auxiliary clock.

The PND for o_1 `parmax` o_2 is shown in figure 12.33. Its construction is supposed to model waiting and uses an auxiliary clock z as well as some auxiliary places. Intuitively, when one of the two underlying PND finishes execution, it checks immediately whether the other PND

[6] In fact, the final transition is *time-deterministic*, since it has a punctual guard $z = 0$, so, marking it uncontrollable would not matter.

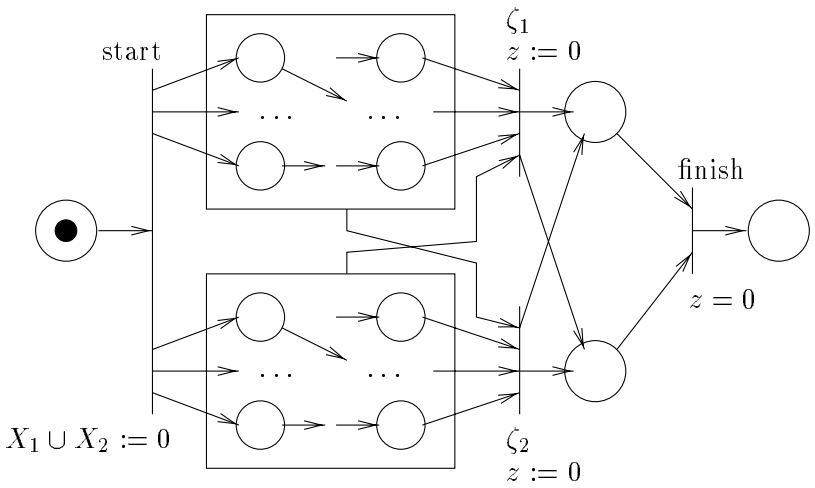


Figure 12.32: The PND for `parmin`.

has also finished. If this is the case, the token goes to the one-before-final place, and in zero time, to the final place. Otherwise, the token goes to one of the middle auxiliary places, and the PND waits for the other one to finish. Regarding controllability, all new transitions are controllable, while the old ones keep their controllability status.

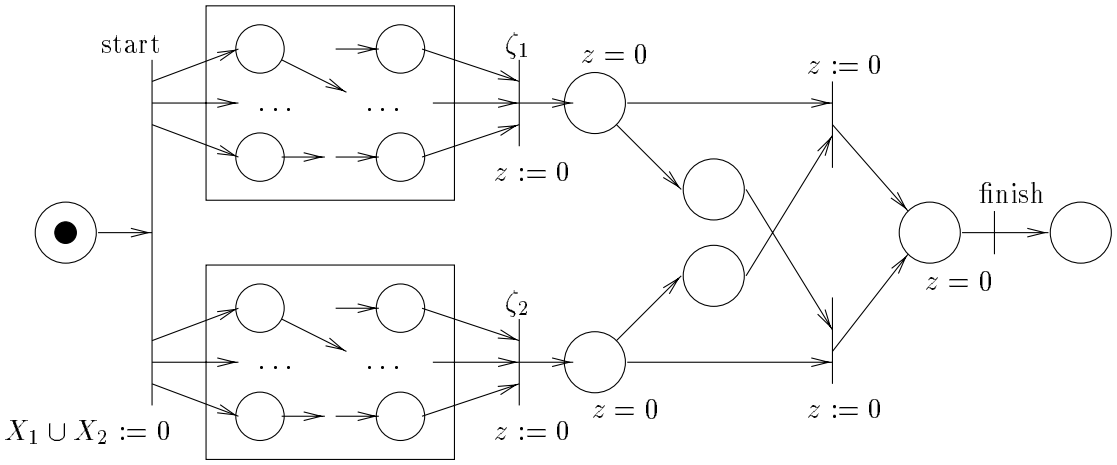


Figure 12.33: The PND for `parmax`.

Finally, the PND for o_1 `master` o_2 is shown in figure 12.34. Its construction is a combination of interruption and waiting, as those for the operators `parmin` and `parmax`. Again, all new transitions are controllable, while the old ones keep their controllability status.

Optimizations

Straightforward as it is, the above construction admits a number of optimizations, to reduce the number of places, transitions and clocks in the final PND.

First of all, the specification can be *pre-processed*: assume that a sub-document o is made up of two basic objects o_1 and o_2 which are both controllable, with duration intervals $[low_i, up_i]$, for $i = 1, 2$, respectively. Then, o can be replaced by a single basic object with duration interval:

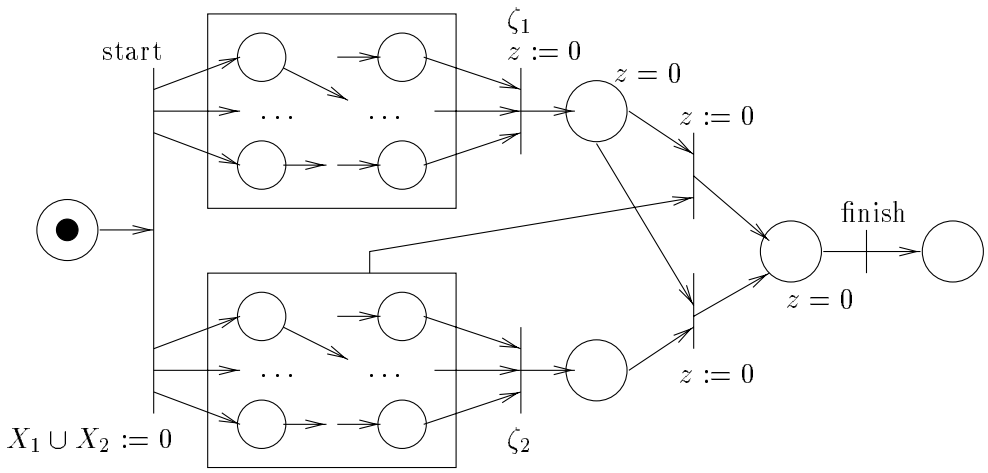


Figure 12.34: The PND for `master`.

- $[low_1 + low_2, up_1 + up_2]$, if $o = o_1$ `meets` o_2,
- $[\max(low_1, low_2), \min(up_1, up_2)]$, if $o = o_1$ `equals` o_2 (in case the interval is empty, the specification is inconsistent),
- $[\min(low_1, low_2), \min(up_1, up_2)]$, if $o = o_1$ `parmin` o_2,
- $[\max(low_1, low_2), \max(up_1, up_2)]$, if $o = o_1$ `parmax` o_2,
- $[low_1, up_1]$, if $o = o_1$ `master` o_2.

Other optimizations can be performed directly on the constructed PND, or the resulting TA. For instance, clocks can be re-used so that a new clock is not introduced for every sub-document: this is possible by applying the `optikron` module to compute clock activities and then identify two or more clocks which are never active at the same time.

A concrete example: the Greeting card

To illustrate the construction presented above, we model a concrete document provided by the MADEUS group. The document models an animated greeting card.

