Université Libre De Bruxelles
Faculté des Sciences
Département d'Informatique

From Timed Models to Timed Implementations

Thèse présentée en vue de
l'obtention du grade de
Docteur en Sciences

Martin De Wulf
Année Académique 2006-2007

From Timed Models to Timed Implementations

Martin De Wulf

Thèse réalisée sous la direction du Pr. Jean-François Raskin et présentée en vue de l'obtention du grade de Docteur en Sciences. Défense le 20 décembre 2006 devant le jury composé de :

- Madame le Pr. Véronique Bruyère (Université de Mons-Hainaut, Belgique)
- Monsieur le Pr. Jean Cardinal (ULB)
- Monsieur le Pr. Raymond Devillers (ULB, président)
- Monsieur le Pr. Kim G. Larsen (Université dAalborg, Danemark)
- Monsieur le Dr. Nicolas Markey (École Normale Supérieure de Cachan, France)
- Monsieur le Pr. Thierry Massart (ULB)
- Monsieur le Pr. Jean-François Raskin (ULB)

Université Libre De Bruxelles
Faculté des Sciences
Département d'Informatique

Résumé

L'informatique fait actuellement face à un grand défi: trouver de bonnes méthodes de conception pour les *systèmes embarqués*. Fondamentalement, un système embarqué est un ordinateur interagissant avec un processus physique. On en trouve, par exemple, dans les systèmes de freinage ou dans les centrales nucléaires. Ils sont difficiles à concevoir pour plusieurs raisons : tout d'abord, ce sont des *systèmes réactifs*, qui interagissent indéfiniment avec leur environnement. Ensuite, ils doivent satisfaire des *contraintes temps-réel* qui spécifient non seulement *comment* ils doivent répondre, mais aussi *quand*. Finalement, leur environnement est souvent *profondément continu*, présentant des dynamiques complexes. Les modèles formels de choix pour spécifier de tels systèmes sont les *automates temporisés et hybrides* pour lesquels les problèmes de *vérification* sont bien étudiés.

Dans la première partie de cette thèse, nous étudions une *méthode complète de conception*, incluant la vérification et la génération de code, pour les automates temporisés. Nous devons définir une nouvelle sémantique pour les automates temporisés, appelée la sémantique AASAP, qui préserve les propriétés de décidabilité pour la vérification et qui, dans le même temps, est implémentable. Notre notion d'implémentabilité est complètement nouvelle, et se base sur la simulation d'une sémantique qui est implémentable de manière évidente sur une plate-forme réelle. Nous avons créé des *outils* qui permettent l'analyse et la génération de code et en illustrons l'usage sur une étude de cas à propos du *protocole audio Philips*, un cas industriel bien connu.

Dans la seconde partie de cette thèse, nous étudions le problème de la *synthèse de contrôleur* pour un environnement spécifié par un automate hybride. Nous donnons une nouvelle solution pour des contrôleurs discrets disposant seulement d'une *information imparfaite* à propos de l'état du système. En résolvant ce problème, nous avons défini un algorithme, basé sur la monotonie de l'*opérateur calculant les prédecesseurs contrôlables*, qui trouve *efficacement* un contrôleur. L'utilisation de

cet algorithme est prometteuse, comme nous le montrons à travers une application à un problème classique : *le test d'universalité pour les automates finis.*

Abstract

Computer Science is currently facing a grand challenge : finding good design practices for *embedded systems*. Embedded systems are essentially computers interacting with some physical process. You could find one in a braking systems or in a nuclear power plant for example. They present several design difficulties : first they are *reactive systems*, interacting indefinitely with their environment. Second, they must satisfy *real-time constraints* specifying *when* they should respond, and not only *how*. Finally, their environment is often *deeply continuous*, presenting complex dynamics. The formal models of choice for specifying such systems are *timed and hybrid automata* for which *model checking* is pretty well studied.

In a first part of this thesis, we study *a complete design approach*, including verification and code generation, for *timed automata*. We have to define a new semantics for timed automata, the AASAP semantics, that preserves the decidability properties for model checking and at the same time is implementable. Our notion of implementability is completely novel, and relies on the simulation of a semantics that is obviously implementable on a real platform. We wrote *tools* for the analysis and code generation and exemplify them on a case study about the well-known *Philips Audio Control Protocol*.

In a second part of this thesis, we study the problem of *controller synthesis* for an environment specified as a hybrid automaton. We give a new solution for discrete controllers having only an *imperfect information* about the state of the system. In the process, we defined a new algorithm, based on the monotonicity of *the controllable predecessors operator*, for efficiently finding a controller and we show some promising applications on a classical problem : *the universality test for finite automata*.

Acknowledgments

First of all, I would like to thank my research advisor Jean-François Raskin. If I have had one good idea during this thesis, it was to come work with him. His astonishing drive, technical skills and farsighted views were totally crucial in this work. Thanks for all that, and the rest.

Then, I want to thank Laurent Doyen. If I have had one stroke of luck during this thesis, it was when Laurent began to work on the same project as me. His rigor, his efficiency and his sheer brilliance never stopped to impress me. I am sure he will lead a great scientific career and I am proud my name stays next to his on six joint papers.

I also would like to thank my two other coauthors, Nicolas Markey and Thomas A. Henzinger. Working with them has learned me a lot.

Thanks also to all the members of this thesis' committee : Véronique Bruyère, Jean Cardinal, Raymond Devillers, Kim G. Larsen, Nicolas Markey and Thierry Massart, with a special thank to Mr Devillers for his careful proofreading of a preliminary version of this thesis.

I will not forget the people of the Computer Science Department and the Verification Group at ULB, I have worked with, or not. In no particular order : Steve Kremer, Gilles Geeraerts, Bram de Wachter, Nicolas Maquet, Laurent Van Begin, Thierry Massart, Frédéric Servais and Pierre Ganty. It was great fun to speak with all of them and they also learned me a lot. A big thank to the secretaries too, and especially to Pascaline Browaeys who allowed me to forget nothing.

All the people supporting me in my personal life have to be thanked too:

I owe to my parents my education, lots of comfort and above all, much love. I will never stop to thank them for all the opportunities they gave me. Thanks also to Thibaut for being such a great big brother.

I would not dare to make a list of my friends, since there are too many, all important, but I really want to thank them all for making life so enjoyable.

Thanks also to the whole Piette-Martin-Bonnier smala. It is good to know you.

And finally, but that is the most important to me, I would like to thank Charlotte. She has been supporting me for those five years of thesis and seems to be willing to proceed. That makes me often think that life could not be better.

À mes futures jumelles
(Ça leur fera une belle jambe...)

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

A good computer scientist is a lazy one. That was one of the favorite aphorisms of my first programming teacher at the university. The idea is that you should try to let the computer work instead of you as much as possible.

In fact, you could see the whole history of computer sciences through this lens. First, there were mathematicians who did not want anymore to perform lengthy (hence error prone) computations by hand and invented machines to do it. Then, there were computer scientists getting bored of assembly programs that are easy to understand for the computer, but hard to write for them. They wanted to use a higher-level language, closer to the human one, and invented compilers and interpreters, programs translating those high-level languages to machine language. In the process, they found that they also could translate the same high-level program to many different assembly language, thus for many different types of machines, reducing further their work. A lot of effort has since been put in raising the abstraction level of the languages, giving us many "generations" of languages. As the implementation details are more and more ignored, what the computer scientists write is more a *formal specification* of what the program should do, than a program in the original sense. An objective of computer science is then to allow the programmer to only write *what* the computer should do rather than *how*. This has been achieved to a certain extent in classical programming, through for example functional programming, but there are still efficiency problems and in lots of domains this is still mostly a dream.

In this work, the people who need to simplify their work are the software designers for embedded systems. Embedded systems are those tiny computers found in a car braking system, a camera, a fridge, a nuclear power plant, a watch,

a electrocardiograph, a doll, a plane, and so forth. For various reasons, you do not want the designers of those systems to mess things up, but they are facing a lot of difficulties: first, they are dealing with *reactive systems*, that have a non stopping interaction with their environment. This is fundamentally different from the sequential computing tasks of early computer science. Second, their program should not only output the good answer to the inputs, but also do this at the right time, not too early, not too late. Your car braking system should for example slow down the car at most 10 milliseconds after you push the pedal. Those softwares are working under *real-time constraints*, and hence are called *real-time (or timed) systems*. Third, embedded systems are often controlling *deeply continuous environments*. The mix of discrete behavior of the computers and continuous evolutions of variables, like temperature, give rise to difficult analysis problems. Finally, from a more down-to-earth point of view, embedded systems often offer very few debugging mechanisms. A modem, for example, has only a few leds. In fact, currently, those problems make software, paradoxically, the most costly and least reliable part of embedded systems. Lots of people consider this as one of the most difficult problems in computer science nowadays [HS06]. Many related ways are currently explored to tackle the problem.

Model checkers The model-checking community wants to free our fellow computer scientists from the lengthy correction proofs of their designs: a model checker is a program verifying properties on formal specification of designs. A toy example would for example be a program that, given as input a directed graph, could tell if there is a path from a certain vertex to another. This could be for example very useful for a network planner. In practice, designers of model checkers try to find a trade-off between the expressiveness of the formal models and of the queries allowed (finite graph and reachability queries in our example), and the time and memory consumption. The main problem is that the model-checker has to examine all possible states of a model before deciding and that the number of states is often huge (if not infinite, as it is the case for timed systems): hence this problem is known as the *state explosion problem*. [AD94, HNSY92] Furthermore, there are so expressive models that no computer can answer any interesting question on it : this is the problem of *undecidability*. Model Checking, however a still young field, is pretty well studied and is more and more used in the industry [HJMS03, HLS99, cov06].

Code generation Once a designer has written a formal design, and maybe verified it through model checking, he would appreciate to just push a button to obtain the code of his application. Unfortunately, for real-time systems, the problem is not fully solved because of the holistic approach needed in this case. You have to take into account all parameters to find a good solution : hardware real-time specification, operating system scheduling, and correctness of the outputs. Some tools do generate code, like Simulink [Tew02], but the emphasis is more on performance optimization than on correctness, which seems to be putting the cart before the horse. Happily, lots of people are currently working on the correctness issues [AIK+03, HKSP03, AFM+02].

Program synthesis An even greater blessing could come down on embedded systems designers than code generation and model checking: it is *program synthesis*, also known as *controller synthesis*. In this approach, you simply specify the problem to be solved, and you ask a program to generate a strategy to enforce your objective: for example, keep a tank above a certain temperature and below boiling point, with measures precise only up to 10 degrees. The controller synthesis problem is often presented as a game. You simply set up the rules, which represents the constraints of the environment (the temperature cannot rise instantaneously, for example) and on the means offered to the controller (imprecise sensors, limited heating power) and then you search for a strategy for the controller. Compared to this approach, model-checking is a much easier problem, as it simply consists in verifying that a strategy is correct, and not generating a winning one.

Contributions In this thesis, we contribute to the three approaches: model checking, code generation and program synthesis. Our initial goal was to generate code from the probably most used formalism for real-time: timed automata [AD94]. It was already known that some specifications in those formalisms were simply not implementable. For example, you can easily specify an automaton making an infinite number of moves in a finite amount of time. No hardware could ever ensure such a behavior. We pinpointed some other implementability problems and slightly modified the meaning, in other words *the semantics*, of timed automata. Our new semantics, called the AASAP[1] semantics, has two interesting properties : first, you

[1]For Almost As Soon As Possible

can use model-checking on it and second, we proved, that it is implementable. More precisely, we proved that the properties you verified on the model can be preserved by a practical implementation, provided that a simple constraint, linking speed parameters of the model and of the implementation platform, is satisfied.

To prove the practicability of the approach, we then tackled a big case study. We verified the AASAP semantics of the Audio Control Protocol, an industrial example introduced by Vaandrager et al. [BPV94]. We had to cope with efficiency issues in the verification phase to be able to treat such an example. We also developed a first tool for generating code from timed automata specifications, and had to resolve a bunch of methodological questions in the process.

Finally, we looked more closely to program synthesis for environment specified as timed or hybrid automata. This lead us to a new solution for finding a strategy to a *game with imperfect information*. In this kind of games, one of the player has limited informations on the state of the game: for one state, he may see different observations and for one observation that he sees, there may be many different states corresponding. In the process we introduced a powerful improvement to the computation of strategy for games that allowed performance gain in solving classical problems like testing the universality of finite automata.

The main concern of all this work is the *robustness* of the specification or generated model: the correctness of a controller should not rely on too simplifying assumptions, like the *synchrony hypothesis*, that assumes that the computation times of the implementation can be ignored, or the *perfect information hypothesis*, that assumes that a controller can know at any moment the exact state of its environment. Those hypotheses can serve in a first time as interesting working assumptions, but should be *formally validated*.

1.2 Thesis Overview

The following chapters of this thesis are organized as follows:

Chapter 2 *Formalisms For Real-Time* This chapter presents the formal definitions of the main formalisms we will use throughout this thesis: untimed and timed transitions systems that we will use as our main semantics models, and timed and hybrid automata that will be our main objects of study. We

recall interesting properties of (un)decidability for those objects: first, the region construction and the decidability of the reachability question for timed automata, and second, semi-decision procedures for the analysis of hybrid automata, since the reachability question is unfortunately undecidable. We also discuss the use of the *synchronized product* as a modelling of control.

Chapter 3 AASAP *Semantics : Implementable Semantics for Timed Automata* We tackle here the implementability problem for timed automata. We show that it is really easy to write specifications, using timed automata, that could not be implemented, no matter how fast and precise the hardware we are provided. We do not discard the classical semantics for timed automata, as it is a very useful tool for reasoning but we propose a new semantics, the AASAP semantics that offers two interesting features. First, we can perform automatic verification of interesting properties and second the semantics has been conceived to be implementable. For proving formally the implementability, we give a semantics to timed automata that is as reasonably close to a real implementation as possible and we show that our AASAP semantics can simulate it, under easy to check constraints.

Chapter 4 *Practical Verification of the* AASAP *Semantics* In Chapter 4, we describe a tool implementing, with extensive improvements, the verification method sketched in the previous chapter. The whole process is illustrated through a case-study of a classical industrial case study: the Philips Audio Control Protocol.

Chapter 5 *Practical Real-Time Code Generation* Then we describe the process of correct-by-construction code generation from a specification given as a timed automaton. As an example, we implemented a tool generating code for the Lego Mindstorms platform. We also describe the methodological problems that had to be solved in the process.

Chapter 6 *Games of Imperfect Information* We then look more carefully at the generation of a controller from the specification of an environment under the constraint that the controller does not have a perfect information about the state of the environment. The only information available can be imprecise about the state of the controller. We show how to solve those games for

finite state problems and generalize it to the generation of discrete controller (each move is separated by the exact same amount of time) for environments specified as rectangular hybrid automata.

In the process of solving games of imperfect information, we found an efficient way of reducing the computing work needed. The improvement over classical techniques is based on the observation that if a player has a winning strategy from any state of a given set S, i.e. he *controls* this set, then he *controls* any subset of S. We show how our technique can be fruitfully applied to the classical problem of deciding the universality of a finite automata.

Chapter 7 *Conclusion*

1.3 Chronology

The material of chapter 3 has been presented at *the Seventh International Workshop on Hybrid Systems : Computation and Control (HSCC 2004)* [DDR04] and in a paper published in the journal *Formal Aspects of Computing* [DDR05a]. Some additional remarks are inspired by a paper published in the proceeding of the *FORMATS/FTRTFT 04 Conference* [DDMR04].

Chapters 4 and 5 presents results partially published at the *Formal Methods 2005* conference [DDR05b].

Finally, the bulk of chapter 6 is a paper published at the *the Ninth International Workshop on Hybrid Systems: Computation and Control(HSCC 06)*[DDR06b]. Some interesting applications presented in this chapter have been published at the Conference *CAV'06 - Computer-aided Verification, 2006*[DDR06a].

Chapter 2

Formalisms For Real-Time

In this chapter, we introduce the notion of *timed* and *untimed transition systems*, to represent the behavior of real-time systems. Those transition systems will be used as the basic tool for defining semantics in this thesis, as well for programs than for more abstract formalisms. It allows us to compare those semantics and check for example if one is a refinement, in some sense, of another. To formalize this notion of refinement, we use *simulation relations*. We then introduce the formalism of the *timed automaton*, which will be our main modeling tool, and its generalization: the *hybrid automaton*. We remind some useful decidability results in this framework. We end the chapter by a discussion on the use of *synchronized product* as a modeling of *control*.

2.1 The Basic Semantic Model: Timed Transition systems

In this thesis, we are interested in continuous real-time systems where the domain for time will be $\mathbb{R}^{\geq 0}$, the set $\{x \in \mathbb{R} \mid x \geq 0\}$ of the nonnegative real numbers. To define the semantics of such systems, we use the notion of timed transition systems, which are a very simple model for representing a set of states and transitions. The transitions of those transition systems are equipped with labels : a label in $\mathbb{R}^{\geq 0}$ corresponds to the passing of some time, a label in Σ corresponds to an instantaneous discrete transition.

Notation We will often need to partition a set into subsets. To express that the sets $S_1, S_2, \ldots, S_n$ constitute a partition of the set S, we write $S = S_1 \uplus S_2 \uplus \cdots \uplus S_n$. In other words, $S_1 \uplus S_2 \uplus \cdots \uplus S_n$ stands for the union of the sets $S_1, \ldots, S_n$ but

also implicitly states that the intersection of any pair of different sets is empty. On the contrary of classical partitions in mathematics, those partitions do not impose that a set S_i can not be empty. Furthermore, given a set of state S and a set $\mathrm{F} \subseteq S$, we denote by $\overline{\mathrm{F}}$ the complement of F, that is the set $S \backslash \mathrm{F}$.

Definition 2.1 (TTS- Timed Transition System)
A timed transition system $\mathcal{T}$ *is a tuple* $\langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *where:*

- S *is a set of states,*
- $\mathrm{E} \subseteq S$ *is the set of possible initial states,*
- $\mathrm{F} \subseteq S$ *is the set of final states (here generally considered as the* bad states*),*
- Σ *is a finite set of labels, also called an* alphabet*, always including the* silent label τ*,*
- $\rightarrow \subseteq S \times (\Sigma \uplus \mathbb{R}^{\geq 0}) \times S$ *is the transition relation.*

We often write $s \xrightarrow{\sigma} s'$ as a shortcut for $(s, \sigma, s') \in \rightarrow$. The non silent labels of the transition system are used to model the *synchronization* between several TTSs (in particular, the labels in $\mathbb{R}^{\geq 0}$ represents synchronization too: the time elapses for all TTS simultaneously). The special silent label τ is added to the alphabet of every TTS to denote transitions that do not allow synchronization with other TTSs.

We call a transition $s \xrightarrow{\sigma} s'$:

- *discrete* if σ belongs to $\Sigma \backslash \{\tau\}$;
- *timed* if σ belongs to $\mathbb{R}^{\geq 0}$;
- *silent* if the label is τ.

Definition 2.2 (Reachable States of a TTS)
A state $s \in S$ *of a* TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *is* reachable *if there exists a finite sequence* $s_0, s_1 \ldots s_n$ *of states such that* $s_0 \in \mathrm{E}$*,* $s_n = s$ *and for any* i*,* $0 \leq i < n$*, there exists* $\sigma \in \Sigma \cup \mathbb{R}^{\geq 0}$ *such that* $(s_i, \sigma, s_{i+1}) \in \rightarrow$*. The set of reachable states of* $\mathcal{T}$ *is noted* $\mathsf{Reach}(\mathcal{T})$*.*

We sometimes write $s \xrightarrow{\sigma_1} \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_k} s'$ as a shortcut for $\exists s_1, \ldots, s_{k+1} \in S$ such that $s_1 = s$, $s_{k+1} = s'$ and $s_i \xrightarrow{\sigma_i} s_{i+1}$ for $1 \leq i \leq k$.

A *run* of a TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ over a finite word $w = \sigma_1 \ldots \sigma_n \in (\Sigma \uplus \mathbb{R}^{\geq 0})^+$ is a sequence $r = \ell_0 \ell_1 \ldots \ell_n$ of states such that (1) $\ell_0 \in \mathrm{E}$ and (2) $(\ell_i, \sigma_{i+1}, \ell_{i+1}) \in \rightarrow$ for all $0 \leq i < n$. The run r is *accepting* iff $\ell_n \in \mathrm{F}$. The *language* $\mathsf{Lang}(\mathcal{T})$ accepted by $\mathcal{T}$ is the set of words $w \in \Sigma^*$ such that $\mathcal{T}$ has an accepting run over w[1].

In this thesis, the main problem that we will address is the *emptiness problem*, assuming that F is the set of bad states.

Definition 2.3 (Emptiness Problem for TTS)
A TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *is* empty *if and only if* $\mathsf{Reach}(\mathcal{T}) \cap \mathrm{F} = \varnothing$. *The* emptiness problem *for a* TTS $\mathcal{T}$ *asks if* $\mathcal{T}$ *is empty.*

Another interesting problem is the *safety problem*:

Definition 2.4 (Safety Problem for TTS)
The safety problem *for a* TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *asks if* $\mathsf{Reach}(\mathcal{T}) \subseteq \mathrm{F}$.

Those two problems are *dual* for TTS in the following sense:

Lemma 2.1
For any TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *we have that* $\mathsf{Reach}(\mathcal{T}) \subseteq \mathrm{F}$ *iff* $\mathsf{Reach}(\mathcal{T}) \cap \overline{\mathrm{F}} = \varnothing$.

In chapter 6, we will also need *untimed* systems:

Definition 2.5 (UTS- Untimed Transition System)
An untimed transition system is a TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *where* $\nexists (s, t, s') : t \in \mathbb{R}^{\geq 0}$.

If the set of states S is finite, such a transition system is classically called a (*nondeterministic*) *finite automaton*, NFA for short. A *deterministic* finite automaton, DFA for short, is an NFA $A = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ such that for all states $s \in S$ and all letters $\sigma \in \Sigma$, there exists at most one state $s' \in S$ such that $s \xrightarrow{\sigma} s'$.

[1] We will only be interested in languages in Chapter 6 , when we talk about universality of finite automata. This is why the definition of language does not include durations.

2.2 The Main Proof Tool: Simulation Relations

To check if a transition system is the refinement of another one, we use the notion of *simulation* [Mil80].

Definition 2.6 (Strong simulation relation for TTS)
Given two TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *and* $\mathcal{T}' = \langle S', \mathrm{E}', \mathrm{F}', \Sigma', \rightarrow' \rangle$*, we say that* $\mathcal{T}'$ strongly simulates $\mathcal{T}$*, noted* $\mathcal{T} \preceq \mathcal{T}'$*, if* $\Sigma = \Sigma'$ *and there exists a relation* $R \subseteq S \times S'$ *(called a* strong simulation relation *or a* simulation relation for $\mathcal{T} \preceq \mathcal{T}'$*) such that:*

- $\forall s \in \mathrm{E}, \exists s' \in \mathrm{E}' : (s, s') \in R$*;*
- $\forall (s, s') \in R : s \in \mathrm{F} \implies s' \in \mathrm{F}'$
- *for all* $(s_1, s_1') \in R$*, for all* $\sigma \in \Sigma \cup \mathbb{R}^{\geq 0}$*, for all* s_2 *such that* $(s_1, \sigma, s_2) \in \rightarrow$*, there exists* $s_2' \in S'$ *such that* $(s_1', \sigma, s_2') \in \rightarrow'$ *and* $(s_1', s_2') \in R$*.*

Definition 2.7 (Strong Bisimulation)
Two TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *and* $\mathcal{T}' = \langle S', \mathrm{E}', \mathrm{F}', \Sigma', \rightarrow' \rangle$ *are* strongly bisimilar *if there exists a relation* $R \subseteq S \times S'$ *such that* R *is a simulation relation for* $\mathcal{T} \preceq \mathcal{T}'$ *and* R^{-1} *is a simulation relation for* $\mathcal{T}' \preceq \mathcal{T}$*.* R *is then called a* strong bisimulation *relation between* $\mathcal{T}$ *and* $\mathcal{T}'$*.*

The relation $\mathcal{T} \preceq \mathcal{T}'$ demands that every transition, even silent, of $\mathcal{T}$ must be mimicked by $\mathcal{T}'$. This is often too strong. This is why one usually introduces stutter-closed transition relations and weak simulation relations.

Definition 2.8 (Stutter-Closed Transition Relation)
Given the TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$*, we define the corresponding* stutter-closed transition relation $\twoheadrightarrow \subseteq S \times (\Sigma \backslash \{\tau\} \cup \mathbb{R}^{\geq 0}) \times S$ *as follows:*

- *if* $\sigma \in \Sigma \backslash \{\tau\}$ *then* $s \overset{\sigma}{\twoheadrightarrow} s'$ *iff* $s \overset{\tau}{\rightarrow} \cdots \overset{\tau}{\rightarrow} \overset{\sigma}{\rightarrow} \overset{\tau}{\rightarrow} \cdots \overset{\tau}{\rightarrow} s'$ *(there can be an arbitrary number of* τ*-labelled transitions before and after* σ*);*
- *if* $t \in \mathbb{R}^{\geq 0}$ *then* $s \overset{t}{\twoheadrightarrow} s'$ *iff there exists a finite sequence* $t_0, \ldots, t_k \in \mathbb{R}^{\geq 0}$ *such that* $s_0 \overset{t_0}{\rightarrow} \overset{\tau}{\rightarrow} \overset{t_1}{\rightarrow} \overset{\tau}{\rightarrow} \cdots \overset{t_k}{\rightarrow} s'$ *and* $\sum_{i=0}^{k} t_i = t$ *.*

Definition 2.9 (Weak simulation relation for TTS)
Given two TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *and* $\mathcal{T}' = \langle S', \mathrm{E}', \mathrm{F}', \Sigma', \rightarrow' \rangle$ *, we say that* $\mathcal{T}'$ weakly simulates $\mathcal{T}$*, noted* $\mathcal{T} \preceq_w \mathcal{T}'$*, if* $\Sigma = \Sigma'$ *and there exists a relation* $R \subseteq S \times S'$ *(called a* weak simulation relation *or a* simulation relation for $\mathcal{T} \preceq_{\mathrm{w}} \mathcal{T}'$*) such that:*

- $\forall s \in \mathrm{E}, \exists s' \in \mathrm{E}' : (s, s') \in R$*;*
- $\forall (s, s') \in R : s \in \mathrm{F} \implies s' \in \mathrm{F}'$
- *for all* $(s_1, s'_1) \in R$*, for all* $\sigma \in (\Sigma \backslash \{\tau\}) \cup \mathbb{R}^{\geq 0}$*, for all* s_2 *such that* $(s_1, \sigma, s_2) \in \twoheadrightarrow$*, there exists* $s'_2 \in S'$ *such that* $(s'_1, \sigma, s'_2) \in \twoheadrightarrow'$ *and* $(s'_1, s'_2) \in R$*.*

Definition 2.10 (Weak Bisimulation)
Two TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *and* $\mathcal{T}' = \langle S', \mathrm{E}', \mathrm{F}', \Sigma, \rightarrow' \rangle$ *are* weakly bisimilar *if there exists a relation* $R \subseteq S \times S'$ *such that* R *is a simulation relation for* $\mathcal{T} \preceq_w \mathcal{T}'$ *and* R^{-1} *is a simulation relation for* $\mathcal{T}' \preceq_w \mathcal{T}$*.* R *is then called a* weak bisimulation *relation.*

Definition 2.11 (Mutual Simulation)
Two TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *and* $\mathcal{T}' = \langle S', \mathrm{E}', \mathrm{F}', \Sigma', \rightarrow' \rangle$ *are mutually similar if there exists a relation* $R \subseteq S \times S'$ *such that* R *is a simulation relation for* $\mathcal{T} \preceq \mathcal{T}'$ *and a relation* $R' \subseteq S' \times S$ *such that* R' *is a simulation relation for* $\mathcal{T}' \preceq \mathcal{T}$*.*

Bisimulation is a stronger notion than mutual simulation. Indeed, bisimilarity of two TTS implies mutual similarity, but the opposite is not true. We will not need in this work the notion of weak mutual simulation and we thus do not define it.

Simulation can be used to define a notion of refinement. We say that the TTS $\mathcal{T}_1$ *refines* the TTS $\mathcal{T}_2$, if $\mathcal{T}_1 \preceq_{\mathrm{w}} \mathcal{T}_2$. In the following, we use simulation relations because they preserve safety properties [AL91], but they also preserve stronger properties such as the ones expressed in the logics LTL [Pnu77] or ACTL [CBG88].

Another important notion is *hiding*. It is used when one TTS $\mathcal{T}'$ could simulate another one, $\mathcal{T}$, if it did not insert some of its own labels in the sequences. In this case, we replace the labels that appear only in $\mathcal{T}'$ by *silent* transition and we prove the existence of a simulation relation between the obtained TTS and $\mathcal{T}$.

Definition 2.12 (Hiding)
The hiding *of a set of label* Σ'' *in a* TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$*, is a new* TTS $\langle S', \mathrm{E}', \mathrm{F}', \Sigma', \rightarrow' \rangle$*, denoted* $\mathcal{T}[\Sigma'' := \tau]$ *where:*

- $S' = S$;
- $\mathrm{E}' = \mathrm{E}$;
- $\mathrm{F}' = \mathrm{F}$;
- $\Sigma' = \Sigma \backslash \Sigma''$;
- $\rightarrow' = \{(s, \tau, s') \mid (s, \sigma, s') \in \rightarrow \wedge \sigma \in \Sigma''\} \cup \{(s, \sigma, s') \mid (s, \sigma, s') \in \rightarrow \wedge \sigma \notin \Sigma''\}$

2.3 Timed Automata

A timed automaton can be viewed as a finite automaton equipped with clocks. Clocks are real-valued variables which value increases with time, with first derivative equal to one: they *count time*. Their value can be set to zero while traversing an edge of the automaton. Edges can also be decorated with constraints, called *guards*, like $x > 3$ for example, which specify the moments when the transition can be fired and by symbols of an alphabet, which allows discrete synchronizations with other automata. Finally, each vertex ℓ, usually called a *location*, of the automaton is associated with a constraint, called an *invariant*, which specifies when the automaton can be in ℓ.

2.3.1 Definitions and Semantics

In this thesis we will denote the truth value of a predicate as follows: $\top$ for *true*, and $\bot$ for *false*.

Definition 2.13 (Rectangular Predicate)
Let Var *be a finite set of real-valued variables, usually called* clocks*. A valuation for* Var *is a function* $v : \mathsf{Var} \rightarrow \mathbb{R}$*. We write* $[\mathsf{Var} \rightarrow \mathbb{R}]$ *for the set of all valuations for* Var*. Given a set of variables* Var*, a valuation* $v : \mathsf{Var} \rightarrow \mathbb{R}$ *and a subset* Var' *of* Var*,* $v_{|\mathsf{Var}'}$ *denotes the projection of the valuation on* Var'*, that is a valuation* $v' : \mathsf{Var}' \rightarrow \mathbb{R}$ *such that* $v(x) = v'(x)$ *for all* $x \in \mathsf{Var}'$*.*

A closed rectangular guard *over* Var *is a finite formula* φ_c *defined by the following grammar rule:*

$$\varphi_c ::= \bot \mid \top \mid x \leq a \mid x \geq a \mid x = a \mid \varphi_c \wedge \varphi_c$$

where $x \in \mathsf{Var}$ *and* $a \in \mathbb{Q}$. *An* open rectangular guard *over* Var *is a finite formula* φ_o *defined by the following grammar rule:*

$$\varphi_o ::= \bot \mid \top \mid x < a \mid x > a \mid \varphi_o \wedge \varphi_o$$

where $x \in \mathsf{Var}$ *and* $a \in \mathbb{Q}$.

We denote by $\mathsf{Rect}_c(\mathsf{Var})$ *(resp.* $\mathsf{Rect}_o(\mathsf{Var})$*) the class of closed (resp. open) rectangular predicates built using variables in* Var. *A* rectangular predicate *over* Var *is a formula* φ *defined by the grammar rule:*

$$\varphi ::= \varphi_c \mid \varphi_o \mid \varphi_c \wedge \varphi_o$$

where $\varphi_c \in \mathsf{Rect}_c(\mathsf{Var})$ *and* $\varphi_o \in \mathsf{Rect}_o(\mathsf{Var})$. *We denote by* $\mathsf{Rect}(\mathsf{Var})$ *the class of rectangular predicates over* Var.

For a rectangular predicate g, we denote by $\mathsf{vars}(g)$ the set of variables appearing in g.

Definition 2.14 ($\models$)
Given a valuation $v : \mathsf{Var} \to \mathbb{R}$ *and a predicate* $\varphi \in \mathsf{Rect}(\mathsf{Var})$, *we write* $v \models \varphi$ *and say that* v satisfies φ *if and only if (recursively):*

- $\varphi \equiv \top$,
- *or* $\varphi \equiv x \bowtie a$ *for* $\bowtie \in \{<, \leq, =, \geq, >\}$ *and* $v(x) \bowtie a$,
- *or* $\varphi \equiv \varphi_1 \wedge \varphi_2$ *and* $v \models \varphi_1$ *and* $v \models \varphi_2$.

We denote by $[\![\varphi]\!]$ the set of valuations that satisfy a rectangular predicate φ. Such a set is called a *rectangle*. We denote by $[\![\varphi]\!]_x$ the *projection* of a rectangle $[\![\varphi]\!]$ on a variable x.

Let $v : \mathsf{Var} \to \mathbb{R}$ be a valuation and $\mathsf{Var}' \subseteq \mathsf{Var}$, then $v[\mathsf{Var}' := 0]$ denotes the valuation v' such that $v'(e) = 0$ if $e \in \mathsf{Var}'$ and $v'(e) = v(e)$ if $e \notin \mathsf{Var}'$. In the sequel, we sometimes write $v[x := 0]$ instead of $v[\{x\} := 0]$. Let $v : \mathsf{Var} \to \mathbb{R}$

be a valuation, for any $t \in \mathbb{R}^{\geq 0}$, $v - t$ is a valuation such that for any $x \in \mathsf{Var}$, $(v-t)(x) = v(x) - t$. We define $v + t$ in a similar way. We extend this definition to valuations v in $[\mathsf{Var} \to \mathbb{R} \cup \{\bot\}]$ as follows: $(v+t)(x) = v(x) + t$, if $v(x) \in \mathbb{R}$ and $(v+t)(x) = \bot$, if $v(x) = \bot$.

In the following, we will sometimes handle a valuation $v : \mathsf{Var} \to \mathbb{R}$ as an element of R^n where $n = \#\mathsf{Var}$. For $v \in R^n$ and $x \in \mathsf{Var}$, we define $v_{|x}$ as the value of the component of v corresponding to x.

We are now equipped to define timed automata [AD94] and their *classical* semantics.

Definition 2.15 (Timed Automata)
A timed automaton *over a set of variables* Var *is a tuple*

$$\langle \mathsf{Loc}, \mathsf{Init}, \mathsf{Final}, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg} \rangle$$

where:

- Loc *is a finite set of locations, denoted by the letter* ℓ *with subscripts or superscripts when needed, representing the discrete states of the automaton.*
- $\mathsf{Init} \subseteq \mathsf{Loc}$ *is the set of initial locations. The automaton starts in a location* ℓ *of* Init *with all values of its clocks (variables in* Var*) equal to zero.* [2]
- $\mathsf{Final} \subseteq \mathsf{Loc}$ *is the set of final locations, corresponding to the error states of the automaton.*
- $\mathsf{Inv} : \mathsf{Loc} \to \mathsf{Rect}(\mathsf{Var})$ *is the invariant condition. The automaton can stay in location* ℓ *as long* $\mathsf{Inv}(\ell)$ *is satisfied by the current valuation of the variables. We require that the invariant of each initial location allows that all clocks are set to zero, to ensure the existence of an initial state.*
- Lab *is a finite alphabet of labels that are used on transition, mainly to allow synchronization between many automata.*
- $\mathsf{Edg} \subseteq \mathsf{Loc} \times \mathsf{Loc} \times \mathsf{Rect}(\mathsf{Var}) \times \mathsf{Lab} \times 2^{\mathsf{Var}}$ *is a finite set of edges. An edge* $(\ell, \ell', g, \sigma, R)$ *represents a discrete transition from location* ℓ *to location* ℓ' *with*

[2] More general initial condition are possible but this kind of initialization is immediately encodable in the two tools we use (UPPAAL [LPY97] and HYTECH [HHWT95b]).

guard g, *label* σ *and a subset* $R \subseteq \mathsf{Var}$ *of variables to be reset . The guard* g *is a rectangular predicate that must be satisfied by the current valuation for the transition to be fired.*

Definition 2.16 (Semantics of Timed automata)
Let $A = \langle \mathsf{Loc}, \mathsf{Init}, \mathsf{Final}, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg} \rangle$ *be a timed automaton over the set of variables* Var*. The semantics of* A *is the* TTS $[\![A]\!] = (S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow)$ *where:*

- $S = \{(\ell, v) \mid \ell \in \mathsf{Loc} \wedge v \in [\mathsf{Var} \rightarrow \mathbb{R}] \wedge v \models \mathsf{Inv}(\ell)\}$;
- $\mathrm{E} = \{(\ell, v) \mid \ell \in \mathsf{Init} \wedge v \in [\mathsf{Var} \rightarrow \mathbb{R}] \wedge v(x) = 0, \forall x \in \mathsf{Var}\}$;
- $\mathrm{F} = \{(\ell, v) \mid \ell \in \mathsf{Final} \wedge v \in [\mathsf{Var} \rightarrow \mathbb{R}]\}$;
- $\Sigma = \mathsf{Lab}$
- *the transition relation* $\rightarrow \subseteq S \times (\Sigma \cup \mathbb{R}^{\geq 0}) \times S$ *is defined as follows:*
  (a) *discrete transitions:* $((\ell, v), \sigma, (\ell', v')) \in \rightarrow$ *iff there exists an edge* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$ *such that* $v \models g$ *and* $v' = v[R := 0]$;
  (b) *continuous transitions:* $((\ell, v), t, (\ell', v')) \in \rightarrow$ *iff* $\ell = \ell'$, $t \in \mathbb{R}^{\geq 0}$, $\forall x \in \mathsf{Var} : v'(x) = v(x) + t$ *and* $\forall t' \in [0, t] : v + t' \models \mathsf{Inv}(\ell)$

2.3.2 Model Checking of Timed Automata

The fundamental theorem about timed automata is the following:
Theorem 2.1
The emptiness problem for timed automata is decidable. In other words, there exists an algorithm which, given a timed automaton A, can decide if $[\![A]\!]$ *is empty [AD94]. The problem is PSpace-Complete.*

Theorem 2.1 stems from a reduction of the emptiness problem for timed automata to the reachability problem on a finite graph. It is based on the so-called *region construction*.

For this construction, we first assume that all constants appearing in the automaton are integers. If this is not the case, it suffices to multiply each constant with the least common multiple of the denominators to obtain a timed automaton with the same language, except for scaling. Let $\lfloor x \rfloor$ denote the integer part of x (the greatest integer $k \leq x$), and $\langle x \rangle$ denote its fractional part: $x = \lfloor x \rfloor + \langle x \rangle$.

Definition 2.17 (Clock regions)
A clock region *is an equivalence class of the relation $\sim_m$ defined over the clock valuations in $\mathsf{Var} \to \mathbb{R}^{\geq 0}$. The variable m is an integer constant. In the following, m will be the greatest constant appearing in an automaton A. We have $v \sim_m w$ iff all the following conditions hold:*

- $\forall x \in \mathsf{Var} : \lfloor v(x) \rfloor = \lfloor w(x) \rfloor$, *or both* $v(x)$ *and* $w(x)$ *are strictly greater than* m.
- $\forall x, y \in \mathsf{Var}$ *such that* $v(x) < m$ *and* $w(x) < m$, *we have* $\langle v(x) \rangle \leq \langle v(y) \rangle$ *iff* $\langle w(x) \rangle \leq \langle w(y) \rangle$.
- $\forall x \in \mathsf{Var}$ *such that* $v(x) < m$, *we have* $\langle v(x) \rangle = 0$ *iff* $\langle w(x) \rangle = 0$. □

Figure 2.1 exhibits the set of clock regions for the valuations over two variables x and y for the relation $\sim_2$. In this figure, a region can be either the inside of a triangle, the crossing point between two lines, or a segment between two crossings.

We write $]v[$ for the clock region containing v. The region $]v[$ contains the valuations that agree with v on the integer part of the variables, which is needed to know which guard is satisfied, and on the ordering of their fractional part, which is needed to know which integer part will change first.

There is only a finite number of regions for $\sim_m$, that is $|\mathsf{Var}|! \cdot 4^{|\mathsf{Var}|} \cdot \Pi_{x \in \mathsf{Var}}(m+1)$ if all the constants in the constraints are integers [AM04]. This value is exponential in the number of clocks and has to be multiplied by the least common multiple of all the denominators if the constants are rational, but not all integers. In the following, m will be the maximal constant in the constraints of the considered automaton.

Definition 2.18 (Region graph)
Given the TTS $[\![A]\!] = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \to_A \rangle$ *of a timed automaton A, we define the corresponding* region graph $G = \langle C, \to_G \rangle$ *of A :*

- $C = \{(\ell,]v[) \mid (\ell, v) \in S\}$ *is the set of regions.*
- $\to_G \subseteq C \times C$ *such that* $((\ell,]v[), (\ell',]v'[)) \in \to_G$ *if and only iff* $(\ell, v) \to_A (\ell', v')$ *and* $(\ell,]v[) \neq (\ell',]v'[)$.

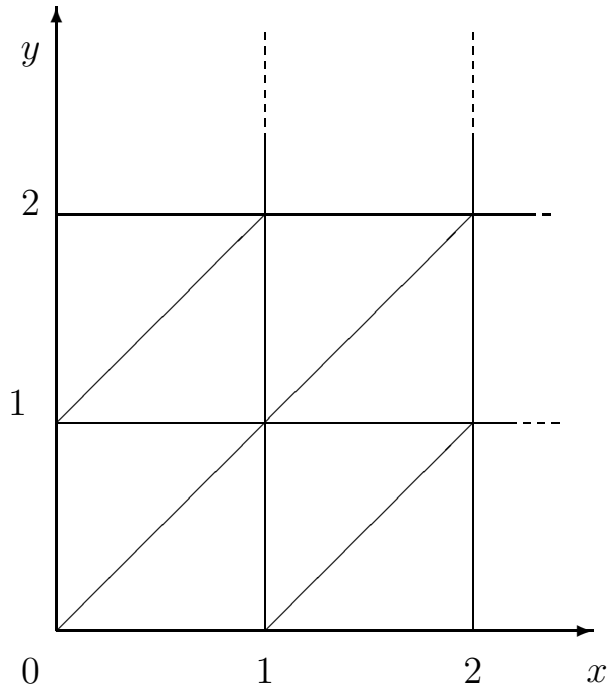

Figure 2.1: Clock regions in two dimensions.

C is finite and whenever $(\ell,]v[) \rightarrow_G (\ell',]v'[)$, for any $s \in]v[$ there exists $s' \in]v'[$ such that $(\ell, s) \rightarrow_A (\ell', s')$, and for any $s' \in]v'[$ there exists $s \in [v]$ such that $(\ell, s) \rightarrow_A (\ell', s')$ [AD94]. This property allows to reason on the finite region graph instead of the infinite TTS and is the base of the decidability result of Theorem 2.1. Indeed, the reachability problem for finite graphs, asking if there is a path from one vertex to another, requires linear time in the size of the graph.

A good summary of the classical results on timed automata is [Yov96]. A survey with more recent results can be found in [AM04].

2.4 Hybrid Automata

Hybrid automata are a natural extension of timed automata, where variables can have richer continuous behavior (hence they may correspond to something else than a clock, like a temperature) and updates of the variables are not limited to reset. They were first introduced in [MMP92].

The original definition of hybrid automata is very permissive, allowing the use of any type of differential equation to specify the behavior of continuous variables and complex update relation for the edges. In consequence, the reachability problem for such hybrid systems is undecidable. We will restrict ourselves in this thesis to a subclass having interesting decidability properties : the rectangular automata.

2.4.1 Definitions and Semantics

The main difference between timed automata and rectangular automata is that the first derivative of a variable is not fixed to 1 anymore, but must satisfy a rectangular predicate, potentially different in each location. This is what is specified through the Flow component of the rectangular automata. The other differences with timed automata are less crucial and could indeed have been introduced in timed automata while preserving the decidability of the emptiness question: first, initial and final condition are specified by rectangular predicates. Second, the value of a variable can be set non deterministically to any value satisfying a rectangular predicate, instead of only the value 0.