The card starts off with a spoken and written invitation for pressing a button. When the button is pressed the text "Merry Christmas" appears accompanied by Christmas music. The texts "and" and "Happy New Year" follow, the latter accompanied by another piece of music. At the end, the text "Smith" moves over the screen. After displaying the word "and", an animation is executed. When it ends, the text "family" appears and moves over the screen. Finally, the two textual elements "family" and "Smith" come to stop synchronously, one above the other. In parallel to all this, a sequence of background pictures is displayed. It ends with a photo of the family Smith.

The basic objects involved in the document are the following:

Ready	/*	text	*/	$[1, \infty)$	controllable;
Intro	/*	audio	*/	$[8, \infty)$	controllable;
Go	/*	button	*/	$[0, 9]$	uncontrollable;
XmasT	/*	text	*/	$[4, 10)$	controllable;
XmasM	/*	audio	*/	$[5, 7]$	uncontrollable;
And	/*	text	*/	$[2, 5]$	controllable;
Smiley	/*	animation	*/	$[3, 4]$	uncontrollable;
Family	/*	text	*/	$[5, 15]$	controllable;
YearT	/*	text	*/	$[5, 15]$	controllable;
Smith	/*	text	*/	$[5, 15]$	controllable;
YearM	/*	audio	*/	$[5, 7]$	controllable;
Pict1	/*	picture	*/	$[4, 5]$	controllable;
Pict2	/*	picture	*/	$[4, 5]$	controllable;
Pict3	/*	picture	*/	$[4, 5]$	controllable;
Pict4	/*	picture	*/	$[4, 5]$	controllable;
Photo	/*	picture	*/	$[4, 5]$	controllable;

The specification is given by the following expression:

```
Background := Pict1 meets Pict2 meets Pict3 meets Pict4 meets Photo
Beginning  := Go master (Ready parmax Intro)
Next       := (XmasT parmin XmasM) meets
              ( (And meets (YearT parmin YearM) meets Smith) equals
                (Smiley meets Family)
              )
Card       := Background equals (Beginning meets Next)
```

After pre-processing, we end up with the simplified specification:

```
Card := Photos equals
        ( Go meets (XmasT parmin XmasM) meets
        (AYS equals (Smiley meets Family) )
        )
```

where "Photos" and "AYS" are controllable basic objects, corresponding to composite objects "Background" and "And meets (YearT parmin YearM) meets Smith", respectively. The duration intervals of "Photos" and "AYS" are:

Photos	$[20, 25]$	controllable;
AYS	$[12, 27]$	controllable;

The PND for the above specification is shown in figure 12.35. Places are labeled with the name of the corresponding basic objects (places with no name are auxiliary). Uncontrollable transitions are drawn in dashed lines. After optimizing the usage of clocks, we end up with three clocks: x counts the duration of "Photos"; y counts the duration of "XmasT" and "XmasM", then of "Smiley", and finally of "Family"; z counts the duration of "AYS" and also serves as an auxiliary clock for modeling "XmasT parmin XmasM".

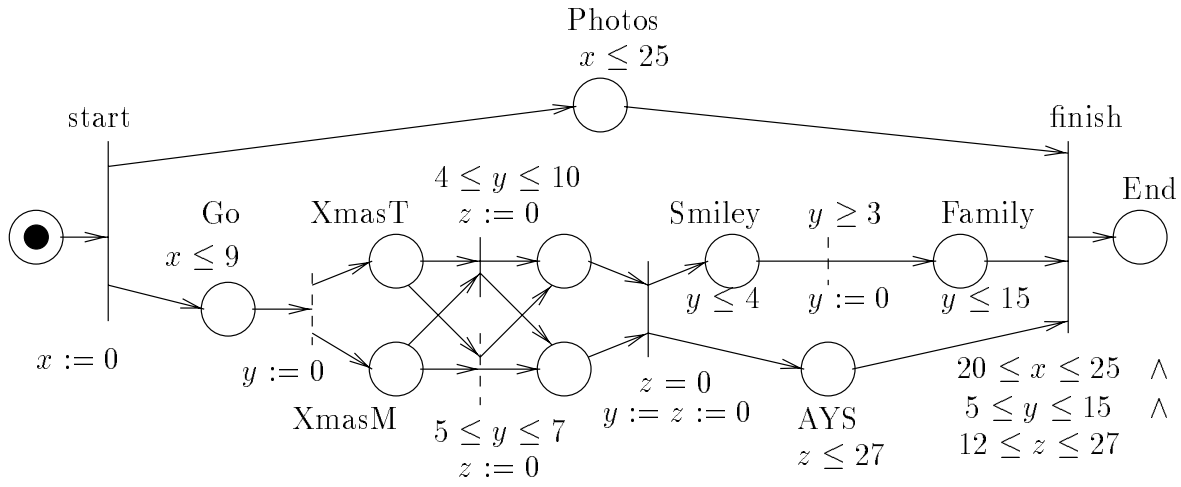


Figure 12.35: The PND for the greeting card example.

12.6.3 Controller synthesis for multimedia documents

The first step in the analysis of a multimedia document is to generate its PND and translate it into a CTA. For the greeting-card example, the PND of figure 12.35 is translated into the CTA of figure 12.36. Notice that, due to the special structure of the PND, the CTA is also acyclic, with a single final discrete state (in fact, a sink state). We label this state "End".

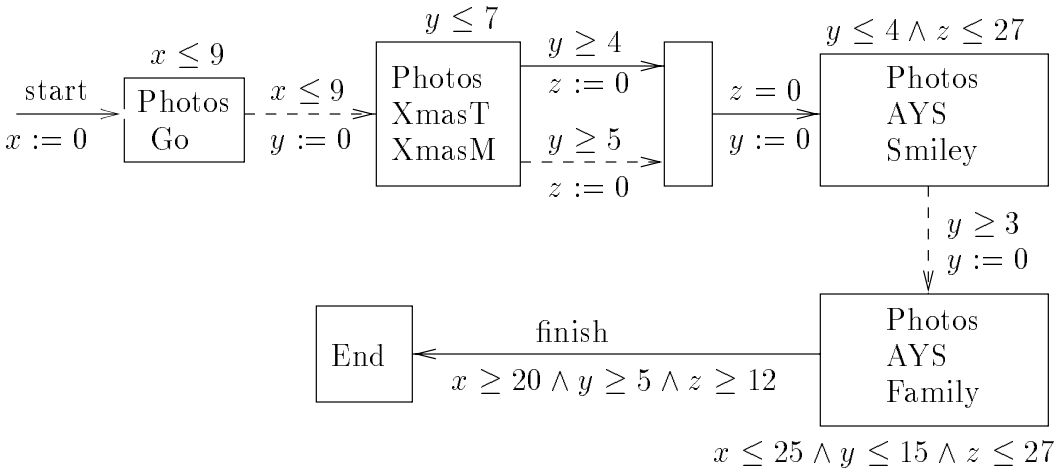


Figure 12.36: The CTA for the greeting card example.

Checking executability of the document means solving controller synthesis for reachability of "End". If a winning strategy exists, it corresponds to the document's controller.

We have used `synth-kro` to compute a controller for the example above. As explained in section 9.2, the tool computes the set of winning states and returns the restriction of the initial CTA to the winning states. Execution takes only a few seconds.

The restricted CTA representing the controller is shown in figure 12.37. Essentially, it differs from the initial CTA in the allowed choices for the controller at state "Photos,XmasT,XmasM". The additional invariant $x \le 13$ shows that control is *adaptive*: if the environment is late then the controller must react earlier than usual. For example, if the button "Go" is pressed at

$x = 8.5$, then "XmasM" is interrupted by "XmasT" at $y = 4.5$ the latest, so that the invariant $x \leq 13$ is not violated. The initial CTA could fail in this case, since the only invariant at state "Photos,XmasT,XmasM" is $y \leq 7$.

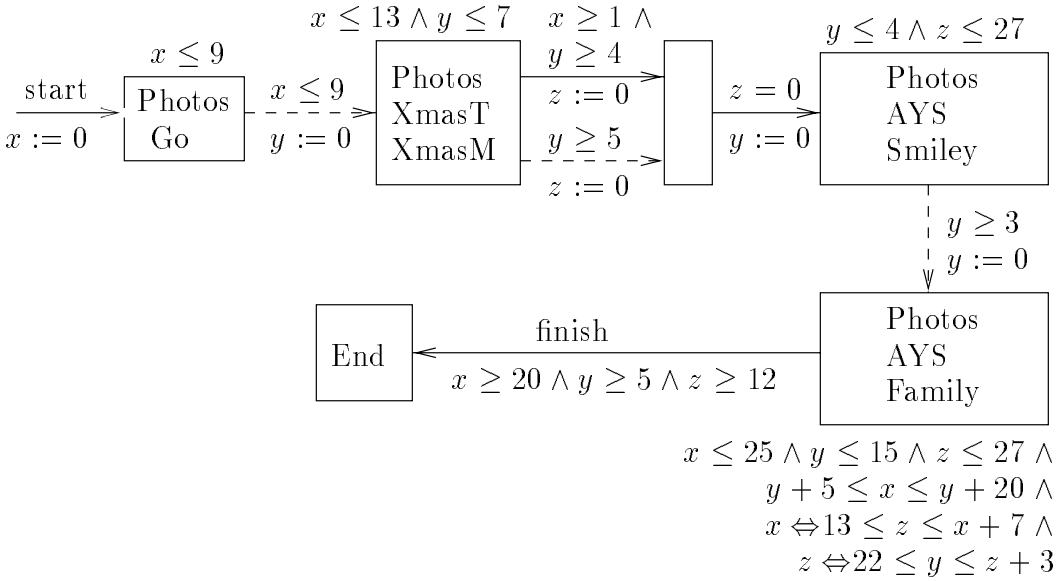


Figure 12.37: The restricted CTA for the greeting card example.

Relation to the literature

MADEUS is inspired from the logic of intervals [All83], with the addition of interruption and waiting operators. The specification of multimedia documents been previously considered in [SDdSS94], using a complicated model of time Petri nets with many different operators for transition synchronization. PND can capture this model, as shown in [BST98], and can also be used for verification and synthesis.

Chapter 13

Conclusions

We have presented a complete formal framework for the analysis of timed systems. Our focus has been on practicality. On the model level, we have adopted the existing model of dense-time automata, believing it to be especially suitable for asynchronous timed systems. As specification languages, we have chosen both linear- and branching-time formalisms covering a large spectrum of quantitative-time properties.

We have provided an analysis methodology based on abstractions which reduce the state space of TA, while preserving properties of interest. These abstractions are defined automatically, that is, no effort is required from the user to invent them. We have shown how to compute the abstractions efficiently, in some cases combining them with the analysis on-the-fly. We have presented a method for providing concrete diagnostics as feedback to the user. We have treated both problems of verification and controller synthesis. For the latter, we have introduced a framework for syntactic and semantic parallel composition of timed systems in the presence of controllability and have developed an on-the-fly controller-synthesis algorithm.

Concretely, we have implemented tools for timed verification and controller synthesis. The input of these tools is given in terms of parallel compositions of TA extended with bounded discrete variables. This extension makes the model quite practical for case studies. The parallel composition of the automata and the discrete state space are generated on-the-fly. The types of analysis that can be performed by the tools go from simple reachability, deadlock- and timelock-detection, to linear- and branching-time model-checking with concrete diagnostics, to controller synthesis.

Experimentation has not only been a way to evaluate the methods and tools, but also a source of inspiration. We have treated a number of case studies including communication protocols, asynchronous circuits, real-time schedulers and multimedia controllers. Most of the case studies come from the industrial world and all of them have non-trivial modeling and analysis complexity. Our analysis has been helpful: inconsistencies were found in the cases of BANG&OLUFSEN's and CNET's protocols and a real schedulability problem has been solved in the case of multimedia documents. Experimental results show that the new techniques bring a significant improvement in performance, often orders of magnitude better than previous attempts.

Perspectives

In his concluding remarks, [Alu91] sets hopes for a practical verification of timed systems, and beyond. Seven years later, the field seems to be a bit closer to these goals: real-world case

studies of non-trivial modeling and analysis complexity have been treated using non-industrial tools; techniques for tackling the state explosion problem have been and are being devised; a prototype tool for dense-time controller synthesis exists, with practical experiments which are quite encouraging.

Still, a long way remains in order for such a framework to become common practice in the industrial world. In the near future, the following research directions seem quite promising:

- Efficient schemes for symbolic representation: a homogeneous way to represent discrete and continuous variables is still missing. Such a representation scheme is important to have, since most practical timed systems include a significant part of discrete variables.

- Application-oriented techniques: although quite general, TA are a relatively low-level model. A number of higher-level timed languages are currently used as a front-end to TA which are then used for the analysis. For example, KRONOS has been interface to AORTA [BHKR95], GRAFCET [MLP96], ET-LOTOS [Her98] and SHIFT [AGS96].

  In order to achieve better usability for domain-specific applications, restrictions of the model and optimization of the algorithms can be envisaged. In this direction, `kronos-open` may be a useful start, since it is based on the compiler philosophy, where particularities of the domain can be taken into account.

- Composition of timed systems: the TA model still lacks a robust theory of composition, which would avoid deadlocks but also preserve the independency of the urgency requirements of each of the components. Following [SY96], work is in progress to define such a theory [BS97, BST98, Bor98]. It still remains to see how analysis techniques can be extended to take into account such composition frameworks. A more ambitious goal concerns compositional verification of timed systems, where properties proven in the components of a system can be assembled to a property that holds in the global system. The work of [TAKB96] is in this direction.

Bibliography

[ABK+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In *Proc. of the Intl. Workshop on Hybrid and Real-Time Systems*, 1997.

[ABT97] DTR/NA 52809: Resource Management Procedures and Cases of their Possible Usages, part 1, ABT/DT : protocol procedures. ETSI, NA4, September 1997.

[ACD+92] A. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *RTSS'92*. IEEE, 1992.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.

[ACH+92] R. Alur, C.Courcoubetis, N. Halbwachs, D.L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *3rd Conference on Concurrency Theory CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 340–354. Springer-Verlag, 1992.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD90] R. Alur and D. Dill. Automata for modeling real-time systems. In *17th ICALP*, LNCS 443, 1990.

[AGS96] A.Deshpande, A. Göllü, and L. Semenzato. The SHIFT programming language and run-time system for dynamic networks of hybrid automata. Technical report, PATH, 1996.