Definition 2.19 (Rectangular Automaton)
A rectangular automaton H *over a set of variables* Var *is a tuple*

$$\langle \mathsf{Loc}, \mathsf{Init}, \mathsf{Final}, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg}, \mathsf{Flow} \rangle$$

where:

- $\mathsf{Loc} = \{\ell_1, \ldots, \ell_m\}$ *is a finite set of* locations*;*
- $\mathsf{Init} : \mathsf{Loc} \rightarrow \mathsf{Rect}(\mathsf{Var})$ *gives the* initial condition $\mathsf{Init}(\ell)$ *of location* ℓ*. The automaton can start in* ℓ *with an initial valuation* v *lying in* $[\![\mathsf{Init}(\ell)]\!]$ *and we impose that* $\exists \ell \in \mathsf{Loc} : [\![\mathsf{Init}(\ell)]\!] \neq \varnothing$*;*
- $\mathsf{Final} : \mathsf{Loc} \rightarrow \mathsf{Rect}(\mathsf{Var})$ *gives the* final condition $\mathsf{Final}(\ell)$ *of location* ℓ*;*
- $\mathsf{Inv} : \mathsf{Loc} \rightarrow \mathsf{Rect}(\mathsf{Var})$ *gives the* invariant condition $\mathsf{Inv}(\ell)$ *of location* ℓ*. The automaton can stay in* ℓ *as long as the values of its variables lie in* $[\![\mathsf{Inv}(\ell)]\!]$*;*
- Lab *is a finite set of* labels*;*

- $\mathsf{Edg} \subseteq \mathsf{Loc} \times \mathsf{Loc} \times \mathsf{Rect}(\mathsf{Var}) \times \mathsf{Lab} \times \mathsf{Rect}(\mathsf{Var})$ *is a finite set of* edges $(\ell, \ell', g, \sigma, \mathsf{update})$. *An edge* $(\ell, \ell', g, \sigma, \mathsf{update})$ *represents a discrete transition from location* ℓ *to location* ℓ' *with* guard g, *label* σ *and a update* update *of variables to be reset . The guard* g *is a rectangular predicate that must be satisfied by the current valuation for the transition to be fired. Each variable* x *appearing in* update *is updated nondeterministically to an arbitrary new value in the interval* $[\![\mathsf{update}]\!]_x$, *while the other variables keep their value.*

- $\mathsf{Flow} : \mathsf{Loc} \rightarrow \mathsf{Rect}(\dot{\mathsf{Var}})$ *governs the evolution of the variables in each location.* $\dot{\mathsf{Var}}$ *is the set of dotted variables* $\dot{x}$ *such that* $x \in \mathsf{Var}$. *Those dotted variables denote the first derivative of the original ones.*

Definition 2.20 (Semantics of Rectangular Automata)
The semantics *of a rectangular automaton* $H = \langle \mathsf{Loc}, \mathsf{Init}, \mathsf{Final}, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg}, \mathsf{Flow} \rangle$ *is the* TTS $[\![H]\!] = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *where*

- $S \subseteq \mathsf{Loc} \times [\mathsf{Var} \rightarrow \mathbb{R}]$ *is the* state space, *the set of pairs* (ℓ, v) *such that* $v \models \mathsf{Inv}(\ell)$,
- $\mathrm{E} = \{(\ell, v) \in S \mid v \models \mathsf{Init}(\ell)\}$ *is the* initial space
- $\mathrm{F} = \{(\ell, v) \in S \mid v \models \mathsf{Final}(\ell)\}$ *is the* final space
- $\Sigma = \mathsf{Lab}$
- *and* $\rightarrow$ *contains all the tuples* $((\ell, v), \sigma, (\ell', v')) \in S \times (\mathsf{Lab} \uplus \mathbb{R}^{\geq 0}) \times S$ *such that*

  1. *for the discrete transitions:* $\sigma \in \mathsf{Lab}$ *and there exists* $e = (\ell, \ell', g, \sigma, \mathsf{update}) \in \mathsf{Edg}$ *such that* $v \models g$, $v' \models \mathsf{update}$ *and* $\forall x \in \mathsf{Var} : x \notin \mathsf{vars}(\mathsf{update}) \implies v(x) = v'(x)$
  2. *for the continuous transitions:* $\sigma \in \mathbb{R}^{\geq 0}$, $\ell = \ell'$ *and there exists* f *a continuously differentiable function* $f : [0, \sigma] \rightarrow [\![\mathsf{Inv}(\ell)]\!]$ *such that for all* $x \in Var$*:*
     - $f(0)_{|x} = v(x)$;
     - $f(\sigma)_{|x} = v'(x)$
     - *for all* $t \in [0, \sigma]$: $\dot{f}(t) \in [\![\mathsf{Flow}(\ell)]\!]$.

2.4.2 Model Checking of Rectangular Automata

The emptiness problem for general hybrid automata is, not surprisingly, undecidable. Unfortunately, even for the class of rectangular automata, the emptiness problem is undecidable.

Theorem 2.2 ([Hen96])
The emptiness problem for rectangular automata is undecidable. In other words, there does not exist an algorithm which, given any rectangular automaton H, can decide if $[\![H]\!]$ is empty.

Proof (Sketch)
The proof works by reduction of the state reachability problem *for two-counter machines [Min67]. A two-counter machine is essentially a finite automaton equipped with two unbounded counters. It can perform three operations:*

- *increment a counter;*
- *decrement a counter;*
- *test if a counter is equal to zero and branch accordingly to another state;*

The state reachability problem *for two-counter machines, that asks whether some state of the automata can be reached with a given value of the counters, is undecidable.*

It is very easy to reduce this problem to the emptiness of a rectangular automaton since the two counters can be mimicked by two variables for which the first derivative can be set, for one unit of time, to 1 *for incrementing, and to* -1 *for decrementing. Zero testing is also straightforward to express in the syntax of rectangular automata.*

Although the emptiness problem is undecidable in general for rectangular automata, there are still interesting questions that can be solved.

- First, there are subclasses of rectangular automata, besides timed automata, for which the emptiness problem is decidable. For example, the class of the *initialized* rectangular automata. In this class, on an edge going from location ℓ to ℓ', variables for which the flow condition change must be updated

to a value that is independent from the former value. More formally: in an initialized rectangular automaton $H = \langle \mathsf{Loc}, \mathsf{Init}, \mathsf{Final}, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg}, \mathsf{Flow} \rangle$ on a set of variables Var, for all $y \in \mathsf{Var}$, for all edge $e = (\ell, \ell', g, \sigma, \mathsf{update}) \in \mathsf{Edg}$, $[\![\mathsf{Flow}(\ell)]\!]_{\dot{y}} \neq [\![\mathsf{Flow}(\ell')]\!]_{\dot{y}}$ implies that $y \in \mathsf{vars}(\mathsf{update})$.

- Second, even if the emptiness of rectangular automata is undecidable in general, some tools like HyTech [HHWT95b] or PHAVer [Fre05] implements semi-algorithms that often terminate in practice. These procedures are based on a symbolic computation of the post operator: given a TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$, the *direct successor operator* $\mathsf{post}_{\mathcal{T}} : 2^S \rightarrow 2^S$ is an operator that, given a set of states, returns the set of direct successors of those states in $\mathcal{T}$. Formally, for any $S' \subseteq S$, we have that:

$$\mathsf{post}_{\mathcal{T}}(S') = \big\{ s \in S \mid \exists s' \in S' : (\exists \sigma \in \Sigma : s' \xrightarrow{\sigma} s \vee \exists t \in \mathbb{R}^{\geq 0} : s' \xrightarrow{t} s) \big\}.$$

This operator is computable for any TTS that is the semantics of a rectangular automaton [HPR94]. Furthermore, the set $\mathsf{Reach}(\mathcal{T})$ can be defined as the least solution of the equation:

$$X = \big(\mathrm{E} \cup \mathsf{Post}_{\mathcal{T}}(X)\big),$$

where X ranges over sets of states.

As this operator is monotone for the subset order, we know by Tarski's theorem that this solution can be computed by successive approximations[Tar55]. Unfortunately, since the emptiness problem is undecidable for rectangular automata, we know that the solution is not necessarily reached within a finite number of steps and thus that tools are not guaranteed to terminate, even if they do rather often.

2.5 Synchronized Product as a Modeling of Control

The labels present on the edges of hybrid automata are used to synchronize two automata. In the resulting *synchronized product* a transition on a label σ can be fired only if both automata have edges coming out of their current location, labeled with σ and with compatible guards and assignments.

Definition 2.21 (Synchronized Product of Rectangular Automata) *The* synchronized product *of two rectangular automata*

$$H_1 = \langle \mathsf{Loc}_1, \mathsf{Init}_1, \mathsf{Final}_1, \mathsf{Inv}_1, \mathsf{Lab}_1, \mathsf{Edg}_1, \mathsf{Flow}_1 \rangle$$

over Var_1 *and*

$$H_2 = \langle \mathsf{Loc}_2, \mathsf{Init}_2, \mathsf{Final}_2, \mathsf{Inv}_2, \mathsf{Lab}_2, \mathsf{Edg}_2, \mathsf{Flow}_2 \rangle$$

over Var_2 *is a third rectangular automaton*

$$H_1 \| H_2 = \langle \mathsf{Loc}, \mathsf{Init}, \mathsf{Final}, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg}, \mathsf{Flow} \rangle$$

over $\mathsf{Var}_1 \cup \mathsf{Var}_2$ *such that:*

- $\mathsf{Loc} = \mathsf{Loc}_1 \times \mathsf{Loc}_2$ *;*
- $\mathsf{Init}(\ell_1, \ell_2) = \mathsf{Init}_1(\ell_1) \wedge \mathsf{Init}_2(\ell_2)$ *for each* $(\ell_1, \ell_2) \in \mathsf{Loc}$*;*
- $\mathsf{Final}(\ell_1, \ell_2) = \mathsf{Final}_1(\ell_1) \vee \mathsf{Final}_2(\ell_2)$ *for each* $(\ell_1, \ell_2) \in \mathsf{Loc}$*;*
- $\mathsf{Inv}(\ell_1, \ell_2) = \mathsf{Inv}_1(\ell_1) \wedge \mathsf{Inv}_2(\ell_2)$ *for each* $(\ell_1, \ell_2) \in \mathsf{Loc}$*;*
- $\mathsf{Lab} = \mathsf{Lab}_1 \cup \mathsf{Lab}_2$*;*
- Edg *is the set of edges* $((\ell_1, \ell_2), (\ell_1', \ell_2'), g, \sigma, \mathsf{update})$ *such that*
  - *either* $\sigma \in (\mathsf{Lab}_1 \cap \mathsf{Lab}_2) \backslash \{\tau\}$ *and*
    - $\exists (\ell_1, \ell_1', g_1, \sigma, \mathsf{update}_1) \in \rightarrow_1$
    - $\exists (\ell_2, \ell_2', g_2, \sigma, \mathsf{update}_2) \in \rightarrow_2$
    - $g = g_1 \wedge g_2$
    - $\mathsf{update} = \mathsf{update}_1 \wedge \mathsf{update}_2$
  - *or* $(\sigma = \tau \vee \sigma \notin \mathsf{Lab}_2)$
    - $(\ell_1, \ell_1', g, \sigma, \mathsf{update}) \in \rightarrow_1$
    - $\ell_2 = \ell_2'$*;*
  - *or* $(\sigma = \tau \vee \sigma \notin \mathsf{Lab}_1)$
    - $\ell_1 = \ell_1'$

$* \ (\ell_2, \ell_2', g, \sigma, \mathsf{update}) \in \rightarrow_2$

- $\mathsf{Flow}(\ell_1, \ell_2) = \mathsf{Flow}_1(\ell_1) \wedge \mathsf{Flow}_2(\ell_2)$ *for each* $(\ell_1, \ell_2) \in \mathsf{Loc}$*;*

Observe that a very similar notion of product can be defined for timed automata.

Synchronized product is often used to model a notion of control.

Definition 2.22 (Safety Control for Hybrid Automata)
Given two hybrid automata Cont *and* Env *modeling on one hand a controller and on the other hand an environment, we say that* Cont controls Env *if* $[\![\mathsf{Cont} \| \mathsf{Env}]\!]$ *is empty, that is, the final (assumed to be bad) states of either the controller or the environment are not reachable. Usually the set of bad states of the controller is empty.*

From this definition of control emerge some problems, as we will see through the following example. Figure 2.2 presents an example of hybrid automaton. It models the evolution of the water level in a leaking tank alimented by a pump. The automaton, that we will call Env, uses two variables: y is the level, and x is a timer used for limiting the time spent changing mode. There are two modes: either the pump is on, and the level is increasing ($\dot{y}$ is in the interval $[0, 1]$), or the pump is off and the level is decreasing ($\dot{y}$ is in the interval $[-2, -1]$). There are two informations that a controller polls through actuators: either level has increased to 10 inches (H) or it has decreased to 5 inches (L). The controller must ensure that the level stays between 1 and 12 inches at every time by switching the pump on and off. The difficulty is that the switching takes up to 2 units of time. For this automaton $\mathsf{Init}(A) := x = 0 \wedge y = 2$ and $\mathsf{Init}(\ell) := \bot$ for $\ell \neq A$ (the arrow with no source indicates the initial location).

A controller, called Cont, is presented in Figure 2.3. Its strategy is very simple: every time the level reaches 10, switch the pump off, and every time it reaches 5, switch the pump back on. Observe that it is a timed automaton, which will often seem reasonable for a controller as it is implemented in hardware and has no variable with a complex continuous behavior, only timers. For this automaton $\mathsf{Init}(1) := z = 0$ and $\mathsf{Init}(\ell) := \bot$ for $\ell \neq A$.

The synchronized product of those two automata is presented in Figure 2.4. One can easily check that if the bad states of Cont are defined to be the states

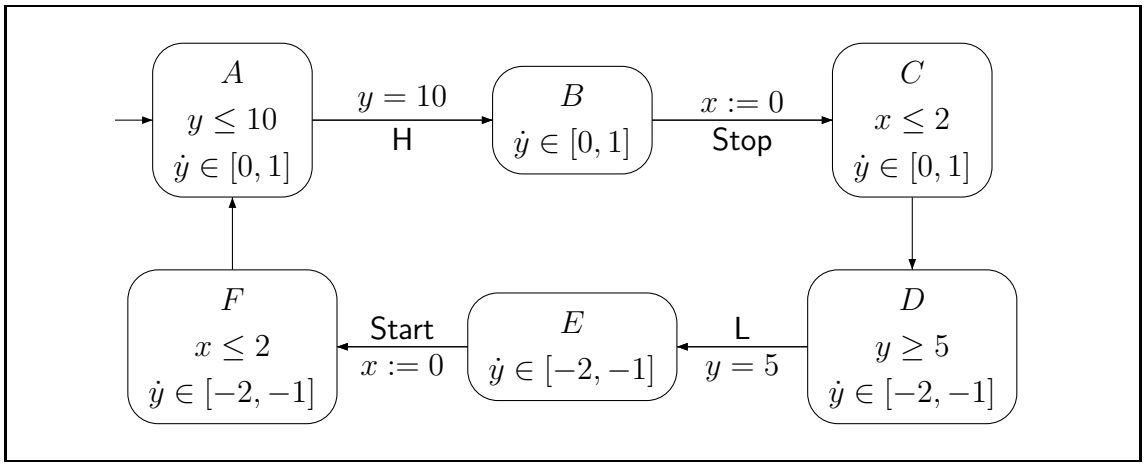

Figure 2.2: Hybrid systems modelling the evolution of the water level in a tank ($\dot{x} = 1$ in all locations)

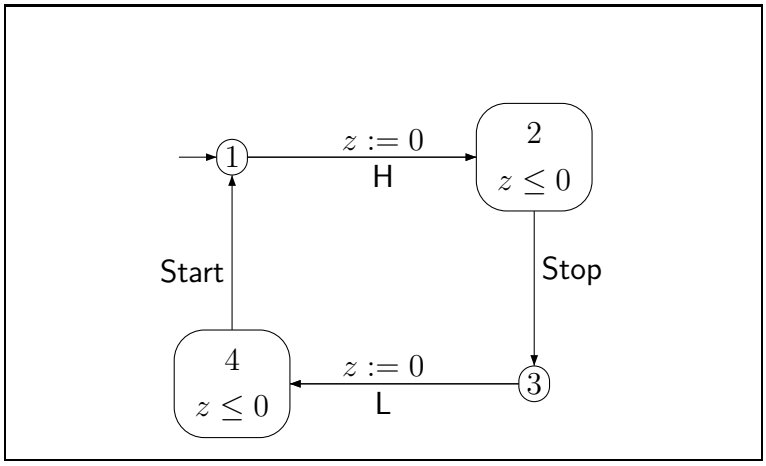

Figure 2.3: A controller for the tank of Figure 2.2 ($\dot{z} = 1$ in all locations)

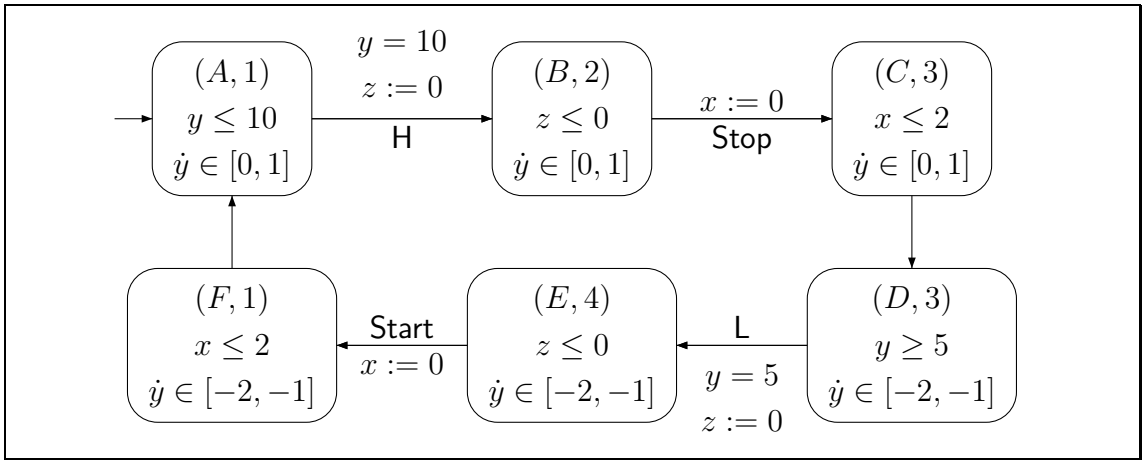

Figure 2.4: Synchronized product of automata of Figure 2.2 and Figure 2.3 ($\dot{x} = 1$ and $\dot{z} = 1$ in all locations)

where $y < 1$ or $y > 12$, we have that Cont controls Env, as explained in the previous section.

This notion of control is worth some comments:

- It would be meaningful to separate the inputs of one automaton from its output. For example, for Cont, H is clearly an input and Stop an output. One could then formalize the fact that a controller must not reach his goal by *blocking* the environment. In our example, if we give as controller an automaton that has all the labels of Env in its label set but offers no edge at all, it fulfills the definition of control by impeaching Env to ever leave location A where y is forced by the invariant to stay lower than 10. In this case the controller would reach his goal by *blocking time*, which is clearly not a realistic strategy. To avoid such problems, in the next chapter, we will force a controller to be *input enabled*, that is to always offer a transition on its input labels.

- In the above example, we clearly use invariants to force automata to move. Notably, in location 2 and 4 of Cont, we have the invariant $z \leq 0$ that allows no time to pass before an answer is given to the H or L events. Invariants are very often used to this modeling purpose : force progress. One other way to do this is to give an ASAP semantics to the automata: that is, semantics in which transitions have to be fired as soon as possible (ASAP). The use

of invariants is often preferred since it allows more flexibility. Furthermore controllers taking no time to react to inputs are not realistic, even if this *synchrony hypothesis* is often used to simplify the analysis.

- Observe also that no variable is shared between Env and Cont. This often seems natural in the examples we handle, where the controllers have to poll the environment to get information about its state. The problem of modelings where there is communication through shared variables was partially managed in [DW02]. In this work, we proposed a transformation that moved guards and invariants from an automaton to another, depending on the *membership* of the variables. We will not treat this problem further in this work and assume there is no variable shared by the environment and the controller.

- In this modeling of a tank, we assume that the tank emits an event *exactly* when some level has been reached. One could never assume such a precision in a real implementation. That is why we will introduce the notion of *imperfect information* in Chapter 6;

- One final interesting observation is that the correctness of Cont heavily depends on instantaneity of reaction to inputs. Any delay in its reactions would lead the environment in a bad state. In fact there is no *robust* controller for this environment. It would be interesting to be able to detect such a case.

In the following chapter, we mix different, albeit similar, formalisms for modeling controllers and environment. To formalize such a mix, we will compose directly the TTS semantic of the models.

Definition 2.23 (Synchronized Product of TTS)
The synchronized product *of two* TTS

$$\mathcal{T}_1 = \langle S_1, \mathrm{E}_1, \mathrm{F}_1, \Sigma_1, \rightarrow_1 \rangle$$

and

$$\mathcal{T}_2 = \langle S_2, \mathrm{E}_2, \mathrm{F}_2, \Sigma_2, \rightarrow_2 \rangle$$

is a third TTS

$$\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$$

such that:

- $S = S_1 \times S_2$;
- $\mathrm{E} = \{(s_1, s_2) \mid s_1 \in \mathrm{E}_1 \wedge s_2 \in \mathrm{E}_2\}$;
- $\mathrm{F} = \{(s_1, s_3) \mid s_1 \in \mathrm{F}_1 \vee s_2 \in \mathrm{F}_2\}$;
- $\Sigma = \Sigma_1 \cup \Sigma_2$;
- $\rightarrow$ *is such that for any* $\sigma \in \Sigma \cup \mathbb{R}^{\geq 0}$, *we have that* $((s_1, s_2), \sigma, (s_1', s_2')) \in \rightarrow$ *iff one of the following assertions holds:*
  - $\sigma \in (\Sigma_1 \backslash \Sigma_2) \cup \{\tau\}$ *and* $(s_1, \sigma, s_1') \in \rightarrow_1$ *and* $s_2 = s_2'$;
  - $\sigma \in (\Sigma_2 \backslash \Sigma_1) \cup \{\tau\}$ *and* $(s_2, \sigma, s_2') \in \rightarrow_2$ *and* $s_1 = s_1'$;
  - $\sigma \in \big((\Sigma_1 \cap \Sigma_2) \backslash \{\tau\}\big) \cup \mathbb{R}^{\geq 0}$ *and* $(s_1, \sigma, s_1') \in \rightarrow_1$ *and* $(s_2, \sigma, s_2') \in \rightarrow_2$

Definition 2.24 (Safety Control for TTS)
Given two TTS Cont *and* Env *modeling the semantics on one hand of a controller and on the other hand of an environment, we say that* Cont controls Env *if* Cont$\|$Env *is empty, that is, the final (assumed to be bad) states of either semantics are not reachable.*

Finally, it is interesting to state that the relation of controls remains true if one replace one of the TTS by another one, that is simulated by the original one.

Lemma 2.2 ([DDR04])
For all TTS $\mathcal{T}_1, \mathcal{T}_1', \mathcal{T}_2, \mathcal{T}_2'$, *if* $\mathcal{T}_1 \preceq_w \mathcal{T}_1'$, $\mathcal{T}_2 \preceq_w \mathcal{T}_2'$ *and* $\mathcal{T}_1'$ *controls* $\mathcal{T}_2'$ *then* $\mathcal{T}_1$ *controls* $\mathcal{T}_2$.

Proof (Sketch)
Let R_1 *and* R_2 *be simulation relations for respectively* $\mathcal{T}_1 \preceq_w \mathcal{T}_1'$ *and* $\mathcal{T}_2 \preceq_w \mathcal{T}_2'$. *It is easy to show that* $R_{12} = \left\{ \big((s_1, s_2), (s_1', s_2')\big) \mid (s_1, s_1') \in R_1 \text{ and } (s_2, s_2') \in R_2 \right\}$ *is a simulation relation for* $\mathcal{T}_1 \| \mathcal{T}_2 \preceq \mathcal{T}_1' \| \mathcal{T}_2'$.

2.6 Conclusion

In this chapter, we have introduced the main formalisms we will use throughout this thesis: timed transition systems, timed and rectangular automata. They are

designed to deal with *continuous* time behavior, where actions can occur at any instant in time, instead of *discrete* time behavior, where actions can occur only at integer point in time. There has been a lot of discussions in the past about the best way to model time. For example Abadi and Lamport advocate the idea that time can be modeled as any ordinary program variable in a system [AL94]. On the contrary, Henzinger underlines that time is fundamentally different from usual program variables since it is *continuous, monotonic and divergent* [Hen91]. Let us just say that this thesis clearly works on the continuous side, since we are looking for the *robustness* of our specification : assuming that the timing of a controller can be so precise that actions only happen at integer points in time does not seem reasonable if we consider the most general cases.

Chapter 3

AASAP Semantics : Implementable Semantics for Timed Automata

3.1 Introduction

As we have explained in the previous chapter, the model-checking problem for timed automata is well studied. Several tools, like UPPAAL [LPY97] or KRONOS [Yov97], allow the verification of models presented as timed automata. When a high level description of a controller has been proven *correct* it would be valuable to ensure that an implementation of that design can be obtained in a systematic way in order to ensure the *preservation of correctness*. This is often called program refinement: given a high-level description P_1 of a program, refine that description into another description P_2 such that the "important" properties of P_1 are maintained. Usually, P_2 is obtained from P_1 by reducing nondeterminism. To reason about the correctness of P_2 w.r.t. P_1, we will use the notion of simulation [Mil80] which is powerful enough to ensure the preservation of LTL properties for example.

We met several difficulties, however, to adapt this elegant schema in the context of timed automata.

First, the notion of time used by hybrid automata is based on a dense set of values (usually the real numbers). This is unarguably an interesting notion of time at the modeling level but when implemented, a digital controller manipulates timers that are digital clocks. Digital clocks have *finite granularity* and take their values in a discrete domain. Furthermore, those clocks may also be subject to drifts and so may not be perfectly accurate. As a consequence, any control strat-

egy that requires clocks with infinite precision can not be implemented. Second, hybrid automata can be called "*instantaneous devices*" in that they are able to instantaneously react to time-outs or incoming events by taking discrete transitions without any delay. Again, while this is a convenient way to see reactivity and synchronization at the modeling level, any control strategy that relies for its correctness on that instantaneity cannot be implemented by any physical device, no matter how fast it is. Those problems are known and have already attracted some attention from our research community. For example, it is well-known that timed automata may describe controllers that control their environment by playing a so called *zeno strategy*, that is, by taking an infinite number of actions in a finite amount of time. This is widely considered as unacceptable even by authors making the synchrony hypothesis [AFP+03]. But even if we prove our controller model non-zeno, that does not mean that it can be implemented. In fact, in [CHR02], Cassez, Henzinger and Raskin showed that there are (very simple) timed automata that enforce faster and faster reactions, say at times $0, \frac{1}{2}, 1, 1\frac{1}{4}, 2, 2\frac{1}{8}, 3, 3\frac{1}{16}, \ldots$. So, timed automata may model control strategies that cannot be implemented because the control strategy does not maintain a minimal bound between two control actions. A direct consequence is that we cannot hope to define for the entire class of timed automata (using the traditional semantics) a notion of refinement such that if a model of a real-time controller has been proven correct then it can be systematically implemented in a way that preserves its correctness.

The infinite precision and instantaneity characteristics of the traditional semantics given to timed automata is very closely related to the *synchrony hypothesis* that is commonly adopted in the community of synchronous languages [Ber00]. Roughly speaking, the synchrony hypothesis can be stated as follows: "*the program reacts to inputs of the environment by emitting outputs instantaneously*". The rationale behind the synchrony hypothesis is that the speed at which a digital controller reacts is usually so high w.r.t. the speed of the environment that the reaction time of the controller can be neglected and considered as null. This hypothesis *greatly simplifies* the work of the designer of an embedded controller: he/she does not have to take into account the performances of the platform on which the system will be implemented. We agree with this view at the modeling level. But, as any hypothesis, the synchrony hypothesis *should be validated*, not only by informal arguments but formally, if we want to transfer correctness prop-

erties from models to implementations. We show in this chapter how this can be done *formally* and *elegantly* using a semantics called the Almost ASAP semantics (AASAP-semantics).

The AASAP-semantics is a parametric semantics that leaves the *reaction time* of the controller as a parameter. This semantics relaxes the synchrony hypothesis in that it does not impose the controller to react instantaneously but imposes on the controller to react *within* Δ *time units* when a synchronization or a control action has to take place (is urgent). The designer acts as if the synchrony hypothesis was true, i.e. he/she models the environment and the controller strategy without referring to the reaction delay. This reaction delay is taken into account during the *verification phase*: we compute the largest Δ (or a Δ large enough) for which the controller is still correct w.r.t. to the properties that it has to enforce (to avoid the environment to enter bad states for example).

We show that the AASAP semantics has several important and interesting properties. First, the semantics is such that "faster is better". That is, if the controller is correct for a reaction delay bounded by Δ then it is correct for any smaller Δ'. Second, any controller which is correct for a reaction delay bounded by $\Delta > 0$ can be implemented by a program on a hardware provided that the hardware is *fast enough* and provides *sufficiently fine granular digital clocks*. Third, the semantics can be analyzed using existing tools, like HyTech.

3.2 Definitions

As explained in the example of the previous chapter, it will be meaningful to split the set of labels of an automaton into input and output, but also internal labels. This is why we introduce *structured set of labels*.

Definition 3.1 (Structured set of labels)
We say that a finite set of labels Lab *is* structured *if it is partitioned into three subsets:* $\mathsf{Lab}^{\mathsf{in}}$ *the set of input labels,* $\mathsf{Lab}^{\mathsf{out}}$ *the set of output labels, and* Lab^{τ} *the set of internal labels. Put differently :* $\mathsf{Lab} = \mathsf{Lab}^{\mathsf{in}} \uplus \mathsf{Lab}^{\mathsf{out}} \uplus \mathsf{Lab}^{\tau}$.

In the sequel, we use one TTS to model a timed controller and one to model the environment in which the controller is embedded. We model the communication between the two TTS using the mechanism of synchronization on common labels.

This is a blocking communication mechanism. Nevertheless, on one hand we want to verify that the controller does not control the environment by refusing to synchronize on its output, and on the other hand, we do not want our controller to issue outputs that can not be accepted by the environment. To avoid such problems we impose *input enabledness* of the TTS that we compose, which means that input labels have the property of being enabled in every state. *Input enabledness* is a concept introduced in [LT87]. Formally :

Definition 3.2 (Input Enabled TTS)
A TTS $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$, *where* $\Sigma = \Sigma^{\mathsf{in}} \uplus \Sigma^{\mathsf{out}} \uplus \Sigma^{\tau}$ *is a structured set of labels, is* input enabled *if for all* $\sigma \in \Sigma^{\mathsf{in}}$, *for all* $s_1 \in S$ *there exists* $s_2 \in S$ *such that* $(s_1, \sigma, s_2) \in \rightarrow$.

Lemma 3.1 (Input Enabledness is Preserved by Simulation)
For all TTS $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *on the same structured alphabet such that* $\mathcal{T}_1 \preceq \mathcal{T}_2$, *if* $\mathcal{T}_1$ *is input enabled, then* $\mathcal{T}_2$ *is input enabled.*

We chose this semantics for inputs because it clarifies the presentation, but other semantics are possible: for instance, in a preliminary version of this work [DDR04], we imposed *receptiveness* of controllers. Under this assumption, a controller must be fast enough to treat each occurrence of an event before the next occurrence arrives. One could also imagine a semantics where inputs arriving at the wrong time are simply ignored. Those aspects are orthogonal to the implementability aspects of the AASAP semantics.

Definition 3.3 (Input Compatible Synchronized Product of TTSs)
The synchronized product of two STTS

$$\mathcal{T}_1 = \langle S_1, \mathrm{E}_1, \mathrm{F}_1, \Sigma_1, \rightarrow_1 \rangle$$

and

$$\mathcal{T}_2 = \langle S_2, \mathrm{E}_2, \mathrm{F}_2, \Sigma_2, \rightarrow_2 \rangle$$

is input compatible *if* $\Sigma_1 = \Sigma_1^{\mathsf{in}} \uplus \Sigma_1^{\tau} \uplus \Sigma_1^{\mathsf{out}}$ *and* $\Sigma_2 = \Sigma_2^{\mathsf{in}} \uplus \Sigma_2^{\tau} \uplus \Sigma_2^{\mathsf{out}}$ *are structured set of labels such that* $\Sigma_1^{\mathsf{in}} = \Sigma_2^{\mathsf{out}}$ *and* $\Sigma_1^{\mathsf{out}} = \Sigma_2^{\mathsf{in}}$, *and if* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *are input-enabled.*

In the following of this chapter, we will always assume that the synchronized products we build are *input compatible*.

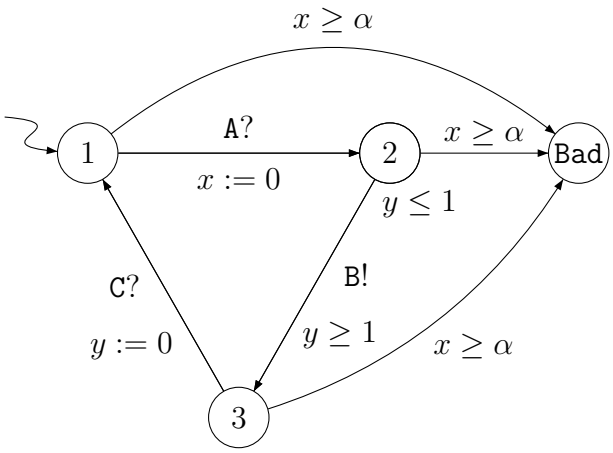

Figure 3.1: The environment of the running example.

Running example. As we already pointed out in the introduction and the example of the previous chapter, the classical semantics given in Definition 2.16 is problematic for the controller part if our goal is to transfer the properties verified on the model to an implementation. Below, we illustrate the properties of the classical semantics that makes it impossible to both implement the controller and ensure formally that the properties of the model are preserved.

The timed automaton of Figure 3.1 models a simple environment (a plant): when a request `A` is received, the response `B` is emitted when $y = 1$, and then the event `C` is accepted, which resets the clock y. A "!" corresponds to an output event and a "?" to an input event. Moreover, the event `A` *must* occur at least every 2 time units, and the reaction `C` should occur before the timeout condition $x \geq \alpha$ becomes true. If it is not the case, the environment enters the location `Bad`, modeling a fatal error. We want to control the environment for $\alpha = 1$ and $\alpha = 2$.

The role of the controller is to produce an event `A` at least every 2 time units, to accept the subsequent event `B` and to output `C` while respecting the timing constraint. An example of such a controller is given in Figure 3.2. The designer has chosen here to output an `A` every 1 time unit, and to react to the event `B` as quickly as possible by emitting a `C`. Given this controller for the system, we must verify that it gives orders in such a way that any resulting behavior of the

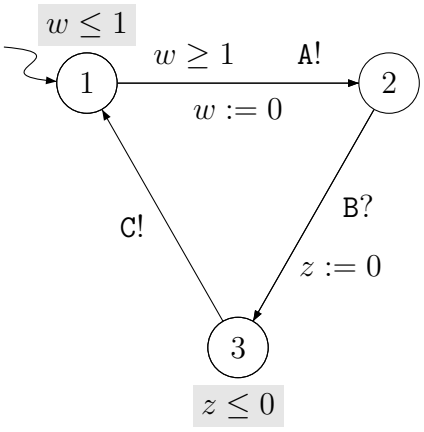

Figure 3.2: The ASAP controller of the running example.

environment avoids to enter the set of bad states (all the states in which the environment is in control location `Bad`). Observe that since the semantics of timed automata is *input enabled*, we are sure that the controller could not simply control the environment to avoid `Bad` by refusing to synchronize with `B`.

The reader can check that, with the classical semantics of timed automata, the controller controls the environment such that the location `Bad` is not reachable for $\alpha = 1$ and $\alpha = 2$. Later, we will see that if $\alpha = 1$ then the controller is not implementable. On the other hand, if $\alpha = 2$ then the controller can be implemented in such a way that it controls the environment to avoid `Bad`.

First, note that invariants (grayed constraints in the example of Figure 3.2 are used to force the controller to take actions. Invariants can be removed if we assume an ASAP semantics for the controller: any action is taken as soon as possible; this is also called the *maximal progress assumption*. So, in the example, the transition labeled with `A!` fires exactly when $w = 1$, and the transition labeled with `C!` proceeds exactly when $z = 0$, *i.e.* instantaneously. Clearly, no hardware can guarantee that the transition will always be taken without any delay. Second, synchronizations between the environment and the controller (*e.g.* transitions labeled `B`) cannot be implemented as instantaneous: some time is needed by the hardware to detect the incoming event `B` and for the software that implements the control strategy to

take this event into account. Third, the use of real-valued clocks is only possible in the model: implementations use digital clocks with finite granularity. It is then necessary to show that digital clocks can replace the real-valued clocks while preserving the verified safety properties.

These three problems illustrate that even if we have formally verified our control strategy, we cannot conclude that an implementation will preserve any of the properties that we have proven on the model. This is unfortunate. If we simplify, there are two options to get out of this situation: (*i*) we ask the designer to give up the synchrony hypothesis and ask her/him to model the platform on which the control strategy will be implemented, as in [IKL+00] and [AT05], or (*ii*) we let the designer go on with the synchrony hypothesis at the modeling level but relax the ASAP semantics during the verification phase in order to *formally validate the synchrony hypothesis*.

We think that the second option is *much more appealing* and we propose in the next section a framework that makes the second option possible *theoretically* but also feasible *practically*. The framework we propose is centered on a relaxation of the ASAP semantics that we call the AASAP semantics. The main characteristics of this semantics are summarized below:

- any transition that can be taken by the controller becomes urgent only after a small delay Δ (which may be left as a parameter);
- a distinction is made between the occurrence of an event in the environment (sending σ), and in the controller (receiving $\widetilde{\sigma}$); however the time difference between the two events is bounded by Δ;
- guards are enlarged by some small amount depending on Δ.

We will now define formally this semantics and we will show in section 3.3.2 that it is *robust* in the sense that it defines a *tube of strategies* (instead of a unique strategy as in the ASAP semantics) which can be refined in a formal way into an implementation while preserving the safety properties imposed by this tube of strategies.

As stated previously, invariants are useful when modeling controllers with the classical semantics in order to force the controller to take actions but they are useless with an ASAP semantics. This is also true for the semantics we define

in this section. So, we restrict our attention to the subclass of timed automata without invariants and with closed guards. In the rest of the paper, we call the controller specified by this subclass ELASTIC[1] controllers.

Definition 3.4 (ELASTIC Controllers)
An ELASTIC *controller* A *is a tuple* $\langle \mathsf{Loc}, \ell_0, \mathsf{Var}, \mathsf{Lab}, \mathsf{Edg} \rangle$ *where:*

- Loc *is a finite set of locations;*
- $\ell_0 \in \mathsf{Loc}$ *is the initial location;*
- $\mathsf{Var} = \{x_1, \ldots, x_n\}$ *is a finite set of clocks;*
- $\mathsf{Lab} = \mathsf{Lab}^{\mathsf{in}} \uplus \mathsf{Lab}^{\mathsf{out}} \uplus \mathsf{Lab}^{\tau}$ *is a finite structured alphabet of labels, partitioned into input labels* $\mathsf{Lab}^{\mathsf{in}}$*, output labels* $\mathsf{Lab}^{\mathsf{out}}$*, and internal labels* Lab^{τ}*;*
- Edg *is a set of edges of the form* $(\ell, \ell', g, \sigma, R)$ *where* $\ell, \ell' \in \mathsf{Loc}$ *are locations,* $\sigma \in \mathsf{Lab}$ *is a label,* $g \in \mathsf{Rect_c}(\mathsf{Var})$ *is a guard and* $R \subseteq \mathsf{Var}$ *is a set of clocks to be reset.*

Before defining the AASAP semantics we need some more notations:

Definition 3.5 (True Since)
We define the function "True Since", noted $\mathsf{TS} : [\mathsf{Var} \to \mathbb{R}^{\geq 0}] \times \mathsf{Rect_c}(\mathsf{Var}) \to \mathbb{R}^{\geq 0} \cup \{-\infty\}$*, as follows:*

$$\mathsf{TS}(v, g) = \begin{cases} t & \text{if } v \models g \wedge v - t \models g \wedge \forall t' > t : v - t' \not\models g \\ -\infty & \text{otherwise} \end{cases}$$

This definition is meaningful since $g \in \mathsf{Rect_c}(\mathsf{Var})$ defines a *closed* set, the first derivative of clocks is the constant 1 and rectangular predicates define convex sets.

Definition 3.6 (Guard Enlargement)
In the following, we will make the non-restrictive assumption that all rectangular constraint p *are in the form* $p \equiv a \sim x \sim b$ *where* $\sim$ *stands for* $<$ *or* $\leq$*,* x *is a clock and* $a, b \in \mathbb{Q} \uplus \{\infty, -\infty\}$*. Given* $\Delta_1, \Delta_2 \in \mathbb{Q}$*, the symbol* $\langle$ *standing either for* [

[1] ELASTIC stands for Event-based LAnguage for Simple TImed Controllers; we also give to those timed controllers a semantics which is elastic in a sense that will soon be made clear to the reader.

or (*and the symbol* ⟩ *standing either for*] *or*), *we define the notation* ${}_{\Delta_1}\langle p\rangle_{\Delta_2}$ *for the parametric rectangular constraint:*

$$a - \Delta_1 \sim'_1 x \sim'_2 b + \Delta_2$$

where $\sim'_1$ *stands either for* $\leq$ *if* $\langle$ *is* [, *or for* $<$ *if* $\langle$ *is* (, *and* $\sim'_2$ *is interpreted symmetrically.*

For example, let $p \equiv 2 \leq x \leq 5$, then ${}_{-\frac{1}{3}}(p]_{\Delta} \equiv 2 + \frac{1}{3} < x \leq 5 + \Delta$. The notation is naturally extended to rectangular predicates, in which we also assume that there is at most one rectangular constraint per variable x. For a rectangular constraint $p \equiv a \sim x \sim b$, we define $lb(p)$ to be the value a and $rb(p)$ to be the value of b (rb and lb stand for right bound and left bound respectively).

Definition 3.7 (Perception of Events)
In the following semantics, we split two different aspects of an event: first there is the occurrence and then there is the perception, the viewing, by a controller. To denote this link, for an event α *we use the notation* $\widetilde{\alpha}$ *for the perception, and we keep the label unchanged,* α*, for the occurrence. For a set of labels* Lab*, we denote by* $\widetilde{\mathsf{Lab}}$ *the same set of labels equipped with* $\tilde{}$*.*

We are now ready to define the AASAP semantics. Intuitions are given right after the definition.

Definition 3.8 (AASAP semantics)
Given an Elastic *controller*

$$A = \langle \mathsf{Loc}, \ell_0, \mathsf{Var}, \mathsf{Lab}, \mathsf{Edg} \rangle$$

and $\Delta \in \mathbb{Q}^{\geq 0}$*, the* AASAP *semantics of* A*, noted* $[\![A]\!]_{\Delta}^{\mathsf{AAsap}}$ *is the* TTS

$$\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$$

where:

(*A*1) S *is the set of tuples* (ℓ, v, I, d) *where* $\ell \in \mathsf{Loc}$*,* $v \in [\mathsf{Var} \rightarrow \mathbb{R}^{\geq 0}]$*,* $I \in [\Sigma^{\mathsf{in}} \rightarrow \mathbb{R}^{\geq 0} \cup \{\bot\}]$ *and* $d \in \mathbb{R}^{\geq 0}$*;*

(*A*2) $\mathrm{E} = \{(\ell_0, v, I, 0)\}$ *where* v *is such that for any* $x \in \mathsf{Var} : v(x) = 0$*, and* I *is such that for any* $\sigma \in \Sigma^{\mathsf{in}}$*,* $I(\sigma) = \bot$*;*

($A3$) $\mathrm{F} = \varnothing$;

($A4$) $\Sigma = \Sigma^{\mathsf{in}} \uplus \Sigma^{\mathsf{out}} \uplus \Sigma^{\tau} \uplus \mathbb{R}^{\geq 0}$ *with* $\Sigma^{\mathsf{in}} = \mathsf{Lab}^{\mathsf{in}}$, $\Sigma^{\mathsf{out}} = \mathsf{Lab}^{\mathsf{out}}$, *and* $\Sigma^{\tau} = \mathsf{Lab}^{\tau} \cup \widetilde{\mathsf{Lab}^{\mathsf{in}}} \cup \{\tau\}$;

($A5$) *The transition relation is defined as follows:*

– *for the discrete transitions, we distinguish five cases:*

($A5.1$) *let* $\sigma \in \mathsf{Lab}^{\mathsf{out}}$. *We have* $((\ell, v, I, d), \sigma, (\ell', v', I, 0)) \in \rightarrow$ *iff there exists* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$ *such that* $v \models_{\Delta} [g]_{\Delta}$ *and* $v' = v[R := 0]$;

($A5.2$) *let* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$. *We have* $((\ell, v, I, d), \sigma, (\ell, v, I', d)) \in \rightarrow$ *iff*

· *either* $I(\sigma) = \bot$ *and* $I' = I[\sigma := 0]$;

· *or* $I(\sigma) \neq \bot$ *and* $I' = I$.