[AH92] R. Alur and T.A. Henzinger. Logics and models of real time: a survey. In *Real Time: Theory in Practice*, LNCS 600, 1992.

[AIKY92] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. In *Proceedings of the 4th Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[AJ98] P. Abdulla and B. Jonsson. Verifying networks of timed processes. In *Tools and Algorithms for the Construction and Analysis of Systems '98, Lisbon, Portugal*, volume 1384 of *LNCS*. Springer-Verlag, 1998.

[AK83] S. Aggarwal and R.P. Kurshan. Modelling elapsed time in protocol specification. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing and Verification, III*, pages 51–62. Elsevier Science Publisers B.V., 1983.

[AK96] R. Alur and R.P. Kurshan. Timing analysis in COSPAN. In *Hybrid Systems III*, LNCS 1066, 1996.

[AKV98] R. Alur, R.P. Kurshan, and M. Viswanathan. Membership questions for timed and hybrid automata. In *RTSS'98*, 1998.

[AL91] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *REX workshop "Real-time: theory in practice"*, number 600 in LNCS, pages 1–27. Springer-Verlag, 1991.

[All83] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[Alt98] K. Altisen. Génération automatique d'ordonnancements pour systèmes temporisés. Technical report, Mémoire de DEA, Ensimag, Grenoble, 1998. In french.

[Alu91] Rajeev Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Department of Computer Science, Stanford University, 1991.

[AMP98] E. Asarin, O. Maler, and A. Pnueli. On the discretization of delays in timed automata and digital circuits. In *Concur'98*, 1998.

[Bal96] F. Balarin. Approximate reachability analysis of timed automata. In *Proc. 17th IEEE Real-Time Systems Symposium*, 1996.

[BB91a] J.C.M. Baeten and J.A. Bergstra. Real-time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[BB91b] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *IEEE Proceedings*, 79:1270–1282, September 1991.

[BCD+90] J.B. Burch, E.M. Clarke, D.Dill, L.J. Hwang, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. In *5th LICS*, pages 428–439. IEEE, 1990.

[BD91] B. Berthomieu and M. Diaz. Modeling and verification of time-dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[BD98] D. Bošnački and D. Dams. Integrating real time into spin: A prototype implementation. In *Proceedings of the FORTE/PSTV XVIII Conference*. Chapman and Hall, 1998.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: a model-checking tool for real-time systems. In *CAV'98*, 1998.

[BFH+92] A. Bouajjani, J.C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BGK+96] J. Bengtsson, W. Griffioen, K. Kristorffersen, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In *CAV'96*, LNCS 1102, 1996.

[BHKR95] S. Bradley, W. Henderson, D. Kendall, and A. Robson. Validation, verification and implementation of timed protocols using AORTA. In P. Dembinski and Sredniawa M, editors, *Proc. 15th PSTV*, Warsaw, Poland, June 1995. IFIP, Chapman & Hall.

[BJLY98] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reductions for timed systems. In *9th International Conference on Concurrency Theory*, 1998.

[BLL+95] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – a tool suite for automatic verification of real-time systems. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.

[BLY96] A. Bouajjani, Y. Lakhnech, and S. Yovine. Model Checking for Extended Timed Temporal Logics. In *Proc. Intern. Symp. on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'96)*. LNCS 1135, 1996.

[BM98] W. Belluomini and C. Myers. Verification of timed systems using partially ordered sets. Technical report, University of Utah, 1998.

[BMPY97] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Proc. of the 8th Conference on Computer-Aided Verification*, 1997.

[BMS99] M. Bozga, O. Maler, and S.Tripakis. Efficient verification of timed automata using dense and discrete time semantics. Submitted to CAV'99, 1999.

[BMSU97] N. Bjørner, Z. Manna, H. Sipma, and T. Uribe. Deductive verification of real-time systems using STeP. In *ARTS'97*, LNCS, 1997.

[Bor98] S. Bornot. *De la composition des systèmes hybrides*. PhD thesis, Université Joseph Fourrier de Grenoble, 1998. In french.

[Boz97] M. Bozga. SMI: An open toolbox for symbolic protocol verification. Technical report, Verimag, March 1997.

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 1986.

[BS94] J. Brzozowski and C. Seger. *Asynchronous Circuits*. Springer, 1994.

[BS97] S. Bornot and J. Sifakis. Relating time progress and deadlines in hybrid systems. In *International Workshop, HART'97*, pages 286–300, Grenoble, France, March 1997. Lecture Notes in Computer Science 1201, Spinger-Verlag.

[BST98] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality*, LNCS 1536, 1998. To appear.

[BT92] P.E. Boyer and D.P. Tranchier. A reservation principle with applications to the atm traffic control. *Computer Networks and ISDN Systems*, 24:321–334, 1992.

[BTY97] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model checking for real-time systems. In *Proc. of the 18th IEEE Real-Time Systems Symposium, San Francisco, CA*, pages 232–243. IEEE, December 1997.

[Büc62] J.R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science 1960*, pages 1–12. Stanford University Press, 1962.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. POPL*, 1977.

[CDCT92] C. Courcoubetis, D. Dill, M. Chatzaki, and P. Tzounakis. Verification with real-time COSPAN. In *Proceedings of the Fourth Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 1994.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8(2):117–141, April 1974.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symp. POPL*, 1987.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992. A preliminary version appeared in the proceedings of CAV'90 (also in Springer Verlag LNCS).

[Daw98] C. Daws. *Méthodes d'analyse de systèmes temporisés: de la théorie à la pratique.* PhD thesis, Institut National Polytechnique de Grenoble, 1998. In french.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 1997.

[Dil89] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, pages 197–212. Springer–Verlag, 1989.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[DOY94] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In D. Hogrefe and S. Leue, editors, *Proc. 7th. IFIP WG G.1 International Conference of Formal Description Techniques, FORTE'94*, pages 227–242, Bern, Switzerland, October 1994. Formal Description Techniques VII, Champan & Hall.

[DT98] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems '98, Lisbon, Portugal*, volume 1384 of *LNCS*. Springer-Verlag, 1998.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proc. 17th IEEE Real-Time Systems Symposium, RTSS'96*, 1996.

[EC81] E.A. Emerson and E. Clarke. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*. LNCS 131, 1981.

[EH86] E.A. Emerson and J.Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *ACM journal*, 33(1):151–178, 1986.

[EL85] E. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. In *12th ACM Symp. POPL*, 1985.

[FGM+92] J.Cl. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A tool box for the verification of lotos programs. In *14th International Conference on Software Engineering*, 1992.

[FM91] J.Cl. Fernandez and L. Mounier. "On the fly" verification of behavioural equivalences and preorders. In Springer Verlag, editor, *Workshop on Computer-Aided Verification, Aalborg University, Denmark, LNCS 575*, 1991.

[Gar98] H. Garavel. OPEN-CAESAR: An open software architecture for verification, simulation and testing. In *Tools and Algorithms for the Construction and Analysis of Systems '98, Lisbon, Portugal*, LNCS 1384. Springer-Verlag, 1998.

[GPV94] A. Göllü, A. Puri, and P. Varaiya. Discretization of timed automata. In *33rd CDC*, 1994.

[Gre97] M. Greenstreet. STARI: Skew tolerant communication. *IEEE Transactions on Computers*, 1997.

[GW91] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *4th CAV*, July 1991.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In *5th Conference on Computer-Aided Verification*. LNCS 697, 1993.

[HBAB93] H. Hulgaard, S. Burns, T. Amon, and G. Borriello. Practical applications of an efficient time separation of events algorithm. In *ICCAD'93*, 1993.

[Her98] Christian Hernalsteen. *Specification, Validation and Verification of Real-Time Systems in ET-LOTOS*. PhD thesis, Université Libre de Bruxelles, 1998.

[HHW97] T. Henzinger, P.-H. Ho, and H. Wong Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1, 1997.

[HK89] Z. Har'El and R. Kurshan. Automatic verification of coordinating systems. In *CAV*, LNCS 407, 1989.

[HK97] T. Henzinger and P. Kopke. Discrete-time control for rectangular hybrid automata. In *ICALP '97*, 1997.

[HKV96] T. Henzinger, O. Kupferman, and M. Vardi. A Space-Efficient On-the-Fly Algorithm for Real-Time Model-Checking. In *CONCUR'96*. LNCS 1119, 1996.

[HMP92] T. Henzinger, Z. Manna, , and A. Pnueli. What good are digital clocks? In *ICALP'92*, LNCS 623, 1992.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[HSLL97] K. Havelund, A. Skou, K. Larsen, and K. Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using Uppaal. In *Proceedings of the 18th IEEE Real-Time Systems Symposium San Francisco, CA*, pages 2–13, December 1997.

[HT87] T. Hafer and W. Thomas. Computation Tree Logic CTL* and Path Quantifiers in the Monadic Theory of the Binary Tree. In *ICALP'87*. LNCS 267, 1987.

[HT96] M.R. Henzinger and J.A. Telle. Faster algorithms for the nonemptiness of street automata and for communication protocol prunning. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory*, pages 10–20, 1996.

[HW91] G. Hoffmann and H. Wong Toi. The input-output control of real-time discrete event systems. In *30th IEEE Conf. on Decision and Control*, 1991.

[Jai94] R. Jain. *FDDI handbook: high-speed networking using fiber and other media.* Addison-Wesley, 1994.

[JLSIR97] M. Jourdan, N. Layaïda, L. Sabry-Ismail, and C. Roisin. An integrated authoring and presentation environment for interactive multimedia documents. In *Proc. of the 44th Conference on Multimedia Modelling*, Singapore, November 1997. World Scientific Publishing.

[KLL+97] Kristoffersen, F. Laroussinie, K. Larsen, P. Petterson, and W. Yi. A compositional proof of a real time mutual exclusion protocol. In *Proc. of the 7th Intl. Conf. on the Theory and Practice of Software Development*, 1997.

[KP92] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In *Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium*, volume 571 of *LNCS*, 1992.

[LA90] N.A. Lynch and H. Attiya. Using mappings to prove timing properties. In *9th ACM Symp. on Principles of Distributed Computing*, pages 265–280, 1990.

[Lam80] L. Lamport. Sometimes is sometimes "not never"– on the temporal logic of programs. In *7th ACM Symp. POPL*, pages 174–185, 1980.

[Lam83] L. Lamport. What good is temporal logic? In R.E.A. Mason, editor, *Information Processing 83: Proceedings of the Ninth IFIP World Computer Congress*, pages 657–668. Elsevier Science Publishers, 1983.

[Lew89] H.R. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical report, Harvard University, 1989.

[Lew90] H. Lewis. A logic of concrete time intervals. In *5th IEEE Symp. LICS*, 1990.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 1995.

[LL93] G. Leduc and L. Léonard. A timed LOTOS supporting dense time domain and including new timed operators. In *Formal Description Techniques V*, pages 87–102, 1993.

[LLPY97] K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proceedings of the 18th IEEE Real-Time Systems Symposium San Francisco, CA*, pages 14–24, December 1997.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *12th ACM Symp. POPL*, pages 97–107, New Orleans, January 1985.

[LPY95] K. Larsen, P. Pettersson, and W. Yi. Diagnostic model-checking for real-time systems. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.

[LPY97] K. Larsen, P. Petterson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1/2), October 1997.

[LWYP98] K. Larsen, C. Weise, W. Yi, and J. Pearson. Clock difference diagrams. Technical Report Nr 98/99, ISSN 0283-0574, DoCS, Uppsala University, August 1998.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. In *ACM Symp. on Theory of Computing*, 1992.

[LY93] K. Larsen and W. Yi. Timed abstracted bisimulation: implicit specification and decidability. In *Proc. MFPS'93*, 1993.

[Mer74] P. Merlin. A study of the recoverability of computer systems. Master's thesis, University of California, Irvine, 1974.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[MLP96] L. Marcé, D. L'Her, and P. Le Parc. Modelling and verification of temporized GRAFCET. In *Proc. CESA IMACS*, Lille, France, July 1996.

[Mou93] L. Mounier. *Méthodes de Vérification de Spécifications Comportementales : étude et mise en œuvre*. PhD thesis, Université Joseph Fourrier de Grenoble, 1993. In french.

[MP95a] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In *CHARME'95*, LNCS 987, 1995.

[MP95b] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS '95*, 1995.

[NMV90] R. De Nicola, U. Montanari, and F.W. Vaandrager. Back and forth bisimulations. Technical report, CWI, Netherlands, May 1990.

[NRSV90] X. Nicollin, J.-L. Richier, J. Sifakis, and J. Voiron. ATP: an Algebra for Timed Processes. In *IFIP TC 2*, 1990.

[Oli94] A. Olivero. *Modélisation et analyse de systèmes temporisés et hybrides*. PhD thesis, Institut National Polytechnique de Grenoble, 1994. In french.

[Pel94] D. Peled. Combining partial order reductions with on–the–fly model checking. In *6th CAV*, june 1994.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *ACM Symp. POPL*, 1989.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive and algorithmic verification. In *Proc. 8th Conference Computer-Aided Verification, CAV'96, Rutgers, NJ*, volume 1102 of *LNCS*, 1996.

[PT87] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6), 1987.

[PV94] A. Puri and P. Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In *CAV'94*, LNCS 818, 1994.

[Ram74] C. Ramchandani. Analysis of asynchronous concurrent systems by petri nets. Technical Report MAC TR-120, MIT, 1974.

[RR88] G.M. Reed and A.W. Roscoe. A timed model for Communicating Sequential Processes. *Theoretical Computer Science*, 58:249–261, 1988.

[RW87] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1), January 1987.

[Sai97] Hassen Saidi. The Invariant-Checker : Automated deductive verification of reactive systems. In *Proceedings of the 9th Conference on Computer-Aided Verification, CAV'97*. Springer Verlag, 1997.

[SDdSS94] P. Sénac, M. Diaz, and P. de Saqui-Sannes. Toward a formal specification of multimedia scenarios. *Annals of telecomunications*, 49(5-6):297–314, 1994.

[Sif77] J. Sifakis. Use of petri nets for performance evaluation. In *Measuring, modelling and evaluating computer systems*, pages 75–93. North-Holland, 1977.

[Sif82] J. Sifakis. A unified approach for studying the properties of transition systems. *Theoretical Computer Science*, 18, 1982.