($A5.3$) *let* $\widetilde{\sigma} \in \widetilde{\mathsf{Lab}^{\mathsf{in}}}$. *We have* $((\ell, v, I, d), \widetilde{\sigma}, (\ell', v', I', 0)) \in \rightarrow$ *iff there exists* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$, $v \models_{\Delta} [g]_{\Delta}$, $I(\sigma) \neq \bot$, $v' = v[R := 0]$ *and* $I' = I[\sigma := \bot]$;

($A5.4$) *let* $\sigma \in \mathsf{Lab}^{\tau} \backslash (\{\tau\} \cup \widetilde{\mathsf{Lab}^{\mathsf{in}}})$. *We have* $((\ell, v, I, d), \sigma, (\ell', v', I, 0)) \in \rightarrow$ *iff there exists* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$, $v \models_{\Delta} [g]_{\Delta}$, *and* $v' = v[R := 0]$;

($A5.5$) *let* $\sigma = \tau$. *We have for any* $(\ell, v, I, d) \in S$: $((\ell, v, I, d), \tau, (\ell, v, I, d)) \in \rightarrow$.

– *for the continuous transitions:*

($A5.6$) *for any* $t \in \mathbb{R}^{\geq 0}$, *we have* $((\ell, v, I, d), t, (\ell, v + t, I + t, d + t)) \in \rightarrow$ *iff the following two conditions are satisfied:*

· *for any edge* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$ *with* $\sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^{\tau}$, *we have that:*

$$\forall t' : 0 \leq t' \leq t : (d + t' \leq \Delta \vee \mathsf{TS}(v + t', g) \leq \Delta)$$

· *for any edge* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$ *with* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, *we have that:*

$$\forall t' : 0 \leq t' \leq t : (d + t' \leq \Delta \vee \mathsf{TS}(v + t', g) \leq \Delta \vee (I + t')(\sigma) \leq \Delta)$$

Comments on the AASAP semantics. Rule ($A1$) defines the states that are tuples of the form $\langle \ell, v, I, d \rangle$. The first two components, location ℓ and valuation v, are the same as in the classical semantics; I and d are new. The function I records, for each input event σ, the time elapsed since its oldest "untreated" occurrence.

The treatment of an event σ happens when a transition labelled $\widetilde{\sigma}$ is fired. Once this oldest occurrence is treated, the function returns $\perp$ for σ until a new occurrence of σ, forgetting about the σ's that happened between the oldest occurrence and the treatment. The time elapsed since the last location change in the controller is recorded by d. Rule $(A2)$, $(A3)$ and $(A4)$ are straightforward. Rules $(A5.1-6)$ require more explanations. Rule $(A5.1)$ defines when it is allowed for the controller to emit an output event. The only difference with the classical semantics is that we enlarge the guard by the parameter Δ. Rules $(A5.2)$ defines how inputs from the environment are received by the controller. The controller maintains, through the function I, a list of events that have occurred and are not treated yet. An input event σ can be received at any time, but only the age of the oldest untreated σ is stored in the I function. ℓ, v and d are unchanged at that point. Note that the rule $(A5.2)$ ensures input enabledness of the controller. Rule $(A5.3)$ defines when inputs are treated by the controller. An input σ is treated when a transition with an enlarged guard and labelled with $\widetilde{\sigma}$ is fired. Once σ has been treated, the value of $I(\sigma)$ goes back to $\perp$. Rule $(A5.4)$ is similar to $(A5.1)$. Rule $(A5.5)$ expresses that the τ event can always be emitted. Rule $(A5.6)$ specifies how much time can elapse. Intuitively, time can pass as long as no transition starting from the current location is *urgent*. A transition labeled with an output or an internal event is urgent in a location ℓ when the control has been in ℓ for more than Δ time units $(d + t' > \Delta)$ and the guard of the transition has been true for more than Δ time units $(\mathsf{TS}(v + t', g) > \Delta)$. A transition labeled with an input event σ is urgent in a location ℓ when the control has been in ℓ for more than Δ time units $(d + t' > \Delta)$, the guard of the transition has been true for more that Δ time units $(\mathsf{TS}(v + t', g) > \Delta)$ and the last untreated occurrence of σ event has been emitted by the environment at least Δ time units ago $(I(\sigma) + t' > \Delta)$ (we define $\perp$ to be smaller than any rational value). This notion of urgency parameterized by Δ is the main difference between the AASAP semantics and the usual ASAP semantics.

Problems formulation We now define three problems that can be formulated about the AASAP semantics of an ELASTIC controller.

Definition 3.9 (Parametric safety control problem)
Let Env *be a timed automaton and let* Cont *be an* ELASTIC *controller, the* parametric safety control problem *asks*

- **[Fixed]** *whether* $[\![\mathsf{Cont}]\!]_{\Delta}^{\mathsf{AAsap}} \parallel [\![\mathsf{Env}]\!]$ *is empty for a given fixed value of* Δ*;*
- **[Existence]** *whether there exists* $\Delta > 0$ *such that* $[\![\mathsf{Cont}]\!]_{\Delta}^{\mathsf{AAsap}} \parallel [\![\mathsf{Env}]\!]$ *is empty;*
- **[Maximization]** *to maximize* Δ *such that* $[\![\mathsf{Cont}]\!]_{\Delta}^{\mathsf{AAsap}} \parallel [\![\mathsf{Env}]\!]$ *is empty.*

As we will see later, the problem [Fixed] is useful when we know the characteristics of the hardware on which the control will be implemented, the problem [Existence] is useful to determine if the controller is implementable at all and the problem [maximization] is useful to determine what is the slowest hardware on which the controller can be implemented.

3.3 Properties of the AASAP Semantics

3.3.1 Faster is Better

We now state a first property of the AASAP semantics. The following theorem and corollary state formally the informal statement "faster is better", that is if an environment is controllable with an ELASTIC controller reacting within the bound Δ_1 then this environment is controllable by the same controller for any reaction time $\Delta_2 \leq \Delta_1$. This is clearly a desirable property.

Theorem 3.1
Let A *be an* ELASTIC *controller, for any* $\Delta_1, \Delta_2 \in \mathbb{Q}^{\geq 0}$ *such that* $\Delta_2 \leq \Delta_1$ *we have that* $[\![A]\!]_{\Delta_2}^{\mathsf{AAsap}} \preceq [\![A]\!]_{\Delta_1}^{\mathsf{AAsap}}$.

Proof
It is clear that the identity relation between the set of states of the two STTS $[\![A]\!]_{\Delta_2}^{\mathsf{AAsap}}$ *and* $[\![A]\!]_{\Delta_1}^{\mathsf{AAsap}}$ *is an appropriate simulation relation between them.*

Lemma 2.2 and Theorem 3.1 allow us to state the following corollary:

Corollary 3.1
Let E *be a timed automaton and* A *be an* ELASTIC *controller. For any* $\Delta_1, \Delta_2 \in \mathbb{Q}^{\geq 0}$*, such that* $\Delta_1 \geq \Delta_2$*, if* $[\![A]\!]_{\Delta_1}^{\mathsf{AAsap}}$ *controls* $[\![E]\!]$ *then* $[\![A]\!]_{\Delta_2}^{\mathsf{AAsap}}$ *controls* $[\![E]\!]$.

3.3.2 Implementability of the AASAP Semantics

In this section, we show that any ELASTIC controller which controls (with $\Delta > 0$) an environment E for a safety property modeled by a set of bad states B can be implemented provided there exists a hardware sufficiently fast and providing sufficiently fine granular digital clocks.

To establish this result, we proceed as follows. First, we define what we call the program semantics of an ELASTIC controller. The so-called program semantics can be seen as a formal semantics for the following procedure interpreting ELASTIC controllers. This procedure repeatedly executes what we call *execution rounds*. An execution round is defined as follows:

- first, the current time is read in the clock register of the CPU and stored in a variable, say `T`;
- the list of input events to treat is updated: the input sensors are checked for new events issued by the environment;
- guards of the edges of the current locations are evaluated with the value stored in `T`. If at least one guard evaluates to true then take nondeterministically one of the enabled transitions;
- the next round is started.

All we require from the hardware is to respect the following two requirements: (i) the clock register of the CPU is incremented every Δ_P time units and (ii) the time spent in one loop is bounded by a fixed value Δ_L. We choose this semantics for its simplicity and also because it is obviously implementable. There are more efficient ways to interpret ELASTIC controllers but as we only want to prove that the AASAP semantics is implementable in a way or another, this implementation semantics is good enough. In Chapter 5, we show how to use this semantics in the context of the LEGO MINDSTORMS™ platform.

This semantics is close to the one of PLC-automata introduced by Dierks [Die01]. The main difference is that we explicitly model the granularity of clocks.

We proceed now with the definition of the program semantics. This semantics manipulates digital clocks, so we need the following definition:

Definition 3.10 (Clock Rounding)
Let $T \in \mathbb{R}^{\geq 0}, \Delta \in \mathbb{Q}^{>0}$, $\lfloor T \rfloor_\Delta = \lfloor \frac{T}{\Delta} \rfloor \Delta$ *where* $\lfloor x \rfloor$ *is the greatest integer* k *such that* $k \leq x$. *Symmetrically,* $\lceil T \rceil_\Delta = \lceil \frac{T}{\Delta} \rceil \Delta$ *where* $\lceil x \rceil$ *is the smallest integer* k *such that* $k \geq x$.

Lemma 3.2 follows directly from the definition above.

Lemma 3.2
For any $T \in \mathbb{R}^{\geq 0}$, *any* $\Delta \in \mathbb{Q}^{>0}$, $T - \Delta < \lfloor T \rfloor_\Delta \leq T$ *and* $T \leq \lceil T \rceil_\Delta < T + \Delta$.

We are now ready to define the program semantics. Intuitions are given right after the definition.

Definition 3.11 (Program Semantics)
Let A *be an* ELASTIC *controller and* Δ_L, $\Delta_P \in \mathbb{Q}^{>0}$. *Let* $\Delta_S = \lceil \Delta_L + \Delta_P \rceil_{\Delta_P}$. *The* (Δ_L, Δ_P) *program semantics of* A, *noted* $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L,\Delta_P}$ *is the structured timed transition system* $\mathcal{T} = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *where:*

($P1$) S *is the set of tuples* (ℓ, r, T, I, u, d, f) *such that* $\ell \in \mathsf{Loc}$, r *is a function from* Var *into* $\mathbb{R}^{\geq 0}$, $T \in \mathbb{R}^{\geq 0}$, I *is a function from* $\mathsf{Lab}^{\mathsf{in}}$ *into* $\mathbb{R}^{\geq 0} \cup \{\bot\}$, $u \in \mathbb{R}^{\geq 0}$, $d \in \mathbb{R}^{\geq 0}$, *and* $f \in \{\top, \bot\}$;

($P2$) $\mathrm{E} = \{(\ell_0, r, 0, I, 0, 0, \bot)\}$ *where* r *is such that for any* $x \in \mathsf{Var}$, $r(x) = 0$, I *is such that for any* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, $I(\sigma) = \bot$;

($P3$) $\mathrm{F} = \varnothing$;

($P4$) $\Sigma = \Sigma^{\mathsf{in}} \uplus \Sigma^{\mathsf{out}} \uplus \Sigma^{\tau} \uplus \mathbb{R}^{\geq 0}$ *and* $\Sigma^{\mathsf{in}} = \mathsf{Lab}^{\mathsf{in}}$, $\Sigma^{\mathsf{out}} = \mathsf{Lab}^{\mathsf{out}}$, $\Sigma^{\tau} = \mathsf{Lab}^{\tau} \cup \widetilde{\mathsf{Lab}^{\mathsf{in}}} \cup \{\tau\}$;

($P5$) *the transition relation* $\rightarrow$ *is defined as follows:*

- *for the discrete transitions:*

($P5.1$) *let* $\sigma \in \mathsf{Lab}^{\mathsf{out}}$. $((\ell, r, T, I, u, d, \bot), \sigma, (\ell', r', T, I, u, 0, \top)) \in \rightarrow$ *iff there exists* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$ *such that* $\lfloor T \rfloor_{\Delta_P} - r \models {}_{\Delta_S}[g]_{\Delta_S}$ *and* $r' = r[R := \lfloor T \rfloor_{\Delta_P}]$.

($P5.2$) *let* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$. $((\ell, r, T, I, u, d, f), \sigma, (\ell, r, T, I', u, d, f)) \in \rightarrow$ *iff*
- *either* $I(\sigma) = \bot$ *and* $I' = I[\sigma := 0]$;
- *or* $I(\sigma) \neq \bot$ *and* $I' = I$.

($P5.3$) *let* $\widetilde{\sigma} \in \widetilde{\mathsf{Lab}^{\mathsf{in}}}$. $((\ell, r, T, I, u, d, \bot), \widetilde{\sigma}, (\ell', r', T, I', u, 0, \top)) \in \longrightarrow$ *iff there exists* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$ *such that* $\lfloor T \rfloor_{\Delta_P} - r \models {}_{\Delta_S}[g]_{\Delta_S}$, $I(\sigma) > u$, $r' = r[R := \lfloor T \rfloor_{\Delta_P}]$ *and* $I' = I[\sigma := \bot]$;

($P5.4$) *let* $\sigma \in \mathsf{Lab}^{\tau}$. $((\ell, r, T, I, u, d, \bot), \sigma, (\ell', r', T, I, u, 0, \top)) \in \longrightarrow$ *iff there exists* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$ *such that* $\lfloor T \rfloor_{\Delta_P} - r \models {}_{\Delta_S}[g]_{\Delta_S}$ *and* $r' = r[R := \lfloor T \rfloor_{\Delta_P}]$.

($P5.5$) *let* $\sigma = \tau$. $((\ell, r, T, I, u, d, f), \sigma, (\ell, r, T+u, I, 0, d, \bot)) \in \longrightarrow$ *iff either* $f = \top$ *or the two following conditions hold:*

- *for any* $\widetilde{\sigma} \in \widetilde{\mathsf{Lab}^{\mathsf{in}}}$, *for any* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$, *we have that either* $\lfloor T \rfloor_{\Delta_P} - r \not\models {}_{\Delta_S}[g]_{\Delta_S}$ *or* $I(\sigma) \leq u$
- *for any* $\sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^{\tau}$, *for any* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$, *we have that* $\lfloor T \rfloor_{\Delta_P} - r \not\models {}_{\Delta_S}[g]_{\Delta_S}$

– *for the continuous transitions:*

($P5.6$) $((\ell, r, T, I, u, d, f), t, (\ell, r, T, I+t, u+t, d+t, f)) \in \longrightarrow$ *iff* $u + t \leq \Delta_L$.

Comments on the program semantics. Rule ($P1$) defines the states which are tuples (ℓ, r, T, I, u, d, f), where ℓ is the current location, r maps each clock to the digital time when it has last been reset, T records the (exact) time at which the last round has started; I, as in the AASAP semantics, records the time elapsed since the arrival of the last untreated input event, u records the time elapsed since the last round was started (so that $T + u$ is the exact current time), d records the time elapsed since the last location change, and f is a flag which is set to $\top$ if a location change has occurred in the current round. Rules ($P2$) to ($P4$) are straightforward. We comment rules ($P5.1 - 6$). First, we make some general comments on digital clocks and guards of discrete transitions of the controller. Note that in those rules, we evaluate the guards with the valuation $\lfloor T \rfloor_{\Delta_P} - r$ for the clocks, that is, for variable x, the difference between the digital value of the variable T at the beginning of the current round and the digital value of x at the beginning of the round when x was last reset. This value approximates the real time difference between the exact time at which the guard is evaluated and the exact time at which the clock x has been reset. Let t be this exact time difference, then we know that: $\lfloor T \rfloor_{\Delta_P} - r(x) - \Delta_L - \Delta_P \leq t \leq \lfloor T \rfloor_{\Delta_P} - r(x) + \Delta_L + \Delta_P$. Also note that the guard g has been enlarged by the value $\Delta_S = \lceil \Delta_L + \Delta_P \rceil_{\Delta_P}$; this

ensures that any event enabled at some point will be enabled sufficiently long so that the change can be detected by the procedure. The reason of the rounding to the least superior multiple of Δ_P is that Δ_S is a constant intended to be written in real code. Thus it has to be be expressed in the unit of time of the system. Rule (P5.1) expresses when transitions labeled with output events can be taken. Note that variables are reset to the digital time of the current round. Rule (P5.2) records the exact time at which the last untreated occurrence of an input event from the environment occurred. This rule simply ensures that the function I is updated when a new event, for which no occurrence is pending, is issued by the environment. Note that this rule ensures input enabledness. Rule (P5.3) says when an input of the environment can be treated by the controller: it has to be present at the beginning of the current round and the enlargement of the guard labelling the transition has to be true for digital values of the clocks at the beginning of the round, and no other discrete transition should have been taken in the current round. Rule (P5.4) is similar to rule (P5.1) but applies to internal events. Rule (P5.5) expresses that the event τ is issued when the current round is finished and the system starts a new round. Note that this is only possible if the program has taken a discrete transition or there were no discrete transition to take. This ensures that the program always takes discrete transitions when possible. Rule (P5.6) expresses that the program can always let time elapse unless it violates the maximal time spent in one round. This obviously ensures that the program semantics cannot implement zeno strategies

The following simulation theorem expresses formally that if the hardware on which the program is implemented is fast enough (parameter Δ_L) and fine granular enough (parameter Δ_P) then the program semantics can be simulated by the AASAP semantics.

Theorem 3.2 (Strong Simulation)
Let A be an ELASTIC *controller: for any rationals $\Delta, \Delta_L, \Delta_P \in \mathbb{Q}^{>0}$ such that $3\Delta_L + 4\Delta_P < \Delta$, we have $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L,\Delta_P} \preceq [\![A]\!]^{\mathsf{AAsap}}_{\Delta}$.*

Proof
Let $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L,\Delta_P} = (S_1, \mathrm{E}_1, \mathrm{F}_1, \Sigma, \rightarrow_1)$ and $[\![A]\!]^{\mathsf{AAsap}}_{\Delta} = (S_2, \mathrm{E}_2, \mathrm{F}_2, \Sigma, \rightarrow_2)$. Consider the relation $R \subseteq S_1 \times S_2$ that contains all the pairs:

$$(s_1, s_2) = ((\ell_1, r_1, T_1, I_1, u_1, d_1, f_1), (\ell_2, v_2, I_2, d_2))$$

such that the following conditions hold:

$(R1)$ $\ell_1 = \ell_2$*;*

$(R2)$ *for any* $x \in \mathsf{Var}$, $|v_2(x) - (T_1 - r_1(x) + u_1)| \leq \Delta_L + \Delta_P$

$(R3)$ *for any* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, $I_1(\sigma) = I_2(\sigma)$*;*

$(R4)$ $d_1 = d_2$*;*

$(R5)$ *there exists* $(\ell_2'', v_2'', I_2'', d_2'')$ *such that:* $((\ell_2, v_2, I_2, d_2), \Delta_L - u_1, (\ell_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.

Let us show that R *is a strong simulation relation.*

1. $\forall s \in \mathrm{E}, \exists s' \in \mathrm{E}' : (s, s') \in R$. *We have to check the 5 rules of the simulation relation for the only element present in* E*, which is paired with the only element present in* E′.

   $(R1), (R2), (R3)$ *and* $(R4)$ *are clearly true.*

   $(R5)$ *To establish this property, we first note that* $d_2 = 0$ *and so* $d_2 + \Delta_L < \Delta$ *which implies* $\forall t' \leq \Delta_L : d_2 + t' < \Delta$*. Hence the two conditions of rule* $(A5.6)$ *are verified.*

2. $\forall (s, s') \in R : s \in \mathrm{F} \implies s' \in \mathrm{F}'$. *This condition is trivially satisfied as* $\mathrm{F} = \varnothing$.

3. *Let us assume that* $(s_1, s_2) = ((\ell_1, r_1, T_1, I_1, u_1, d_1, f_1), (\ell_2, v_2, I_2, d_2)) \in R$ *and that* $(s_1, \sigma, s_1') \in \rightarrow_1$ *(with* $s_1' = (\ell_1', r_1', T_1', I_1', u_1', d_1', f_1')$*). We must prove that for each value of* σ*, there exists a state* $s_2' \in S_2$ *such that* $(s_2, \sigma, s_2') \in \rightarrow_2$ *and* $(s_1', s_2') \in R$.

   Since $(s_1, s_2) \in R$ *we know that:*

   $(H1)$ $s_2 = (\ell_1, v_2, I_1, d_1)$

   $(H2)$ $\forall x \in \mathsf{Var} : T_1 - r_1(x) + u_1 - \Delta_L - \Delta_P \leq v_2(x) \leq T_1 - r_1(x) + u_1 + \Delta_L + \Delta_P$

   $(H3)$ *there exists* $s_2'' = (\ell_2'', v_2'', I_2'', d_2'') \in S_2$ *such that:* $((\ell_2, v_2, I_2, d_2), \Delta_L - u_1, (\ell_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.

   The rest of the proof works case by case on the different possible types of σ*:*

case (a) *let* $\sigma \in \Sigma^{\mathsf{in}}$

Since $(s_1, \sigma, s_1') \in \rightarrow_1$ *we know that:*

$$s_1' = \begin{cases} (\ell_1, r_1, T_1, I_1[\{\sigma\} := 0], u_1, d_1, f_1) & \text{if} \quad I_1(\sigma) = \bot \\ (\ell_1, r_1, T_1, I_1, u_1, d_1, f_1) & \text{if} \quad I_1(\sigma) \neq \bot \end{cases}$$

Let us first prove that $\exists s_2' \in S_2 : (s_2, \sigma, s_2') \in \rightarrow_2$. *This is immediate since the* AASAP *semantics is* input enabled. *Now that we know* s_2' *exists we can say that:*

$$s_2' = \begin{cases} (\ell_1, v_2, I_1[\{\sigma\} := 0], d_1) & \text{if} \quad I_1(\sigma) = \bot \\ (\ell_1, v_2, I_1, d_1) & \text{if} \quad I_1(\sigma) \neq \bot \end{cases}$$

It is now easy to prove that $(s_1', s_2') \in R$. *Indeed, it is obvious that* s_2' *fulfills the five conditions of the simulation relation if* $(s_1, s_2) \in R$.

case (b) *let* $\sigma \in \Sigma^{\mathsf{out}}$

Since $(s_1, \sigma, s_1') \in \rightarrow_1$ *we know that:*

$$(J1) \begin{cases} \exists (\ell_1, \ell_1', g, \sigma, R) \in \mathsf{Edg} : \lfloor T_1 \rfloor_{\Delta_P} - r_1 \models {}_{\Delta_S}[g]_{\Delta_S} \\ s_1' = (\ell_1', r_1[R := \lfloor T_1 \rfloor_{\Delta_P}], T_1, I_1, u_1, 0, \top) \end{cases}$$

Let us first prove that $\exists s_2' \in S_2 : (s_2, \sigma, s_2') \in \rightarrow_2$. *We use the same edge as in the implementation semantics (see* $(J1)$*). This amounts to prove that:* $\forall x \in \mathsf{Var} : v_2(x) \models {}_{\Delta}[g]_{\Delta}(x)$. *Let* $a_x = lb(g(x))$ *and* $b_x = rb(g(x))$. *We know that* $\forall x \in \mathsf{Var}$*:*

$$
\begin{array}{ll}
 & a_x - \Delta_S \leq \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) \leq b_x + \Delta_S \\
 & (J1) \\
\Longrightarrow & a_x - \Delta_S - \Delta_P \leq T_1 - r_1(x) \leq b_x + \Delta_S + \Delta_P \\
 & (\textit{Lemma 3.2}) \\
\Longrightarrow & a_x - \lceil \Delta_L + \Delta_P \rceil_{\Delta_P} - \Delta_P \leq T_1 - r_1(x) \\
 & \leq b_x + \lceil \Delta_L + \Delta_P \rceil_{\Delta_P} + \Delta_P \\
 & (\textit{def. of } \Delta_S) \\
\Longrightarrow & a_x - \Delta_L - 3\Delta_P \leq T_1 - r_1(x) \leq b_x + \Delta_L + 3\Delta_P \\
 & (\textit{Lemma 3.2}) \\
\Longrightarrow & a_x - 2\Delta_L - 4\Delta_P + u_1 \leq T_1 - r_1(x) + u_1 - \Delta_L - \Delta_P \wedge \\
 & T_1 - r_1(x) + u_1 + \Delta_L + \Delta_P \leq b_x + 2\Delta_L + 4\Delta_P + u_1 \\
\Longrightarrow & a_x - 2\Delta_L - 4\Delta_P + u_1 \leq v_2(x) \leq b_x + 2\Delta_L + 4\Delta_P + u_1 \\
 & (H2) \\
\Longrightarrow & a_x - 2\Delta_L - 4\Delta_P \leq v_2(x) \leq b_x + 3\Delta_L + 4\Delta_P \\
 & (0 \leq u_1 \leq \Delta_L) \\
\Longrightarrow & a_x - \Delta \leq v_2(x) \leq b_x + \Delta \\
 & (3\Delta_L + 4\Delta_P < \Delta) \\
\Longrightarrow & v_2(x) \models {}_{\Delta}[g]_{\Delta}
\end{array}
$$

Now that it is established that $\exists s_2' \in S_2 : (s_1', \sigma, s_2') \in \rightarrow_2$ *we know that* $s_2' = (\ell_1', v_2[R := 0], I_1, 0)$.

It remains to prove that $(s_1', s_2') \in R$ *which means we must check the five rules of the simulation relation.* $(R1)$,$(R3)$,$(R4)$ *and* $(R5)$ *are clearly true.*

To prove $(R2)$ *we have to prove that:*

$$
\forall x \in \mathsf{Var} : \begin{cases} T_1 - r_1[R := \lfloor T_1 \rfloor_{\Delta_P}](x) + u_1 - \Delta_L - \Delta_P \leq v_2[R := 0](x) \\ v_2[R := 0](x) \leq T_1 - r_1[R := \lfloor T_1 \rfloor_{\Delta_P}](x) + u_1 + \Delta_L + \Delta_P \end{cases}
$$

This proposition is the same as $(H2)$ *for* $x \notin R$. *For* $x \in R$, *it amounts to prove:*

$$
T_1 - \lfloor T_1 \rfloor_{\Delta_P} + u_1 - \Delta_L - \Delta_P \leq 0 \leq T_1 - \lfloor T_1 \rfloor_{\Delta_P} + u_1 + \Delta_L + \Delta_P.
$$

which is implied by $T_1 - \Delta_P - \lfloor T_1 \rfloor_{\Delta_P} \leq 0 \leq T_1 + \Delta_P - \lfloor T_1 \rfloor_{\Delta_P}$ *since* $u_1 - \Delta_L \leq 0$. *This is a consequence of Lemma 3.2. This establishes* $(R2)$.

case (c) *let* $\sigma \in \Sigma^\tau = \mathsf{Lab}^\tau \cup \widetilde{\mathsf{Lab}^{\mathsf{in}}} \cup \{\tau\}$. *The proof for the first two sets is similar to the previous case. Let* $\sigma = \tau$.

Since $(s_1, \tau, s_1') \in \rightarrow_1$ *we know by* $(P5.5)$ *that*

$(K1)$ $s_1' = (\ell_1, r_1, T_1 + u_1, I_1, 0, d_1, \bot)$

$(K2)$ $f_1 = \top$ *or*

- *for any* $\widetilde{\sigma}$ *such that* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, *for any* $(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$, *we have that either* $\lfloor T_1 \rfloor_{\Delta_P} - r_1 \not\models_{\Delta_S} [g]_{\Delta_S}$ *or* $I_1(\sigma) \leq u_1$
- *for any* $\sigma \in \mathsf{Lab}^{\mathsf{out}}$, *for any* $(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$, *we have that* $\lfloor T_1 \rfloor_{\Delta_P} - r_1 \not\models_{\Delta_S} [g]_{\Delta_S}$

By rule $(A5.5)$ *of the* AASAP*-semantics, we know that there exists* $s_2' \in S_2$ *such that* (s_2, τ, s_2') *and*

$(K3)$ $s_2' = s_2 = (\ell_1, v_2, I_1, d_1)$.

Now we have to prove that $(s_1', s_2') \in R$. $(R1), (R3)$ *and* $(R4)$ *are clearly true. Proving* $(R2)$ *amounts to prove that*

$$\forall x \in \mathsf{Var} : T_1 + u_1 - r_1(x) + 0 - \Delta_L - \Delta_P \leq v_2(x) \leq T_1 + u_1 - r_1(x) + 0 + \Delta_L + \Delta_P$$

which turns out to be equivalent to $(H2)$.

Let us now prove that there exists s_2'' *s.t.* $((\ell_1, v_2, I_1, d_1), \Delta_L, s_2'') \in \rightarrow_2$. *According to rule* $(A5.6)$, *it amounts to prove that*

$(L1)$ *for any edge* $(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$ *with* $\sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^\tau$, *we have that:*

$$\forall t' : 0 \leq t' \leq \Delta_L : (d_1 + t' \leq \Delta \vee \mathsf{TS}(v_2 + t', g) \leq \Delta)$$

$(L2)$ *for any edge* $(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$ *with* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, *we have that:*

$$\forall t' : 0 \leq t' \leq \Delta_L : (d_1 + t' \leq \Delta \vee \mathsf{TS}(v_2 + t', g) \leq \Delta \vee (I_1 + t')(\sigma) \leq \Delta)$$

If $f_1 = \top$ *it implies that the program has made a discrete transition during the last loop, which means that* $d_1 \leq \Delta_L$ *and thus that* $d_1 + \Delta_L \leq 2\Delta_L \leq \Delta$ *because we know that, by hypothesis,* $2\Delta_L < \Delta$, *which makes* $(L1)$ *and* $(L2)$ *true for any* t'.

If $f_1 \neq \top$, the proof is less trivial. We first make a proof for labels of $(L1)$.

$\forall(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$ *with* $\sigma \in \mathsf{Lab}^{\mathsf{out}}$ *we have* $\lfloor T_1 \rfloor_{\Delta_P} \not\models {}_{\Delta_S}[g]_{\Delta_S}$ *by* $(K2)$. *Let* $a_x = lb(g(x))$ *and* $b_x = rb(g(x))$. *There are two possible cases:*

(a) $\exists x \in \mathsf{Var}$ *such that*

$$
\begin{array}{ll}
 & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) < a_x - \Delta_S \\
\Longrightarrow & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) < a_x - \lceil \Delta_L + \Delta_P \rceil_{\Delta_P} \\
 & (\Delta_S = \lceil \Delta_L + \Delta_P \rceil_{\Delta_P}) \\
\Longrightarrow & T_1 - r_1(x) < a_x - \Delta_L \\
 & (\textit{Lemma 3.2}) \\
\Longrightarrow & T_1 - r_1(x) + u_1 + \Delta_L + \Delta_P < a_x + u_1 + \Delta_P \\
\Longrightarrow & v_2(x) < a_x + u_1 + \Delta_P \\
 & (H2) \\
\Longrightarrow & v_2(x) < a_x + \Delta_L + 2\Delta_P \\
 & (u_1 \leq \Delta_L + \Delta_P) \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : v_2(x) + t' \leq a_x + 2\Delta_L + 2\Delta_P \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : \mathsf{TS}(v_2(x) + t', g(x)) \leq 2\Delta_L + \Delta_P \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : \mathsf{TS}(v_2(x) + t', g(x)) \leq \Delta \\
 & (2\Delta_L + 2\Delta_P < \Delta) \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : \mathsf{TS}(v_2 + t', g) \leq \Delta
\end{array}
$$

(b) $\exists x \in \mathsf{Var}$ *such that*

$$
\begin{array}{ll}
 & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) > b_x + \Delta_S \\
\Longrightarrow & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) > b_x + \lceil \Delta_L + \Delta_P \rceil_{\Delta_P} \\
 & (\Delta_S = \lceil \Delta_L + \Delta_P \rceil_{\Delta_P}) \\
\Longrightarrow & T_1 - r_1(x) > b_x + \Delta_L + \Delta_P \\
 & (\textit{Lemma 3.2}) \\
\Longrightarrow & T_1 - r_1(x) + u_1 - \Delta_L - \Delta_P > b_x + u_1 \\
\Longrightarrow & v_2(x) > b_x + u_1 \\
 & (H2) \\
\Longrightarrow & v_2(x) > b_x \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : v_2(x) + t' > b_x
\end{array}
$$

$$\begin{array}{lll} \implies & \forall t' : 0 \leq t' \leq \Delta_L : \mathsf{TS}(v_2(x) + t', g(x)) \leq \Delta & \\ & (v_2(x) + t' \not\models g(x)) & \\ \implies & \forall t' : 0 \leq t' \leq \Delta_L : \mathsf{TS}(v_2 + t', g) \leq \Delta & \end{array}$$

Thus, both cases imply that $(L1)$ *is true.*

The proof for $(L2)$ *is the same if we have, by* $(K2)$, $\lfloor T_1 \rfloor_{\Delta_P} \not\models {}_{\Delta_S}[g]_{\Delta_S}$. *Otherwise, we have* $I_1(\sigma) < u_1$ *which also proves* $(L2)$. *Indeed*

$$\begin{array}{lll} & I_1(\sigma) < u_1 & \\ \implies & I_1(\sigma) < \Delta_L & (u_1 \leq \Delta_L) \\ \implies & \forall t' : 0 \leq t' \leq \Delta_L : I_1(\sigma) + t' < 2\Delta_L & \\ \implies & \forall t' : 0 \leq t' \leq \Delta_L : I_1(\sigma) + t' < \Delta & (2\Delta_L \leq \Delta) \end{array}$$

case (d) *let* $\sigma \in \mathbb{R}^{\geq 0}$. *For the sake of clarity let us consider that* $\sigma = t$.

Since $(s_1, t, s_1') \in \rightarrow_1$ *we know by* $(P5.6)$ *that*

$(M1)$ $s_1' = (\ell_1, r_1, T_1, I_1 + t, u_1 + t, d_1 + t, f_1)$;

$(M2)$ $u_1 + t \leq \Delta_L$.

With those facts, we know that there exists $s_2' = (\ell_2', v_2', I_2', d_2') \in S_2$ *such that* $(s_2, t, s_2') \in \rightarrow_2$ *because* $(s_2, \Delta_L - u_1, s_2'') \in \rightarrow_2$ *by* $(H3)$ *and* $t \leq \Delta_L - u_1$ *by* $(M2)$.

Now we have $(s_2, t, s_2') \in \rightarrow_2$ *and we know that:*

$(M3)$ $s_2' = (\ell_1, v_2 + t, I_1 + t, d_1 + t)$

We can now prove that $(s_1', s_2') \in R$. *We have to check the five points of the simulation relation:* $(R1), (R2), (R3)$ *and* $(R4)$ *are easy to prove using hypothesis* $(H1)$ *to* $(H3)$ *and* $(M1)$ *to* $(M3)$.

For $(R5)$, *since by* $(H3)$, *there exists* $s_2'' = (\ell_2'', v_2'', I_2'', d_2'') \in S_2$ *such that* $((\ell_2, v_2, I_2, d_2), \Delta_L - u_1, (\ell_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$, *we have* $((\ell_1, v_2 + t, I_1 + t, d_1 + t), (\Delta_L - u_1 - t), (\ell_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.

Theorem 3.3 (Simulability)
For any ELASTIC *controller* A, *for any* $\Delta \in \mathbb{Q}^{>0}$, *there exists* $\Delta_L, \Delta_P \in \mathbb{Q}^{>0}$ *such that* $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L, \Delta_P} \preceq [\![A]\!]^{\mathsf{AAsap}}_{\Delta}$.

Proof
For any $\Delta > 0$, *since parameters* Δ_L *and* Δ_P *are in the rational numbers, they can always be chosen such that* $3\Delta_L + 4\Delta_P < \Delta$.

And so, given a sufficiently fast hardware with a sufficiently small granularity for its clock, we can implement any controller that has been proved correct. This is expressed by the following corollary:

Corollary 3.2 (Implementability)
Let E be a timed automaton. For any Elastic *controller A, for any $\Delta \in \mathbb{Q}^{>0}$, such that $[\![A]\!]^{\mathsf{AAsap}}_{\Delta}$ controls $[\![E]\!]$, there exist $\Delta_L, \Delta_P \in \mathbb{Q}^{>0}$ such that $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L,\Delta_P}$ controls $[\![E]\!]$.*

3.3.3 Implementability with Clock Drifts

In the previous section, we have shown that a control strategy that has been shown correct for the AASAP semantics can be implemented with a sufficiently fast hardware equipped with a sufficiently fine granular clock. To keep the proof and the exposition of the concepts simple, we have assumed that the hardware clock was delivering exactly spaced ticks. If we want to model the imprecision due to clock drifts in the hardware, we have to modify our program semantics. This is done in the Definition 3.12.

There are two modifications in the definition of the program semantics :

1. The main change is in rule $(Q5.5)$: to model a possible drift of ε every time unit, we change the way the variable T holding the current time of the system is updated. T is not incremented by the exact value u since its last update, but with a value $u' \in [(1-\varepsilon)u, (1+\varepsilon)u]$. This clearly models a drift bounded by ε every time unit.

2. As a drifting clock is less accurate, we have to enlarge the guards a bit more than in the previous semantics to ensure that no guard is missed.

Definition 3.12 (Program Semantics with clock drifts)
Let A be an Elastic *controller, Δ_L, $\Delta_P \in \mathbb{Q}^{>0}$, $\varepsilon \in \mathbb{Q}^{\geq 0}$ with $\varepsilon < 1$. Let $M \in \mathbb{N}$ be the largest constant a clock is compared with in A and finally, let $\Delta_S = \lceil (1+\varepsilon)(\Delta_L + \Delta_P) + \varepsilon M \rceil_{\Delta_P}$. The $(\Delta_L, \Delta_P, \varepsilon)$ program semantics of A, noted $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L,\Delta_P,\varepsilon}$ is the* input enabled *structured timed transition system $\mathcal{T} = \langle S, \mathrm{E}, \Sigma, \rightarrow \rangle$ where:*

($Q1$) *S is the set of tuples (ℓ, r, T, I, u, d, f) such that $l \in \mathsf{Loc}$, r is a function from* Var *into* $\mathbb{R}^{\geq 0}$*, $T \in \mathbb{R}^{\geq 0}$, I is a function from* $\mathsf{Lab}^{\mathsf{in}}$ *into* $\mathbb{R}^{\geq 0} \cup \{\bot\}$*, $u \in \mathbb{R}^{\geq 0}$, $d \in \mathbb{R}^{\geq 0}$, and $f \in \{\top, \bot\}$;*

($Q2$) $\mathrm{E} = \{(\ell_0, r, 0, I, 0, 0, \bot)\}$ *where r is such that for any $x \in \mathsf{Var}$, $r(x) = 0$, I is such that for any $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, $I(\sigma) = \bot$;*

($Q3$) $\mathrm{F} = \varnothing$;

($Q4$) $\Sigma = \Sigma^{\mathsf{in}} \uplus \Sigma^{\mathsf{out}} \uplus \Sigma^{\tau} \uplus \mathbb{R}^{\geq 0}$ *and* $\Sigma^{\mathsf{in}} = \mathsf{Lab}^{\mathsf{in}}$, $\Sigma^{\mathsf{out}} = \mathsf{Lab}^{\mathsf{out}}$, $\Sigma^{\tau} = \mathsf{Lab}^{\tau} \cup \widetilde{\mathsf{Lab}^{\mathsf{in}}} \cup \{\tau\}$;

($Q5$) *the transition relation $\rightarrow$ is defined as follows:*

- *for the discrete transitions:*

($Q5.1$) *let $\sigma \in \mathsf{Lab}^{\mathsf{out}}$. $((\ell, r, T, I, u, d, \bot), \sigma, (\ell', r', T, I, u, 0, \top)) \in \rightarrow$ iff there exists $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$ such that $\lfloor T \rfloor_{\Delta_P} - r \models {}_{\Delta_S}[g]_{\Delta_S}$ and $r' = r[R := \lfloor T \rfloor_{\Delta_P}]$.*

($Q5.2$) *let $\sigma \in \mathsf{Lab}^{\mathsf{in}}$. $((\ell, r, T, I, u, d, f), \sigma, (\ell, r, T, I', u, d, f)) \in \rightarrow$ iff*

- *either $I(\sigma) = \bot$ and $I' = I[\sigma := 0]$;*
- *or $I(\sigma) \neq \bot$ and $I' = I$.*

($Q5.3$) *let $\widetilde{\sigma} \in \widetilde{\mathsf{Lab}^{\mathsf{in}}}$. $((\ell, r, T, I, u, d, \bot), \widetilde{\sigma}, (\ell', r', T, I', u, 0, \top)) \in \rightarrow$ iff there exists $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$ such that $\lfloor T \rfloor_{\Delta_P} - r \models {}_{\Delta_S}[g]_{\Delta_S}$, $I(\sigma) > u$, $r' = r[R := \lfloor T \rfloor_{\Delta_P}]$ and $I' = I[\sigma := \bot]$;*

($Q5.4$) *let $\sigma \in \mathsf{Lab}^{\tau}$. $((\ell, r, T, I, u, d, \bot), \sigma, (\ell', r', T, I, u, 0, \top)) \in \rightarrow$ iff there exists $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$ such that $\lfloor T \rfloor_{\Delta_P} - r \models {}_{\Delta_S}[g]_{\Delta_S}$ and $r' = r[R := \lfloor T \rfloor_{\Delta_P}]$.*

($Q5.5$) *let $\sigma = \tau$. $((\ell, r, T, I, u, d, f), \sigma, (\ell, r, T + u', I, 0, d, \bot)) \in \rightarrow$ where $u' \in [(1 - \varepsilon)u, (1 + \varepsilon)u]$ iff either $f = \top$ or the two following conditions hold:*

- *for any $\widetilde{\sigma} \in \widetilde{\mathsf{Lab}^{\mathsf{in}}}$, for any $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$, we have that either $\lfloor T \rfloor_{\Delta_P} - r \not\models {}_{\Delta_S}[g]_{\Delta_S}$ or $I(\sigma) \leq u$*
- *for any $\sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^{\tau}$, for any $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}$, we have that $\lfloor T \rfloor_{\Delta_P} - r \not\models {}_{\Delta_S}[g]_{\Delta_S}$*

- *for the continuous transitions:*

$(Q5.6)$ $((\ell, r, T, I, u, d, f), t, (\ell, r, T, I+t, u+t, d+t, f)) \in \rightarrow$ *iff* $u+t \leq \Delta_L$.

The following simulation theorem expresses formally that if the hardware on which the program is implemented is fast enough (parameter Δ_L), fine granular enough (parameter Δ_P), and precise enough(parameter ε) then the program semantics can be simulated by the AASAP semantics.

Theorem 3.4 (Simulation with drifting clocks)
Let A be an ELASTIC *controller. Let M be the largest constant a clock is compared with in A. For any $\Delta, \Delta_L, \Delta_P \in \mathbb{Q}^{>0}$, $\varepsilon \in \mathbb{Q}^{\geq 0}$ with $\varepsilon < 1$ such that $\frac{2\varepsilon M+(3+\varepsilon)\Delta_L+(4+2\varepsilon)\Delta_P}{1-\varepsilon} < \Delta$, we have $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L,\Delta_P,\varepsilon} \preceq [\![A]\!]^{\mathsf{AAsap}}_{\Delta}$.* ■

Proof
Let $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L,\Delta_P,\varepsilon} = (S_1, \mathrm{E}_1, \mathrm{F}_1, \Sigma, \rightarrow_1)$ and $[\![A]\!]^{\mathsf{AAsap}}_{\Delta} = (S_2, \mathrm{E}_2, \mathrm{F}_2, \Sigma, \rightarrow_2)$. Consider the relation $R \subseteq S_1 \times S_2$ that contains the pairs:

$$(s_1, s_2) = ((\ell_1, r_1, T_1, I_1, u_1, d_1, f_1), (\ell_2, v_2, I_2, d_2))$$

such that the following conditions hold:

$(R1)$ $\ell_1 = \ell_2$;

$(R2)$ *for any* $x \in \mathsf{Var}$, $\frac{T_1 - r_1(x) - (1+\varepsilon)(\Delta_L+\Delta_P)}{1+\varepsilon} + u_1 \leq v_2(x) \leq \frac{T_1 - r_1(x) + (1+\varepsilon)(\Delta_L+\Delta_P)}{1-\varepsilon} + u_1$

$(R3)$ *for any* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, $I_1(\sigma) = I_2(\sigma)$;

$(R4)$ $d_1 = d_2$;

$(R5)$ *there exists* $(\ell_2'', v_2'', I_2'', d_2'')$ *such that:* $((\ell_2, v_2, I_2, d_2), \Delta_L - u_1, (\ell_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.

Let us show that R is a simulation relation.

1. $\forall s \in \mathrm{E}, \exists s' \in \mathrm{E}' : (s, s') \in R$. *We have to check the 5 rules of the simulation relation for the only element present in* E, *which is paired with the only element present in* E′.

   $(R1), (R2), (R3)$ *and* $(R4)$ *are clearly true.*

   $(R5)$ *To establish this property, we first note that $d_2 = 0$ and so $d_2 + \Delta_L < \Delta$ which implies $\forall t' \leq \Delta_L : d_2 + t' < \Delta$. Hence the two conditions of rule* $(A5.6)$ *are verified.*

2. $\forall (s, s') \in R : s \in \mathrm{F} \implies s' \in \mathrm{F}'$. *This condition is trivially satisfied as* $\mathrm{F} = \varnothing$.

3. *Let us assume that* $(s_1, s_2) = ((\ell_1, r_1, T_1, I_1, u_1, d_1, f_1), (\ell_2, v_2, I_2, d_2)) \in R$ *and that* $(s_1, \sigma, s_1') \in \rightarrow_1$ *(with* $s_1' = (\ell_1', r_1', T_1', I_1', u_1', d_1', f_1')$*). We must prove that for each value of* σ*, there exists a state* $s_2' \in S_2$ *such that* $(s_2, \sigma, s_2') \in \rightarrow_2$ *and* $(s_1', s_2') \in R$.

   Since $(s_1, s_2) \in R$ *we know that:*

   $(H1)$ $s_2 = (\ell_1, v_2, I_1, d_1)$

   $(H2)$ $\forall x \in \mathsf{Var} : \frac{T_1 - r_1(x) - (1+\varepsilon)(\Delta_L + \Delta_P)}{1+\varepsilon} + u_1 \leq v_2(x) \leq \frac{T_1 - r_1(x) + (1+\varepsilon)(\Delta_L + \Delta_P)}{1-\varepsilon} + u_1$

   $(H3)$ *there exists* $s_2'' = (\ell_2'', v_2'', I_2'', d_2'') \in S_2$ *such that:* $((\ell_2, v_2, I_2, d_2), \Delta_L - u_1, (\ell_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.

   The rest of the proof works case by case on the different possible types of σ*:*

case (a) *let* $\sigma \in \Sigma^{\mathsf{in}}$

Since $(s_1, \sigma, s_1') \in \rightarrow_1$ *we know that:*

$$s_1' = \begin{cases} (\ell_1, r_1, T_1, I_1[\{\sigma\} := 0], u_1, d_1, f_1) & \text{if} \quad I_1(\sigma) = \bot \\ (\ell_1, r_1, T_1, I_1, u_1, d_1, f_1) & \text{if} \quad I_1(\sigma) \neq \bot \end{cases}$$

Let us first prove that $\exists s_2' \in S_2 : (s_2, \sigma, s_2') \in \rightarrow_2$. *This is immediate since the* AASAP *semantics is* input enabled. *Now that we know* s_2' *exists we can say that:*

$$s_2' = \begin{cases} (\ell_1, v_2, I_1[\{\sigma\} := 0], d_1) & \text{if} \quad I_1(\sigma) = \bot \\ (\ell_1, v_2, I_1, d_1) & \text{if} \quad I_1(\sigma) \neq \bot \end{cases}$$

It is now easy to prove that $(s_1', s_2') \in R$. *Indeed, it is obvious that* s_2' *fulfills the five conditions of the simulation relation if* $(s_1, s_2) \in R$.

case (b) *let* $\sigma \in \Sigma^{\mathsf{out}}$

Since $(s_1, \sigma, s_1') \in \rightarrow_1$ *we know that:*

$$(J1) \begin{cases} \exists (\ell_1, l_1', g, \sigma, R) \in \mathsf{Edg} : \lfloor T_1 \rfloor_{\Delta_P} - r_1 \models_{\Delta_S} [g]_{\Delta_S} \\ s_1' = (\ell_1', r_1[R := \lfloor T_1 \rfloor_{\Delta_P}], T_1, I_1, u_1, 0, \top) \end{cases}$$

Let us first prove that $\exists s'_2 \in S_2 : (s_2, \sigma, s'_2) \in \rightarrow_2$. *We use the same edge as in the implementation semantics (see* $(J1)$*). This amounts to prove that:* $\forall x \in \mathsf{Var} : v_2(x) \models_{\Delta} [g]_{\Delta}(x)$. *Let* $a_x = lb(g(x))$ *and* $b_x = rb(g(x))$. *We know that* $\forall x \in \mathsf{Var}$*:*

$$
\begin{array}{ll}
 & a_x - \Delta_S \leq \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) \leq b_x + \Delta_S \\
 & (J1) \\
\Longrightarrow & a_x - \lceil (1+\varepsilon)(\Delta_L + \Delta_P) + \varepsilon M \rceil_{\Delta_P} \leq \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) \wedge \\
 & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) \leq b_x + \lceil (1+\varepsilon)(\Delta_L + \Delta_P) + \varepsilon M \rceil_{\Delta_P} \\
 & (\textit{def. of } \Delta_S) \\
\Longrightarrow & a_x - (1+\varepsilon)(\Delta_L + \Delta_P) - \varepsilon M - \Delta_P \leq T_1 - r_1(x) \wedge \\
 & T_1 - r_1(x) \leq b_x + (1+\varepsilon)(\Delta_L + \Delta_P) + \varepsilon M + 2\Delta_P \\
 & (\textit{Lemma 3.2}) \\
\Longrightarrow & \frac{a_x - 2(1+\varepsilon)(\Delta_L + \Delta_P) - \varepsilon M - \Delta_P}{1+\varepsilon} + u_1 \leq \frac{T_1 - r_1(x) - (1+\varepsilon)(\Delta_L + \Delta_P)}{1+\varepsilon} + u_1 \wedge \\
 & \frac{T_1 - r_1(x) + (1+\varepsilon)(\Delta_L + \Delta_P)}{1-\varepsilon} + u_1 \leq \frac{b_x + 2(1+\varepsilon)(\Delta_L + \Delta_P) + \varepsilon M + 2\Delta_P}{1-\varepsilon} + u_1 \\
\Longrightarrow & \frac{a_x - 2(1+\varepsilon)(\Delta_L + \Delta_P) - \varepsilon M - \Delta_P}{1+\varepsilon} + u_1 \leq v_2(x) \wedge \\
 & v_2(x) \leq \frac{b_x + 2(1+\varepsilon)(\Delta_L + \Delta_P) + \varepsilon M + 2\Delta_P}{1-\varepsilon} + u_1 \\
 & (H2) \\
\Longrightarrow & \frac{a_x - (2+2\varepsilon)\Delta_L - (3+2\varepsilon)\Delta_P - \varepsilon M}{1+\varepsilon} \leq v_2(x) \wedge \\
 & v_2(x) \leq \frac{b_x + (3+\varepsilon)\Delta_L + (4+2\varepsilon)\Delta_P + \varepsilon M}{1-\varepsilon} \\
 & (0 \leq u_1 \leq \Delta_L) \\
\Longrightarrow & a_x - \frac{2\varepsilon M + (2+2\varepsilon)\Delta_L + (3+2\varepsilon)\Delta_P}{1+\varepsilon} \leq v_2(x) \wedge \\
 & v_2(x) \leq b_x + \frac{2\varepsilon M + (3+\varepsilon)\Delta_L + (4+2\varepsilon)\Delta_P}{1-\varepsilon} \\
 & (M \geq a_x \wedge M \geq b_x) \\
\Longrightarrow & a_x - \Delta \leq v_2(x) \leq b_x + \Delta \\
 & (\frac{2\varepsilon M + (3+\varepsilon)\Delta_L + (4+2\varepsilon)\Delta_P}{1-\varepsilon} < \Delta) \\
\Longrightarrow & v_2(x) \models_{\Delta} [g]_{\Delta}
\end{array}
$$

Now that it is established that $\exists s'_2 \in S_2 : (s'_1, \sigma, s'_2) \in \rightarrow_2$ *we know that:* $s'_2 = (\ell'_1, v_2[R := 0], I_1, 0)$

It remains to prove that $(s'_1, s'_2) \in R$ *which means we must check the five rules of the simulation relation.* $(R1)$,$(R3)$,$(R4)$ *and* $(R5)$ *are clearly true.*

To prove $(R2)$ *we have to prove that*

$$\forall x \in \mathsf{Var} : \begin{cases} \frac{T_1 - r_1[R:=\lfloor T_1 \rfloor_{\Delta_P}](x) - (1+\varepsilon)(\Delta_L+\Delta_P)}{1+\varepsilon} + u_1 \leq v_2[R:=0](x) \\ v_2[R:=0](x) \leq \frac{T_1 - r_1[R:=\lfloor T_1 \rfloor_{\Delta_P}](x) + (1+\varepsilon)(\Delta_L+\Delta_P)}{1-\varepsilon} + u_1 \end{cases}$$

This proposition is the same as $(H2)$ *for* $x \notin R$*. For* $x \in R$*, it amounts to prove:*

$$\frac{T_1 - \lfloor T_1 \rfloor_{\Delta_P} - (1+\varepsilon)(\Delta_L+\Delta_P)}{1+\varepsilon} + u_1 \leq 0$$

and

$$0 \leq \frac{T_1 - \lfloor T_1 \rfloor_{\Delta_P} + (1+\varepsilon)(\Delta_L+\Delta_P)}{1-\varepsilon} + u_1.$$

which is implied by $T_1 - \Delta_P - \lfloor T_1 \rfloor_{\Delta_P} \leq 0 \leq T_1 + \Delta_P - \lfloor T_1 \rfloor_{\Delta_P}$*, which is a consequence of Lemma 3.2, and* $u_1 - \Delta_L \leq 0$*. This establishes* $(R2)$*.*

case (c) *let* $\sigma \in \Sigma^\tau$*.* $\Sigma^\tau = \mathsf{Lab}^\tau \cup \widetilde{\mathsf{Lab}^{\mathsf{in}}} \cup \{\tau\}$*. The proof for the first two sets is similar to the previous case. Let* $\sigma = \tau$*.*

Since $(s_1, \tau, s_1') \in \longrightarrow_1$ *we know by* $(Q5.5)$ *that*

$(K1)$ $s_1' = (\ell_1, r_1, T_1 + u_1', I_1, 0, d_1, \bot)$ *where* $u_1' \in [(1-\varepsilon)u_1, (1+\varepsilon)u_1]$*;*

$(K2)$ $f_1 = \top$ *or*

- *for any* $\widetilde{\sigma}$ *such that* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$*, for any* $(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$*, we have that either* $\lfloor T_1 \rfloor_{\Delta_P} - r_1 \not\models {}_{\Delta_S}[g]_{\Delta_S}$ *or* $I_1(\sigma) \leq u_1$*, and*
- *for any* $\sigma \in \mathsf{Lab}^{\mathsf{out}}$*, for any* $(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$*, we have that* $\lfloor T_1 \rfloor_{\Delta_P} - r_1 \not\models {}_{\Delta_S}[g]_{\Delta_S}$

By rule $(A4.5)$ *of the* AASAP*-semantics, we know that there exists* $s_2' \in S_2$ *such that* (s_2, τ, s_2') *and*

$(K3)$ $s_2' = s_2 = (\ell_1, v_2, I_1, d_1)$*.*

Now we have to prove that $(s_1', s_2') \in R$*.* $(R1), (R3)$ *and* $(R4)$ *are clearly true. Proving* $(R2)$ *amounts to prove that*

$$\forall x \in \mathsf{Var} : \frac{T_1 + u_1' - r_1(x) - (1+\varepsilon)(\Delta_L+\Delta_P)}{1+\varepsilon} \leq v_2(x) \leq \frac{T_1 + u_1' - r_1(x) + (1+\varepsilon)(\Delta_L+\Delta_P)}{1-\varepsilon}$$

which is implied by $(H2)$ *and* $(1-\varepsilon)u_1 \leq u_1' \leq (1+\varepsilon)u_1$ *.*

Let us now prove that there exists s_2'' *s.t.* $((\ell_1, v_2, I_1, d_1), \Delta_L, s_2'') \in \longrightarrow_2$*. According to rule* $(A4.6)$*, it amounts to prove that*

$(L1)$ *for any edge* $(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$ *with* $\sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^{\tau}$*, we have that:*

$$\forall t' : 0 \leq t' \leq \Delta_L : (d_1 + t' \leq \Delta \vee \mathsf{TS}(v_2 + t', g) \leq \Delta)$$

$(L2)$ *for any edge* $(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$ *with* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$*, we have that:*

$$\forall t' : 0 \leq t' \leq \Delta_L : (d_1 + t' \leq \Delta \vee \mathsf{TS}(v_2 + t', g) \leq \Delta \vee (I_1 + t')(\sigma) \leq \Delta)$$

If $f_1 = \top$ *it implies that the program has made a discrete transition during the last loop, which means that* $d_1 \leq \Delta_L$ *and thus that* $d_1 + \Delta_L \leq 2\Delta_L \leq \Delta$ *because we know that, by hypothesis,* $\Delta > 2\Delta_L$*, which makes* $(L1)$ *and* $(L2)$ *true for any* t'*.*

If $f_1 \neq \top$*, the proof is less trivial. We first make a proof for labels of* $(L1)$*.*

$\forall (\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$ *with* $\sigma \in \mathsf{Lab}^{\mathsf{out}}$ *we have* $\lfloor T_1 \rfloor_{\Delta_P} \not\models {}_{\Delta_S}[g]_{\Delta_S}$ *by* $(K2)$*. Let* $a_x = lb(g(x))$ *and* $b_x = rb(g(x))$*. There are two possible cases:*

(a) $\exists x \in \mathsf{Var}$ *such that*

$$\begin{array}{lll}
 & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) < a_x - \Delta_S & \\
 & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) < a_x - \lceil (1+\varepsilon)(\Delta_L + \Delta_P) + \varepsilon M \rceil_{\Delta_P} & \\
 & (\textit{def. of } \Delta_S) & \\
\Longrightarrow & T_1 - r_1(x) - \Delta_P < a_x - (1+\varepsilon)(\Delta_L + \Delta_P) - \varepsilon M & \\
 & (\textit{Lemma 3.2}) & \\
\Longrightarrow & \frac{T_1 - r_1(x) + (1+\varepsilon)(\Delta_L + \Delta_P)}{1-\varepsilon} + u_1 < \frac{a_x + \Delta_P - \varepsilon M}{1-\varepsilon} + u_1 & \\
\Longrightarrow & v_2(x) < \frac{a_x + \Delta_P - \varepsilon M}{1-\varepsilon} + u_1 & \\
 & (H2) & \\
\Longrightarrow & v_2(x) < \frac{a_x + \Delta_P - \varepsilon M}{1-\varepsilon} + \Delta_L & \\
 & (u_1 \leq \Delta_L) & \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : v_2(x) + t' \leq \frac{a_x + \Delta_P - \varepsilon M}{1-\varepsilon} + 2\Delta_L & \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : v_2(x) + t' \leq a_x + \frac{\varepsilon a_x + 2(1-\varepsilon)\Delta_L + \Delta_P - \varepsilon M}{1-\varepsilon} & \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : v_2(x) + t' \leq a_x + \frac{2(1-\varepsilon)\Delta_L + \Delta_P}{1-\varepsilon} & \\
 & (M \geq a_x) & \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : \mathsf{TS}(v_2(x) + t', g(x)) \leq \frac{2(1-\varepsilon)\Delta_L + \Delta_P}{1-\varepsilon} & \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : \mathsf{TS}(v_2(x) + t', g(x)) \leq \Delta & \\
 & (\frac{2(1-\varepsilon)\Delta_L + \Delta_P}{1-\varepsilon} < \Delta) & \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : \mathsf{TS}(v_2 + t', g) \leq \Delta &
\end{array}$$

(b) $\exists x \in \mathsf{Var}$ *such that*

$$
\begin{array}{ll}
 & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) > b_x + \Delta_S \\
 & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) > b_x + \lceil (1+\varepsilon)(\Delta_L + \Delta_P) + \varepsilon M \rceil_{\Delta_P} \\
 & (\textit{def. of } \Delta_S) \\
\Longrightarrow & T_1 - r_1(x) > b_x + (1+\varepsilon)(\Delta_L + \Delta_P) + \varepsilon M \\
 & (\textit{Lemma 3.2}) \\
\Longrightarrow & \frac{T_1 - r_1(x) - (1+\varepsilon)(\Delta_L + \Delta_P)}{1+\varepsilon} + u_1 > \frac{b_x + \varepsilon M}{1+\varepsilon} + u_1 \\
\Longrightarrow & v_2(x) > \frac{b_x + \varepsilon M}{1+\varepsilon} + u_1 \\
 & (H2) \\
\Longrightarrow & v_2(x) > \frac{b_x + \varepsilon M}{1+\varepsilon} \\
 & (u1 \geq 0) \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : v_2(x) + t' > \frac{b_x + \varepsilon M}{1+\varepsilon} \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : v_2(x) + t' > b_x + \frac{-\varepsilon b_x + \varepsilon M}{1+\varepsilon} \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : v_2(x) + t' > b_x + \frac{-\varepsilon M + \varepsilon M}{1+\varepsilon} \\
 & (b_x \leq M) \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : v_2(x) + t' > b_x \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : \mathsf{TS}(v_2(x) + t', g(x)) \leq \Delta \\
 & (v_2(x) + t' \not\models g(x)) \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : \mathsf{TS}(v_2 + t', g) \leq \Delta
\end{array}
$$

Thus, both cases imply that $(L1)$ *is true.*

The proof for $(L2)$ *is the same if we have, by* $(K2)$, $\lfloor T_1 \rfloor_{\Delta_P} \not\models {}_{\Delta_S}[g]_{\Delta_S}$. *If not, we have* $I_1(\sigma) < u_1$ *which also proves* $(L2)$. *Indeed*

$$
\begin{array}{lll}
 & I_1(\sigma) < u_1 & \\
\Longrightarrow & I_1(\sigma) < \Delta_L & (u_1 \leq \Delta_L) \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : I_1(\sigma) + t' < 2\Delta_L & \\
\Longrightarrow & \forall t' : 0 \leq t' \leq \Delta_L : I_1(\sigma) + t' < \Delta & (2\Delta_L \leq \Delta)
\end{array}
$$

case (d) *let* $\sigma \in \mathbb{R}^{\geq 0}$. *For the sake of clarity let us consider that* $\sigma = t$.

Since $(s_1, t, s_1') \in \rightarrow_1$ *we know by* $(Q5.6)$ *that*

$(M1)$ $s_1' = (\ell_1, r_1, T_1, I_1 + t, u_1 + t, d_1 + t, f_1)$;

$(M2)$ $u_1 + t \leq \Delta_L$.

With those facts, we know that there exists $s_2' = (\ell_2', v_2', I_2', d_2') \in S_2$ *such that* $(s_2, t, s_2') \in \rightarrow_2$ *because* $(s_2, \Delta_L - u_1, s_2'') \in \rightarrow_2$ *by* $(H3)$ *and* $t \leq \Delta_L - u_1$ *by* $(M2)$.

Now we have $(s_2, t, s_2') \in \rightarrow_2$ *and we know that:*

$(M3)$ $s_2' = (\ell_1, v_2 + t, I_1 + t, d_1 + t)$

We can now prove that $(s_1', s_2') \in R$. *We have to check the five points of the simulation relation:* $(R1), (R2), (R3)$ *and* $(R4)$ *are easy to prove using hypothesis* $(H1)$ *to* $(H3)$ *and* $(M1)$ *to* $(M3)$.

For $(R5)$, *since by* $(H3)$, *there exists* $s_2'' = (\ell_2'', v_2'', I_2'', d_2'') \in S_2$ *such that* $((\ell_2, v_2, I_2, d_2), \Delta_L - u_1, (\ell_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$, *we have* $((\ell_1, v_2 + t, I_1 + t, d_1 + t), (\Delta_L - u_1 - t), (\ell_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.

One can observe that for $\varepsilon = 0$ we get the same constraint as in Theorem 3.2.

We can now immediately state the following theorem and corollary, which are the counterparts with clock drifts of Theorem 3.3 and Corollary 3.2 of the previous section.

Theorem 3.5 (Simulability with clock drifts)
For any ELASTIC *controller* A, *for any* $\Delta \in \mathbb{Q}^{>0}$, *there exists* $\Delta_L, \Delta_P, \varepsilon \in \mathbb{Q}^{>0}$ *with* $\varepsilon < 1$, *such that* $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L, \Delta_P, \varepsilon} \preceq [\![A]\!]^{\mathsf{AAsap}}_{\Delta}$.

Proof
Let M *be the largest constant a clock is compared with in* A. *For any* $\Delta > 0$, *since parameters* Δ_L, Δ_P *and* ε *are in the non-negative rational numbers, they can always be chosen such that* $\frac{2\varepsilon M + (3+\varepsilon)\Delta_L + (4+2\varepsilon)\Delta_P}{1-\varepsilon} < \Delta$.

And so, given a sufficiently fast hardware with a sufficient precision and a sufficiently small granularity for its clock, we can implement any controller that have been proved correct for the AASAP semantics. This is expressed by the following corollary:

Corollary 3.3 (Implementability with clock drifts)
Let E *be a timed automaton and* A *be an* ELASTIC *controller. For any* $\Delta \in \mathbb{Q}^{>0}$, *such that* $[\![A]\!]^{\mathsf{AAsap}}_{\Delta}$ *controls* $[\![E]\!]$, *there exist* $\Delta_L, \Delta_P, \varepsilon \in \mathbb{Q}^{>0}$ *with* $\varepsilon < 1$, *such that* $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L, \Delta_P, \varepsilon}$ *controls* $[\![E]\!]$.

3.3.4 Verifiability

In this section, we show that the AASAP semantics can be analyzed automatically, since its reachability problem can be reduced to the reachability problem for timed

automata. In this chapter, we remain at a high-level point of view, but in the next chapter, we will tackle the problem of using real model-checker tools like UPPAAL [PL00] or HYTECH [HHWT95a].

The reduction works as follows : for any $\Delta \in \mathbb{Q}^{\geq 0}$, for any ELASTIC controller A, the AASAP semantics of A can be encoded using the classical semantics of a timed automaton $\mathcal{A}^{\Delta}$ constructed from A and Δ.

Theorem 3.6

For any ELASTIC *controller A, for any $\Delta \in \mathbb{Q}^{>0}$, we can construct effectively a timed automaton $\mathcal{A}^{\Delta} = \mathcal{F}(A, \Delta)$ such that $[\![A]\!]_{\Delta}^{\mathsf{AAsap}}$ is strongly bisimilar to $[\![\mathcal{A}^{\Delta}]\!]$.*

Proof

We give the construction of $\mathcal{F}(A, \Delta)$. Let A be the ELASTIC *automaton*

$$\langle \mathsf{Loc}_1, \ell_1^0, \mathsf{Var}_1, \mathsf{Lab}_1, \mathsf{Edg}_1 \rangle$$

and let $\mathcal{F}(A, \Delta)$ be the timed automaton $\langle \mathsf{Loc}_2, \mathsf{Init}, \mathsf{Final}, \mathsf{Inv}_2, \mathsf{Lab}_2, \mathsf{Edg}_2 \rangle$ over the set of clocks Var_2 such that:

- $\mathsf{Loc}_2 = \{(\ell, b) \mid \ell \in Loc_1 \wedge b \in [\Sigma^{\mathsf{in}} \to \{\top, \bot\}]\}$;
- $\mathsf{Init} = \{(\ell_1^0, b_\bot)\}$ *where $b_\bot$ is such that $b_\bot(\sigma) = \bot$ for any $\sigma \in \mathsf{Lab}_2^{\mathsf{in}}$;*
- $\mathsf{Final} = \varnothing$
- $\mathsf{Var}_2 = \mathsf{Var}_1 \uplus \{y_\sigma \mid \sigma \in \Sigma^{\mathsf{in}}\} \cup \{d\}$;
- $\mathsf{Lab}_2 = \underbrace{\mathsf{Lab}_2^{\mathsf{in}}} \uplus \mathsf{Lab}_2^{\mathsf{out}} \uplus \mathsf{Lab}_2^{\tau}$ *and* $\mathsf{Lab}_2^{\mathsf{in}} = \mathsf{Lab}_1^{\mathsf{in}}$, $\mathsf{Lab}_2^{\mathsf{out}} = \mathsf{Lab}_1^{\mathsf{out}}$ *and* $\mathsf{Lab}_2^{\tau} = \mathsf{Lab}_1^{\tau} \cup \widetilde{\mathsf{Lab}_1^{\mathsf{in}}}$;
- Edg_2 *is defined as follows.*

  $((\ell, b), (\ell', b'), {}_{\Delta}[g]_{\Delta}, \sigma, R') \in \mathsf{Edg}_2$ *iff one of the following condition holds:*

  - $\sigma \in \mathsf{Lab}_1^{\mathsf{in}}$ *and*
    1. $\ell' = \ell$
    2. $b(\sigma) = \bot$
    3. $b' = b[\sigma := \top]$
    4. $g = \mathsf{true}$

5. $R' = \{y_\sigma\}$

– $\sigma \in \mathsf{Lab}_1^{\mathsf{in}}$ *and*

1. $\ell' = \ell$
2. $b(\sigma) = \top$
3. $b' = b$
4. $g = \mathsf{true}$
5. $R' = \varnothing$

– $\sigma \in \mathsf{Lab}_1^{\mathsf{out}}$ *and*

1. *there exists* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}_1$
2. $b' = b$
3. $R' = R \cup \{d\}$

– $\sigma \in \mathsf{Lab}_1^{\tau}$ *and*

1. *there exists* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}_1$
2. $b' = b$
3. $R' = R \cup \{d\}$

– $\sigma = \widetilde{\alpha} \in \widetilde{\mathsf{Lab}_1^{\mathsf{in}}}$ *and*

1. *there exists* $(\ell, \ell', g, \alpha, R) \in \mathsf{Edg}_1$
2. $b(\alpha) = \top$
3. $b' = b[\alpha := \bot]$
4. $R' = R \cup \{d\}$

– $\sigma = \tau$ *and*

1. $\ell' = \ell$
2. $b' = b$
3. $g = \mathsf{true}$
4. $R' = \varnothing$

- *The function* Inv_2 *is defined as follows. Let* $EVT((\ell, b)) = \{((\ell, \ell', g, \sigma, R) \in \mathsf{Edg}_1 \mid \sigma \in \widetilde{\mathsf{Lab}_1^{\mathsf{in}}} \wedge b(\sigma) = \top\}$. *Let* $ACT((\ell, b)) = \{(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}_1 \mid \sigma \in \mathsf{Lab}_1^{\mathsf{out}} \cup \mathsf{Lab}_1^{\tau}\}$. *Then* $\mathsf{Inv}_2((\ell, b)) = \varphi_1(\ell, b) \wedge \varphi_2(\ell, b)$ *where*

$$\varphi_1(\ell, b) = \bigwedge_{(\ell,\ell',g,\sigma,R)\in EVT((\ell,b))} \big(d \leq \Delta \vee \neg({}^{\Delta}g) \vee y_\sigma \leq \Delta\big)$$
$$\varphi_2(\ell, b) = \bigwedge_{(\ell,\ell',g,\sigma,R)\in ACT((\ell,b))} \big(d \leq \Delta \vee \neg({}^{\Delta}g)\big)$$

and ${}^{\Delta}g(x)$ *is the expression* $x \in (a + \Delta, b]$ *if* $g(x)$ *is the expression* $x \in [a, b]$.

To establish that the construction above is correct, we proceed as follows. We show that there exists a relation $R \subseteq S \times S'$ *such that*

(1) R *is a simulation relation for* $[\![A]\!]_\Delta^{\mathsf{AAsap}} \preceq [\![\mathcal{F}(A, \Delta)]\!]$

(2) R^{-1} *is a simulation relation for* $[\![\mathcal{F}(A, \Delta)]\!] \preceq [\![A]\!]_\Delta^{\mathsf{AAsap}}$

Let $[\![A]\!]_\Delta^{\mathsf{AAsap}} = (S_1, \mathrm{E}_1, \mathrm{F}_1, \Sigma_1, \rightarrow_1)$ *and* $[\![\mathcal{F}(A, \Delta)]\!] = (S_2, \mathrm{E}_2, \mathrm{F}_2, \Sigma_2, \rightarrow_2)$.
To prove (1) *we can use the simulation relation* $R \subseteq S_1 \times S_2$ *such that*

$$((\ell_1, v_1, I_1, d_1), ((\ell_2, b_2), v_2)) \in R$$

iff:

1. $\ell_1 = \ell_2$
2. *for any* $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, $\begin{cases} b_2(\sigma) = \bot & \text{iff } I_1(\sigma) = \bot \\ b_2(\sigma) = \top \wedge v_2(y_\sigma) = I_1(\sigma) & \text{iff } I_1(\sigma) \neq \bot \end{cases}$
3. $v_{2|\mathsf{Var}_1} = v_1$ *(* $v_{|X}$ *is the restriction of* v *to* X*)*
4. $v_2(d) = d_1$

Let us prove that R *is a simulation relation.*

1. *We want to prove that* $\forall s \in \mathrm{E}_1, \exists s' \in \mathrm{E}_2 : (s, s') \in R$. *The only element in* E_1 *is* $q^1 = (\ell_0^1, v_0, I_0, 0)$ *where* v_0 *is such that for any* $x \in \mathsf{Var} : v_0(x) = 0$, *and* I_0 *is such that for any* $\sigma \in \Sigma^{\mathsf{in}}$, $I_0(\sigma) = \bot$. *On the other hand, the only element of* E_2 *is* $q^2 = ((\ell_0^1, b_\bot), v_0)$ *where* $b_\bot(\sigma) = \bot$ *for any* $\sigma \in \mathsf{Lab}^{\mathsf{in}}{}_2$. *It is easy to check that* $(q^1, q^2) \in R$.

2. *As* $\mathrm{F}_1 = \mathrm{F}_2 = \varnothing$, *we have trivially that* $\forall (s, s') \in R : s \in \mathrm{F}_1 \implies s' \in \mathrm{F}_2$

3. *Let us assume that* $(s_1, s_2) = ((\ell_1, v_1, I_1, d_1), ((\ell_2, b_2), v_2)) \in R$ *and that* $(s_1, \sigma, s_1') \in \rightarrow_1$ *(with* $s_1' = (\ell_1', v_1', I_1', d_1')$*).*

For each value of σ*, we must establish the existence of a state* $s_2' \in S^2$ *such that* $(s_2, \sigma, s_2') \in \rightarrow_2$ *and* $(s_1', s_2') \in R$*.*

Since $(s_1, s_2) \in R$ *we know that:*

$(H0)$ $s_2 = ((\ell_1, b_2), v_2)$

$(H1)$ $v^2_{1|\mathsf{Var}^1} = v_1$*,* $v_2(d) = d_1$*, and* $v_2(y_\sigma) = I_1(\sigma)$ *iff* $I_1(\sigma) \neq \bot$*.*

Let $\sigma \in \mathbb{R}^{\geq 0}$ *(other cases are straightforward and left to the reader). For the sake of clarity, let us consider that* $\sigma = t$*.*

Since $(s_1, t, s_1') \in \rightarrow_1$ *we know by* $(A4.6)$ *that*

$(H2)$ *for any edge* $(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$ *with* $\sigma \in \mathsf{Lab}_1^{\mathsf{out}} \cup \mathsf{Lab}_1^{\tau}$*, we have that:*

$$\forall t' : 0 \leq t' \leq t : (d_1 + t' \leq \Delta \vee \mathsf{TS}(v_1 + t', g) \leq \Delta)$$

$(H3)$ *for any edge* $(\ell_1, \ell', g, \sigma, R) \in \mathsf{Edg}$ *with* $\sigma \in \mathsf{Lab}_1^{\mathsf{in}}$*, we have that:*

$$\forall t' : 0 \leq t' \leq t : d_1 + t' \leq \Delta \vee \mathsf{TS}(v_1 + t', g) \leq \Delta \vee (I_1 + t')(\sigma) \leq \Delta$$

It is easy to prove by $(H1)$ *that:*

- $d_1 + t' \leq \Delta \Leftrightarrow v_2(d) + t' \leq \Delta$
- $(I_1 + t')(\sigma) \leq \Delta \Leftrightarrow v_2(y_\sigma) + t' \leq \Delta$
- $\mathsf{TS}(v_1 + t', g) \leq \Delta \Leftrightarrow v_2 + t' \models \neg({}^{\Delta}g)$

The third proposition can be proved as follows (let $a_x = lb(g(x))$ *and* $b_x = rb(g(x))$*):*

$$
\begin{array}{ll}
 & \mathsf{TS}(v_1 + t', g) \leq \Delta \\
\Leftrightarrow & (v_1 + t' \models g) \implies \exists x \in \mathsf{Var}_1 : v_1(x) + t' - \Delta \leq a_x \\
\Leftrightarrow & (v_1 + t' \models g) \implies \\
 & (\exists x \in \mathsf{Var}_1 : a_x \leq v_1(x) + t' \leq b_x \ \wedge \ v_1(x) + t' - \Delta \leq a_x) \\
\Leftrightarrow & (v_1 + t' \not\models g) \vee (\exists x \in \mathsf{Var}_1 : a_x \leq v_1(x) + t' \leq \min(b_x, a_x + \Delta)) \\
\Leftrightarrow & (\exists x \in \mathsf{Var}_1 : v_1(x) + t' < a_x \vee v_1(x) + t' > b_x) \\
 & \vee (\exists x \in \mathsf{Var}_1 : a_x \leq v_1(x) + t' \leq \min(b_x, a_x + \Delta)) \\
\Leftrightarrow & \exists x \in \mathsf{Var}_1 : v_1(x) + t' \leq a_x + \Delta \vee v_1(x) + t' > b_x \\
\Leftrightarrow & \exists x \in \mathsf{Var}_1 : v_1(x) + t' \notin (a_x + \Delta, b_x] \\
\Leftrightarrow & v_1 + t' \not\models {}^{\Delta}g \\
\Leftrightarrow & v_1 + t' \models \neg({}^{\Delta}g) \\
\Leftrightarrow & v_2 + t' \models \neg({}^{\Delta}g)
\end{array}
$$

Consequently, using $(H2)$ *and* $(H3)$*, we have* $\forall 0 \leq t' \leq t : v_2 + t' \models \mathsf{Inv}_2(\ell_2)$*. Thus, the state* $s'_2 = ((\ell_2, b_2), v'_2)$ *where* $v'_2 = v_2 + t$ *is such that* $(s_2, t, s'_2) \in \rightarrow_2$*.*

The reader can easily check that $(s'_1, s'_2) \in R$*.*

To prove (2) *we can use the simulation relation* R' *such that* $R'(s_2, s_1)$ *iff* $R(s_1, s_2)$*. The proof is similar since we only used equivalence in our reasonings.*

Corollary 3.4
For any Elastic *controller* A*, for any* $\Delta \in \mathbb{Q}^{>0}$*, for any timed automaton* E *, we have that* $[\![A]\!]_{\Delta}^{\mathsf{AAsap}}$ *controls* $[\![E]\!]$ *iff* $[\![\mathcal{F}(A, \Delta)]\!]$ *controls* $[\![E]\!]$*.*

Observe that the reduction to timed automata we propose is not very efficient: indeed, the number of *locations* of the initial Elastic controller is multiplied by $2^{|\mathsf{Lab}^{\mathsf{in}}|}$, that is an value exponential in the number of inputs of the automaton. The obtained model has thus a *description* that is exponentially larger than the initial specification. This is not a problem for the following theoretical discussions, but in practice it does not allow to handle realistic examples. We tackle this problem in the next chapter.

3.4 Discussions

3.4.1 Verification in Practice

In practice, we can use Corollary 3.4 to reduce the controllability problem to a reachability problem. We first construct $\mathcal{F}(A, \Delta)$ (where we can leave Δ as a parameter) and generate a HyTech file with a description of $\mathcal{F}(A, \Delta)$ and E (which encapsulates the description of the bad states). We then use HyTech to synthesize the weakest constraint on Δ that ensures the correctness of the control strategy for the AASAP semantics. In other words, we ask HyTech for which parameter value $[\![\mathcal{F}(A, \Delta)]\!] \parallel [\![E]\!])$ is empty. If HyTech terminates, it returns a linear constraint $\psi(\Delta)$ over Δ which allows to solve the three problems of Definition 3.9:

- **[Fixed]** For a given value $D \in \mathbb{Q}^{\geq 0}$ for the parameter Δ, answering the [Fixed] question amounts to ask if $\psi(D)$ is true.
- **[Existence]** To solve the [Existence] question, it suffices to ask if there exists $D \in \mathbb{Q}^{\geq 0}$ such that $\psi(D)$.
- **[Maximization]** To solve the [Maximization] question, we ask if
  - either there exists $D_{max} \in \mathbb{Q}^{\geq 0}$ such that $\psi(D_{max})$ and $\forall D > D_{max} : \neg\psi(D)$;
  - or there exists $D_{sup} \in \mathbb{Q}^{\geq 0}$ such that $\neg\psi(D_{sup})$ and $\forall 0 \leq D < D_{sup} : \psi(D)$.

All those question are expressed as formulas of the additive theory of the reals and are thus solvable [Wei99] .

In case the HyTech analysis does not terminate, we still have a practical solution for two of the problems of Definition 3.9 if the environment is specified as a timed automaton:

- **[Fixed]** In this case we can obviously respond to the [Fixed] version of the correctness problem as it amounts to a reachability question on timed automata [AD94],

- [**Maximization**] In this case, we can approximate as close as needed the solution of the [Maximization] question thanks to the "faster is better" property of the AASAP semantics: by doing a binary search on the value space of the parameter Δ, using the [Fixed] question, we can approximate the maximal value of Δ for which the controller is correct up to any precision.

To answer those two previous questions, we are not restricted to the use of HyTech or other parametric tools, and we can use for example Uppaal [PL00], as we will see in the next chapter.

Running example If we apply the construction of Corollary 3.4 to our running example (Figure 3.2 and Figure 3.1), we can ask HyTech to establish for which value of Δ, the tube of control strategies defined by the timed automaton obtained by the construction of Corollary 3.4 is valid.

First, consider the case ($\alpha = 1$). The condition of correctness is $\Delta = 0$. We can interpret this as follows: we have a correct model w.r.t to the classical ASAP semantics (the controller imposes the repetition of events ABC, at integer points in time, that is the divergent timed word $((\texttt{ABC})^\omega, \tau)$ with $\tau_{3i} = \tau_{3i+1} = \tau_{3i+2} = i+1$, $i \in \mathbb{N}$). But if we accept a positive *fixed* delay (no matter how small it is) between the event B and the reaction C, we cannot anymore guarantee that the environment will not eventually reach Bad. The insightful reader has maybe noticed that if the delay between B and C can vary (and here decrease), then it is possible to avoid Bad. However, the greatest lower bound of the delays will be zero so that implementation is not possible. For example, a (non-zeno) controller could issue orders such that $\tau = (1, 1, 1\frac{1}{2})(2, 2\frac{1}{2}, 2\frac{3}{4})(3, 3\frac{3}{4}, 3\frac{7}{8})(4, 4\frac{7}{8}, 4\frac{15}{16})\ldots$ Clearly this time sequence, however divergent, is not acceptable as the output of an implementable controller because there is no lower bound on the difference between two consecutive time instants. This shows that some non zeno controllers also require infinitely fast hardware to be implemented. Hence, it must be admitted that the synchrony hypothesis is not only a matter of non-zenoness; in this example, the condition $\Delta = 0$ shows that it is impossible for a finite-speed hardware to implement the controller.

In the second case ($\alpha = 2$), the model is correct for any $\Delta < \frac{1}{3}$. If we assume that the unit of time is the second, Theorem 3.4 then tells us that, to preserve the desired property with a systematic implementation of the Elastic controller (as

described in Section 3.3.2), we should have a platform with loop time Δ_L clock granularity Δ_P and clock precision ε such that $\frac{2\varepsilon M+(3+\varepsilon)\Delta_L+(4+2\varepsilon)\Delta_P}{1-\varepsilon} < 333\text{ms}$, where $M = 2$ is the greatest constant appearing in the controller. For instance, we can implement the controller on the LEGO MINDSTORMS™ platform, since it allows Δ_L to be as low as 6ms, offers a digital clock with $\Delta_P = 1\text{ms}$ and a drift certainly not greater than 1%. We give more details about this platform and automatic generations of code for it in Chapter 5.

3.4.2 Relaxing the AASAP Semantics for more Efficiency

The construction of Theorem 4.1 gives a model that can be used in practice to verify the implementability of a controller (Corollary 3.4). This model is a timed automaton that contains *exactly* the same reachability information than the AASAP semantics. This is why the construction is not as simple as one might expect. Other constructions can be used (for example if it facilitates the verification), at the condition that it can *simulate* the AASAP semantics. Also, if a controller controls an environment E', it ensures that it can control any refinement E of E'. This results from Lemma 2.2, and Theorem 4.1.

Corollary 3.5
For any ELASTIC *controller A, for any $\Delta \in \mathbb{Q}^{>0}$, for any timed automata E' and E such that $[\![E]\!] \preceq [\![E']\!]$; for any timed automaton C such that $[\![\mathcal{F}(A,\Delta)]\!] \preceq [\![C]\!]$, we have that if $[\![C]\!]$ controls $[\![E']\!]$ then $[\![A]\!]_\Delta^{\mathsf{AAsap}}$ controls $[\![E]\!]$.*

In Corollary 3.5, the automaton C is an *over-approximation* of the AASAP semantics. For verification concerns, such a promising over-approximation is, given a timed automaton T, to close and enlarge (left and right) every guard by Δ, this leads to a timed automaton we call $\mathcal{X}(T,\Delta)$. The model $\mathcal{X}(\mathcal{F}(A,\Delta),\Delta)$ has interesting properties. We show in [DDMR04] that the problem to decide whether there exists a rational value Δ such that it controls $\mathcal{X}(E,\Delta)$ to avoid B is decidable.

3.5 Conclusion and Related Works

In this chapter, we have introduced the notion of Almost ASAP semantics for timed automata. This semantics is a relaxation of the usual ASAP semantics: the controller does not have to react instantaneously to events and time-outs: it only has

to do so within a given time bound (that can be leaved as a parameter). We have shown that this semantics is useful to formally reason about the implementability of mathematical models for timed controllers: any controller that has been shown correct for the Almost ASAP semantics can be systematically implemented. The properties that have been proven on the model are transfered to the implementation (without making the synchrony hypothesis) provided that the implementation is executed on a hardware which is sufficiently fast and which uses a sufficiently fine granular digital clock subject to a sufficiently small drift.

We now compare our work with some recent related works and also point out several future research directions.

Related works. In [AFM+02, AFP+03], Yi et al. present a tool called TIMES that generates executable code (C code for the LEGO MINDSTORMS™ platform) from timed automata models. The code is generated with the synchrony hypothesis. This work does not tackle the problem on which we concentrate in this chapter. The properties proved on the models are not guaranteed to be preserved by their code generation. On the other hand, this work also integrates an interesting schedulability analysis. In our work, we have only concentrated on simple control centered programs. In our approach, tasks that perform expensive computing , should be modeled explicitly (with their worst-case execution time for example). This is coherent with the approach they propose.

In [KMTY04], Krčál et al. agree that an event must remain observable during some (usually small but not singular) period. They propose a digitalized semantics for timed automata in order to model the fact that the environment cannot be observed continuously, but only at discrete instants. However, they do not make a formal link with automatic code generation. On the contrary, we propose such a link, and our semantics is more high-level in that it is continuous-time and thus closer to the designer's point of view.

In [AIK+03], Alur et al. introduce a methodology to generate code from hybrid automata. The class of models they consider is larger than the class we consider here, i.e. the ELASTIC controllers. As in the work of Yi et al., they adopt the synchrony hypothesis. Nevertheless, they plan to explore further this translation in order to see how to achieve the translation without the synchrony hypothesis. The work in this thesis should be useful in that context.

In [HKSP03], Henzinger et al. introduce a programming model for real-time embedded controllers called GIOTTO. GIOTTO is an embedded software model that can be used to specify a solution to a given control problem independently of an execution platform but which is closer to executable code than a mathematical model. So, GIOTTO can be seen an intermediary step between mathematical models like hybrid automata and real execution platform.

In [IKL+00], Larsen et al. show how to model code for real-time controllers using UPPAAL models in order to formally verify the code behavior. Usually, they encounter the problem that the obtained description is difficult to analyze because the time unit at the controller level (time slice of the real-time OS for example) is much smaller than the natural time unit of the environment. This leads to what they call *symbolic state space fragmentation*. They proposed in [HL02] a partial solution to that problem. In our framework, we do not encounter that problem. In fact, the larger reaction delay computed during the analysis phase of the AASAP semantics is usually close to the time unit of the environment to control, and usually much larger than the time unit of the hardware on which the control program is executed. The program is generated automatically from the ELASTIC model and is guaranteed to be correct by construction (no need to verify it).

In [AT05], Altisen and Tripakis explicitly refer to the material presented in this Chapter as the motivation of their work. Like us, they want to formally validate the synchrony hypothesis but they underline that the question of implementability for timed automata could be handled without introducing new semantics for timed automata, but by using modelling. In this work, the platform is modelled through a product of timed automata and the program through an untimed automaton. In this framework, the author reveal subtle problems about the replacement of the platform by a faster one (with higher sampling rate) that leads to the non preservation of properties.

To summarize, we could say that our specification language is maybe not as rich as the one offered in other research works but that is the price we pay for fully formal treatment of the *synchrony hypothesis*. The argument most often used in other works is that the speed of the controller is of another scale than the speed of the environment. This may be true in general, but in communication protocols for example, as in the case study of the next chapters, this is not so true anymore.