[SS95] O. Sokolsky and S. Smolka. Local Model Checking for Real-Time Systems. In *CAV'95*. LNCS 939, 1995.

[STA98] R. Spelberg, H. Toetenel, and M. Ammerlaan. Partition refinement in real-time model checking. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Lyngby, Denmark*, volume 1486 of *LNCS*. Springer-Verlag, 1998.

[SV96] J. Springintveld and F. Vaandrager. Minimizable timed automata. In B. Jonsson and J. Parrow, editors, *Proc. of the 4th International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'96)*, volume 1135 of *Lecture Notes in Computer Science*, pages 130–147, Uppsala, Sweden, 1996. Springer-Verlag.

[SY96] J. Sifakis and S. Yovine. Compositional specification of timed systems. In *13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, pages 347–359, Grenoble, France, February 1996. Lecture Notes in Computer Science 1046, Spinger-Verlag.

[TAKB96] S. Tasiran, R. Alur, R.P. Kurshan, and R. Brayton. Verifying abstractions of timed systems. In *CONCUR'96*, LNCS 1119, 1996.

[Tar55] A. Tarski. A lattice theoretical fix-point theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.

[Tar72] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–170, 1972.

[TB97] S. Tasiran and R.K. Brayton. STARI: A case study in compositional and hierarchical timing verification. In *CAV'97*, LNCS 1254, 1997.

[TC96] S. Tripakis and C. Courcoubetis. Extending promela and spin for real time. In *TACAS'96, Passau, Germany*, volume 1055 of *LNCS*. Springer-Verlag, 1996.

[Tra93] D.P. Tranchier. *Fast Reservation Protocol / DT : Multiplexage statistique dans les re'seaux ATM*. PhD thesis, Université de Rennes I, 1993.

[TY96] S. Tripakis and S. Yovine. Analysis of timed systems based on time–abstracting bisimulations. In *Proc. 8th Conference Computer-Aided Verification, CAV'96, Rutgers, NJ*, volume 1102 of *LNCS*, pages 232–243. Springer-Verlag, July 1996.

[TY98] S. Tripakis and S .Yovine. Verification of the Fast-Reservation Protocol with Delayed Transmission using the tool Kronos. In *4th IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[Č92] K. Čerāns. Decidability of bisimulation equivalence for parallel timer processes. In *Proceedings of the Fourth Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science, 1992.

[Val90] A. Valmari. Stubborn sets for reduced state space generation. LNCS 483, 1990.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. POPL*, pages 184–192, St. Petersburgh, January 1986.

[WT95] H. Wong-Toi. *Symbolic Approximations for Verifying Real-Time Systems.* PhD thesis, Stanford University, 1995.

[WTD94] H. Wong-Toi and D.L. Dill. Approximations for verifying timing properties. In *Theories and Experiences for Real-Time System Development.* World Scientific Publishing, 1994.

[Yi90] W. Yi. Real-time behavior of asynchronous agents. In *Concur'90*, LNCS 458, 1990.

[YL93] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real-time transition systems. In *Fifth Conference on Computer-Aided Verification*, LNCS 697, Elounda, Greece, june 1993.

[Yov93] S. Yovine. *Méthodes et outils pour la vérification symbolique de systèmes temporisés.* PhD thesis, Institut National Polytechnique de Grenoble, 1993. In french.

[Yov97] S. Yovine. KRONOS: a verification tool for real-time systems. *Software Tools for Technology Transfer*, 1997.

Appendix A

Higher-level modeling

A.1 Adding finite-domain variables to the timed-automata model

We show how the model of TA can be extended with discrete variables of finite domains without affecting the theoretical results. In fact, we can view this extension as pure "syntactic sugar" since the extended model can be translated to the basic TA model.

Let $Q = K_1 \times \cdots \times K_m$, where $K_1, ..., K_m$ are finite domains. The extended TA model (ETA) is similar to a *guarded-command* language. Each ETA has:

- a finite set of clocks $\mathcal{X}$,
- a set of variables $\{\kappa_1, ..., \kappa_m\}$, κ_i ranging in K_i, for $i = 1, ..., m$,
- an initial assignment of the variables $q_0 \in Q$,
- a finite set of *commands*: each command has the form $(\zeta \wedge f(\cdot)) \stackrel{a}{\Leftrightarrow} (X, g(\cdot))$, where:
  - $(\zeta \wedge f(\cdot))$ is the guard of the command, decomposed in a convex $\mathcal{X}$-polyhedron ζ (the clock guard) and a boolean function f on Q (the discrete guard);
  - a is the label of the command;
  - $(X, g(\cdot))$ is the assignment of the command, decomposed in a set of clocks to be reset X and a function $g : Q \mapsto Q$ (the discrete assignment).
- an invariant function invar associating to each $q \in Q$ a convex $\mathcal{X}$-polyhedron $\mathsf{invar}(q)$.

Notice that the control states are not explicit, however, they can be modeled by adding a special variable representing the "program counter".

Translating an ETA to a TA A is straightforward. The set of discrete states of A is Q and the initial state is q_0. The invariant of A is $\mathsf{invar}()$. For each $q \in Q$ and each command $(\zeta \wedge f(\cdot)) \stackrel{a}{\Leftrightarrow} (g(\cdot), X)$ such that $f(q) = \mathsf{true}$, A has an edge $(q, \zeta, a, X, g(q))$.

We have already used ETA informally in the case studies. Most of the times, the above translation to TA has not been necessary, since the new generation of KRONOS, `kronos-open`, works directly with ETA and builds the discrete state space on-the-fly.

A.2 Modeling atomic states

We show how to model atomic states using auxiliary variables. For simplicity, we consider TA extended with boolean variables. The latter can be encoded in the discrete structure of the TA as described in appendix A.1.

Informally, the semantics of atomic states for a network of TA are as follows:

1. while some automaton is in an atomic state, time stops;
2. when an automaton A enters an atomic state, it has to exit before any other automaton can take a discrete step.

Atomic states are sometimes useful for compact specifications. In particular, they correspond to the "committed locations" of UPPAAL, and have been used in modeling the BANG&OLUFSEN protocol (section 12.3). Since KRONOS does not support atomic states directly, we have used the modeling method described below.

We introduce an auxiliary global boolean variable *atom* and an auxiliary global clock z (a single boolean variable and a single clock suffice, no matter how many the atomic states are). The invariant that must hold during execution is that *atom* is set iff some automaton is in an atomic state and that the time spend in atomic states is zero.

For each automaton A in the global system, if $e = (q, \_, \_, q')$ is an edge of A, then:

- If q is not atomic, then we add the boolean guard $\neg atom$ to e.
- If q is atomic, then we add the clock guard $z = 0$ to e.
- If q' is atomic, then we add the assignment $atom :=$ true and the clock reset $z := 0$ to e.
- If q' is not atomic, then we add the assignment $atom :=$ false to e.

The construction is illustrated in figure A.1.