Chapter 4

Practical Verification of the AASAP Semantics

4.1 Introduction

In the previous chapter we have introduced a new semantics (the AASAP semantics) for a family of timed automata (the ELASTIC controllers). This semantics has the advantage of being implementable. More formally, we have proved that if the AASAP semantics of an ELASTIC controller satisfies some safety property with a value of its parameter strictly greater than zero, then there exists an implementation that satisfies the same safety property. Furthermore, we have proved that the verification of reachability properties is decidable on the AASAP semantics as long as the values of the parameters are fixed, which encompasses the speed of the processor and the granularity and drift of the clocks. This proof of decidability works by reduction to the reachability problem for timed automata but the translation is not very useful in practice since we build from an ELASTIC automaton a timed automaton that is larger by an exponential factor in the number of inputs. This problems stems from the memory used to encode if an event has arrived or not. For every event σ, each location ℓ of the original automaton is split into two locations, to differentiate the cases where σ has arrived or not. Consequently, the resulting automaton has a size multiplied by $2^{\#\mathsf{Lab}^{\mathsf{in}}}$.

In this chapter, we describe a tool that allows the verification of the AASAP semantics in practice. We give reductions of the problem to the reachability problem for timed automata. These new reductions have the advantage of being *compositional*, and avoiding a blowup in the size of the specification. The state space of the model stays the same, but verification tools do not necessarily build the whole

state space and instead build, *on-the-fly*, only the reachable part of it. This gives us the possibility of verifying more complex models.

To encode the AASAP semantics of a controller into a compositional construction in HyTech [HHWT95a], we used heavily the mechanism of Asap edges, that impose that if a transition on a certain label has become possible for a product of automata, the time cannot elapse until this transition has been fired. This mechanism allows to specify *urgency* in a distributed manner, and was very useful for our compositional construction.

The translation to HyTech gave us a first experimentation tool and the power of the HyTech script language allowed us to answer in most cases all kinds of questions presented in Section 3.4.1 ([Fixed], [Existence] and [Maximization]); but rapidly, we also met the limits of the verification engine of this tool. We thus tried to use the tool Uppaal [LPY97], because of its efficiency in the verification of timed automata, although it does not allow to answer parametric questions or to use rectangular automata for specifying the environment.

The design of a compositional construction with Uppaal is also more complicated than with HyTech, since Uppaal main synchronization mechanism only allows *two* automata to synchronize at once. We had to use repeated synchronizations to simulate one transition of the compositional construction for HyTech. Consequently, our translation to Uppaal relies heavily on the use of *committed* locations : if some automata of a product are in a committed location, no transition can take place that does not make one of these automata move. This notion was introduced in [BGK+02] to allow *atomic sequences of transitions*, which is exactly what we are intending to do.

In this chapter we describe the two translations, to the HyTech and Uppaal modelling languages. First, in Section 4.2, we introduce, in a high-level fashion, these modelling languages. Second, in Section 4.3, we give the translation to the HyTech language, for which the proof can be found in [Doy06]. Third, in Section 4.4, we give the translation, built on the previous one, to the Uppaal language and we prove its correctness. In Section 4.5, we give some practical details about our tool. Finally, in Section 4.6 we present a verification case study about the Philips Audio Control Protocol, a well known industrial example.

This chapter is essentially an extended version of [DDR05b].

4.2 Preliminaries

The syntax and semantics for timed automata introduced in the previous chapters were used for theoretical work only. Now, we handle fairly complicated syntax and semantics that are designed to be almost as expressive as both the UPPAAL and HYTECH semantics.

The first extension to the timed automata model of the previous sections is the use of discrete variables, in guards and invariants but also through updates added on the edges. In the following, we will assume that the discrete variables are all bounded along time at any moment. Under this assumption, the discrete variables add no expressive power to timed automata, they are just a useful modelling tool allowing more succinct specifications. They could be encoded using the discrete states of timed automata. The use of discrete variable is possible directly in both HYTECH and UPPAAL.

The other extensions are directly related to the *special features* offered in the tools HYTECH and UPPAAL. The *special features* coming from UPPAAL[1] are the following:

- *committed locations*: if some of the automata in the product are in a *committed* location, no transition can be fired that does not make one of those automata leave its current location (possibly using a self-loop). This notion has been introduced in order to allow atomic sequences of transitions [BGK+02].

- *urgent locations* : if some of the automata in the product are in an *urgent* location, no timed transition can be made before they are all in a non urgent location.

- *urgent labels*: an urgent label allows to flag many locations as urgent at one time. Indeed, every location source of an edge labelled with an urgent label becomes an urgent location, even if the guard of the edge is not satisfied. It is a purely syntactical mechanism not taking into account the valuation of the variables or the states of the other automata.

- *broadcast labels*: in UPPAAL edges are labeled either as output (denoted by an exclamation mark '!' in the figures) or as input (denoted by a question

[1] In version 3.4.11 of UPPAAL.

mark '?' in the figures). An output edge for which the label is broadcast can be fired as soon as its guard is true. When the transition is fired, all automata having an input edge with the same label have to make this transition (By definition, all these *input edges* have guards equal to $\top$.) In other word, it is a rendez-vous between a leader an a group of followers. The leader moves and all followers able to do the same move (possibly none) must do it.

- *pairwise edges*: For a label that is not broadcasted, one needs two automata to offer the transition, one with an input edge with this label, and one with an output edge on this label. It is a synchronous rendez-vous for two automata. This is the main synchronization mechanism of UPPAAL.

The *special features* coming from HYTECH are the following:

- *multiway labels*: for the product to fire a transition with a multiway label σ, all automata having σ in their set of labels must participate. In HYTECH this is the only type of label.

- ASAP edges: the ASAP flag can only be used on a transition labeled with no label or with a multiway label. An Asap transition labeled σ is not urgent until *every* automaton having σ in its alphabet is in a location offering edges labelled by σ.

We are now ready to formally define all these extensions to the model of timed automata. We begin by handling discrete variables.

Definition 4.1 (Rectangular Predicate with Discrete variables)
Let X be a finite set of real-valued variables,usually called clocks*, and D a finite set of integer-valued variables. A valuation for $X \uplus D$ is a function $v : (X \uplus D) \to \mathbb{R}$ such that $v(d) \in \mathbb{Z}, \forall d \in D$. We write $[X \uplus D \to \mathbb{R}]$ for the set of all valuations for $X \uplus D$.*

A closed rectangular guard *over $X \uplus D$ is a finite formula φ_c defined by the following grammar rule:*

$$\varphi_c ::= \bot \mid \top \mid x \leq a \mid x \geq a \mid x = a \mid \varphi_c \wedge \varphi_c$$

where $x \in X \uplus D$ and $a \in \mathbb{Q}$. An open rectangular guard *over $X \uplus D$ is a finite formula*

$$\varphi_o ::= \bot \mid \top \mid x < a \mid x > a \mid \varphi_o \wedge \varphi_o$$

where $x \in X \uplus D$ *and* $a \in \mathbb{Q}$.

We denote by $\mathsf{Rect}_c(X \uplus D)$ *(resp* $\mathsf{Rect}_o(X \uplus D)$*) the class of rectangular predicates built using variables in* $X \uplus D$. *A* rectangular predicate *over* $X \uplus D$ *is a formula* φ *defined by the grammar rule:*

$$\varphi ::= \varphi_c \mid \varphi_o \mid \varphi \wedge \varphi$$

where $\varphi_c \in \mathsf{Rect}_c(X \uplus D)$ *and* $\varphi_o \in \mathsf{Rect}_o(X \uplus D)$. *We denote by* $\mathsf{Rect}(X \uplus D)$ *the class of rectangular predicates.*

Finally, a multi-rectangular predicate *over* $X \uplus D$ *is a formula* φ *defined by the grammar rule:*

$$\varphi ::= \varphi_c \mid \varphi_o \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

where $\varphi_c \in \mathsf{Rect}_c(X \uplus D)$ *and* $\varphi_o \in \mathsf{Rect}_o(X \uplus D)$. *We denote by* $\mathsf{MultRect}$ *the class of multi-rectangular predicates.*

The satisfaction relation $\models$ *is defined as expected.*

In the following, the clocks can only be reset to the value 0 but we allow more general updates for the discrete variables.

Definition 4.2 (Discrete Update)
A discrete update *over* D *is a finite formula defined by the following grammar:*

$$\mathsf{update} ::= d' = \mathsf{rexpr}$$

where

$$\mathsf{rexpr} ::= c \mid c \times d \mid \mathsf{rexpr} + \mathsf{rexpr}$$

where $c \in \mathbb{Z}$ *and* $d \in D$. *We will add sets of such formula on the transitions of timed automata, with the constraint that there is at most one update with left expression* d' *for each* $d \in D$. *This constraint is called the* consistency condition. *The semantics of such a set of updates* Update *relatively to a valuation* v *is a function* $[\![\mathsf{Update}]\!]_v \colon D \to \mathbb{Z}$. *The semantics is defined as follows:*

$$[\![\mathsf{rexpr}]\!]_v = \begin{cases} c & \text{if } \mathsf{rexpr} \equiv c \\ c \times v(d) & \text{if } \mathsf{rexpr} \equiv c \times d \\ [\![\mathsf{rexpr}_1]\!]_v + [\![\mathsf{rexpr}_2]\!]_v & \text{if } \mathsf{rexpr} \equiv \mathsf{rexpr}_1 + \mathsf{rexpr}_2 \end{cases}$$

and

$$[\![\mathsf{Update}]\!]_v\ (d) = \begin{cases} [\![\mathsf{rexpr}]\!]_v & \text{if } \exists\ "d' = \mathsf{rexpr}" \in \mathsf{Update} \\ v(d) & \textit{otherwise} \end{cases}$$

Let $v : (X \uplus D) \to \mathbb{R}$ *be a valuation and* Update *be a set of discrete updates, then* $v[\mathsf{Update}]$ *denotes the valuation* v' *such that* $v'(d) = [\![\mathsf{Update}]\!]_v\ (d), \forall d \in D$ *and* $v'(x) = v(x), \forall x \in X$.

We denote by $\mathsf{Disc}(D)$ *the class of sets of discrete updates built using variables in* D *and respecting the* consistency condition.

In the product of automata we will define shortly, we need the possibility to assign initially non null values to the discrete variables. This is the reason for the following definition:

Definition 4.3 (Discrete Initialization)
A discrete initialization *over* D *is a finite formula defined by the following grammar:*

$$\mathsf{initialize} ::= d' = c$$

where $c \in \mathbb{Z}$ *and* $d \in D$. *The* consistency condition *is extended naturally to a set of discrete initializations. We denote by* $\mathsf{DiscInit}(D)$ *the class of sets of discrete initializations built using variables in* D *and respecting the* consistency condition. *The semantics of* $\mathsf{Initialize} \in \mathsf{DiscInit}(D)$ *is a function* $[\![\mathsf{Initialize}]\!]$: $D \to Z$ *defined as follows:* $[\![\mathsf{Initialize}]\!]\ (d) = c$ *if "*$d' = c$*" belongs to* Initialize *and* 0 *otherwise.*

We make a(nother) little abuse of notation by extending the following notations. Let $v : (X \uplus D) \to \mathbb{R}$ be a valuation, for any $t \in \mathbb{R}^{\geq 0}$, $v + t$ is a valuation such that for any $x \in X$, $(v+t)(x) = v(x)+t$ and for any $d \in D : (v+t)(x) = v(x)$. In other words, the values of the discrete variables are not affected by the passage of time. We define $v - t$ in a similar way.

In the following, for a location $\ell = (\ell_1, \ell_2, \ldots \ell_k)$ of a synchronized product $\langle A_1, A_2, \ldots, A_k, \mathsf{Out}, \mathsf{Broad}, \mathsf{Urg}, \mathsf{Comm}, \mathsf{Mult}, \mathsf{Asap}\rangle$, we define $\ell(A_i) = \ell_i$ for $1 \leq i \leq k$.

Syntax We are now ready to define the syntax of our product of automata. Observe that in this case, because of the special features introduced by both tool, it is easier to define directly the semantics of a product rather than defining the composition as a syntactical operation on timed automata.

Definition 4.4 (Product of Timed automata)
A product of timed automata over a set of clocks X and discrete variables D is a tuple

$$\langle A_1, A_2, \ldots, A_k, \mathsf{Out}, \mathsf{Broad}, \mathsf{Urg}, \mathsf{Comm}, \mathsf{Mult}, \mathsf{Asap}\rangle$$

where

- *each A_i is an automaton, i.e. a tuple $\langle \mathsf{Loc}_i, \mathsf{Init}_i, \mathsf{Final}_i, \mathsf{Inv}_i, \mathsf{Lab}_i, \mathsf{Edg}_i\rangle$ where*
  - *Loc_i is a finite set of locations, denoted by the letter ℓ, representing the discrete states of the automaton. We impose that $\forall 1 \leq i < j \leq k$: $\mathsf{Loc}_i \cap \mathsf{Loc}_j = \varnothing$.*
  - *$\mathsf{Init}_i \in \mathsf{Loc}_i \times \mathsf{DiscInit}(D)$ specifies a constraint $(\ell, \mathsf{Initialize}_i)$ on the initial state of the product. In the initial state (ℓ, v) of the product, the location of the automaton A_i must be $\ell(A_i)$, the valuation v must assigns the value zero to all clocks and a value specified by the set $\mathsf{Initialize}_i$ for the discrete variables. We require that the set $\bigcup_{i=1}^{k} \mathsf{Initialize}_i$ satisfies the consistency condition and that it contains an initialization for every variable of D.*
  - *$\mathsf{Final}_i \subseteq \mathsf{Loc}_i$ is the set of final locations, corresponding to the error states of the automaton.*
  - *$\mathsf{Inv}_i : \mathsf{Loc}_i \to \mathsf{MultRect}(X)$ is the invariant condition. The automaton can stay in location $\ell(A_i)$ as long as $\mathsf{Inv}_i(\ell(A_i))$ is satisfied by the current valuation of the variables.*
  - *Lab_i is a finite alphabet of labels that are used on edges to allow the synchronization between automata.*
  - *$\mathsf{Edg}_i \subseteq \mathsf{Loc}_i \times \mathsf{Loc}_i \times \mathsf{Rect}(X \uplus D) \times \mathsf{Lab}_i \times 2^X \times \mathsf{Disc}(D)$ is a set of edges. An edge $(\ell, \ell', g, \sigma, R, \mathsf{update})$ represents a discrete transition from location ℓ to location ℓ' with* guard *g, label σ , a subset $R \subseteq X$ of the variables to be reset and a set of discrete updates* update *defining the values of the discrete variables in the next state. The guard g is a rectangular predicate that must be satisfied by the current valuation for the transition to be fired. We impose that for all $\sigma \in \big((\cup_{i=1}^{k} \mathsf{Lab}_k)\backslash\{\tau\}\big)$, for any set of edges $\{e_1, e_2, \ldots, e_j\}$ such that each edge belongs to a*

different automaton and is labelled by σ, the union of the update sets of those edges satisfies the consistency condition, that is, the updates do not assign two different values to a discrete variable[2].

- $\mathsf{Out} \subseteq \bigcup_{i=1}^{k} \mathsf{Edg}_i$ *is the set of output edges and we define the set of input edges* $\mathsf{In} = (\bigcup_{i=1}^{k} \mathsf{Edg}_i) \backslash \mathsf{Out}$ *to be its complement;*

- $\mathsf{Broad} \subseteq \bigcup_{i=1}^{k} \mathsf{Lab}_i$ *is the set of* broadcast labels. *We impose*[3] *that each edge in* In *with a label in* Broad *has a guard equal to* $\top$*;*

- $\mathsf{Urg} \subseteq \bigcup_{i=1}^{k} (\mathsf{Loc}_i \cup \mathsf{Lab}_i)$ *is the set of* urgent locations and labels. *We impose, as* Uppaal *that* $\forall 1 \leq i \leq k \cdot \forall \sigma \in \mathsf{Urg} \cdot \forall e = (\ell, \ell', G, \sigma, R) \in Edg_i : G = \top$ *;*

- $\mathsf{Comm} \subseteq \bigcup_{i=1}^{k} \mathsf{Loc}_i$ *is the set of* committed locations*;*

- $\mathsf{Mult} \subseteq \bigcup_{i=1}^{k} \mathsf{Lab}_i$ *is the set of multiway labels. We impose*[4] *that* $\mathsf{Broad} \cap \mathsf{Mult} = \varnothing$*.*

- $\mathsf{Asap} \subseteq \bigcup_{i=1}^{k} \mathsf{Edg}_i$ *is the set of* Asap *edges.*

Observe that this syntax does not cover all possible correct automata for the Uppaal syntax. For example, in our syntax, it is impossible to have and edge that belonging to Out and to In. This syntax does not cover all correct automata for the HyTech syntax neither, since HyTech allows the specification of rectangular automata. Nevertheless, we think that this syntax covers all *interesting* (i.e., practical) subcases of the HyTech and Uppaal syntaxes for timed automata.

Semantics Before defining the formal semantics for this kind of product, we need to define some additional notations. We denote by Asap_i the set $\mathsf{Asap} \cap \mathsf{Edg}_i$. We define similarly Broad_i, Out_i, and Urg_i.

We denote the following semantics for a product of automata P by $[\![P]\!]^{\mathsf{UH}}$ because it is compatible with both Uppaal and HyTech.

[2]This is a restriction to both HyTech and Uppaal syntaxes.

[3]This restriction comes directly from the Uppaal tool. Without it, our translation to the Uppaal language presented in the following could have been a bit simpler.

[4]For labels in Mult, belonging to Out would not matter in the semantics.

Definition 4.5 (Semantics for a Product of Timed automata)
Let

$$P = \langle A_1, A_2, \ldots, A_k, \mathsf{Out}, \mathsf{Broad}, \mathsf{Urg}, \mathsf{Comm}, \mathsf{Mult}, \mathsf{Asap} \rangle$$

be a product of timed automata over the set of clocks X and variables D. The semantics of P is the TTS $[\![P]\!]^{\mathsf{UH}} = (S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow)$ *where:*

- $S = \{(\ell, v) \mid \ell \in (\mathsf{Loc}_1 \times \cdots \times \mathsf{Loc}_k) \wedge v \in [X \uplus D \rightarrow \mathbb{R}] : v \models \bigwedge_{i=1}^{k} \mathsf{Inv}_i(\ell(A_i))\}$;
- $\mathrm{E} = \{(\ell, v) \mid (\ell, v) \in S \wedge \forall x \in X : v(x) = 0 \wedge \forall 1 \leq i \leq k : \mathsf{Init}_i = (\ell(A_i), \mathsf{Initialize}_i) \wedge \forall d \in D : v(d) = [\![\bigcup_{i=1}^{k} \mathsf{Initialize}_i]\!](d)\}$. *Because of the* consistency condition *this set is thus a singleton;*
- $\mathrm{F} = \{(\ell, v) \mid (\ell, v) \in S \wedge \exists 1 \leq i \leq k : \ell(A_i) \in \mathsf{Final}_i\}$;
- $\Sigma = \bigcup_{i=1}^{k} \mathsf{Lab}_i$
- *the transition relation $\rightarrow$ is defined as follows:*
  (a) *For the discrete transitions, $((\ell, v), \sigma, (\ell', v')) \in \rightarrow$ iff*
    - *(***Pairwise***) $\sigma \notin \mathsf{Broad} \wedge \sigma \notin \mathsf{Mult}$ and $\exists i, j \in \{1, \ldots, k\}$ such that $i \neq j$ and*
      - $(\exists c : \ell(A_c) \in \mathsf{Comm}) \implies (\ell(A_i) \in \mathsf{Comm} \vee \ell(A_j) \in \mathsf{Comm})$
      - $\exists (\ell(A_i), \ell'(A_i), G_i, \sigma, R_i, \mathsf{update}_i) \in \mathsf{Out}$
      - $\exists (\ell(A_j), \ell'(A_j), G_j, \sigma, R_j, \mathsf{update}_j) \in \mathsf{In}$
      - $\forall c$ *s.t.* $c \neq i$ *and* $c \neq j : \ell'(A_c) = \ell(A_c)$
      - $v \models (G_i \wedge G_j) \wedge v' := v[R_i \cup R_j := 0][\mathsf{update}_i \cup \mathsf{update}_j]$
    - *(***Broadcast***) $\sigma \in \mathsf{Broad}$ and $\exists i \in \{1, \ldots, k\}, \exists J \subseteq \{1, \ldots, k\} \backslash \{i\}$ and*
      - $(\exists c : \ell(A_c) \in \mathsf{Comm}) \implies (\exists j \in J \cup \{i\} : \ell(A_j) \in \mathsf{Comm})$
      - $\exists (\ell(A_i), \ell'(A_i), G_i, \sigma, R_i, \mathsf{update}_i) \in \mathsf{Out}$
      - $\forall j \in J : \exists (\ell(A_j), \ell'(A_j), G_j, \sigma, R_j, \mathsf{update}_j) \in \mathsf{In}$
      - $\forall c \notin J : c \neq i \implies \big(\ell(A_c) = \ell'(A_c) \wedge \forall e = (\ell(A_c), \ell'(A_c), G_c, \sigma, R_c) : v \not\models G_c\big)$

· $v \models (G_i \wedge (\bigwedge_{j\in J} G_j))$ *and*

$$v' := v[R_i \cup (\bigcup_{j\in J} R_j) := 0][\mathsf{update}_i \cup (\bigcup_{j\in J} \mathsf{update}_j)]$$

∗ *(***Multiway***)* $\sigma \in \mathsf{Mult}$ *and there exists* $I, J \subseteq \{1,\dots,k\}$ *such that* $I \uplus J = \{1,\dots,k\}$ *and*

· $(\exists c : \ell(A_c) \in \mathsf{Comm}) \implies \exists i \in I : \ell(A_i) \in \mathsf{Comm}$

· $\forall j \in J : \sigma \notin \mathsf{Lab}_j$ *and* $\ell(A_j) = \ell'(A_j)$

· $\forall i \in I : \exists(\ell(A_i), \ell'(A_i), G_i, \sigma, R_i, \mathsf{update}_i) \in \mathsf{Edg}_i$ *such that* $v \models G_i \wedge v' = v[\bigcup_{i\in I} R_i := 0][\bigcup_{i\in I} \mathsf{update}_i]$*;*

∗ *(***Silent***)* $\sigma = \tau$ *and* $\exists i \in \{1,\dots,k\}$ *such that*

· $(\exists c : \ell(A_c) \in \mathsf{Comm}) \implies \ell(A_i) \in \mathsf{Comm}$

· $\exists(\ell(A_i), \ell'(A_i), G_i, \tau, R_i, \mathsf{update}_i) \in \mathsf{Edg}_i$

· $\forall c \neq i : \ell(A_c) = \ell'(A_c)$

· $v \models G_i \wedge v' := v[R_i := 0][\mathsf{update}_i]$

(*b*) *For the continuous transitions,* $((\ell, v), t, (\ell', v')) \in \rightarrow$ *iff*

∗ $\ell = \ell'$ *and* $t \in \mathbb{R}^{\geq 0}$ *and* $v' = v + t$ *and* $\forall t' \in [0, t] : v + t' \models \bigwedge_{1\leq i\leq k} \mathsf{Inv}(\ell_i)$ *and* $\forall 1 \leq i \leq k : \ell(A_i) \notin \mathsf{Comm} \cup \mathsf{Urg}$

∗ $\nexists \sigma \in \mathsf{Mult} \cdot \exists I, J : I \uplus J = \{1,\dots k\}$

· $\forall j \in J : \sigma \notin \Sigma_j$

· $\forall i \in I : \exists(\ell(A_i), \ell'_i, G_i, \sigma, R_i) \in Edg_i$

· $\exists i \in I : \exists(\ell(A_i), \ell'_i, G_i, \sigma, R_i) \in \mathsf{Asap}$

∗ $\nexists \sigma \in \mathsf{Urg} \cap \mathsf{Broad} \cdot \exists i \in \{1,\dots,k\} \cdot \exists J \subseteq \{1,\dots,k\}\backslash\{i\}$ *and*

· $\exists(\ell(A_i), \ell'(A_i), G_i, \sigma, R_i) \in \mathsf{Out}$

· $\forall j \in J : \exists(\ell(A_j), \ell'(A_j), G_j, \sigma, R_j) \in \mathsf{In}$

∗ $\nexists \sigma \in \mathsf{Urg}$ *such that* $\sigma \notin \mathsf{Broad} \wedge \sigma \notin \mathsf{Mult}$ *and* $\exists i, j \in \{1,\dots,k\}$ *such that* $i \neq j$ *and*

· $\exists(\ell(A_i), \ell'(A_i), G_i, \sigma, R_i) \in \mathsf{Out}$

· $\exists(\ell(A_j), \ell'(A_j), G_j, \sigma, R_j) \in \mathsf{In}$

Comments on the semantics of the product of automata

- Each rule for a discrete transition begins with the constraint that if some of the automata in the product are in a committed location, then the discrete transition must imply a move of those automata.
- The rule for timed transition is split in several parts :
  - The first part specifies that during the elapse of time, the invariants must stay true at every intermediary moment and that a continuous transition can happen only if no automaton is in a committed or urgent location.
  - The second part specifies that a continuous transition can happen only if there is no ASAP transition available.
  - The third part specifies that a continuous transition can happen only if no urgent transition on a broadcast label is possible;
  - Finally, the fourth part specifies that a continuous transition can happen only if no urgent transition on a pair of pairwise edge with the same label is possible;

We introduce this semantics for the product of automata to facilitate a translation from one formalism to the other in the following. Indeed, the part of the UPPAAL and HYTECH language we use for products of timed automata are subsets of the language in definition 4.4.

First, observe that if no discrete variables are used, Broad, Urg, Comm, and Asap are empty sets, $\mathsf{Mult} = \bigcup_{i=1}^{k} \mathsf{Lab}_i$ and all invariants are in $\mathsf{Rect}(X)$ (and not in MultRect), then this product and its semantics are very similar to the definition 2.21 of the classical product of timed automata. The main difference is that the latter was specified for two automata only.

Second, if Urg, Comm and Broad are empty, $\mathsf{Mult} = \bigcup_{i=1}^{k} \mathsf{Lab}_i$ and all invariants are in $\mathsf{Rect}(X)$ (and not in MultRect), we obtain an HYTECH product of automata, that is a product that can be encoded directly in the HYTECH language.

And finally, if Mult and Asap are empty and all invariants are in $\mathsf{Rect}(X)$ (and not in MultRect), we have an UPPAAL product of automata, that is a product that can be encoded directly in the UPPAAL language.

Remark The compositional translation that follows generate automata with invariants in MultRect, but this is not really a problem since if the invariant of a location ℓ is a multi-rectangle, we can roughly split ℓ in as many location as there are rectangles and link all those location with dummy edges (no guard, silent label, no reset, no update). This problem is thoroughly treated in the thesis of Laurent Doyen [Doy06].

4.3 Compositional Translation to HyTech

We now define the compositional construction that will encode the AASAP semantics of an Elastic controller into a product of automata respecting Definition 4.4.

In this section, we will only consider HyTech automata and we will omit the discrete updates on the edges, since we are not using them in the translation. Furthermore we will not consider the problem of naming conflicts throughout this chapter: each time a new name is created, it is assumed that it is not already the name of an existing part of the automata (e.g. a clock, a location, ...).

The main idea underlying our compositional construction is to treat the incoming events (issued by the environment) independently of the control structure of the Elastic controller, with a product of automata. This leads to technical difficulties that we explain and address in this section. Following the rule (A4.6) of the AASAP semantics (Definition 3.8), defining *almost urgency* of the AASAP semantics, there are essentially three reasons for allowing time to pass:

- either the controller has been in its current location for less than Δ time units,
- or all last untreated occurrences of an event have been issued by the environment less than Δ time units ago,
- or finally the guards of the outgoing transitions have not been enabled for more than Δ time units.

Those conditions will be checked in our compositional construction by respectively A^2, which is a transformation of the Elastic controller A, and two types of widgets: the *event-watchers* and the *guard-watchers*.

In classical timed automata, not UPPAAL or HYTECH ones, there is essentially one way for modeling urgency: invariants on locations. If we have a transition guarded by a lower bound constraint g, it can be forced as soon as it is enabled by adding as invariant in its source location the closure of $\neg g$. E.g. for a guard $x \geq 3$ we can add the invariant $x \leq 3$. This way, time is blocked when the guard is satisfied and the discrete transition is forced. If we enlarge the invariant by Δ ($x \leq 3 + \Delta$), we get the *almost urgency* we need. To formalize this idea, we will need to introduce some more notations:

Additional notations

- Given an ELASTIC controller $A = \langle \mathsf{Loc}, \ell_0, \mathsf{Var}, \mathsf{Lab}, \mathsf{Edg} \rangle$ and a location $\ell \in \mathsf{Loc}$, let $G_{\mathsf{act}}(\ell) = \{g \mid (\ell, \ell', g, \sigma, R) \in \mathsf{Edg} \wedge \sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^{\tau}\}$ be the set of guards labelling output transitions or internal transitions, and for $\alpha \in \mathsf{Lab}^{\mathsf{in}}$, let $G_{\mathsf{evt}}(\ell, \alpha) = \{g \mid (\ell, \ell', g, \alpha, R) \in \mathsf{Edg}\}$ be the set of guards labelling input transitions with event α.

- Then define $\bar{\varphi}_{\mathsf{a}}(\ell) = \bigwedge_{g \in G_{\mathsf{act}}(\ell)} \neg({}_{-\Delta}(g)_0)$ and $\bar{\varphi}_e(\ell, \alpha) = \bigwedge_{g \in G_{\mathsf{evt}}(l,\alpha)} \neg({}_{-\Delta}(g)_0)$.

For example, let $G_{\mathsf{act}}(\ell) = \{2 \leq x \leq 5, 0 \leq y \leq 1\}$, then $\bar{\varphi}_{\mathsf{a}}(l) \equiv (x \leq 2 + \Delta \vee x \geq 5) \wedge (y \leq \Delta \vee y \geq 1)$.

Those constraints will be used as invariant to match the third part of rule ($A5.6$) of the AASAP semantics (see Definition 3.8 for a reminder). The constraint $\bar{\varphi}_{\mathsf{a}}(\ell)$ will be used as an invariant for location ℓ in A^2 to force an output transition when it becomes possible. The constraint $\bar{\varphi}_e(\ell, \alpha)$ will be used in the *guard-watchers*, to ensure that when a guard has been true for enough time, the corresponding transition becomes urgent (as long as it is allowed by other parts of rule ($A5.6$)).

Those invariants are central to our construction, but if we want a compositional construction (a product of automata), invariants are too restrictive to express urgency since urgency also depends on the current state of the other automata offering enabled synchronizations in the product. Hence, we should not block time simply when a transition is enabled in *one* automaton but only when it is enabled in *every* automaton of the product. Therefore, some compositional mechanism is needed to model urgency in a product: we will use the Asap flag in HYTECH

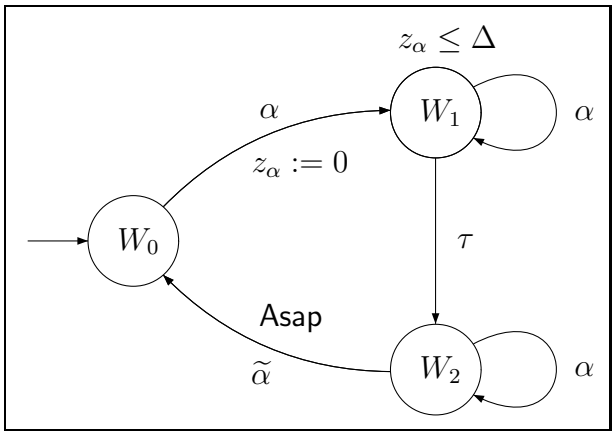

Figure 4.1: Event-Watcher W_α.

automata. Remember that the Asap flag expresses the fact that a transition is urgent as soon as it is enabled in the whole product.

The formal definition of our construction is given in Definition 4.8. From an Elastic controller A and a parameter Δ we construct $\mathcal{F}(A, \Delta)$ as a product of three types of components: event-watchers, guard-watchers and A^2 directly obtained from A. There is one event-watcher W_σ for each event σ of A and there is one guard-watcher W_α^ℓ for each pair of event $\alpha \in \Sigma^{\mathsf{in}}$ and location $\ell \in \mathsf{Loc}$. $\mathcal{F}(A, \Delta)$ is the product of timed automata

$$\langle A^2, W_1, \ldots, W_{i_W}, \mathrm{GW}_1, \ldots, \mathrm{GW}_{i_{\mathrm{GW}}}, \mathsf{Broad}, \mathsf{Urg}, \mathsf{Comm}, \mathsf{Mult}, \mathsf{Asap} \rangle$$

where $\{W_1, \ldots, W_{i_W}\} = \{W_\sigma \mid \sigma \in \Sigma^{\mathsf{in}}\}$ and $\{\mathrm{GW}_1, \ldots, \mathrm{GW}_{i_{\mathrm{GW}}}\} = \{\mathrm{GW}_\alpha^\ell \mid \alpha \in \Sigma^{\mathsf{in}} \wedge \ell \in \mathsf{Loc}\}$.

Broad, Urg, and Comm are empty sets, Mult contains all labels and Asap will be defined in the following. Remark that $\mathcal{F}(A, \Delta)$ is an HyTech automaton.

Event-Watcher Associated to an event $\alpha \in \Sigma^{\mathsf{in}}$, we define W_α (see Figure 4.1) that records the event α. It has a clock z_α encoding the value of $I(\alpha)$ in the AASAP semantics: z_α records the time elapsed since the last untreated event α was issued by the environment (when $I(\alpha) \neq \bot$, the value of the clock z_α is equal to $I(\alpha)$ of the AASAP semantics).

Definition 4.6 (Event-Watcher)
Given an ELASTIC *controller* $A = \langle \mathsf{Loc}^1, \ell_0, \mathsf{Lab}^1, \mathsf{Edg}^1 \rangle$*, for* $\alpha \in \Sigma^{\mathsf{in}}$ *define the timed automaton* Event-Watcher $W_\alpha = \langle \mathsf{Loc}, \{W_0\}, \varnothing, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg} \rangle$ *where:*

- $\mathsf{Loc} = \{W_0, W_1, W_2\}$
- $\mathsf{Inv}(W_0) = \mathsf{Inv}(W_2) = \top$ *and* $\mathsf{Inv}(W_1) = z_\alpha \leq \Delta$
- $\mathsf{Lab} = \{\alpha, \tau, \widetilde{\alpha}\}$
- $\mathsf{Edg} = \{e_1, e_2, e_3, e_4, e_5\}$ *where:*
  - $e_1 = (W_0, W_1, \top, \alpha, \{z_\alpha\})$
  - $e_2 = (W_1, W_2, \top, \tau, \varnothing)$
  - $e_3 = (W_2, W_0, \top, \widetilde{\alpha}, \varnothing)$.
  - $e_4 = (W_1, W_1, \top, \alpha, \varnothing)$
  - $e_5 = (W_2, W_2, \top, \alpha, \varnothing)$

For this automaton, $e_3 \in \mathsf{Asap}$ *and* $e_1, e_2, e_4, e_5 \notin \mathsf{Asap}$.

This widget is intended to record the occurrence of the events α (as expressed by rule $(A5.2)$ in the definition of the AASAP semantics), and then to propose a synchronization on $\widetilde{\alpha}$ with an Asap flag in location W_2. Remember that the notation $\widetilde{\alpha}$ corresponds to the detection of event α by the controller. From the invariant of location W_1, this synchronization will not become urgent before Δ time units.

Guard-Watchers. We introduce *Guard-Watchers* (see Figure 4.2) to monitor the truth value of a set of guards. They are associated to an event $\alpha \in \Sigma^{\mathsf{in}}$ and a location $\ell \in \mathsf{Loc}$. When the controller is not in location ℓ, the guard-watchers $W^\ell_\alpha(G)$ do not influence the execution, being in location U_0 and offering a self-loop synchronization on $\widetilde{\alpha}$. When location ℓ is reached, the synchronization on in_ℓ forces $W^\ell_\alpha(G)$ to enter location U_1 and to become active. The watcher gets back in U_0 as soon as ℓ is exited by out_ℓ. Thus, it is active when it is not in U_0. Its role is then to prevent the label $\widetilde{\alpha}$ to become urgent whenever there is no transition labeled with $\widetilde{\alpha}$ that has been enabled for more than Δ units of time. Hence, we use $W^\ell_\alpha(G)$ with the set of guards $G = \bar{\varphi}_e(\ell, \alpha)$.

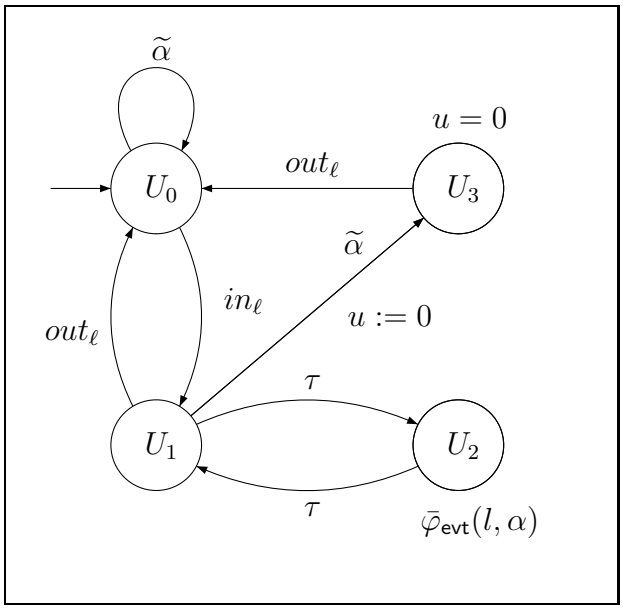

Figure 4.2: Guard-Watcher $W^l_\alpha(\bar{\varphi}_{\mathsf{evt}}(\ell, \alpha))$

Definition 4.7 (Guard-Watchers)
Given an Elastic *controller* $A = \langle \mathsf{Loc}^1, \ell_0, \mathsf{Var}^1, \mathsf{Lab}^1, \mathsf{Edg}^1 \rangle$*, for an event* $\alpha \in \Sigma^{\mathsf{in}}$*, a location* $\ell \in \mathsf{Loc}^1$ *and a set of guards* $G \subseteq \mathsf{Rect}_{\mathsf{c}}(\mathsf{Var})$*, define the timed automaton* Guard-Watcher $W^\ell_\alpha(G) = \langle \mathsf{Loc}, \{U_{\mathsf{init}}\}, \varnothing, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg}, \mathsf{Asap} \rangle$ *where:*

- $\mathsf{Loc} = \{U_0, U_1, U_2, U_3\}$*;*
- $U_{\mathsf{init}} = U_1$ *if* $\ell = \ell_0$ *and* $U_{\mathsf{init}} = U_0$ *otherwise;*
- $\mathsf{Var} = \mathsf{Var}^1 \uplus \{u\}$*;*
- $\mathsf{Inv}(U_0) = \mathsf{Inv}(U_1) := \top$*,* $\mathsf{Inv}(U_2) := \bar{\varphi}_e(\ell, \alpha)$ *and* $\mathsf{Inv}(U_3) := (u = 0)$*;*
- $\mathsf{Lab} = \{\widetilde{\alpha}, in_\ell, out_\ell, \tau\}$*;*
- $\mathsf{Edg} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ *where*
  - $e_1 = \{(U_0, U_1, \top, in_\ell, \varnothing)$*,*
  - $e_2 = (U_1, U_0, \top, out_\ell, \varnothing)$*,*
  - $e_3 = (U_2, U_1, \top, \tau, \varnothing)$*,*

- $e_4 = (U_1, U_2, \top, \tau, \varnothing)$,
- $e_5 = (U_1, U_3, \top, \widetilde{\alpha}, \{u\})$,
- $e_6 = (U_3, U_0, \top, out_\ell, \varnothing)$,
- $e_7 = (U_0, U_0, \top, \widetilde{\alpha}, \varnothing)$

For this automaton, no edge is in the set Asap.

When G is clear from the context, we simply write W^ℓ_α. The guard-watchers $W^\ell_\alpha(G)$ are used to monitor the set of guards G and, in combination with the event-watcher W_α, to make transitions labeled by $\widetilde{\alpha}$ urgent whenever their guard has been satisfied for more than Δ time units (as expressed by rule $(A5.6)$). The guard-watchers $W^\ell_\alpha(G)$ can let time pass in location U_2 when the controller is in location ℓ and there is no enabled transition labeled $\widetilde{\alpha}$. If the controller is not in location ℓ, the guard-watchers $W^\ell_\alpha(G)$ do not influence the execution, being in location U_0 and offering a synchronization on $\widetilde{\alpha}$. Notice that initially, the guard-watchers $W^\ell_\alpha(G)$ are in location U_0 except when $\ell = \ell_0$ is the initial location of A.

Controller transformation

Definition 4.8 (Compositional construction $\mathcal{F}$)
Let $A = \langle \mathsf{Loc}^1, \ell_0^1, \mathsf{Var}^1, \mathsf{Lab}^1, \mathsf{Edg}^1 \rangle$ *be an* ELASTIC *controller where* $\mathsf{Lab}^1 = \mathsf{Lab}^{\mathsf{in}}_1 \uplus \mathsf{Lab}^{\mathsf{out}}_1 \uplus \mathsf{Lab}^{\tau}_1$ *is a structured alphabet.*

The compositional construction $\mathcal{F}(A, \Delta)$, *where* $\Delta \in \mathbb{R}^{\geq 0}$ *is the product*

$$\langle A^2, W_\alpha, \ldots, GW^\ell_\beta(G_{\mathsf{evt}}(\ell, \beta)), \ldots, \mathsf{Broad}, \mathsf{Urg}, \mathsf{Comm}, \mathsf{Mult}, \mathsf{Asap} \rangle$$

where

- *the automata* W_α *are event-watchers for every* $\alpha \in \mathsf{Lab}^{\mathsf{in}}_1$,
- *the automata* $GW^\ell_\beta(G_{\mathsf{evt}}(\ell, \beta))$ *are the guard-watchers for every* $\beta \in \mathsf{Lab}^{\mathsf{in}}_1, \ell \in \mathsf{Loc}^1$,
- A^2 *is the automata* $\langle \mathsf{Loc}^2, \{\ell_0^2\}, \varnothing, \mathsf{Inv}^2, \mathsf{Lab}^2, \mathsf{Edg}^2 \rangle$ *where:*
  - $\mathsf{Loc}^2 = \{PreIn_\ell, In_\ell, Out_\ell, PostOut_{\ell,\ell'} \mid \ell, \ell' \in Loc^1\}$;
  - $\ell_0^2 = In_{\ell_0^1}$;

- *there are no final states;*
- $\mathsf{Var}^2 = \mathsf{Var}^1 \uplus \{d\}$*;*
- $\mathsf{Lab}^2 = \mathsf{Lab}_1^{\mathsf{out}} \uplus \mathsf{Lab}_1^{\tau} \uplus \widetilde{\mathsf{Lab}_1^{\mathsf{in}}} \uplus \{in_\ell, out_\ell\}$*;*
- Edg^2 *contains*
  - *the edges* $(Out_\ell, PostOut_{\ell,\ell'}, {}_\Delta[g]_\Delta, \sigma, R \cup \{d\})$ *such that there exists* $(\ell, \ell', g, \sigma, R) \in \mathsf{Edg}^1$ *with* $\sigma \in \mathsf{Lab}_1^{\mathsf{out}} \cup \mathsf{Lab}_1^{\tau}$
  - *the edges* $(Out_\ell, PostOut_{\ell,\ell'}, {}_\Delta[g]_\Delta, \widetilde{\alpha}, R \cup \{d\})$ *such that there exists* $(\ell, \ell', g, \alpha, R) \in \mathsf{Edg}^1$ *with* $\alpha \in \mathsf{Lab}_1^{\mathsf{in}}$ *and*
  - *the edges* $(PostOut_{\ell,\ell'}, PreIn_{\ell'}, \varnothing, out_\ell, \varnothing)$ *for each* $\ell, \ell' \in Loc^1$*, and the edges* $(PreIn_\ell, In_\ell, \varnothing, in_\ell, \varnothing)$ *and* $(In_\ell, Out_\ell, \varnothing, \tau, \varnothing)$ *for each* $\ell' \in Loc^1$*.*
- *The function* Inv^2 *is defined as follows. For each* $\ell, \ell' \in Loc^1$*,*
  - $\mathsf{Inv}^2(In_\ell) := d \leq \Delta$
  - $\mathsf{Inv}^2(Out_\ell) := d \leq \Delta \vee \bar{\varphi}_{\mathsf{a}}(\ell)$ *and*
  - $\mathsf{Inv}^2(PreIn_\ell) = \mathsf{Inv}^2(PostOut\ell, \ell') := (d = 0)$*.*

- $\mathsf{Broad} = \varnothing$*;*
- $\mathsf{Urg} = \varnothing$*;*
- Mult *is the union of all labels appearing in the product;*
- Asap *contains some edges of the event-watchers, as before; no edge of the modified automaton* A^2 *or of the guard-watchers is in the* Asap *set.*

We illustrate the transformation of the Elastic controller with an example. The timed automaton A^2 corresponding to the Elastic controller A of Figure 4.3 is depicted on Figure 4.4. The automaton A^2 has a similar structure to A. It is used to:

- Guarantee a maximum delay of Δ when location changes (as modeled by the variable d in the AASAP semantics, in rule ($A5.6$) of Definition 3.8), before any action is possible. When A enters in the location ℓ, A^2 enters location In_ℓ, where no action is possible. A^2 can stay in In_ℓ for only Δ time units because of the invariant.

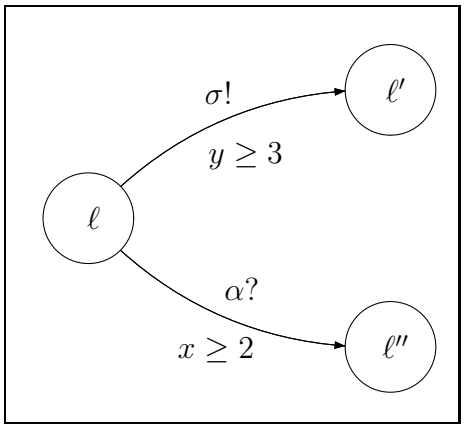

Figure 4.3: An ELASTIC controller A.

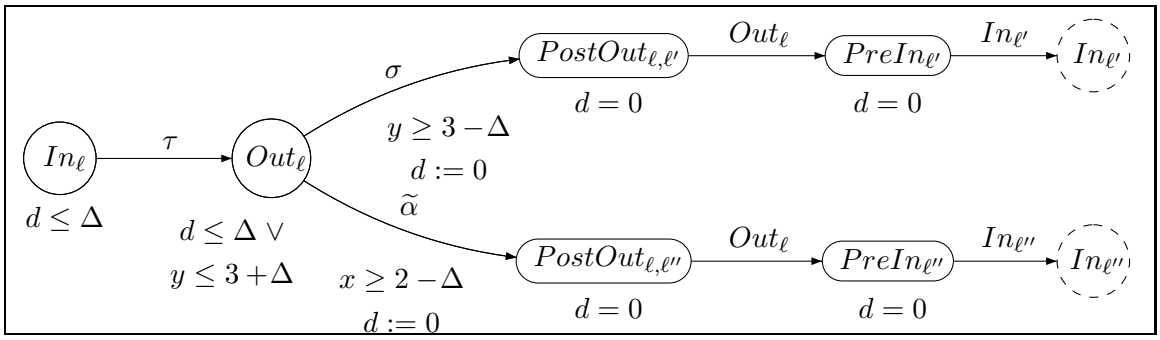

Figure 4.4: The timed automaton A^2 associated to the ELASTIC controller A of Figure 4.3.

- Make transitions labeled with actions $\sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^{\tau}$ urgent when their guard has been satisfied for more than Δ time units, as it is specified in rules $(A5.6)$ of the AASAP semantics[5]. This is obtained through invariant of Out_ℓ.

- Enlarge the guards of the controller's transitions (as expressed by rules $(A5.1)$, $(A5.3)$ and $(A5.4)$).

The soundness of our compositional construction is established by the following theorem.

[5]Notice that the second parts of rule $(A5.3)$ and $(A5.4)$ are encoded by the watchers

Theorem 4.1
For any ELASTIC *controller A, for any rectangular automaton E modelling an environment, for any $\Delta \in \mathbb{Q}^{\geq 0}$, $[\![A]\!]_\Delta^{\mathsf{AAsap}} \parallel [\![E]\!]$ is empty iff $[\![\mathcal{F}(A,\Delta)]\!] \parallel [\![E]\!]$ is empty.*

Proof
The complete proof is given in [Doy06]. It is a classical proof using two simulation relations as witnesses of the mutual similarity of $[\![A]\!]_\Delta^{\mathsf{AAsap}}$ and $[\![\mathcal{F}(A,\Delta)]\!]$ where we hide the in_ℓ and out_ℓ labels (see Definition 2.12 for hiding).

The translation to the HYTECH language has been implemented in a tool called ELASTIC, after the name of the language it handles. We give some practical details about this tool in Section 4.5 but first we describe the second important task of this tool: create models for UPPAAL encapsulating the AASAP semantics of ELASTIC controllers.

4.4 The Translation to UPPAAL

In this section we explain how we translate a HYTECH specification obtained by the compositional construction of the previous section into an UPPAAL specification.

For translating a HYTECH product of automata to an UPPAAL product, we have one major problem : the possibility in HYTECH to make an automaton synchronize with more than one other automata, while pairwise synchronization is the basic mechanism of UPPAAL. This is further complicated when some of the involved edges are in Asap. We will thoroughly handle this problem and then briefly sketch the solutions employed in our tool for other types of synchronizations.

Multiway Labels: Rendez-Vous for Three at Least To handle multiway labels that are shared by more than two automata, we will define a transformation that will add to the product, for each such label σ, an automaton, called a *multiwatcher* (see Figure 4.5, where one can notice that the word "Comm." on the side of a location means that it is commited). This added automaton simply counts, in a discrete variable $\mathsf{counter}_\sigma$, how many automata are in a location in

$\mathsf{Sources}(\sigma)$. We shall denote $\mathsf{MW}(\sigma, i_\sigma)$, the multiwatcher for the label σ, shared by i_σ automata of the original product (see Definition 4.11).

Before describing the transformation, we need some additional definitions:

Definition 4.9
Let $P = \langle A_1, \ldots, A_k, \mathsf{Out}, \mathsf{Broad}, \mathsf{Urg}, \mathsf{Comm}, \mathsf{Mult}, \mathsf{Asap}\rangle$ be a product of timed automata. For this product we define some functions and a set:

- $\mathsf{Sources}(\sigma)$ *is the set of locations ℓ, belonging to any of the automata A_1 to A_k, such that there exists an edge that is outgoing from ℓ and labelled by σ. We also define* $\mathsf{Sources}_i(\sigma)$ *to be* $\mathsf{Sources}(\sigma) \cap \mathsf{Loc}_i$*;*

- $\mathsf{Label}(e)$ *is the label of the edge e;*

- $\mathsf{Reset}(e)$ *is the reset set of the edge e;*

- $\mathsf{ToTreat}$ *is the set of labels σ such that $i_\sigma > 2$ and $\sigma \in \mathsf{Mult}$.*

To allow the counting, we make each automaton of the original product emit a special label $\mathsf{Ready}\sigma$ before entering a location of $\mathsf{Sources}(\sigma)$, and $\mathsf{NotReady}\sigma$ after leaving it by emitting another label than σ. The multiwatcher for σ synchronizes on the σ, Ready_σ and $\mathsf{NotReady}_\sigma$ labels and thus can know how many automata are in a location in $\mathsf{Sources}(\sigma)$.

When the multiwatcher $\mathsf{MW}(\sigma, i_\sigma)$ detects that all the automata sharing σ in their alphabet are in a location in $\mathsf{Sources}(\sigma)$, it switches to location MW_2 from its initial location MW_1. For the product, the transition on σ then becomes possible. In practice, the original transition on the multiway label σ is realized through an (almost) atomic sequence of i_σ pairwise synchronizations on σ, initiated by the multiwatcher. The fact that σ appears always on Asap edges of one of the original automata is reflected through the optional[6] urgent status of the location MW_2. If all transitions on σ were urgent in the original product P, it will also be the case in the transformed product, since when a transition on σ becomes possible, the corresponding multiwatcher will be in an urgent location MW_2.

For one label σ, the partial translation from HyTech to Uppaal of the product P is defined below in the function $\mathsf{TM}_\sigma(P)$ that simply adds the multiwatcher

[6]This explains the square brackets around the word urgent for MW_2 in the Figure 4.6

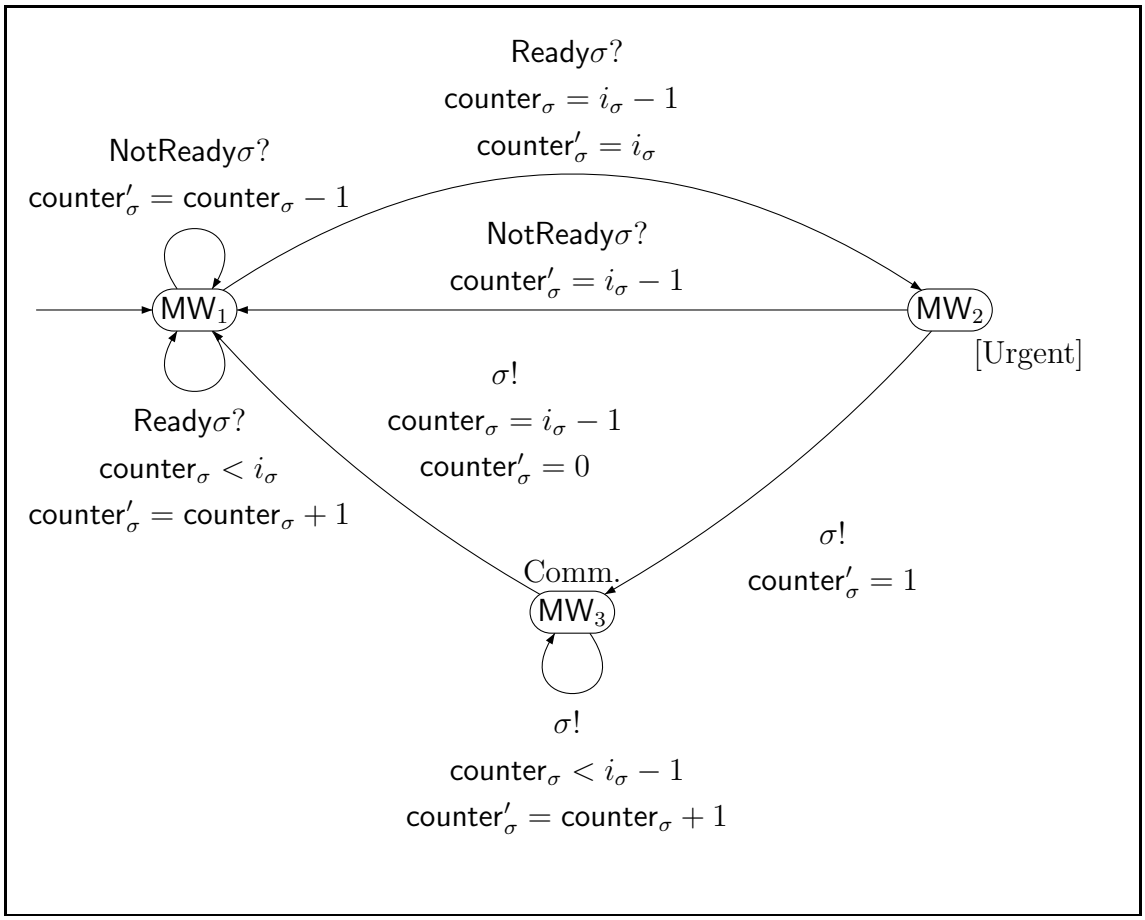

Figure 4.5: MW(i_σ, σ)

MW(σ, i_σ) to the product and applies the function TM$_\sigma(A)$ to each automaton A of the product.

The transformation for a label σ of an automaton A is defined below in the function TM$_\sigma(A)$. The output of this function is an automaton A', where we add some committed locations and some edges that will ensure the emission of the events Readyσ and NotReadyσ. We give an example of the repeated use of this function in Figures 4.7 and 4.8 for the automaton H of Figure 4.6,.

The complete translation to UPPAAL then simply applies iteratively the functions TM$_\sigma(\cdot)$ to each multiway label σ of the specification until all synchronizations are either broadcast or pairwise, which are the types of labels of UPPAAL and there are no more multiway labels, which are the extra type of label of HYTECH.

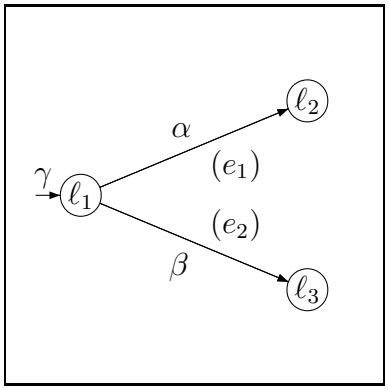

Figure 4.6: The fragment example of an HyTech automaton H to translate to Uppaal

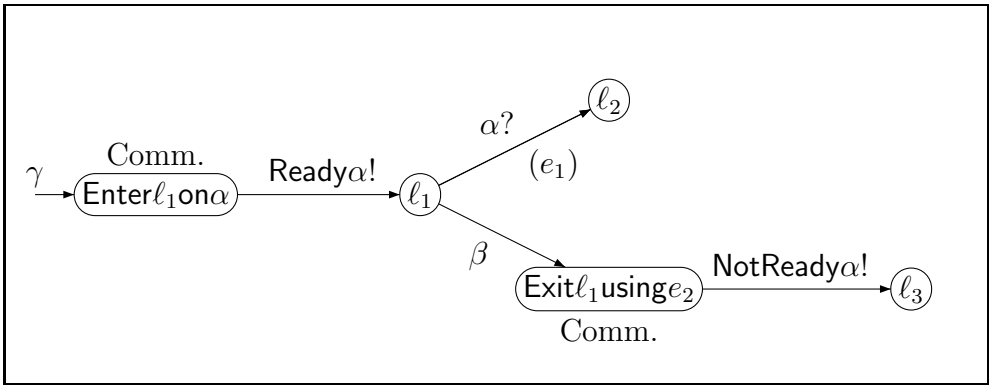

Figure 4.7: Fragment of $\mathsf{TM}_\alpha(H)$

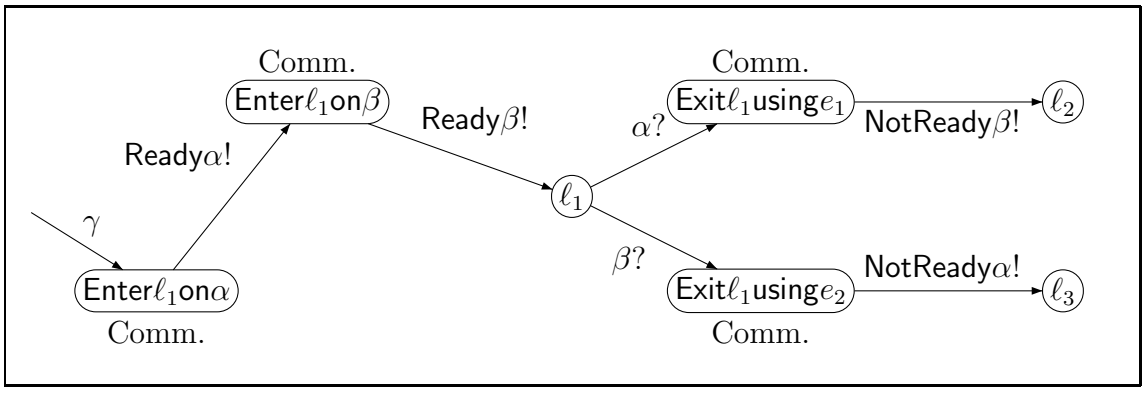

Figure 4.8: Fragment of $\mathsf{TM}_\beta\big(\mathsf{TM}_\alpha(H)\big)$

We are now ready to give the formal definitions of the transformation and to prove it, but first, observe that the translation is not designed to handle *any* HYTECH specification but only a subclass including the HYTECH specification corresponding to the AASAP semantics of an ELASTIC automata as detailed in Section 4.3:

Definition 4.10 (Globally Asap)
A product of of automata $P = \langle A_1, A_2, \ldots, A_k, \mathsf{Out}, \mathsf{Broad}, \mathsf{Urg}, \mathsf{Comm}, \mathsf{Mult}, \mathsf{Asap}\rangle$ *is* globally asap *if for every automaton* A_i*, for every label* σ *in* Lab_i*,*

$$(\exists e \in \mathsf{Asap} \cap \mathsf{Edg}_i : \mathsf{Label}(e) = \sigma)$$

implies

$$(\forall e \in \mathsf{Edg}_i : \mathsf{Label}(e) = \sigma \implies e \in \mathsf{Asap})$$

. In other words, a product of automata needs to be globally Asap *if for all automata, the transitions on a label* σ *are all* Asap *or no one is.*

Furthermore, for every $\sigma \in \mathsf{Mult}$*, let us define* $i_\sigma = \#\{\Sigma_i \mid 1 \leq i \leq k \wedge \sigma \in \Sigma_i\}$*. If* $i_\sigma > 2$*, for every set of edges* $\{e_1, e_2, \ldots, e_{i_\sigma}\}$ *such that* $\forall 1 \leq i \leq i_\sigma : \mathsf{Label}(e_i) = \sigma$ *and all these edges belong to different automata, we impose that all edges, but possibly one, have guards equal to* $\top$*.*

The second part of the definition is added because in the following translation, the firing of a transition labelled by a multiway label σ can be split in an (almost) atomic sequence of pairwise transitions all labelled by σ, and if guards and reset are used in at least two consecutive transitions, the arrival state may differ in the translation from the original product. This condition could be removed if guards were allowed on edges labelled with broadcast edges but this is not the case in the language of UPPAAL.

Observe that the product $\mathcal{F}(A, \Delta)$ obtained in the previous section is *globally* Asap. Indeed, the only edges in Asap are the edges labelled by the viewing of an event, $\widetilde{\alpha}$ for example. For such a label, three types of automata synchronize : the transformed automaton A^2, some guard-watchers and some event-watchers. Since for any event-watcher, the only edge on the viewing of an event is in Asap and there is no edge in Asap for the transformed automaton A^2 and the guard-watchers, $\mathcal{F}(A, \Delta)$ satisfies this part of the definition. The second part of the definition is satisfied too, since only A^2 has constraining guards (for any label).

Definition 4.11 (TU_σ)
Let $P = \langle A_1, \ldots, A_k, \mathsf{Out}, \mathsf{Broad}, \mathsf{Urg}, \mathsf{Comm}, \mathsf{Mult}, \mathsf{Asap}\rangle$ be a product of timed automata over a set of clocks X and a set of discrete variables D. The transformation $\mathsf{TU}_\sigma(P)$ is a product of timed automata over a set of clocks X and a set of discrete variables $D \cup \{\mathsf{counter}_\sigma\}$. This product is defined as follows :

$$\mathsf{TU}_\sigma(P)$$

$$=$$

$$\langle \mathsf{TM}_\sigma(A_1), \ldots, \mathsf{TM}_\sigma(A_k), \mathsf{MW}(\sigma, i_\sigma), \mathsf{Out}', \mathsf{Broad}', \mathsf{Urg}', \mathsf{Comm}', \mathsf{Mult}', \mathsf{Asap}'\rangle$$

where

- $\mathsf{TM}_\sigma(A_i)$ *is* A_i *if* $\sigma \notin \mathsf{Lab}_i$, *and is* $A'_i = \langle \mathsf{Loc}'_i, \mathsf{Init}'_i, \mathsf{Final}'_i, \mathsf{Inv}'_i, \mathsf{Lab}'_i, \mathsf{Edg}'_i\rangle$ *otherwise where*
  - $\mathsf{Loc}'_i = \mathsf{Loc}_i \uplus \{\mathsf{Enter}\ell\mathsf{on}\sigma \mid \ell \in \mathsf{Sources}_i(\sigma)\} \uplus \{\mathsf{Exit}\ell\mathsf{using}e \mid \mathsf{Label}(e) \in \Sigma_i \backslash \{\sigma\} \wedge \ell \in \mathsf{Sources}_i(\sigma)\}$
  - $\mathsf{Init}'_i = \mathsf{Init}_i$
  - $\mathsf{Final}'_i = \mathsf{Final}_i$
  - $\mathsf{Inv}'_i(\ell) = \begin{cases} \mathsf{Inv}_i(\ell) & \text{iff} \quad \ell \in \mathsf{Loc}_i \\ \top & \text{iff} \quad \ell \in (\mathsf{Loc}'_i \backslash \mathsf{Loc}_i) \end{cases}$
  - $\mathsf{Lab}'_i = \mathsf{Lab}_i \uplus \{\mathsf{Ready}\sigma, \mathsf{NotReady}\sigma\}$
  - $\mathsf{Edg}'_i = \begin{cases} \big(\mathsf{Edg}_i \backslash \{e \mid \mathsf{Label}(e) = \sigma\}\big) \\ \uplus \ \{(\mathsf{Enter}\ell\mathsf{on}\sigma, \ell, \top, \mathsf{Ready}\sigma, \varnothing) \mid \ell \in \mathsf{Sources}_i(\sigma)\} \\ \uplus \ \{(\ell, \mathsf{Exit}\ell\mathsf{using}e, G, \beta, R) \mid \\ \quad \exists e = (\ell, \ell', G, \beta, R) \in \mathsf{Edg}_i \wedge \beta \neq \sigma \wedge \ell \in \mathsf{Sources}(\sigma) \\ \quad \} \\ \uplus \ \{(\mathsf{Exit}\ell\mathsf{using}e, \ell'', \top, \mathsf{NotReady}\sigma, \varnothing) \mid \\ \quad \exists e = (\ell, \ell', G, \beta, R) \in \mathsf{Edg}_i \wedge \beta \neq \sigma \wedge \ell \in \mathsf{Sources}_i(\sigma) \\ \quad \wedge \Big((\ell'' = \ell' \wedge \ell' \notin \mathsf{Sources}_i(\sigma)) \\ \quad \ \vee (\ell'' = \mathsf{Enter}\ell\mathsf{on}\sigma \wedge \ell' \in \mathsf{Sources}_i(\sigma))\Big) \\ \quad \} \end{cases}$

- $\mathsf{MW}(\sigma, i)$ *is a timed automaton, called a* multiwatcher *in the following,*

$$\langle \mathsf{Loc_{MW}}, \mathsf{Init_{MW}}, \mathsf{Final_{MW}}, \mathsf{Inv_{MW}}, \mathsf{Lab_{MW}}, \mathsf{Edg_{MW}} \rangle$$

where:

  - $\mathsf{Loc_{MW}} = \{\mathsf{MW}^1_\sigma, \mathsf{MW}^2_\sigma, \mathsf{MW}^3_\sigma\}$*;*
  - $\mathsf{Init_{MW}} = (\mathsf{MW}_j, \{\mathsf{counter}'_\sigma = l\})$ *where*
    - $l = \#\{i \mid \ell_i(A_i) \in \mathsf{Sources}(\sigma)$ *and* ℓ_i *is the location appearing in* $\mathsf{Init}_i\}$
    - MW_j *is* MW_2 *iff* $l = i_\sigma$ *and* MW_1 *otherwise.*
  - $\mathsf{Final_{MW}} = \varnothing$*;*
  - $\mathsf{Inv_{MW}}(\ell) = \top, \forall \ell \in \mathsf{Loc_{MW}}$*;*
  - $\mathsf{Lab_{MW}} = \{\sigma, \mathsf{Ready}\sigma, \mathsf{NotReady}\sigma\}$*;*
  - $\mathsf{Edg_{MW}} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ *where*
    - $e_1 = (\mathsf{MW}^1_\sigma, \mathsf{MW}^2_\sigma, \mathsf{counter}_\sigma = i_\sigma - 1, \mathsf{Ready}\sigma, \varnothing, \{\mathsf{counter}'_\sigma = i_\sigma\})$*;*
    - $e_2 = (\mathsf{MW}^2_\sigma, \mathsf{MW}^1_\sigma, \top, \mathsf{NotReady}\sigma, \varnothing, \{\mathsf{counter}'_\sigma = i_\sigma - 1\})$*;*
    - $e_3 = (\mathsf{MW}^1_\sigma, \mathsf{MW}^1_\sigma, \mathsf{counter}_\sigma < i_\sigma - 1, \mathsf{Ready}\sigma, \varnothing, \{\mathsf{counter}'_\sigma = \mathsf{counter}_\sigma + 1\})$*;*
    - $e_4 = (\mathsf{MW}^1_\sigma, \mathsf{MW}^1_\sigma, \top, \mathsf{NotReady}\sigma, \varnothing, \{\mathsf{counter}'_\sigma = \mathsf{counter}_\sigma - 1\})$*;*
    - $e_5 = (\mathsf{MW}^2_\sigma, \mathsf{MW}^3_\sigma, \top, \sigma, \varnothing, \{\mathsf{counter}'_\sigma = 1\})$*;*
    - $e_6 = (\mathsf{MW}^3_\sigma, \mathsf{MW}^3_\sigma, \mathsf{counter}_\sigma < i_\sigma - 1, \sigma, \varnothing, \{\mathsf{counter}'_\sigma = \mathsf{counter}_\sigma + 1\})$*;*
    - $e_7 = (\mathsf{MW}^3_\sigma, \mathsf{MW}^1_\sigma, \mathsf{counter}_\sigma = i_\sigma - 1, \sigma, \varnothing, \{\mathsf{counter}'_\sigma = 0\})$

- $\mathsf{Out}' = \mathsf{Out}'_1 \uplus \cdots \uplus \mathsf{Out}'_k \uplus \mathsf{Out_{MW}}$ *where*
  - $\mathsf{Out}'_i = (\mathsf{Out}_i \cup \{e \in \mathsf{Edg}'_i \mid \mathsf{Label}(e) = \mathsf{Ready}\sigma \vee \mathsf{Label}(e) = \mathsf{NotReady}\sigma\})$ *for* $1 \le i \le k$*;*
  - $\mathsf{Out_{MW}} = \{e_5, e_6, e_7\}$*;*

- $\mathsf{Broad}' = \mathsf{Broad}$*;*

- $\mathsf{Urg}' = \begin{cases} \mathsf{Urg} \uplus \{\mathsf{MW}^2_\sigma\} & \text{if } \exists e \in \mathsf{Asap} : \mathsf{Label}(e) = \sigma \\ \mathsf{Urg} & \text{if not} \end{cases}$

- $\mathsf{Comm}' = \mathsf{Comm}'_1 \uplus \cdots \uplus \mathsf{Comm}'_k \uplus \mathsf{Comm}_{\mathsf{MW}}$ *where*
  - $\mathsf{Comm}'_i = \mathsf{Comm}_i \cup \{\mathsf{Enter}\ell\mathsf{on}\sigma \mid \ell \in \mathsf{Sources}_i(\sigma)\} \cup \{\mathsf{Exit}\ell\mathsf{using}e \mid \mathsf{Label}(e) \neq \sigma \wedge \ell \in \mathsf{Sources}_i(\sigma)\}$*;*
  - $\mathsf{Comm}_{\mathsf{MW}} = \{\mathsf{MW}^3_\sigma\}$*;*
- $\mathsf{Mult}' = \mathsf{Mult}\backslash\{\sigma\}$*;*
- $\mathsf{Asap}' = \mathsf{Asap}\backslash\{e \mid \mathsf{Label}(e) = \sigma\}$*;*

A multiwatcher automaton is illustrated in Figure 4.5.

We now prove that the repeated application (whatever the ordrer) of the function $\mathsf{TU}(\cdot)$ to a globally asap product of automata will give an equivalent product of automata, in the following sense:

Theorem 4.2
For any globally asap product of automata P with $\mathsf{ToTreat} = \{\sigma_1, \ldots, \sigma_n\}$:
$[\![P]\!]^{\mathsf{UH}}$ *is empty iff* $[\![\mathsf{TU}_{\sigma_n}(\ldots \mathsf{TU}_{\sigma_2}(TU_{\sigma_1}(P)))]\!]^{\mathsf{UH}}$ *is empty.*

Proof
The proof we give is based on a simulation relation. We will work in two steps, first, we will define a transition relation for the transformed product where multiple occurrences of the treated label σ are agglomerated. Then we will prove that the transformed product with the agglomerated relation is in bisimulation relation with the original product. This is a particular case of stuttering bisimulation *[BCG88], that is a simulation relation where a non silent transition of a* TTS *can be matched by multiple non silent transitions of the other one.*

Notations

- *Let* $P = \langle A_1, A_2, \ldots, A_k, \mathsf{Out}, \mathsf{Broad}, \mathsf{Urg}, \mathsf{Comm}, \mathsf{Mult}, \mathsf{Asap}\rangle$ *be the globally asap product of automata to handle.*
- $P' = \mathsf{TU}_{\sigma_n}(\ldots \mathsf{TU}_{\sigma_1}(P))$
  $= \langle A'_1, \ldots, A'_n, \mathsf{MW}^1_\sigma, \ldots, \mathsf{MW}^n_\sigma, \mathsf{Out}', \mathsf{Broad}', \mathsf{Urg}', \mathsf{Comm}', \mathsf{Mult}', \mathsf{Asap}'\rangle$ *where:*
  - *the different automata A'_i are obtained through the repeated use of the function* TM *on A_i: $A'_i = \mathsf{TM}_{\sigma_n}(\ldots \mathsf{TM}_{\sigma_2}(\mathsf{TM}_{\sigma_1}(A_i)))$*

– *the different automata* MW^k_σ *are the* multiwatchers $\mathsf{MW}(\sigma_k, i_{\sigma_k})$

- $[\![P]\!]^{\mathsf{UH}} = (S_1, \mathrm{E}_1, \mathrm{F}_1, \Sigma_1, \rightarrow_1)$
- $[\![P']\!]^{\mathsf{UH}}\ [\{\mathsf{Ready}\sigma_1, \mathsf{NotReady}\sigma_1, \ldots, \mathsf{Ready}\sigma_n, \mathsf{NotReady}\sigma_n\} := \tau] =$

$$(S_2, \mathrm{E}_2, \mathrm{F}_2, \Sigma_2, \rightarrow_2)$$

- *D is the set of discrete variables of P and $D \cup \{\mathsf{counter}_\sigma \mid \sigma \in \mathsf{ToTreat}\}$ is the set of discrete variables of P'.*

There are three essential remarks to understand the construction:

- *In each automaton transformed through possibly multiple application of the function* TM*, an edge $e = (\ell, \ell', g, \sigma, R, \mathsf{update})$ of the original product is replaced by a chain of edges from ℓ to ℓ' that runs through a sequence of committed locations. The first edge of this chain keeps the label σ, the reset R and the update* update *intact. The following edges are all labelled by* Readyλ *or* NotReadyλ *where λ belongs to the alphabet of the original automaton, all those additional edges having empty resets and updates and guards equal to $\top$. See Figure 4.8 for an illustration. To be more specific, an edge $e = (\ell, \ell', g, \sigma, R, \mathsf{update})$ is replaced by a sequence of edges:*

  – *beginning with the edge $(\ell, \ell_1, g, \sigma, R, \mathsf{update})$*

  – *then followed by a sequence $(\ell_i, \ell_{i+1}, \top, \mathsf{NotReady}\lambda, \varnothing, \varnothing)$ for $1 \leq i \leq j-1$, where $\lambda \in \mathsf{ToTreat} \wedge \ell \in \mathsf{Sources}(\lambda)$*

  – *and finished by a sequence $(\ell_i, \ell_{i+1}, \top, \mathsf{Ready}\lambda, \varnothing, \varnothing)$ for $j \leq i \leq l-1$ where $\lambda \in \mathsf{ToTreat} \wedge \ell' \in \mathsf{Sources}(\lambda)$ and $\ell_l = \ell'$;*

  The important constraint about those added edges is that in a sequence of edges of the transformed automaton, for any label λ you can never cross two edges labelled subsequently with Readyλ *without crossing exactly one λ or* NotReadyλ *on the way. This ensures that the added automaton $\mathsf{MW}(\lambda, i_\lambda)$ can keep track of the number of automata that are in a location source of an edge labelled by λ.*

- *Thanks to this mechanism and the design of the function* TU*, every time* i_σ *automata of* P *are in a location where* σ *is possible, then in* P'*, the automata* $\mathsf{MW}(\sigma, i_\sigma)$ *is in its second location (called* MW^2_σ *before). From this location,* $\mathsf{MW}(\sigma, i_\sigma)$ *offers two edges:*

  1. *one edge labelled by* σ*, a label that demands pairwise synchronization and that the current multiwatcher is alone to have in its output set. Said differently, only the multiwatcher can initiate a transition on* σ *by emitting a first time the label and going to the location* MW^3_σ*. It will then be followed by* $i_\sigma - 1$ *other pairwise transition on* σ*, one per automaton synchronizing on* σ*. The atomicity of this sequence of transition is obtained through the fact that the location* MW^3_σ *of the multiwatcher is committed, allowing no automaton that is not in a committed location to move. In fact, the sequence of* σ *is not necessarily atomic, since it can be interleaved with silent* τ *transition, due to the synchronizations on all the* Ready *and* NotReady *labels, but this is not a problem for the simulation of* P *since the sequence of* σ *can not be interleaved by other labels of the original product* P*.*
  2. *one edge labelled by the label* NotReadyσ *that leads back to the location* MW^1_σ *while decreasing the value of* $\mathsf{counter}_\sigma$*. This is used when, finally, a label of the original product that is different from* σ *has been fired.*

- *The* ASAP *status*[7] *of* σ *is taken into account by making location* MW^2_σ *urgent or not. So, if every time* P *was ready to fire a* σ*, time was not allowed to pass anymore, this remains the case in* P'*.*

Given the TTS $[\![P']\!]^{\mathsf{UH}}$*, we define the* agglomerated transition relation $\rightsquigarrow \subseteq S_2 \times (\Sigma_2 \cup \mathbb{R}^{\geq 0}) \times S_2$ *as follows:* $s \rightsquigarrow^\sigma s'$ *iff*

$$\begin{cases} s \xrightarrow{\sigma}_2 s' & \text{and } \sigma \notin \mathsf{ToTreat} \\ s \underbrace{\Rightarrow_2^\sigma \Rightarrow_2^\sigma \cdots \Rightarrow_2^\sigma}_{i_\sigma \text{ repeatitions of } \sigma} s' & \text{and } \sigma \in \mathsf{ToTreat} \end{cases}$$

where $\Rightarrow_2$ *is the stutter-closure of the relation* $\rightarrow_2$*.*

[7] That is, if every edge of P labelled by σ is in ASAP or none is.

The transition relation $\rightsquigarrow$ is meaningful since we can ensure through the construction that one transition of the initial product P is transformed into one step of $\rightsquigarrow$. Furthermore, thanks to the previous remarks, we can limit our interest to the states where, for every automaton A_i of P, the location in P' is a location appearing in the original product P .

Let us thus define S'_2 as the set $\{(\ell, v) \in S_2 \mid \ell(A_i) \in \mathsf{Loc}_i, \forall 1 \leq i \leq k\}$.

We now prove that

$$(S_1, \mathrm{E}_1, \mathrm{F}_1, \Sigma_1, \rightarrow_1) \preceq (S'_2, \mathrm{E}_2, \mathrm{F}_2, \Sigma_2, \rightsquigarrow)$$

and

$$(S'_2, \mathrm{E}_2, \mathrm{F}_2, \Sigma_2, \rightsquigarrow) \preceq (S_1, \mathrm{E}_1, \mathrm{F}_1, \Sigma_1, \rightarrow_1)$$

The main element of the proof is the bisimulation relation we use.

The bisimulation relation *$R \in S_1 \times S'_2$ is the set of pairs $((\ell_1, v_1), (\ell_2, v_2))$ such that:*

1. *$\forall x \in X \uplus D : v_1(x) = v_2(x)$ and $\forall \sigma \in \mathsf{ToTreat} : v_2(\mathsf{counter}_\sigma) = \#\{A_i \mid 1 \leq i \leq k \wedge \ell_1(A_i) \in \mathsf{Sources}(\sigma)\}$*
2. *$\ell_1(A_i) = \ell_2(A_i), \forall 1 \leq i \leq k$ and $\ell_2(\mathsf{MW}(\sigma, i_\sigma)) = \mathsf{MW}^2_\sigma$ if $v_2(\mathsf{counter}_\sigma) = i_\sigma$ and MW^1_σ otherwise.*

Proof of $(S_1, \mathrm{E}_1, \mathrm{F}_1, \Sigma_1, \rightarrow_1) \preceq (S'_2, \mathrm{E}_2, \mathrm{F}_2, \Sigma_2, \rightsquigarrow)$ *We have to prove that all the points of definition 2.9 holds:*

1. *$\forall s^1 \in \mathrm{E}^1, \exists s^2 \in \mathrm{E}^2 : (s^1, s^2) \in R$. This is obvious from the definition.*

2. $\forall (s^1, s^2) \in R : s^1 \in \mathrm{F}^1 \implies s^2 \in \mathrm{F}^2$. *This is obviously true since* $\mathrm{F}^1 = \mathrm{F}^2$;

3. *for all* $(s_1, s'_1) \in R$, *for all* $\lambda \in (\Sigma \backslash \{\tau\}) \cup \mathbb{R}^{\geq 0}$, *for all* s_2 *such that* $(s_1, \lambda, s_2) \in \rightarrow_1$, *there exists* $s'_2 \in S'_2$ *such that* $(s'_1, \lambda, s'_2) \in \rightarrow_2$ *and* $(s'_1, s'_2) \in R$

   (a) $\lambda \in \mathbb{R}^{>0}$. *First observe that for a timed transition, if we denote* $s_1 = (\ell_1, v_1)$ *and* $s'_1 = (\ell_1, v_1 + \lambda)$ *we have* $s_2 = (\ell_2, v_2)$ *where* $\ell_2(A_i) = \ell_1(A_i)$ *for* $1 \leq i \leq k$ *and, for every* $\sigma \in \mathsf{ToTreat}$, $\ell_2(\mathsf{MW}(\sigma, i_\sigma))$ *is* MW^2_σ *if all automata knowing* σ *offer* σ *in their current location or* MW^1_σ *if it is not the case. We now have to prove that* $((\ell_2, v_2), \lambda, (\ell_2, v_2 + \lambda)) \in \rightsquigarrow$. *Let us examine the reason why this transition may not belong to* $\rightsquigarrow$. *First, the invariant could forbid the transition. This is not possible, since all invariants are the same for locations common to* A_1 *to* A_k *and* A'_1 *to* A'_k *and all locations of all multiwatchers have* $\top$ *as invariant. Second,* ℓ_2 *could be a committed or urgent location. This is not possible, since for locations* $\ell(A_1)$ *to* $\ell(A_k)$ *the committed and urgent status have been preserved and for a watcher* $\mathsf{MW}(\sigma, i_\sigma)$, *the current location can be* MW^2_σ *and urgent only if* ℓ_1 *offers an* Asap *transition, which would in both cases mean that no timed transition is possible from* s_1, *which contradicts the fact that* $s_1 \xrightarrow{\lambda} s'_1$.

   (b) $\lambda \in \Sigma_2$. *There are two possible cases: either* $\lambda \in \mathsf{ToTreat}$ *or* $\lambda \notin \mathsf{ToTreat}$. *An important remark is that in both cases, the value of the counters* $\mathsf{counter}_\sigma$ *(for* $\sigma \in \mathsf{ToTreat}$*) will really hold the right value after all committed locations have been leaved, that is the number of automata being in a location offering* σ. *This is obtained by the transformation that forces every automaton to emit an event* Readyσ *before entering a location offering a transition on* Σ *and an event* NotReadyσ *after it. What is left to show is that the transitions are still possible after the transformation. This comes easily in both cases:*

      i. $\lambda \in \mathsf{ToTreat}$. *In this case, the transition is obviously possible because* $\mathsf{MW}(\lambda, i_\lambda)$ *is in* MW^2_λ *where a transition labelled by* λ *is possible, the valuations are the same in both products* P *and* P' *for the variables that are not of the kind* $\mathsf{counter}_\sigma$, *the guard of the first edge of the sequence has been kept and the other edges are subsumed by the transition relation* $\rightsquigarrow$.

ii. $\lambda \notin$ ToTreat. *In this case there is no difficulty either, since the transition is kept as such, except for potential stops in committed location for emitting events of the kind* NotReadyσ *or* Readyσ.

Proof of $(S'_2, \mathrm{E}_2, \mathrm{F}_2, \Sigma_2, \rightsquigarrow) \preceq (S_1, \mathrm{E}_1, \mathrm{F}_1, \Sigma_1, \rightarrow_1)$ *This part of the proof is based on thee same arguments (mainly the preservation of the invariants and the urgent and committed status for the locations) and is left to the reader.*

There remain two cases to manage to completely transform an HYTECH automaton into an UPPAAL automaton:

Rendez-Vous for Two For a label $\sigma \in$ Mult that is shared by exactly two automata A and B in a *globally asap* product of automata, the transformation is straightforward:

- remove σ from Mult;
- put all edges of A labelled by σ in Out;
- put all edges of B labelled by σ in In;
- if there exist an edge $e \in$ Asap such that its label is σ put σ in Urg;

There is no need to add any automaton to the product in this case. To convince oneself of the correctness of this transformation, remember that in a globally asap product, if one edge labelled by σ is in Asap, then all edges labelled by σ are in Asap. Furthermore, in the result of the transformation, as in the original product, one of the automaton can only fire a σ if the other do the same at the same instant.

Multiway Alone For a label $\sigma \in$ Mult that is appearing only in the alphabet of one automaton, there is an easy solution: put the label in the Broad set (since, if it stays pairwise, it would need other automata to synchronize in UPPAAL), and possibly in Urg, if all the corresponding edges are in Asap.

4.5 Tool Suite

We implemented a tool, called ELASTIC after the name of the language it handles, for generating either HYTECH or UPPAAL specifications for the AASAP semantics of an ELASTIC controller. The specification language for an ELASTIC controller is an extension of the HYTECH syntax for the specification of the automata. The script language of our tool is also inspired by HYTECH but is strongly reduced[8].

As an example of the input files for our tool, Figure 4.9 presents the specification of the running example of the previous chapter.

Comments on the specification syntax In the specification syntax for ELASTIC, there are two types of automata, ELASTIC controllers and rectangular automata for the environment. The definitions are respectively introduced either by the keywords `elastic automaton` or `automaton`.

The syntax for the environment automata is exactly the same as HYTECH. We even used a part of the syntactical analyzer of HYTECH as a starting point for our tool[9]. We thus refer the reader to the user guide of HYTECH for this part [HHWT95a].

The syntax of ELASTIC automaton is derived from the syntax of HYTECH rectangular automata. Instead of the declaration of synchronization labels with the keyword `synclabs`, we allow the distinction between events, orders and internal labels through the use of the keywords: `eventlabs`, `orderlabs` and `internlabs`.

Furthermore, we use the keywords `put` and `get` to differentiate edges labelled, respectively, by an event or by an input. This distinction in the syntax of the transition is not strictly necessary, since the type of each label is specified at the top of the specification, but it does make the specification more readable.

Internal labels can be useful for allowing an observer to synchronize on some internal action. At specification time, there is one type of label that is not available for synchronization: the "viewing" input, denoted by a tilde in the previous chapter (like $\widetilde{\alpha}$). It may come in handy to have an observer synchronizing on those symbols. This is why we introduce the directive `view` that allows to fix in advance the name used in practice for the viewing of an event, for example:

[8] Observe that, in fact, the script language for HYTECH is Turing powerful.

[9] Which, by the way, made us program in C.

```
define(alpha, 2) -- or 1
var
w, x, y : clock;
-*----------------------Controller----------------------------------*-
elastic automaton controller

eventlabs : B;
internlabs : ;
orderlabs : A, C;

initially c1  & w=0;

loc c1 :
        when w>=1 put A do {w' = 0} goto c2;
loc c2 :
        when get B & True  goto c3;
loc c3 :
        when True put C goto c1;
end
-*----------------------Environment---------------------------------*-
automaton environment
synclabs: A, B, C;
initially e1 & x=0 & y=0;
loc e1: while True wait {}
         when True sync A do {x' = 0} goto e2;
         when x>=alpha goto Bad;
         when True sync C goto Bad;    -- for input enabledness
loc e2: while y<=1 wait {}
         when y>=1 sync B goto e3;
         when x>=alpha goto Bad;
         when True sync A goto Bad;    -- for input enabledness
         when True sync C goto Bad;    -- for input enabledness
loc e3: while True wait {}
         when True sync C do{y'=0}  goto e1;
         when x>=alpha goto Bad;
         when True sync A goto Bad;    -- for input enabledness
loc Bad: while True wait{}
end

init := param[controller]=1/5 ;
bad  := loc[environment] = Bad ;
```

Figure 4.9: Concrete ELASTIC specification of the running example of Chapter 3.

```
view[up]=getup;
```

With this mechanism, one can add an observer synchronizing on `getup`, as in the case study of the next section.

Two methodological remarks about observers automata need to be made at this point: for the correctness of the whole methodology, they are not allowed to constrain in any way the value of the variables of the environment or controller and they should be input-enabled, so as to allow any transition of the observed automata.

To summarize, the main differences with the HyTech syntax are the following: there is no invariants in Elastic automata, since they are assumed to move AASAP and the labels of an Elastic controller are divided between inputs, outputs and internals.