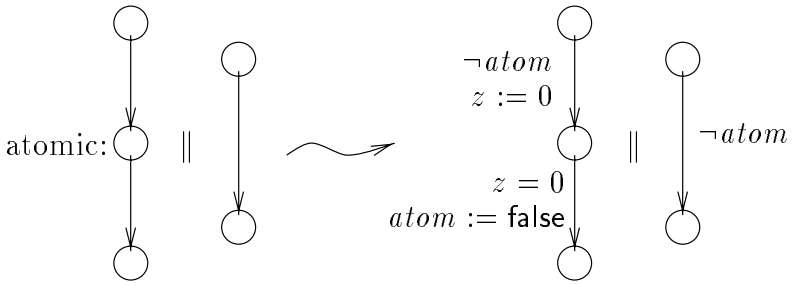


Figure A.1: Modeling atomic states with an auxiliary boolean variable and clock.

A.3 Petri Nets with Deadlines

An *1-safe Petri net* is a pair $(\mathcal{P}, \mathcal{P}_0, \mathcal{T})$ where

- $\mathcal{P}$ is a finite set of *places*. A subset of $\mathcal{P}$ is called a *marking*. $\mathcal{P}_0 \subseteq \mathcal{P}$ is the initial marking.

- $\mathcal{T} \subseteq 2^{\mathcal{P}} \times 2^{\mathcal{P}}$ is a finite set of *transitions*.

Adopting standard Petri-net terminology, given a marking $\mathcal{P}'$, we say that there is a *token* in place q when $q \in \mathcal{P}'$. Places can be viewed as local states of processes. Given $(\mathcal{P}_1, \mathcal{P}_2) \in \mathcal{T}$, places in $\mathcal{P}_1$ and $\mathcal{P}_2$ are the *input* and *output* places of the transition. We say that the transition *consumes* tokens from each place in $\mathcal{P}_1$ and *produces* tokens in each place in $\mathcal{P}_2$. A transition with more than one input places represents a synchronization of several processes.

A *Petri Net with Deadlines* (PND) [BST98] consists of:

- An 1-safe Petri net $(\mathcal{P}, \mathcal{P}_0, \mathcal{T})$.
- A finite set of clocks $\mathcal{X}$.
- A function h mapping each transition $(\mathcal{P}_1, \mathcal{P}_2) \in \mathcal{T}$ into a tuple (ζ, a, X), where ζ is a convex $\mathcal{X}$-polyhedron, a is a label and X is a subset of $\mathcal{X}$.
- A function invar mapping each place into a convex $\mathcal{X}$-polyhedron.

The above PND defines a TA $(\mathcal{X}, 2^{\mathcal{P}}, \mathcal{P}_0, E, \mathsf{invar}')$, where:

- $E = \{(\mathcal{P}_1, \zeta, a, X, \mathcal{P}_2) \mid (\mathcal{P}_1, \mathcal{P}_2) \in \mathcal{T} \wedge h(\mathcal{P}_1, \mathcal{P}_2) = (\zeta, a, X)\}$;
- $\mathsf{invar}'(\mathcal{P}_1) \bigcap_{q \in \mathcal{P}_1} \mathsf{invar}(q)$.

In other words, the discrete structure of the TA corresponds to the marking graph of the PND: the discrete states of the TA are PND markings and its edges are PND transitions. As for the invariant, intuitively, the definition above makes sure that time can pass in a marking iff it can pass in every place in the marking.

Appendix B

Proofs

Proof of lemma 5.9.

Let $\rho = s_0 \xrightarrow{\delta_0} s'_0 \xrightarrow{e_1} s_1 \cdots$ be a run of A. By definition, $s'_0 \in \mathsf{time\text{-}succ}(\{s_0\})$. For $i = 1, 2, ...$, define $S_i = \mathsf{post}(e_i, S_{i-1}, c)$, where $S_0 = \mathsf{time\text{-}succ}(\{s_0\}$. By definition, for all $i = 1, 2, ...$, $s_i \in \mathsf{disc\text{-}succ}(e_i, S_{i-1})$ and $s'_i \in \mathsf{time\text{-}succ}(\mathsf{disc\text{-}succ}(e_i, S_{i-1}))$, thus, $s'_i \in \mathsf{post}(e_i, S_{i-1}, c)$ (since $\mathsf{close}(S, c) \supseteq S$). Then, $\pi = S_0 \xrightarrow{e_1} S_1 \cdots$ is a valid zone path, since no S_i is empty. If π is zeno then ρ is zeno (the argument is similar to the one in the proof of lemma 5.5).

Inversely, let π be a zone path. For simplicity, we assume that π is *ultimately periodic*, that is, $\pi = \pi'(\lambda)^\omega$, where λ is a cycle (this case covers also finite paths). Let $\lambda = S_1 \xrightarrow{e_1} \cdots S_l \xrightarrow{e_l} S_1$. Choose some $s_l \in S_l$ and let C_l be the region of s_l. By the post-stability property, there exist $s'_l \in S_l, s_{l-1} \in S_{l-1}$ such that $s_{l-1} \xrightarrow{e_l} \xrightarrow{\delta_l} s'_l$ and s_l, s'_l are c-equivalent. Observe that c-equivalence implies region-equivalence, thus, $s'_l \in C_l$ and $C_{l-1} \xrightarrow{e_l} C'_l \xrightarrow{\tau} C_l$, where C'_l is some region and C_{l-1} is the region of s_{l-1}. We can continue backwards along the cycle in the same way, finding predecessor regions $C'_{l-1}, C_{l-2}, ..., C_1, C'_l, C^1_l, ...$, and so on. Observe that C'_i, C_i have non-empty intersection with S_i. Since there is a finite number of regions, after a bounded number of iterations we encounter a region we have already seen before, that is, we have found a cycle of regions. This defines an infinite path of regions, and we can use lemma 5.5 to extract a run ρ from it. By definition, ρ is inscribed in the original path π. Also, if π is non-zeno it is easy to verify that the region path is also non-zeno, therefore, ρ can be chosen non-zeno.

The proof can be extended to an infinite path π which is not ultimately periodic: since the simulation graph is finite, π must visit nodes in a SCC from some point on. We apply the backward-propagation technique above to the nodes of the SCC visited infinitely often by π. This yields a SCC of regions, thus, we can "map" π upon an infinite path of regions, from which a run is extracted as before.

Proof of lemma 6.4.

The proof is by induction on the syntax of ϕ. The basis (ϕ is an atomic proposition) comes from the fact that $\approx$ respects P. The case for $\phi_1 \vee \phi_2$ is trivial.

Consider the case where ϕ is of the form $\exists\, \phi_1 \mathcal{U} \phi_2$. Assume that $C' \in \mathsf{ctl\text{-}eval}(\phi)$, $C \in \mathsf{ctl\text{-}eval}(\phi_1)$ and $C \xrightarrow{\tau} C'$. Let $s \in C$. There exists δ such that $s \xrightarrow{\delta} s + \delta$ and $s + \delta \in C'$. By induction, $s + \delta$ satisfies ϕ and s satisfies ϕ_1. Now, for any $\delta' < \delta$, $s + \delta' \in C \cup C'$ (here we use the fact that C' are the immediate time successors of C). By lemma 5.7 and the fact that $s + \delta'$ is STa-bisimilar either to s or $s + \delta$, we have $s + \delta' \models \phi_1$ or $s + \delta' \models \phi$, thus, $s \models \phi$. The case $C \xrightarrow{e} C'$ is similar.