The script language is basic. Each controller has a parameter bounding its speed, which can be fixed using special directives:

```
init := param[sender]= 0 & param[receiver]=0;
```

The bad states are specified using the keyword `bad` and the same language that HyTech uses to specify regions, for example:

```
bad := loc[checkOutput] = cerror ;
```

The use of the tool is really simple; there are three possible options:

1. `-H` for generating a HyTech specification (this is the default.)
2. `-U` for generating an Uppaal specification (if not specified, values of the parameters are set to 0.)
3. `-C` for generating C code for BrickOs [Nie00]. We will give more details about this code generation in the next chapter.

The workflow of the tool is schematized in Figure 4.10. The arrow from the parameter(s) value to the box standing for `elastic -H` is dashed to mean that those values are optional, since HyTech allows to ask parametric questions where we do not need a value for the parameters. This has been explained in Section 3.4.1.

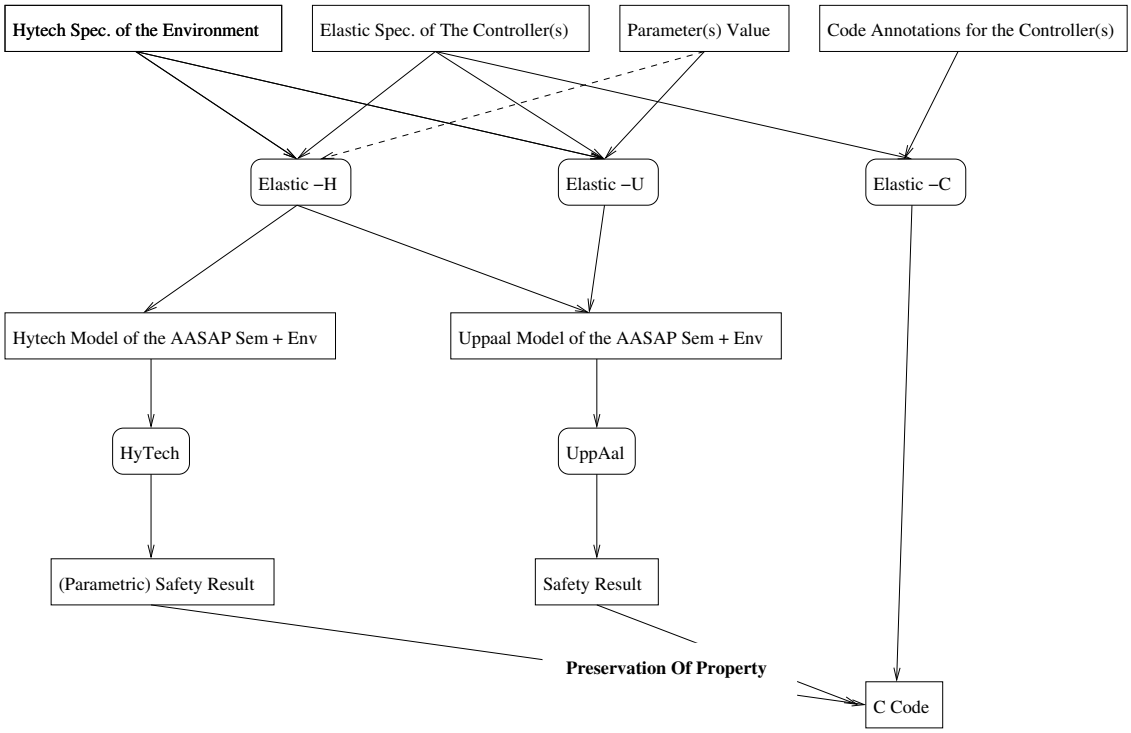

Figure 4.10: ELASTIC workflow.

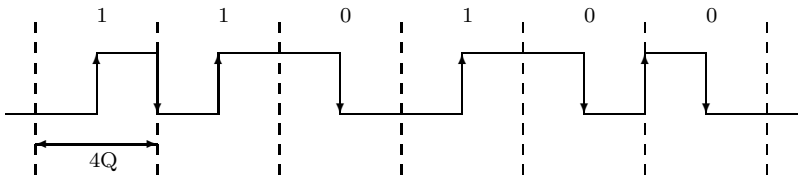

Figure 4.11: Manchester encoding of 110100

The syntax for the code annotations and the process of code generation will be covered in the next chapter.

The source code of the tool is currently (October 2006) available, with a set of examples, at the address:

`http://www.ulb.ac.be/di/ssd/madewulf/aasap/`

We used this tool on the case study of the next section.

4.6 Case Study: the "Philips Audio Control Protocol"

Introduction Bosscher et al study in [BPV94] "a simple protocol for the physical layer of an interface bus that connects the devices of a stereo equipment". This protocol was proposed by Philips engineers. The protocol is based on Manchester encoding to transmit binary sequences on a wire between a single sender and a single receiver. It uses time slots of fixed length.

Manchester encoding uses evenly spaced time slots. To transmit a `1`, the sender must let the signal go from low voltage to high in the middle of a slot and from high to low for a `0`. To repeat a bit, the sender is thus forced between two slots to turn the signal off for a `1` or on for a `0`. The receiver is not able to detect precisely time instants when the signal goes down and then only relies on the UP signals to decode the messages. This implies that a message has to begin by a `1` and that messages ending in `10` or in `1` are not distinguishable without adding information bits. Rather than adding bits, the protocol restricts messages to be either odd in length or to end in `00`. The Manchester encoding of the sequence 110100 is illustrated in Figure 4.11. The *unit of time of the protocol* Q is defined to be a quarter of a time slot needed to send a bit.

The difficulties to implement the protocol are the following:

- although the receiver knows the length of a time slot, it does not know when it begins;
- a receiver does not know the length of the bit string it is receiving;
- only UP signals can be reliably detected by our sensors;
- the sender and the receiver have digital clocks that have a finite granularity, so there will be some imprecision in both sending and receiving times;
- in most operating systems sensors are polled periodically. As a consequence, the time instant at which a bit is perceived can be substantially later than the time instant it has been sent.

The first three difficulties should be solved by the logic of the protocol. The last two difficulties are much lower level, and we would like to forget them when designing a high level version of the protocol. This is exactly what the AASAP semantics allows us to do.

Next, we present the idealized version of the protocol and how we modeled it with two ELASTIC controllers: one for the sender and one for the receiver. Here, the environment is an observer that compares the sequence of bits sent by the sender with the sequence of bits decoded by the receiver. The observer reaches the location *error* whenever the two sequences do not match.

Afterwards, we explain how we can use the AASAP semantics during the verification process and verify the robustness of the protocol. The robustness will allows us to generate code that is correct by construction, as we will explain in the next chapter.

ELASTIC models. Our modelling of the protocol can be found in Figure 4.12 for the sender and in Figure 4.13 for the receiver (the complete ELASTIC specification for this case study is given in Appendix A. There is an additional *observer automaton* playing the role of the environment on Figure 4.14, that allows us to verify the correct transmission of the bits (this observer was proposed by Ho and Wong-Toi in [HWT95]). The unit of time of the model, noted U, is a quarter of the time slot. This unit is not written in the constraints, to alleviate the presentation.

Observe that our modelling uses finite range discrete variables, which were not included in the syntax of ELASTIC controllers in the previous chapter. This is

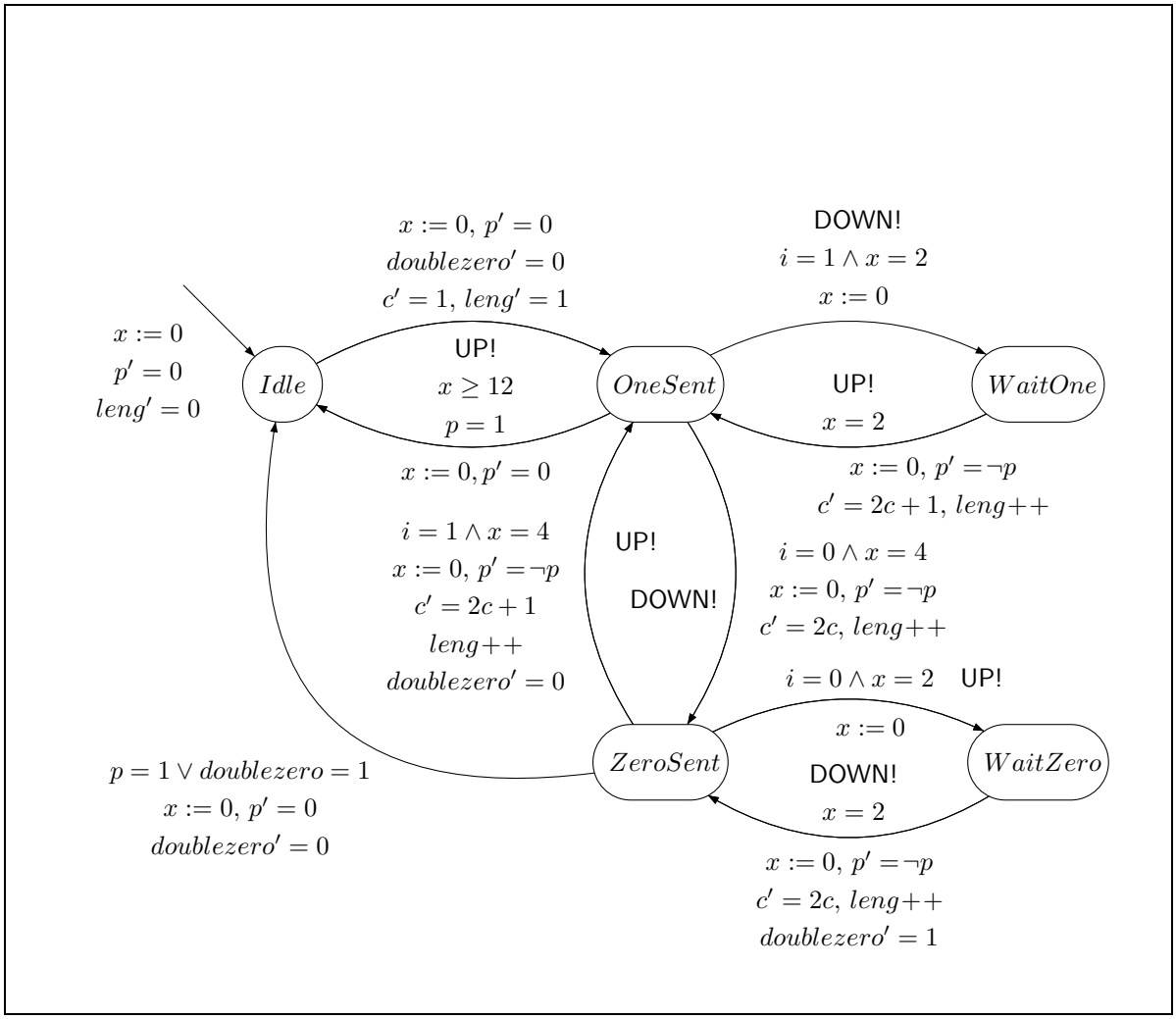

Figure 4.12: The Sender automaton.

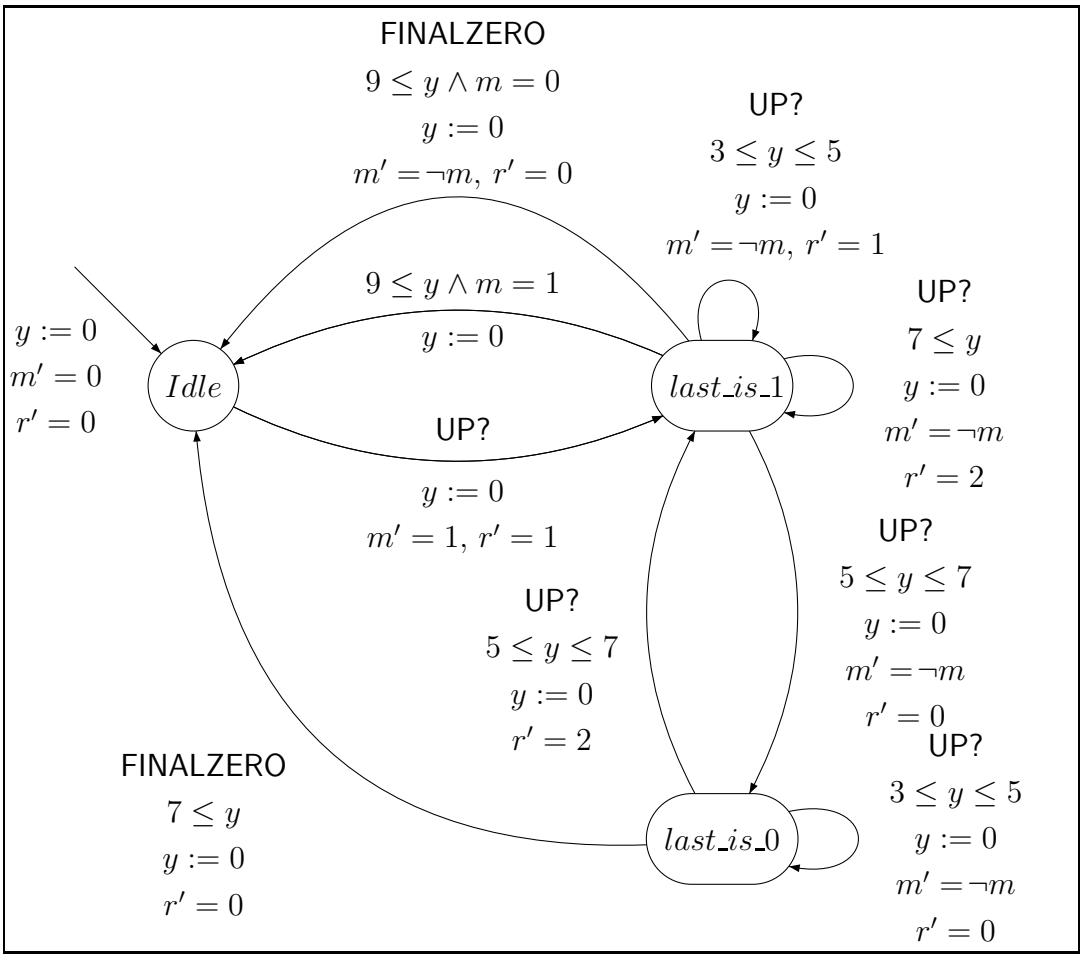

Figure 4.13: The Receiver automaton.

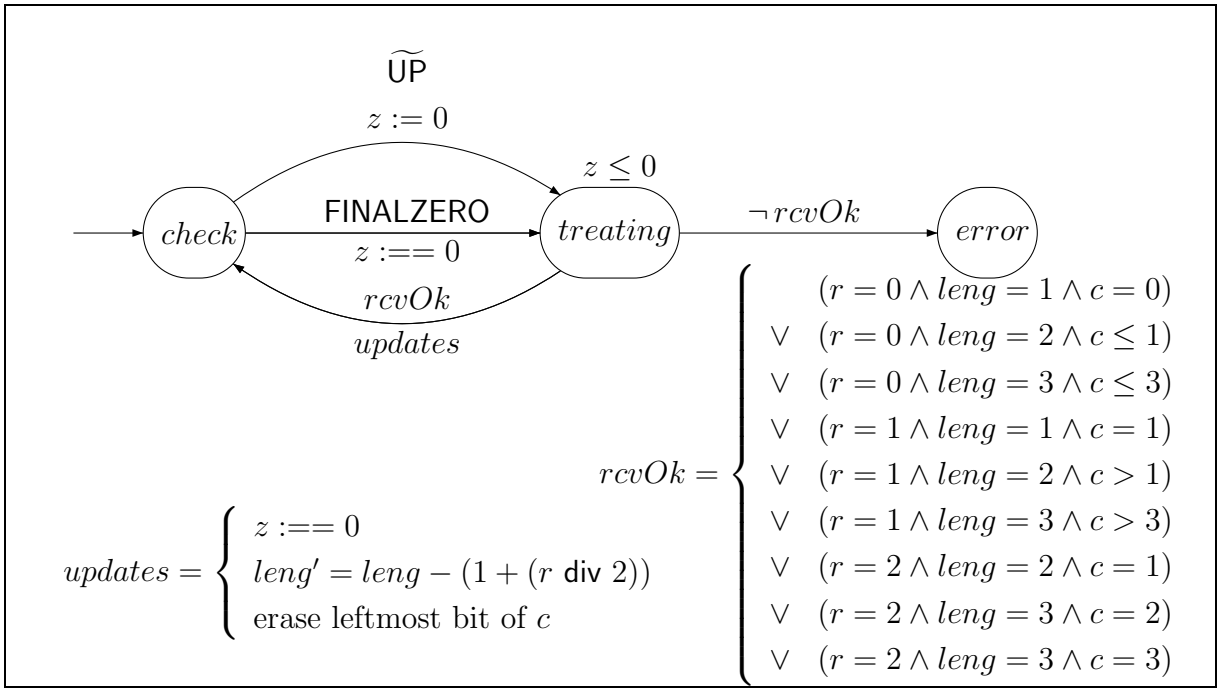

Figure 4.14: The Observer automaton.

not a problem since all those discrete variables are bounded and thus could be encoded in locations. For the sake of clarity, we did not do this. Furthermore, as we have just explained, the tools that we are using allow the use of such finite range discrete variables.

One can easily check that the sender automaton can send any sequence conforming to the protocol restrictions. Arrival in location *OneSent* (*ZeroSent*) means the signal for a 1 (a 0) has just been sent. The clock x is used for the timing of the sequence. The discrete variable i is non-deterministically set to 1 or 0 each time a bit is sent (not shown on the figures). Its value determines which shall be the next bit. The discrete variables p and *doublezero* encode respectively if the current sequence is odd in length and if it ends with 00. Finally, the discrete variables c and *leng* are used to encode the bits that have been sent but not decoded by the receiver yet. c simply encodes in an integer the binary word composed by the last such bits and *leng* is the number of those bits. The decrementing of c and *leng* is done by the observer automaton every time it succeeds in matching a sent bit with a received bit.

The receiver automaton decodes its incoming UP signals by rounding its local time, when it received the signal, to the nearest possible time it expects a signal. This is what makes the protocol robust. If no signal is received in due time, the sequence is interpreted as being complete. The discrete variable m is used to encode the parity of the received sequence. It allows the receiver to know if it has to complete a sequence with an additional 0 to conform to the protocol restrictions. The discrete variable r encodes the one or two bits that were last received. This variable is checked by the observer automaton against c and *leng* of the sender to verify if the sent bits are the same as the received ones. The label FINALZERO does not correspond to an event. It is an internal action done when the receiver understands it must add a 0 to the sequence to end it. The observer automaton then synchronizes on this label to know a new bit has been decoded. As said before, the receiver does not synchronize on DOWN signals.

This modelling uses finite range discrete variables. This is not a problem since all those discrete variables are bounded and thus could be encoded in locations. For the sake of clarity, we did not do this.

Parametric verification Using the transformation of section 4.3 we generate for the sender and the receiver the HyTech specification of their AASAP semantics following Definition 4.8. Those two semantics are noted $[\![\mathsf{Sender}]\!]^{\mathsf{AAsap}}_{\Delta_1}$ and $[\![\mathsf{Receiver}]\!]^{\mathsf{AAsap}}_{\Delta_2}$.

We can first check that if the protocol is executed in an idealized setting, that is for $\Delta_1 = 0$ and $\Delta_2 = 0$, it is correct. This is formalized by the following question: $\mathsf{Reach}([\![\mathsf{Sender}]\!]^{\mathsf{AAsap}}_0 \parallel [\![\mathsf{Receiver}]\!]^{\mathsf{AAsap}}_0 \parallel [\![\mathsf{Observer}]\!]) \cap \mathsf{Bad} \stackrel{?}{\neq} \varnothing$, where Bad are the states in which the observer is in location *error*. With HyTech (or Uppaal), we can easily show that this test is passed successfully by our modelling of the protocol. If this verification had failed then we should have concluded that the protocol was flawed in its logic.

To continue the study of the protocol and determine if it can be implemented, we should check its robustness. In our context, we must determine what are the maximum values of Δ_1 and Δ_2 which ensure that the system $[\![\mathsf{Sender}]\!]^{\mathsf{AAsap}}_{\Delta_1} \parallel [\![\mathsf{Receiver}]\!]^{\mathsf{AAsap}}_{\Delta_2} \parallel [\![\mathsf{Observer}]\!] \cap \mathsf{Bad} = \varnothing$. Those maximal value will be expressed in the unit of time U of the system that we have not fixed so far. Remember U is a quarter of a time slot. By tuning this value, we can then maximize the throughput of the protocol. We should then look for the smallest implementable U on our implementation platform. For BrickOs, the value $\Delta_L U$ (length of the loop in the execution procedure) and $\Delta_P U$ (precision of the clocks) can be set to as low as 6 ms and 1 ms, see next chapter. To guarantee a correct implementation of Sender (and Receiver), we need to have $\Delta > 3\Delta_L + 4\Delta_P$, and so $\Delta U > 22ms$.

So, we know that $\Delta_1 U$ and $\Delta_2 U$ should be strictly below 22 ms. If $\Delta_1 \leq \Delta_2$, the infimum for U is $\frac{22 \text{ ms}}{\Delta_1}$ otherwise it is $\frac{22 \text{ ms}}{\Delta_2}$. Now if we increase the value of one of the parameters Δ_i, the correct value for the other decreases. This is because increasing the parameter value for the AASAP semantics of a controller strictly increases its looseness, forcing the other to be more precise as compensation, which corresponds to a smaller value for its parameter.

Using this fact, we can conclude that the best U for the system will be obtained when Δ_1 and Δ_2 are equal.

Guiding HyTech with this information, by a parametric search, we found that, for ensuring the correctness, the parameters must be strictly less than $\frac{1}{4}U$. This search is not guaranteed to terminate in the general case (remember HyTech uses a semi-algorithm) but it ended here. If it did not, we could have approximated

Tool	Constraint	Result	Time
HyTech	$\Delta_1 + \Delta_2 < 1/2$	Safe	55s
	$\Delta_1 = \Delta_2 = 1/5$	Safe	50s
	$\Delta_1 = \Delta_2 = 1/4$	Unsafe	90s
Uppaal	$\Delta_1 = \Delta_2 = 1/5$	Safe	< 1s
	$\Delta_1 = \Delta_2 = 1/4$	Unsafe	< 1s

Figure 4.15: Execution times for the different models.

this value as close as needed by a bisection search. In fact, we proved that a sufficient condition to avoid the *error* state is that $\Delta_1 + \Delta_2 < \frac{1}{2}$. Our initial guess has been shown correct using HyTech by imposing the constraint and checking that the *error* state is not reachable and then by imposing the negation of the constraint and checking that any possible choice left is bad. Execution times of different analysis are given in Figure 4.15. All experiments were conducted on a Bi processor Linux station (two 3.06Ghz Intel Xeons with 4GB of RAM). Note that to make HyTech terminate, we needed to give some initial constraints. Execution times with Uppaal are very encouraging: the solved problems are simpler as the models are not parametric but this problems are those to be solved in practice as a precise parametric analysis is nice in theory but not required in practice (if the target platform is fixed).

4.7 Conclusion and Related Works

In this chapter, we have shown how we are able to verify the AASAP semantics for case studies that are more than toy examples, using HyTech and Uppaal.

To illustrate the efficiency of our model-checking approach, we tackled the well-studied Philips Audio Protocol. It is interesting to trace the history of this case study. It was first introduced in [BPV94] by Bosscher, Polak, and Vaandrager. The emphasis was on the use of hybrid automata for specification and proof. Their modelling is a bit different from ours, since they explicitly represent drift of the clocks to account for all implementation problems. Under the assumption that this is sufficient to represent all delays in the implementation, they prove the

correctness of the protocol.

The next step was conducted by Ho and Wong-Toi [HHWT95a] who automated the proof using HyTech. One important aspect of their work is that they used their tool to find the maximum drift allowed on the clocks while still preserving the correctness of the protocol. Thanks to the use of an automated tool they were able to experiment more easily with the protocol and to find an improvement, giving more tolerance to clock drifts.

Then, in [BGK+02], Pettersson and Larsen proved an extended version of the protocol, with multiple senders and collision avoidance. The emphasis of this paper is on performances improvement for Uppaal through the introduction of committed locations that allowed to verify more efficiently broadcast communications. We used heavily this idea in this chapter.

The current chapter can be seen as a step forward in the analysis of the protocol, since we were able to perform the verification for implementations with clock drifts, but also, without the synchrony hypothesis. In such a case, where the time scale of the protocol can be very close to the time scale of the platform, it seems important to validate formally this hypothesis. We have also applied our tool to the extended protocol with collision detection, presented by Pettersson and Larsen, and obtained positive verification results, but in this case we are reaching the limits of the verification tool: HyTech is not even able to build the synchronized product of the automata and Uppaal seems to be limited by the memory use, even if we obtained a positive result for the verification when the value of the speed parameters is set to zero. We believe that the state space becomes huge because of the interleaved behavior of the two senders. This could maybe be fixed by using tools using partial order reductions (see [LNZ05] for example).

In the next chapter, we will progress further on this case study in showing how we generated a code that is correct by construction for the Elastic specification of the protocol.

Chapter 5

Practical Real-Time Code Generation

5.1 Introduction

In the previous chapters, we have handled the two most studied aspects of a model-based approach : the specification language (and its semantics) and the verification using model checking. We have put the emphasis on the necessity for the specification to be implementable, i.e. that we can write an implementation that preserves verified properties of the model, as long as we have a sufficiently fast hardware at our disposal.

In this chapter, we handle the practical code generation for real-time controllers specified as ELASTIC controllers. We will ask the designer to provide the code fragments that implement the detection of events, and the commands of our controller. From those *annotations* to the formal model, we will generate correct-by-construction code, under the assumption that the annotations were correct and that the speed of the implementation platform is sufficient. To put our approach in concrete form, we need a platform on which we can perform some case studies. We choose the Lego Mindstorms, running the open source operating system BRICKOS [Nie00].

There is an important line of work in code generation from formal models (e.g. finite automata) when there are no real-time constraints. Compiler construction tools like LEX and YACC could for example be considered as such efforts. For some well known (more or less) formal languages, existing industrial tools (like Rational Rose for UML) perform code generation, but the correctness is in general not guaranteed an still depends on clever manipulations by the designers.

For reactive systems, there exists synchronous languages (like ESTEREL [BC84], LUSTRE [CPHP87], and STATECHARTS [Har87]) that offer rigorous approaches.

For formal models with real-time constraints, in industrial tools, the correctness of the generated implementation, if any, often relies on hand waving arguments. For example, in SIMULINK [Tew02], correctness issues are not satisfactorily handled, more work being put in performance optimization problems. For example, a problem that is not handled there is the scheduling for the generated code, while this is critical for the correctness of the real-time implementation.

This is one methodological problem, but there are many others:

Modelling Language First, as we have seen in the previous chapter, the choice of a good modelling language is not obvious. We need an expressive language which still allows computer aided-verification. For this very reason, hybrid and timed automata are the most frequently used formalisms. In this chapter, we proceed with our ELASTIC language, which are essentially timed automata without invariants.

Nondeterminism As we have seen before, nondeterminism in timed behavior is a prerequisite for a controller to be implementable, if we do not work under the *synchrony hypothesis*. We thus have to find a way for our programs to "implement" this nondeterminism.

Furthermore, nondeterminism is often used in practice because the specification models *all* behaviors of the system, not a particular one. For example, in the audio control protocol of the previous chapter, nondeterminism is used to allow the controller to emit *any* binary message. Obviously, an implementation for which we do not control which message is sent is of no use. In practice, our tool will handle the nondeterminism necessary for implementation in the *timed behavior*, but it is up to the designer to resolve the nondeterminism in the *discrete behavior*. He will have to be careful not to prevent the progress of the controller. One of the original behaviors at least must remain possible at each moment.

Real Time Guarantees from Hardware and Software The platform we use should offer some ways to guarantee real-time properties. The correctness of our programs depends not only on a functional semantics of the implementation

language, as for most discrete programs, but also on the real-time features of the program and thus from the hardware and software that support it. The mode of handling inputs can for example be of great importance: either through polling or interrupts. The worst-case execution times (WCET) of each part of the code has to be computed. This can be really difficult if the structure of the code is complicated through the use of recursion, nested loops, function calls, concurrency and so forth. This is a whole field of research by itself (see [FHL+01] for an example).

Input Handling One problem we will also have to solve is maybe more dependent on our ELASTIC language: the treatment of inputs. In the AASAP semantics, the communication of a controller with the environment is abstracted through the use of input events. In practice, we will seldom have environments feeding the controller with binary information, like buttons pushed, for example. Often, the controller will poll the environment through sensors of temperature, light, speed, and so forth. An event in this case will generally be that a value has exceeded a certain threshold. We must be sure that such an event is not missed because of a polling mechanism. One can imagine a temperature sensor behaving like a medical thermometer that keeps showing the highest reached temperature until it is reset. This kind of sensor can not really miss a threshold but more common sensors proceed by polling the value and could easily miss a threshold by polling the value only before and after the threshold is exceeded, see Figure 5.1 for an illustration.

To ensure that the abstraction we made by using discrete events is correct, we thus have to make some assumptions on the mechanism that generates them and prove by some external argument that our reasoning is correct.

Handling Various Platforms For the methodology to be useful, we should be able to generate code for various platforms. As a first example, we generated code for the BRICKOS platform, but other works have been conducted about code generation from ELASTIC controllers to other platforms, like RTAI, a real-time operating systems for personal computers [Maq06]. The main ideas explained in this chapter apply for other platform.

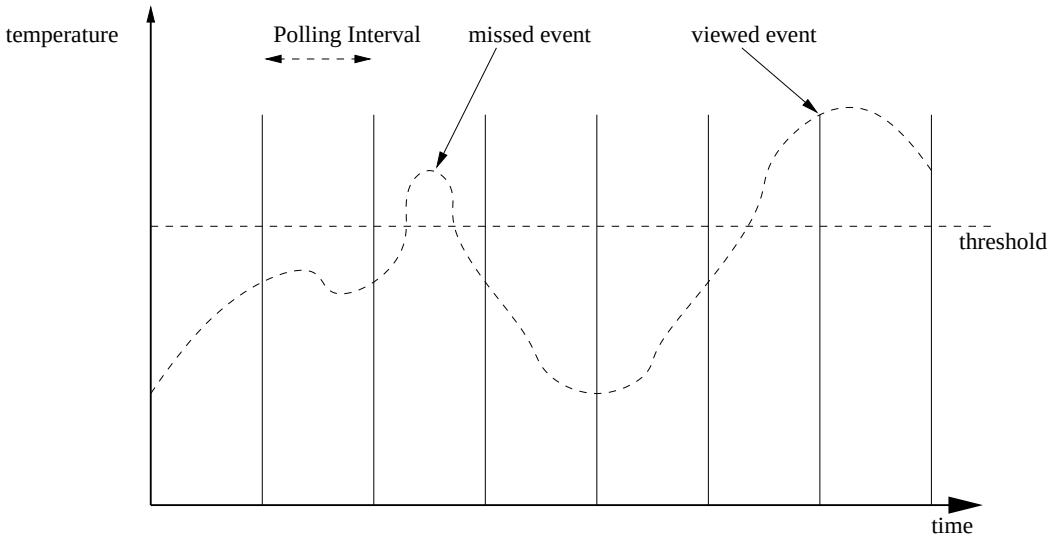

Figure 5.1: Events based on threshold can be missed when using polling.

Structure of the Chapter The rest of this chapter is structured as follows: first, in Section 5.2, we introduce a slightly extended syntax for our ELASTIC controllers, allowing the use of discrete variables. Second, in Section 5.3, we give the needed specification of our implementation platform. Then, Section 5.4 explains how one can annotate the controllers with fragments of code that mainly specify a matching between events and the hardware. We then proceed in Section 5.5 with the explanation of the mechanisms we use in the BRICKOS operating system for our implementation scheme. We are then ready to describe, in Section 5.6, our implementation scheme and argue about its correctness, before we illustrate the use of our tool on our main case study, the Philips Audio Control Protocol in Section 5.7. We close the chapter, in Section 5.8, by some remarks on the generated code for the protocol and a brief review of the literature.

5.2 ELASTIC Controllers with Discrete Variables

In this part of the thesis, as in the case study of the previous chapter, our ELASTIC controllers are extended to manipulate finite range discrete variables.

We do not give their abstract syntax[1] and semantics as they are really straight-

[1]For the interested reader, we gave examples of the concrete syntax in Figure 4.9 and in

forward extensions of respectively the ELASTIC syntax (Definition 3.4) and the AASAP (Definition 3.8) and program semantics (Definition 3.11) using the definitions of rectangular guards with discrete variables (Definition 4.1) and discrete updates (Definition 4.2).

Let us just say that the set of edges of an ELASTIC controller with discrete variables is defined as follows: Edg is a set of edges of the form $(\ell, \ell', g, \sigma, R, \mathsf{Update})$ where $\ell, \ell' \in \mathsf{Loc}$ are locations, $\sigma \in \mathsf{Lab}$ is a label, $g \in \mathsf{Rect_c}(\mathsf{Var})$ is a guard, $R \subseteq \mathsf{Var}$ is a set of clocks to be reset and $\mathsf{Update} \in \mathsf{Disc}(D)$.

Furthermore, let us remind the reader that we assume that every discrete variable takes its values in a finite range and could thus be encoded through a demultiplication of the locations of the controller. These variables add no expressive power to our controllers. They are just a very convenient syntactic addition to the ELASTIC syntax, that allows a more natural representation of timed controllers.

5.3 The Hardware

The Lego Mindstorms were originally designed as toys for learning robotics and programming. The main part of the system, called the RCX (Robotics Command System), is shaped like a big Lego brick, approximatively 10cm long (see Figure 5.2), and is a perfect example of an embedded system. It has a 16 MHz microprocessor with 32K external RAM, and three input gates for connecting sensors (touch, light, etc.), and three output gates designed for connecting motors. Finally an IR port allows to communicate with a computer or other RCX bricks. Programs have to be downloaded to the RCX from an office computer. RCX is somewhat tedious to program since there are very few debugging facilities : you can only use a very small screen (5 characters) or the data sent to a computer to get information on the program executing. Nevertheless, it is possible to install an operating system and to choose a programming language : the classical configurations[2] are C/C++ under BRICKOS and Java under lejOS.

Because of all those features and low cost, Lego Mindstorms have been broadly used in research and teaching [AFP+03, IKL+00, Pro98].

Appendix A.

[2]In the following, we choose to use BRICKOS because lejOS imposes the use of a virtual machine that was too demanding in terms of resources for our implementations.

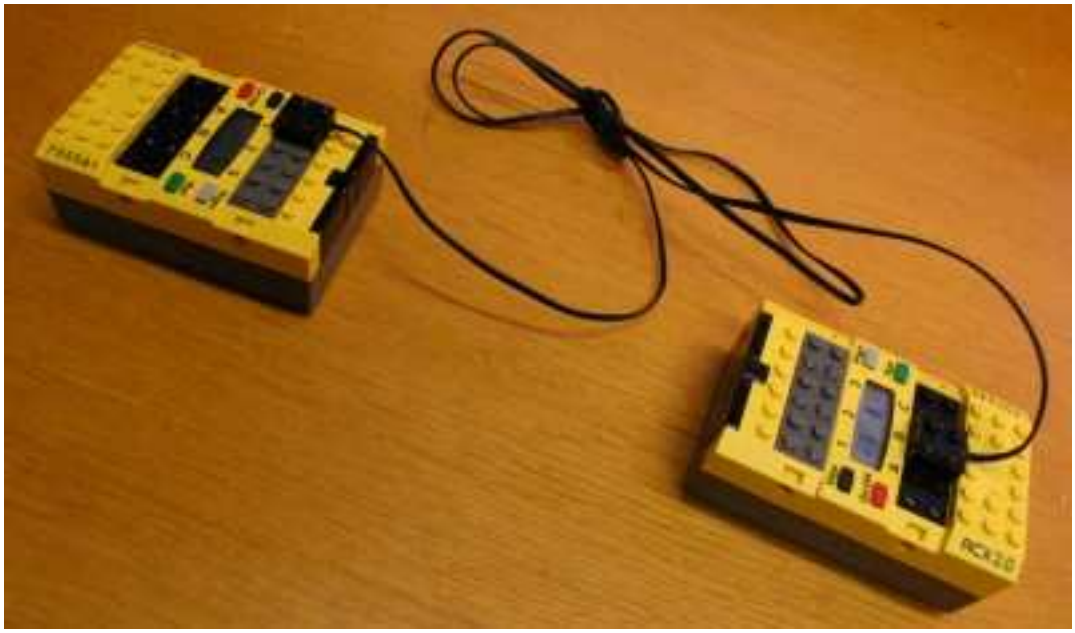

Figure 5.2: Two Lego Mindstorms Bricks connected by a wire

5.4 Annotation of an ELASTIC Specification

During the implementation process of an ELASTIC controller, the designer will have to provide the code for handling inputs and outputs, and to restrict the nondeterminism.

This code is organized in a set of annotations given as an appendix to the ELASTIC model (which concrete syntax has been sketched in Section 4.5). The original ELASTIC model is thus not modified.

We give now informally the syntax and the intended use of each annotation, and illustrate each annotation syntax through an example:

1. **Global Declarations:** the code written here includes the declaration of additional variables, functions needed in other annotations, and so forth... This annotation is parameterized by the name of the controller, since there can be many controller specifications in one ELASTIC file.

   Example:

   ```
   DECLARATIONS sender :
   ```

```
%{
   int message []={1,1,0,1,0,0};
   int j;
%}
```

2. **Controller Initialization:** This is where the designer can initialize the variables he needs or allocate resources. This annotation is also parameterized by the name of the controller.

   Example:

```
INIT sender :
%{
   j=0;
%}
```

3. **Input Handling:**

   In this annotation, the designer has to provide a boolean function that is true when the corresponding event has arrived. This is the most tricky part of the annotations for the designer, as he must ensure that each event is detected once and only once. The following example illustrates this problem:

```
DETECT  button :
%{
   bool button_pressed()
   {
      if (buttonPressedBefore)
      {
         if (! TOUCH1)
           buttonPressedBefore=false;
         return false;
      }
     else
     {
         if (TOUCH1)
```

```
            {
             buttonPressedBefore=true;
             return true;
            }
          else
            return false;
   }
%}
```

In this code, the designer wants to detect if a button is newly pressed and can only check, through the `PUSH1` macro of BRICKOS, if a button is currently pressed. Thus the program must have memorized the state of the button the last time it checked it. This is done through the use of the boolean variable `buttonPressedBefore`. It is now easy to see if the event "the button has just been pressed" has happened : it is the case if the button was not pressed before and is pressed now. Using this memory ensures that an event is not detected many times, but to ensure that an event is indeed detected once, we must know the frequency of the calls to the function `button_pressed()` and the minimal length of a pression on the button. The designer must check those informations by himself.

4. **Input Actions:** In this annotation, the designer puts the code that is executed every time an edge $(\ell, \ell', g, \sigma, R, \mathsf{Update})$ is fired, where σ is the label of the annotation and is an input label. The code for input actions will be executed after the clocks in R have been reset and updates Update have been executed.

   Example:

```
GET UP
%{
   i=message[j++]
%}
```

5. **Output Actions:** This annotation is very similar to input action, except that it applies to output actions. Observe that we could not have chosen a

common keyword for inputs and outputs, as used in the ELASTIC language, since we need to differentiate the cases where the label is the input of one controller or the output of another one. The code for output actions is executed after the possible resets of the clocks and updates to the discrete variables of the controller.

Example:

```
PUT MOTORON
%{
  motor_b_dir (FWD);
  motor_b_speed (MAX_SPEED);
%}
```

6. **Restriction of Nondeterminism through Guards:** Code added here must return a boolean value. This function will be added to the code evaluated when the guard of the transition is fired. This allows the reduction of nondeterminism. In this code, the user is allowed to read the discrete variables of the ELASTIC model, knowing that they are implemented using integer variables.

   Example:

```
RESTRICT loc1 TO loc2
%{
   bool iIsEven()
    {
       return (i%2==0);
    }
%}
```

   There is really a danger that the code added here modifies completely the timed behavior of the implementation. For example, a restriction could completely forbid an action a that was always chosen in some location ℓ , and this way force another action b that never happened in the AASAP semantics of the controller because a was always chosen first. This is why we said

in the introduction that the reductions of nondeterminism provided by the user should always keep one of the possible behaviors at any moment. More formally, for any reachable state in the AASAP semantics of the controller, if there are many edges enabled, the reductions to nondeterminism must keep *at least* one of those edges enabled. We will give an example of such reductions in our case study about the Philips Audio Control Protocol.

7. **Discrete Variable Hiding:** This annotation is in some sense an optimization : it allows to specify which variables should not appear in the implementation. This is typically useful for variables used in the ELASTIC specification only for the communication with observers. It is not necessary that the tool provides the code for those variables in the practical implementation. For example, in the Audio Control Protocol, the receiver stores the two last bits received in a variable `C`, so that the observer can check that those two bits are the same than what was sent. It is not useful to keep this variable in the implementation.

   Example:

   ```
   HIDE : C, LENG;
   ```

   Care must be taken not to hide variables that are meaningful to the implementation.

8. **Time Scale:** One final piece of information needed is the unit of time of the specification. This is provided through the directive `unit`. The value provided is assumed to be a multiple of the millisecond. Thanks to this information, we will be able to scale the constants of the ELASTIC model for the BRICKOS platform. In our framework, where only one type of hardware is handled, this is a sufficient information; but for a PC for example, a lot more parameters have to be provided, e.g. granularity of the clock (see [Maq06]).

   Example:

   ```
   unit : 100;
   ```

Observe that there is no cleanup code in the annotations. This seems natural since the whole purpose of ELASTIC controllers is to model *reactive systems* that interact infinitely often with an environment.

One other type of annotation that we could have added is some code for specific edges. We did not do that because we try to keep a clear separation of concerns between implementation and verification and this would have forced us to add an identifier to each edge, which is not useful at the level of the ELASTIC language.

5.5 Implementation Platform: BRICKOS

BRICKOS [Nie00] is an open source operating system for the Lego Mindstorms, available at `http://brickos.sourceforge.net` (in October 2006). It can be uploaded on the brick, along with user programs, through the infrared port. The programming environment, including the uploader, a cross-compiler[3] and a communication daemon runs on either Linux or Windows.

BRICKOS allows the use of multiple threads and respects the posix standard for semaphores. A semaphore is an integer variable for which there is a special "test and set" atomic operation and a queue of waiting threads. It can be used to manage the concurrent access to a resource. The terminology we use about semaphore says that a semaphore is *posted* when it is incremenented and *consumed* when it is successfully reset. For our implementations we added one operation that allows test and set of two semaphores atomically[4] and that is depending on a rectangular predicate on the variables of the program. This allows us to test for an edge for which, at the same time, the source location and the input are available and the guard is true.

BRICKOS allows preemptive multitasking with a *prioritized round-robin* scheduling policy, i.e., each thread has its own priority and amongst threads of the same priority the CPU is allocated for the same amount of time, called a *timeslice*, to each one in turn, cyclically. A thread can yield the cpu back to the scheduler at any time, using a dedicated function.

BRICKOS has only one interrupt and it is triggered by the timer every ms (this value can be customized). The handling of input and sharing of the processing

[3] The compiler is on a personal computer but creates binaries for the Mindstorms

[4] The atomicity is obtained by making the code execute while interruptions are disabled.

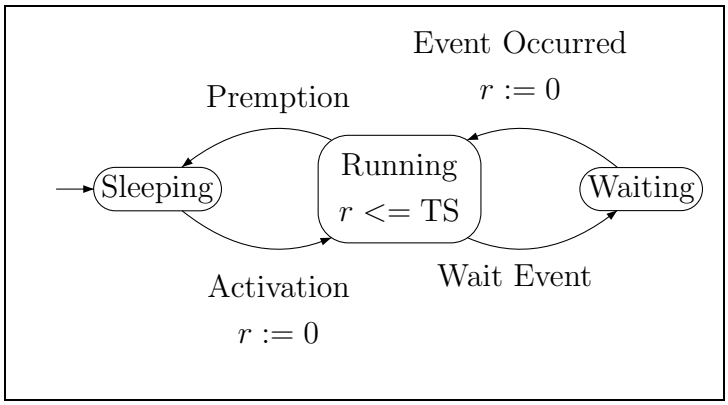

Figure 5.3: Possible states for a thread in BrickOs (TS stands for timeslice)

power are based on those interrupts. At every interrupt, BrickOs polls the memory zones corresponding to the inputs and possibly updates the zone that corresponds to outputs (e.g. setting the speed of the motors). Finally, BrickOs checks if the time that has been allocated to the current process is still smaller than the maximum timeslice and calls the scheduler if this is not the case. One can customize the value of the timeslice between 6 and 255 ms, 20 being the default value.

A thread can be in five states for the scheduler:

- *running*: the thread is running;
- *sleeping*: the thread is idle but ready to run;
- *waiting*: the thread is idle because it is waiting for something to happen: in our case either a timeout, the posting of a semaphore or a new value for some input.
- *dead*: the thread has finished its work and its memory stack has been deallocated;
- *zombie*: the thread has finished its work but its memory stack is still allocated;

We have illustrated in Figure 5.3, for the first three possibilities, the transitions from one state to the other. We are not much interested in the last two possibilities

since, in the programs we will generate, each thread will stay alive forever.