Now, consider the case where ϕ is of the form $\forall\, \phi_1 \mathcal{U} \phi_2$. Let $C \notin \mathsf{ctl\text{-}eval}(\phi)$. Since A is deadlock-free, there is an infinite path in G, $\pi = C \rightarrow C_1 \rightarrow \cdots$, and some i, such that $C_i \notin \mathsf{ctl\text{-}eval}(\phi_1)$ and for all $j < i$, $C_j \notin \mathsf{ctl\text{-}eval}(\phi_2)$. Also remark that π contains only a finite number of τ-transitions, since there are no τ-self-loops in G. Finally, π is non-zeno, since A is strongly non-zeno. Thus, by lemma 5.5, we can extract from π a non-zeno run which falsifies $\forall\, \phi_1 \mathcal{U} \phi_2$.

Proof of lemma 7.3.

Only one of the directions is non-trivial. Let λ be a non-zeno cycle.

`Repeat`: Is λ elementary ? If yes, we are done.

Otherwise, λ visits at least one node S twice, thus, can be divided in two sub-cycles λ_1 and λ_2 rooted at S. We distinguish two cases.

Case 1: there exist two clocks x and y such that x is reset in λ_1 but not in λ_2 and y is reset in λ_2 but not in λ_1. Let $\stackrel{e_1}{\rightarrow} S_1$ be the last edge before S where x is reset in λ_1. We have that $S_1 \subseteq (x \leq y)$, thus, from post-stability and the fact that y is not reset between S_1 and S, we obtain $S \subseteq (x \leq y)$. Reasoning symmetrically on λ_2, we obtain $S \subseteq (y \leq x)$ and from the two facts, we have that $S \subseteq (x = y)$.

Now, let ρ be a non-zeno run inscribed in λ. Whenever ρ passes through S_1, its valuation is such that $x = 0$. It should also be that $y = 0$, since when ρ passes through S it is $x = y$ and the difference between x and y is not changed between S_1 and S (none of the clocks is reset). Reasoning symmetrically on λ_2, we obtain that $x = y = 0$ each time ρ passes through S_2. Then, no time elapses from S_1 to S_2, since y is not reset anywhere between. No time elapses from S_2 to S_1 since x is not reset anywhere between. Thus, no time elapses at all along the run. We have a contradiction since ρ was assumed non-zeno.

Case 2: the negation of case 1. There must be a sub-cycle among λ_1, λ_2 which resets all clocks that are reset in λ. Without loss of generality, we assume that so does λ_1. Every clock not reset in λ remains unbounded in λ, thus, also in λ_1, so that the latter is non-zeno. Then, update λ to λ_1 and `goto Repeat`.

Since cycles are finite structures, the process cannot be repeated ad infinitum. It eventually terminates yielding an elementary cycle.

Proof of lemma 8.3.

We assume that all constraints are in normalized form $y \Leftrightarrow y' \leq c$, where $c \in \mathsf{Z} \cup \{\infty\}$. The proof is by contradiction, assuming that there is a constraint $y \Leftrightarrow y' \leq c$ which is strengthened infinitely often, that is, c decreases without lower bound. Since the operation which "destabilizes" the set of constraints is variable substitution (after each step the set of constraints is fixed), we can assume that $y \Leftrightarrow y' \leq c$ is a constraint of the first iteration, that is, $y, y' \in \{y_0, ..., y_{m-1}\}$.

In order for c to decrease, there should be a "path" of constraints $y \Leftrightarrow w_1 \leq c_0$, $w_1 \Leftrightarrow w_2 \leq c_1$, ..., $w_k \Leftrightarrow y' c_k$, where $w_1, ..., w_k \in Y$, such that $c_0 + \cdots + c_k < c$ (so that we get $y \Leftrightarrow y' = y \Leftrightarrow w_1 + w_1 \Leftrightarrow w_2 + \cdots + w_k \Leftrightarrow y' \leq c_0 + \cdots + c_k$). Now, there should be at least one constraint of the second iteration, say, $w_j \Leftrightarrow w_{j+1} \leq c_j$, which is also strengthened infinitely often. This is because after each step the set of constraints is fixed and there is a finite number of constraint paths like the one above.

Since w_j, w_{j+1} are variables of the second iteration and y' is a variable of the first iteration, there are two implicit constraints $y' \Leftrightarrow w_j \leq 0$ and $y' \Leftrightarrow w_{j+1} \leq 0$. From the latter we conclude that $c_{j+1} + \cdots + c_k \geq 0$, otherwise we would have $0 = y' \Leftrightarrow w_{j+1} + w_{j+1} \Leftrightarrow y' = y' \Leftrightarrow w_{j+1} + w_{j+1} \Leftrightarrow w_{j+2} + \cdots + w_k \Leftrightarrow y' \leq 0 + c_{j+1} + \cdots + c_k < 0$, and the set of constraints would be inconsistent. Thus, there is a constant c' such that $0 \leq c' \leq c_{j+1} + \cdots + c_k$ and $w_{j+1} \Leftrightarrow y' \leq c'$ from some point on in the fix-point computation. (Notice that c' cannot increase, since at each iteration the constraints are strengthened.)

Now, since $w_j \Leftrightarrow w_{j+1}$ is strengthened infinitely often, there exists some point in which $w_j \Leftrightarrow w_{j+1} \leq \Leftrightarrow(c' + 1)$. At this point, we have $0 = y' \Leftrightarrow w_j + w_j \Leftrightarrow w_{j+1} + w_{j+1} \Leftrightarrow y \leq 0 \Leftrightarrow (c' + 1) + c' = \Leftrightarrow 1$, which implies that the set of constraints is inconsistent, contradicting our initial assumption.

Proof of lemma 9.5.

First notice that for any visited node v, `Reach`(v) returns *yes* iff at that point in the execution of the algorithm, $v \in$ *Yes*.

For a node $v \notin \hat{V}$, we prove the following facts, by induction on the number of nodes:

1. If $v \in$ *Maybe* at the end of the algorithm then there exists no winning strategy from v.
2. If $v \in$ *Yes* then at the end of the algorithm *Strat* contains a winning strategy from v.

For fact 1, we shall prove that either there exists $v \xrightarrow{u} w$ such that $w \in$ *Maybe*, or for all $v \xrightarrow{c} v'$, $v' \in$ *Maybe*. The result follows from the induction hypothesis and lemma 9.3. Now, at the moment when v is first visited by `Reach`(v) and its successors are explored, either some uncontrollable successor of v is in *Maybe* (line 1) or no controllable successor of v is in *Yes* (line 2). Otherwise, v would be inserted in *Yes* (line 3). If, during the algorithm, some successor of v is moved from *Maybe* to *Yes*, so that the above condition ceases to hold, then v would be updated by procedure `UndoMaybe` (line 6 or 8).

For fact 2, observe that for a node v to be inserted in *Yes*, procedure `UndoMaybe` has to be called and this is done after having passed either line 4 or line 7, where a controllable edge from v is inserted in *Strat*. Moreover, all uncontrollable edges from v are inserted in *Strat* by `UndoMaybe` (line 5). Thus, by the induction hypothesis and lemma 9.3, *Strat* is a winning strategy.

Facts 1 and 2 settle the "if" and "only if" parts of the proof, respectively.

Index