Every time a thread finishes its work or begins to wait, the scheduler is called. Observe that this means that calls to the scheduler can be very frequent, if the execution time of each thread before a wait is short.

A thread is put to sleep every time it has consumed a whole timeslice and another thread of superior or equal priority is ready to run.

During the execution of the scheduler, interruptions are disabled. This is the mechanism used for ensuring non concurrent accesses to a semaphore : only the scheduler accesses those special variable and, while it is executing, nothing can interfere. This is why we will move much of the critical actions of our implementation to the scheduler.

5.6 Implementation Scheme and Correctness

5.6.1 Architecture of the Generated Code

Several implementation schemes are possible for ELASTIC controllers, as long as they mimic the program semantics. Remember that the program semantics can be seen as a formal semantics for the following procedure, interpreting ELASTIC controllers, that repeatedly executes what we call *execution rounds*. An execution round is defined as follows:

- first, the current time is read from the clock register of the CPU and stored in a variable, say `T`;
- the list of input events to be treated is updated: the input sensors are checked for new events issued by the environment;
- guards of the edges of the current location are evaluated, using the value stored in `T`. If at least one guard evaluates to true then take nondeterministically one of the enabled transitions;
- the next round is started.

We will mimic this functioning by using the mechanisms offered by BRICKOS. Semaphores will be used as binary variables for two kinds of objects, location of the automaton and incoming events:

- A location semaphore is posted when the control of the automaton is in the corresponding location. We need to be sure that two transitions cannot be fired concurrently and we obtain this behavior by making the current location a resource impossible to share concurrently.

- An event semaphore is posted by special environment threads, when the corresponding event has occurred. Those threads are essentially infinite loops running the code for the detection of events provided by the designer. In this case, we use semaphores as we want to avoid problems with concurrent reads and writes.

Handling Clocks, Discrete Variables and Guards: Clocks and guards are dealt with very similarly, as in the implementation semantics for ELASTIC automata (Definition 3.11). The value of every clock is based on the unique system clock. With each clock c is associated a variable x_c that can store values of the system timer. The operation corresponding to a reset of the clock x in the practical implementation is to assign to the variable x_c the value of the system clock, that we denoted by $Time$, at the beginning of the last execution round. Then, when in a guard of the controller, we test if $c < k$, for $k \in \mathbb{N}$, the corresponding operation in practice is $Time - x_c < k$. We furthermore enlarge the guards as in the implementation semantics, using parameters Δ_L and Δ_p for which we will give values a little later.

The discrete variables appearing in the models are straightforwardly implemented using integer variables.

Useful BRICKOS Primitives: In the following, besides basic control structures and the function accessing to the inputs and outputs, that are provided by the users, we will only use very few primitives of the operating system. We present them here in an abstract fashion:

- **wait_event(f)** is a function used by a thread to signal that it is waiting for some event specified by the function f. After the call, the status of the thread is made waiting and the scheduler gets back the cpu. The event that the thread is waiting for is specified by the boolean function `f` which must return `true` when the event has happened. `f` is called a *wakeup function*.

Those *wakeup* functions are executed atomically by the scheduler and should thus have a very short WCET. When they return true, the scheduler gives back the cpu to the waiting thread.

- **wait_event**$(\ell, g, \mathsf{restrict}, \sigma)$ is a modified version of the wait_event(f) function we explained before. The difference is that this function makes the current thread waiting for the availability of the semaphore associated to location ℓ , the availability of the semaphore associated to event σ and the satisfaction of the rectangular guard g and of the boolean function restrict, which use we will explain later. The semaphore for the event can be omitted if necessary, and we write **wait_event**$(\ell, g, \mathsf{restrict})$ in this case. The function has also the side effect of storing the value of the system clock used to evaluate the guards in a global variable T.

- **clearTimers**(R) is a function that performs the resetting of the clocks in the set R. The value given to all the clocks in R is the value hold by the global variable T (that is the value of the system time at the beginning of an execution round);

- **execute**(σ) is a function that simply executes the code associated to the label σ (through the `get` and `put` directives) in the annotations to the ELASTIC model ;

- **sem_post**(s) is a function which increments the value of the semaphore s.

- **do_update**(Update) is a function that performs the discrete updates specified by the updates in $\mathsf{Update} \in \mathsf{Disc}(D)$.

Threads Structure The controller will be built from many threads: one thread per different input label and one thread per edge of the automaton.

Each input label will have a separate thread (called an *input thread* in the following) in charge of its management. Every time it is launched, it will consume its event if it has arrived, and put the corresponding semaphore. Each input edge follows the pattern of Algorithm 1. It is an infinite loop, setting the semaphore σ every time the thread gets the cpu from the scheduler because the boolean function

Algorithm 1: Code for the input thread of σ

```
   begin
1  |  while ⊤ do
2  |  |  wait_event(event_has_arrived(σ));
3  |  |  sem_post(σ);
   end
```

event_has_arrived(σ) is true. This is the function provided in the annotations, precisely in the annotation beginning with `DETECT` σ.

For each edge of the automaton, we define a thread (called an *edge thread*) following the pattern of Algorithm 3 if it is labelled by an input label, or Algorithm 2 if it is labelled by an output label.

For an output edge $(\ell, \ell', \sigma, g, R)$, the thread is woken up whenever the semaphore for ℓ is available and a modification of the guard g is true. This modification is obtained through first the removal (using function hide) of any discrete variable that has been hidden in the annotation, modulo the enlargement of the rectangular predicate on clocks, as in the implementation semantics.

Once awake, the semaphore for ℓ having been consumed, the thread resets the variables associated to the clocks of R, performs the discrete updates (modified accordingly to the `hide` directive) and executes the code that was associated to the label σ in the annotation (through the `put` σ annotation).

Finally, the thread sets the semaphore associated to location ℓ' and gives the cpu back to the scheduler by a call to wait_event($\ell, {}_{\Delta_S}[\mathsf{hide}(g)]_{\Delta_S}, \mathsf{restrict}$).

For an input edge $(\ell, \ell', \sigma, g, R, \mathsf{Update})$, the thread is very similar, except that it awaits for one additional semaphore : the one that signals that the event σ has arrived. Observe that the guards are enlarged, as in the implementation semantics, by the value $\Delta_S = \lceil \Delta_L + \Delta_P \rceil_{\Delta_P}$.

The *edge threads* have all the same priority which is inferior to the priority used for all *input threads*. The scheduler thread structure thus looks like Figure 5.4. Each time the scheduler is called, it first checks the waiting condition of each input thread and executes them if one input has arrived. It is important here that events cannot repeat so closely in time that we never reach the priority level of the edge threads (it would be a kind of *thrashing*).

Algorithm 2: Code for the edge $(\ell, \ell', \sigma, g, R, \mathsf{Update})$ (for $\sigma \in \mathsf{Lab}^{\mathsf{out}}$)

```
   begin
1      while ⊤ do
2          wait_event(ℓ, Δ_S[hide(g)]_{Δ_S}, restrict);
3          clearTimers(R);
4          execute(σ);
5          do_update(hide(Update));
6          sem_post(ℓ′);
   end
```

Algorithm 3: Code for the edge $(\ell, \ell', \sigma, g, R, \mathsf{Update})$ (for $\sigma \in \mathsf{Lab}^{\mathsf{in}}$)

```
   begin
1      while ⊤ do
2          wait_event(ℓ, Δ_S[hide(g)]_{Δ_S}, restrict, σ);
3          clearTimers(R);
4          do_update(hide(Update));
5          execute(σ);
6          sem_post(ℓ′);
   end
```

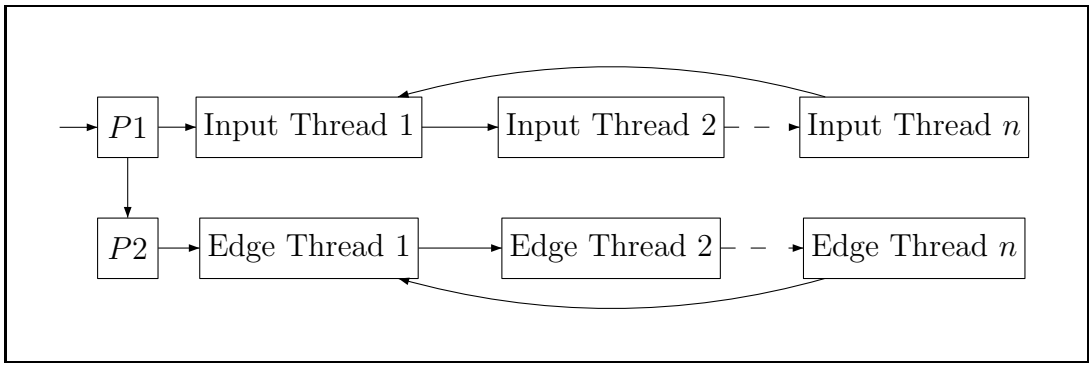

Figure 5.4: Thread structure of our implementations of ELASTIC controllers

After all possible inputs have been handled, one *edge thread* at most is run before the inputs are checked again. Why one at most ? Because when it ends, the scheduler restarts its run through the threads structure by the highest priority levels, thus checking for new inputs. This is the way we match the program semantics, since this mechanism ensures that all inputs are checked between the traversal of two consecutive edges. To be totally sure that this is exactly what happens, we will have to make some assumptions on the WCET of one loop of the input and edge threads. We will elaborate more about this in the next section.

In this scheme, the execution round of the implementation semantics is in fact mainly executed by the scheduler: it performs all the checks for the arrival of events and satisfaction of guards. The only code that is not executed by the scheduler is the code of the edge threads.

5.6.2 Correctness of the Generated Code

We want to argue the correctness of our generated code under the assumption that the AASAP semantics of the controller given in the specification has been proven correct for some safety property and for some strictly positive value for Δ, the parameter of the AASAP semantics. The correctness of the code is due to the direct matching with the implementation semantics, for which we have proven that it is simulated by the AASAP semantics of the ELASTIC controller at hand. Nevertheless, to ensure that our implementation indeed corresponds to the implementation semantics we need to know a number of additional informations,

namely the WCETs of every part of the code:

- the WCET of each edge thread, and for that we will need :
  1. the WCET of the operations wait_event(), sem_post(), clearTimers() and do_update()
  2. the WCET of each annotation of the code;
- the WCET of the scheduler before giving the cpu to a thread;
- the WCET of each input thread;

As stated before, the computation of worst-case execution time is a field by itself and we will not fully tackle this problem here. We will assume that the time required by the scheduler for checking all input threads and edge threads waiting conditions is less than 1 ms, thus meaning that all guards are checked for the same value of the system clock (assuming its granularity has been set to 1ms), which fits well with the program semantics. Since the guards are enlarged by Δ_S and since the value used to evaluate the guards is stored in a global variable T when a guard is true, and since this value is the one used to possibly reset clocks, our implementation scheme complies with the implementation semantics for the management of the clocks.

We furthermore assume that the time additionally needed by the edge thread (the one possibly scheduled) never outreaches the timeslice (that we set to be 6 ms), and thus that the thread is never preempted but instead always gives the hand back to the scheduler by itself. This seems reasonable, as we assume to be in a framework of simple timed controllers, where no time consuming computation has to be performed: 6 ms of time for a processor (the Hitachi H8/300L), running at 16MHz with at most 14 cycles per instruction ([Hit]), means the possible execution of at least 6857 cpu instructions.

This means in the methodology that for the BrickOs platform the values of the parameters Δ_L is set to 6ms and the value of Δ_P is set to 1ms.

We have the sufficient condition that $\Delta > 3\Delta_L + 4\Delta_P$ for the implementation to be correct (see Theorem 3.2 for a reminder.) This condition was known from the previous chapters, but there are other conditions for the implementation to be correct :

- the insurance that no event is missed, that is that the input-checking functions never miss any event;
- the insurance that the restrictions to nondeterminism have kept the controller reactive, that is the controller has kept in any situation at least one of the many behaviors that were allowed initially;
- the insurance that all added code is free of run-time errors and terminating (this can be included in the WCET request);

We will now illustrate the checking of those conditions on our case study, the Philips Audio Control Protocol.

5.7 Case Study: the Audio Control Protocol

As a case study, we have implemented the Audio Control Protocol presented in the previous chapter (Section 4.6). We connected two Lego Mindstorms Bricks, one being the sender and the other the receiver. To connect the two Bricks, we use a wire plugged to an output gate of the sender and to an input gate of the receiver (see Figure 5.2). We then annotated the ELASTIC model explained in the previous chapter. This model and its annotation with BRICKOS code is given in Appendix B.

In practice, on one hand, our implementation of the sender sends again and again the same message that is given in an array of integer:

```
int message[]= {1,0,1,1,0,0,1,1,1,1,0,1,0,0};
```

On the other hand, the receiver checks the message he receives against the awaited message and simply puts "OK" on the lcd screen of the brick if there is a perfect match.

We now describe the annotations and explain how they satisfy the constraints listed in the previous section, so that we can be confident that the code generated by the ELASTIC tool is correct by construction.

Handling inputs and outputs: First, to put some voltage on a wire, that is to create an UP event, we simply use the function provided by BRICKOS to turn a motor on:

```
put up :
%{
  motor_b_dir (brake);
  motor_b_speed (MAX_SPEED);
%}
```

For a DOWN event, we simply use the code for turning a motor off:

```
put down :
%{
  motor_b_dir (brake);
  motor_b_speed (MIN_SPEED);
%}
```

In the Audio Control Protocol, there is only one input : the UP event. To detect this input, we provide the boolean function `isUp` of Figure 5.5. The presence of an electric current on the wire is tested as if it was a button pushed, using the `TOUCH1` macro (provided in BrickOs). The principle is very simple: if there is a current, and there was not before, then there has been an UP event. We use the `lastWasUp` boolean variable to store the previous state of the current on the wire. Since this function is checked every 6 ms at least, the controller cannot miss an UP event since the length of the emission on the wire is always strictly greater than 88ms, which is the lower bound on the unit of time of the protocol as we will explain very soon.

The handling of the UP events by the receiver is then specified through the following annotation:

```
get up :
%{
  switch(r)
  {
     case 0:
       break;
     case 1:
     {
      received[i]=1;
```

```
detect up :
%{
bool getup()
 {
     if (lastWasUp)
          {
            if (TOUCH_1)
               return false;
            else
               {
                 lastWasUp=false;
                 return false;
               }
           }
        else
          {
            if (TOUCH_1)
              {
               lastWasUp=true;
               return true;
              }
            else
               {
                 return false;
               }
           }
  }
%}
```

Figure 5.5: Detection Function for the UP event

```
      i++;
      break;
     }
```

```
      case 2:
      {
       received[i]=1;
       received[i+1]=0;
       i=i+2;
       break;
      }
  }
%}
```

This fragment of code uses the `r` variable, which encodes, in the ELASTIC model, the value of the last bits decoded every time an UP has been seen (see Section 4.6 for a reminder).

Handling Nondeterminism: The ELASTIC model of the sender in the Audio Control Protocol uses nondeterminism to allow the sender to send any binary message. This was done by setting nondeterministically the variable i to 1 or 0 in an annex automaton, and testing this variable to decide the next bit to send (see Figure 4.12 in the previous chapter).

In practice, we want our implementation to send one precise message. This has been resolved very simply by eliminating the test on the i variable from the sender automaton using the `hide` directive and by replacing, using the `restrict` directive, each test on i by a call to a function testing the next bit.

For example, the test $i = 1$ on the edge from $OneSent$ to $WaitOne$ is replaced by a call to the function `oneNext()`:

```
restrict OneSent to WaitOne
%{
   bool oneNext()
   {
      return (j<M_SIZE && message[j++]==1);
   }
%}
```

In the code of this function, `j` is the number of bits that have been sent in the current message, `M_SIZE` is the length of the message and `message` is the array

holding the sequence of bits to send.

A function `zeroNext()` is defined similarly for edges where the test $i = 0$ appears.

One can easily check that the restrictions we made to the nondeterminism will not allow more behaviors of the implementation than what was allowed in the ELASTIC model.

Correctness of the Code To argue the correctness of the code generated, we have to address five points:

1. the WCETs of the different parts of the code
2. the detection of events;
3. the reductions to nondeterminism;
4. the correctness of the added code;
5. the satisfaction of the constraints between Δ , Δ_P and Δ_L for the ELASTIC controller at hand;

For the WCETs, the code provided in the annotations seems sufficiently short to satisfy the assumption that all WCETs are very low and that the firing of an edge will never demand more than 6857 assembly instructions.

We have already explained how we address the detection of inputs and the reduction of nondeterminism correctly. For the correctness of the added code, only a careful reading of the annotations , that are given in Appendix **??**, will convince the reader. Note that the work is not very heavy, since the length of the annotations is only about 150 lines of code. It is interesting to compare this number with the number of lines of code for the generated code : about 400 lines of code for the sender and 650 for the receiver, which is pretty close to the lengths of a first implementation we made "by hand".

Finally, we must check the satisfaction of the constraint $\Delta > 3\Delta_L + 4\Delta_P$. As for our implementation platform $\Delta_L = 6$ms and $\Delta_P = 1$ms, this imposes that $\Delta > 22$ms. On the other hand, remember that for the correctness of the AASAP semantics of the protocol, we must have $\Delta < \frac{U}{4}$ where U is the unit of time of the

protocol. We thus have a lower bound on U of 88ms for us to be able to guarantee the correctness of the implementation.

Thus, provided the unit of time of the protocol is not too low, the code generated by the ELASTIC tool for the Audio Control Protocol is correct by construction and can safely be executed on a pair of LEGO MINDSTORMS™ Bricks as an alternative communication means with real-time guarantees. For that, it suffices to give the highest level of priority to the protocol to ensure its real-time behavior. This should not spoil the behavior of other applications running on the Brick as the resources needed by the protocol are very low.

Now, let us look at the performance of the protocol in our implementation, even if, in real-time, the main concern is generally not speed, but the respect of the timing constraints. The throughput obtained, when the length of the sequence goes to infinity, is around 2.84 bits per seconds (because of the constraint that a quarter of a time slots require at least 88ms). This may look quite low and we could think that a far better throughput could be obtained by a hand-made implementation. But this is not the case. Indeed, we can show, using the results of Ho and Wong-Toi [HWT95], and by taking into account only the imprecision due to reading on digital clocks every time slice, that the throughput of the protocol on LEGO MINDSTORMS™ is bounded from above by around 4.16 bits per seconds. So, the price in term of performance loss to obtain automatically generated and correct code is not too high in our opinion. Let us also note that we were only able to find error by testing when the throughput was set around 7 bits per seconds. That shows the limit of testing, at least when done in a naive way: our simple testing program was not able to provoke the worst-case scenarios analysed through model checking.

5.8 Conclusion

Let us close this chapter by a summary of some interesting works about code generation for real-time specifications in the literature. The main difference between those works and ours is that they use the synchrony hypothesis:

- Alur et al have considered the problem of generating code from models of hybrid systems specified in the language CHARON [AIK+03]. The differences

with our work are the following: first, they allow communication between environment and controllers using shared continuous variables, where we have opted for an abstraction of those mechanisms using discrete events. Second, the correctness of their work relies on the synchrony hypothesis. They consider for example that the computation of a control law takes no time. Finally the emphasis is also put in this work on the reusability and the hierarchical structure of the specifications.

- In this chapter, we use systematically the same fixed priority scheduling algorithm, but other options have been the subject of recent research works. Notably, in [AC05], Alur et al have studied platform independent code generation, where a "dispatch sequence" of the different tasks is established depending on the interdependances between tasks. This dispatch sequence can then be tested for schedulability on different platforms. This idea could be adapted to the implementation of ELASTIC controllers.

- The Times tool [AFM+02] offers the generation of code from models specified as timed automata [AFP+02] but the emphasis of this work is more on a generalization of classical scheduling techniques. The scheduling policy and the pattern of arrivals of the tasks are specified through the use of timed automata. The launch of a task can be associated with some edges of the automaton, which allows much more complex patterns than the classical periodic patterns. The authors proved the decidability of the resulting scheduling problem [FPY02] and provide an implementation based on the UPPAAL verification engine. This work uses the synchrony hypothesis implicitly, but could still model switching times between tasks through the timed automata modelling the scheduling policy.

Chapter 6

Games of Imperfect Information

6.1 Introduction

Consider that we have a hybrid system modelling a plant: the controller state moves discretely between control modes, and in each control mode, the plant state evolves continuously according to physical laws. The distinction between continuous evolutions of the variable and discrete switches of the controller state (which is given by the location, or control mode, of the hybrid automaton) permits a natural formulation of the *safety control problem*: given an unsafe set U of plant states, is there a strategy to switch the controller state in real time so that the plant can be prevented from entering U? In other words, the hybrid automaton specifies a set of possible control modes, together with the plant behavior resulting from each mode, and the control problem asks for deriving a *switching strategy* between control modes that keeps the plant out of trouble.

In the literature, there are algorithms or semi-algorithms (termination is not always guaranteed) to derive such switching strategy. Those semi-algorithms usually comes in the form of symbolic fixed point computations that manipulate sets of states using a well-suited monotonic function like the controllable predecessor operator [AHK02, MPS95]. Those algorithms make a strong hypothesis: they consider that the controller that executes the switching strategy has a *perfect information* about the state of the controlled system. Unfortunately, this is usually an unreasonable hypothesis. Indeed, when the switching strategy has to be implemented by a real hardware, the controller typically acquires information about the

state of the system by reading values on sensors. Those sensors have a finite precision, and so the information about the state in which the system lies is *imperfect*. Let us illustrate this. Consider a controller that monitors the temperature of a tank, and has to maintain the temperature between given bounds by switching on and off a gas burner. The temperature of the tank is the state of the continuous system to control. Assume that the temperature is sensed through a thermometer that returns an integer number and ensures a deviation bounded by one degree Celsius. So, when the sensor returns the temperature c, the controller only knows that the temperature lies in the interval $(c-1, c+1)$ degrees. We say that the sensor reading is an *observation* of the system. This observation gives an imperfect information about the state of the system.

Now, if we fix a set of possible observations of the system to control, the control problem that we want to solve is the *safety control problem with imperfect information*: "given an unsafe set U of plant states and a set of observations, is there an observation based strategy to switch the controller state in real time so that the plant can be prevented from entering U?". While it is well-known (see [Rei84]) that games of perfect information can be won using *memoryless strategies*, this is not the case for games of imperfect information. In his work, Reif studies *games of incomplete information*, which are a subclass of games of imperfect information where the set of observations is a partition of the state space. Notice that this is not the case of our tank example since when the temperature of the water is d, the thermometer, that returns only integer values, may return either $\lceil d \rceil$ or $\lfloor d \rfloor$. To win such games, memory is sometimes necessary: the controller has to remember (part of) the history of observations that it has made so far. In the finite state case, games of incomplete information can be solved algorithmically. Reif proposes an algorithm that first transforms the game of incomplete information into an equivalent game of perfect information using a kind of determinization procedure. The latter game is equivalent to the former in the sense that if there exist a winning strategy for one of the game, there exists a winning strategy for the other.

In this chapter, we propose an alternative method to solve both the already cited safety games, but also reachability games of imperfect (and incomplete) information. Our method comes in the form of a fixed point (semi-)algorithm that iterates a monotone operator on the lattice of antichains of sets of states. Our method benefits from the monotonicity of the controllable predecessors operator to

use this lattice instead of the classical lattice built from sets of subsets of the state space, ordered by subset inclusion. Thus the greatest fixed point of this operator contains exactly the information needed to determine the states from which an observation based control strategy exists and to synthesize such a strategy.

This technique allows a great efficiency in finding strategies, and we proved it experimentally by using the technique for the test of universality of finite automata. Indeed, we reduce the universality problem to a two-player reachability game of incomplete information. We implemented this solution using NuSMV [CCGR99] and the CUDD library [Som98]. To compare the performance of the antichain algorithm to the performance of various implementations of determinization based algorithms, we used a large set of examples generated in the probabilistic framework by Tabakov and Vardi [TV05]. This framework was proposed with the express purpose of comparing the performances of algorithms on finite automata. In their experiments, the authors conclude that explicit determinization as implemented in [Mø04] outperforms the algorithm of Brzozowski [BL80] as well as newer implementations, which use symbolic methods for the determinization. Our experimental results show that our implementation of the antichain algorithm is considerably faster, on the entire parameter space of the probabilistic framework, than the most efficient implementation of the standard algorithm. In particular, on the most difficult instances of the probabilistic framework, the antichain algorithm outperforms [Mø04] by two orders of magnitude. For this comparison, we are limited to automata with approximately 175 states, which is the limit that the classical approach, using a subset construction, can handle on the most expensive instances of the probabilistic framework. On these difficult instances, the antichain approach scales much better: we are able to successfully check universality for automata with several thousands of states in less than 10 seconds.

Furthermore, we prove that our algorithm has an optimal complexity for finite state games and we identify a class of infinite state games for which the greatest fixed point of the operator is computable. Using this class of games and results from [HK99], we show that the discrete-time control problem with imperfect information is decidable for the class of rectangular automata. Strategies that win those games are robust in a sense, as they can be implemented using hardware that senses its environment with finite precision.

Our fixed point method has two main advantages over the algorithmic method

proposed by Reif. First, our method is a little more general than Reif's one, since we are able to handle games of imperfect information, which are more general than games of incomplete information. Second, our method is always more efficient: we even show that there are families of games on which the Reif's algorithm needs exponential time when our algorithm only needs polynomial time. Finally, thanks to its fixed point form, our method is easily applicable to interesting classes of systems like rectangular automata.

This chapter is structured as follows. In Section 6.2, we recall the definition of the lattice of antichains. In Section 6.3 , we show how to use this lattice to solve games of imperfect information. In Section 6.4, we give a fixed point algorithm that is EXPTIME for finite state games, we compare it with the technique of Reif and we show its efficiency in practice, by handling the universality test for finite automata. Finally, in Section 6.5, we identify a class of infinite state games for which we can still use our fixed point algorithm and use this result to solve games of imperfect information for rectangular automata.

6.2 The Lattice of Antichains of Sets of States

First we recall the notion of antichain. An *antichain* on a partially ordered set $\langle X, \leq \rangle$ is a set $X' \subseteq X$ such that for any $x_1, x_2 \in X'$ we have that $x_1 \leq x_2$ implies $x_1 = x_2$, that is X' is a set of incomparable elements of X. We define similarly a *chain* to be a set of comparable elements of X.

Let $q, q' \in 2^{2^S}$ and define $q \sqsubseteq q'$ if and only if $\forall s \in q : \exists s' \in q' : s \subseteq s'$. This relation is a preorder but is not antisymmetric. Since we need a partial order, we consider a set $L \subseteq 2^{2^S}$ for which $\sqsubseteq$ is antisymmetric on L. The set L is a set of antichains on $\langle 2^S, \subseteq \rangle$.

We say that a set $s \subseteq S$ is *dominated* in $q \in 2^{2^S}$ if and only if $\exists s' \in q : s \subset s'$. The set of elements of 2^S that are dominated by elements of q is denoted $\mathsf{Dom}(q)$. The *reduced form* of q is $\lceil q \rceil = q \backslash \mathsf{Dom}(q)$ and dually the *expanded* form of q is $\rceil q \lceil = q \cup \mathsf{Dom}(q)$. The set $\lceil q \rceil$ is an antichain of $\langle 2^S, \subseteq \rangle$. The relation $\sqsubseteq$ has the useful following properties:

Lemma 6.1
Let $q, q' \in 2^{2^S}$. If $q \subseteq q'$ then $q \sqsubseteq q'$.

Lemma 6.2
$\forall q, q' \in 2^{2^S}, \forall q_1, q_2 \in \{q, \lceil q \rceil, \rfloor q \lceil \}, \forall q'_1, q'_2 \in \{q', \lceil q' \rceil, \rfloor q' \lceil \} : q_1 \sqsubseteq q'_1$ *is equivalent to* $q_2 \sqsubseteq q'_2$.

We can now define formally L as the set $\{\lceil q \rceil \mid q \in 2^{2^S}\}$. From now on, we will consider only the relation $\sqsubseteq_L$ defined as $\sqsubseteq \cap (L \times L)$. We will thus allow ourself a little abuse of notation by omitting the L in $\sqsubseteq_L$.

Lemma 6.3
$\langle L, \sqsubseteq \rangle$ *is a partially ordered set.*

Proof
Clearly $\sqsubseteq$ *is reflexive and transitive. We show that* $\sqsubseteq$ *is anti-symmetric on* L, *that is* $\forall q, q' \in L : q \sqsubseteq q' \wedge q' \sqsubseteq q \Rightarrow q = q'$. *Hence, we must show that* $q \subseteq q'$ *and* $q' \subseteq q$. *Let* $s \in q$. *Since* $q \sqsubseteq q'$, *there exists* $s' \in q' : s \subseteq s'$. *And since* $q' \sqsubseteq q$, *there exists* $s'' \in q : s' \subseteq s''$. *Thus* $s \subseteq s' \subseteq s''$. *Since* $s, s'' \in q$ *and* $q \in L$, *this implies that* $s = s''$ *and therefore* $s = s'$ *and* $s \in q'$. *Hence,* $q \subseteq q'$. *By symmetry we have that* $q' \subseteq q$, *hence* $q = q'$.

Lemma 6.4
For $q, q' \in L$, *the greatest lower bound of* q *and* q' *is*

$$q \bigsqcap q' = \lceil \{s \cap s' \mid s \in q \wedge s' \in q'\} \rceil$$

and the least upper bound of q *and* q' *is*

$$q \bigsqcup q' = \lceil q \bigcup q' \rceil$$

.

Proof
On one hand, first, we show that $q \bigsqcap q'$ *is a lower bound of* q *and* q'. *Let* $r \in q \bigsqcap q'$. *Then* $r = s \cap s'$ *for some* $s \in q$ *and* $s' \in q'$. *Hence,* $r \subseteq s$ *and* $r \subseteq s'$ *and so* $q \bigsqcap q' \sqsubseteq q$ *and* $q \bigsqcap q' \sqsubseteq q'$. *Second, we show that for any* q'' *such that* $q'' \sqsubseteq q$ *and* $q'' \sqsubseteq q'$, *we have* $q'' \sqsubseteq q \bigsqcap q'$. *Let* $s'' \in q''$. *Then there exists* $s \in q$ *and* $s' \in q'$ *such that* $s'' \subseteq s$ *and* $s'' \subseteq s'$, *and thus* $s'' \subseteq s \cap s'$ *which is dominated in* $q \bigsqcap q'$.

On the other hand, first, it is obvious that $q \bigsqcup q'$ *is an upper bound of* q *and* q' *for* $\sqsubseteq$, *since* $\forall r \in q : \exists r'' \in q \bigsqcup q' : r \subseteq r''$ *and* $\forall r' \in q' : \exists r'' \in q \bigsqcup q' : r' \subseteq r''$. *Second, we show that* $\forall q, q', q'' : (q \sqsubseteq q'' \wedge q' \sqsubseteq q'') \implies ((q \bigsqcup q') \sqsubseteq q'')$. *Let* $rr \in q \bigsqcup q'$. *It implies that* $rr \in q \bigcup q'$ *and since* $\forall r \in q : \exists r'' \in q'' : r \subseteq r'' \wedge \forall r' \in q' : \exists r'' \in q'' : r' \subseteq r''$, *we have that* $\exists r'' \in q'' : rr \subseteq r''$, *which implies the result.*

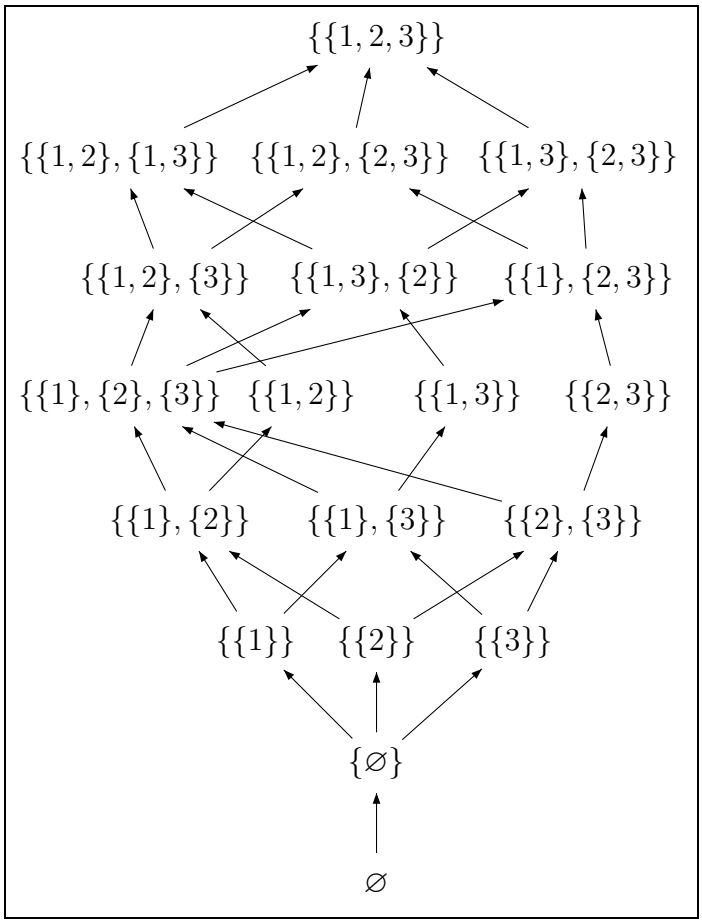

Figure 6.1: The lattice of antichains for $S = \{1, 2, 3\}$. An arrow from A to B implies that $A \sqsubseteq B$. (We omitted arrows obtained by transitivity).

For $Q \subseteq L$, we have $\bigsqcap Q = \lceil \{\bigcap_{q \in Q} s_q \mid s_q \in q\} \rceil$ and $\bigsqcup Q = \lceil \bigcup_{q \in Q} q \rceil$. The least element of L is $\bot = \bigsqcap L = \varnothing$ and the greatest element of L is $\top = \bigsqcup L = \{S\}$.

Lemma 6.5
$\langle L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top \rangle$ *is a complete lattice.*

This lattice is the *lattice of antichains of sets of states*. As an example, the reader can find the lattice of antichains for the set $S = \{1, 2, 3\}$ in Figure 6.1.

6.3 Games of Imperfect Information

6.3.1 Definitions

Notations Given a sequence $\overline{a} = a_0 a_1 \ldots$ of size larger than k (and possibly infinite), we denote by $\overline{a}_k = a_0, \ldots, a_k$ the sequence of the first $k+1$ elements of $\overline{a}$. $\overline{a}_{-1}$ is the empty sequence, also denoted by a dot "$\cdot$".

Definition 6.1 (Two-player games)
A two-player game *is a* UTS $\langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *where S is a (non-empty) set of states,* $\mathrm{E} \subseteq S$ *is the set of* initial *states,* $\mathrm{F} \subseteq S$ *is the set of* final *states, Σ is a finite alphabet of actions, and* $\rightarrow \subseteq S \times \Sigma \times S$ *is a transition relation. We require, without loss of generality, that the transition relation is total, that is* $\forall x \in S \cdot \forall \sigma \in \Sigma \cdot \exists x' \in S : x \xrightarrow{\sigma} x'$

The game is turn-based and played by a controller against an environment. At every turn, the controller chooses one label and the environment resolves the nondeterminism. The goal of the controller can be of two types:

- *(*safety *game) avoid the set of states* F*;*
- *(*reachability *game) reach the set of states* F*;*

We request that $\forall x \in \mathrm{F}$*, the set* $\{x' \in S \mid \exists \sigma \in \Sigma : (x, \sigma, x') \in \rightarrow\}$ *is included in* F*. We will explain later why this constraint is not too restrictive.*

For $s \subseteq S$, let $\mathsf{post}_\sigma(s) = \{x' \in S \mid \exists x \in s : (x, \sigma, x') \in \rightarrow\}$ be the set of *successor states* of s by the action σ, let $\mathsf{pre}_\sigma(s) = \{x \in S \mid \exists x' \in s : (x, \sigma, x') \in \rightarrow\}$ be the set of *predecessor states* of s by the action σ and let $\mathsf{cpre}_\sigma(s) = \{x \in S \mid \forall x' \in S : (x, \sigma, x') \in \rightarrow \text{ implies } x' \in s\}$ be the set of *controllable predecessors* of s by the action σ. A *play* of a game G is an infinite sequence of state $x_0 x_1 \ldots$, such that $x_0 \in \mathrm{E}$ and $\forall i \geq 0 \cdot \exists \sigma : (x_i, \sigma, x_{i+1}) \in \rightarrow$.

The controller has an imperfect view of the game state space in that his choices are based on imprecise observations of the states.

Definition 6.2 (Observation set)
An observation set Obs *of the state space S is a set of subsets (called* observations*) of S (*$\mathsf{Obs} \subseteq 2^S$*) satisfying the constraint that* $\bigcup_{\mathsf{obs} \in \mathsf{Obs}} \mathsf{obs} = S$*, that is, each element of S is included in at least one observation.*

An observation obs is *compatible* with a state x if $x \in \mathsf{obs}$. When the controller observes the current state x of the game, he receives *one* observation compatible with x. The observation is non-deterministically chosen by the environment.

Definition 6.3 (Games of Imperfect, Incomplete, or Perfect information) *A two-player game $\langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ equipped with an observation set Obs of its state space defines a* game of imperfect information $\langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs} \rangle$. *The* size *of the game is the sum of the sizes of the transition relation $\rightarrow$ and the set Obs.*

Let $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs} \rangle$ be a game of imperfect information. We say that G is a game of incomplete information *if for any $\mathsf{obs}_1, \mathsf{obs}_2 \in \mathsf{Obs}$, if $\mathsf{obs}_1 \neq \mathsf{obs}_2$ then $\mathsf{obs}_1 \cap \mathsf{obs}_2 = \varnothing$, that is the observations are disjoint, thus partitioning the state space.*

We say that G is a game of perfect information *if $\mathsf{Obs} = \{\{x\} \mid x \in S\}$.*

The drawback of games of incomplete information is that they are not suited for a robust modelling of sensors. Indeed, real sensors are imprecise and may return different observations for a given state.

Example Figure 6.2 presents a game of imperfect observation with state space $S = \{1, 2, 3, \mathrm{Bad}\}$, set of initial states $\mathrm{E} = \{2, 3\}$ and actions $\Sigma = \{a, b\}$. The observation set is $\mathsf{Obs} = \{\mathsf{obs}_1, \mathsf{obs}_2\}$ with $\mathsf{obs}_1 = \{1, 2, \mathrm{Bad}\}$ and $\mathsf{obs}_2 = \{1, 3, \mathrm{Bad}\}$. An edge from x to x' labelled by $a \mid b$ stands for two arrows with same source and destination, one labelled with a, the other one with b. The states belonging to E are pointed out using arrows with no source state

Definition 6.4 (Observation Based Strategy, Outcome, Reach_λ) *An* observation based strategy *for a game $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs} \rangle$ of imperfect information is a function $\lambda : \mathsf{Obs}^+ \rightarrow \Sigma$. The* outcome *of λ on G is the set $\mathsf{Outcome}_\lambda(G)$ of (finite or infinite) plays $\overline{x} = x_0 x_1 \ldots$ such that there exists a sequence of observations $\overline{\mathsf{obs}} = \mathsf{obs}_0 \mathsf{obs}_1 \ldots$ such that for all $0 \leq i$, $x_i \in \mathsf{obs}_i \wedge x_i \xrightarrow{\lambda(\overline{\mathsf{obs}}_i)} x_{i+1}$. The set of reachable states of G through the strategy λ is defined as follows:*

$$\mathsf{Reach}_\lambda(G) = \{x_i \mid \exists x_0 x_1 \ldots x_i \in \mathsf{Outcome}_\lambda(G)\}.$$

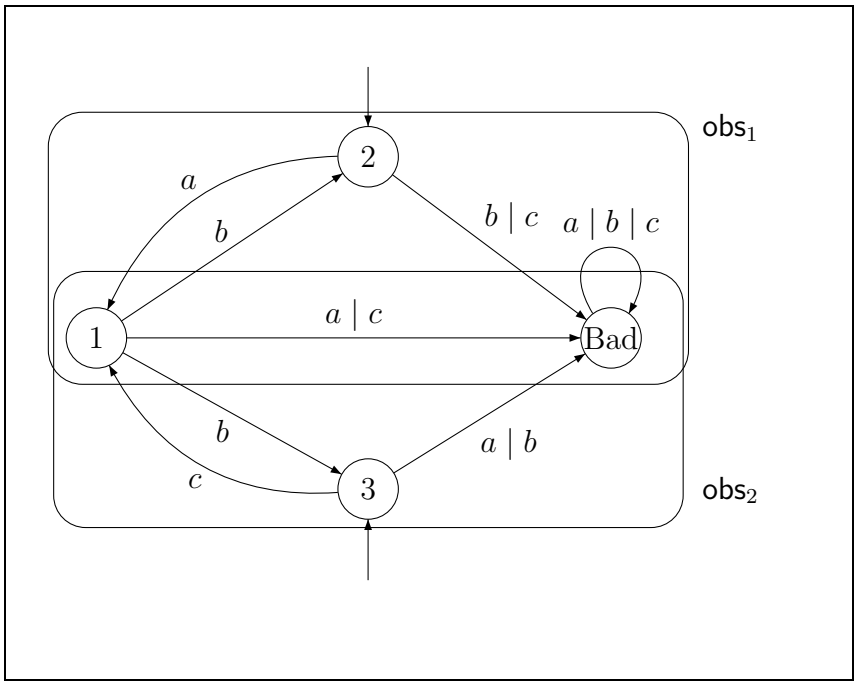

Figure 6.2: A two-player game G_1 with observation set $\{\mathsf{obs}_1, \mathsf{obs}_2\}$.

Definition 6.5 (Safe and Reaching Strategy)
We say that an observation based strategy λ for a game $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs}\rangle$ of imperfect information is safe *if for every play $x_0 x_1 \dots x_i \in \mathsf{Outcome}_\lambda(G)$, we have $x_i \notin \mathrm{F}$, or in other words if $\mathsf{Reach}_\lambda(G) \subseteq \overline{\mathrm{F}}$.*

We say that an observation based strategy λ for a game $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs}\rangle$ of imperfect information is reaching *if there exists $i \geq 0$ such that for every play $x_0 x_1 \dots x_i \in \mathsf{Outcome}_\lambda(G)$, $x_i \in \mathrm{F}$.*

Intuitively, a strategy is *safe* if it can ensure that the set F is never reached by the game and a strategy is reaching if there exist a number i for which the strategy can ensure that the set F is reached after exactly i moves of the controller, whatever the environment does. This request on the existence of i may seem restrictive, as *classical reaching strategies* are defined as follows: an observation based strategy λ for a game $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs}\rangle$ of imperfect information is *classically reaching* if for every play $x_0 x_1 \cdots \in \mathsf{Outcome}_\lambda(G)$, there exists $i \geq 0$ such that $x_i \in \mathrm{F}$. In this classical definition, the constraint that $\forall x \in \mathrm{F}$, the set

$\{x' \in S \mid \exists \sigma \in \Sigma : (x, \sigma, x') \in \rightarrow\}$ is included in F has not to be satisfied by G. It is easy to see that the question of the existence of a classically reaching strategy for this game $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs}\rangle$ can be reduced to the existence of a reaching strategy for the game $G' = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow', \mathsf{Obs}\rangle$, where $\rightarrow'$ is defined as follows: $(x, \sigma, x') \in \rightarrow'$ iff either $x = x' \in \mathrm{F}$ or $x \notin \mathrm{F} \wedge (x, \sigma, x') \in \rightarrow$. In fact, the strategies we look after in both cases are the same. This explains why it is reasonable to ask at the same time that for the game we handle, the final set F can not be escaped once entered and that a reaching strategy ensures that the set of final states is reached after the exact same number of move, whatever the sequence of observations: finding a reaching strategy in this framework allows to find a classical reaching strategy in the classical framework. We chose our definitions because they simplify the proofs for the reachability games and do not influence the problem of finding safe strategies.

Definition 6.6 (History, Knowledge)
Let us an history *of length $k \in \mathbb{N}$ be a couple $(\overline{\mathsf{obs}}, \overline{\sigma}) \in \mathsf{Obs}^k \times \Sigma^k$ such that $\exists \overline{x} \in S^k : x_0 \in \mathrm{E}$ and for all $0 \leq i \leq k$ we have $x_i \in \mathsf{obs}_i$ and for all $0 \leq i < k$ we have $x_i \xrightarrow{\sigma_i} x_{i+1}$. Let us call* knowledge *after an history $(\overline{\mathsf{obs}}, \overline{\sigma})$ the function $K : \cup_{0 \leq k}(\mathsf{Obs}^k \times \Sigma^k) \to 2^S$ defined inductively as follows.*

$$\begin{cases} K(\cdot, \cdot) &= \mathrm{E} \\ K(\overline{\mathsf{obs}}_k, \overline{\sigma}_k) &= \mathsf{post}_{\sigma_k}(K(\overline{\mathsf{obs}}_{k-1}, \overline{\sigma}_{k-1}) \cap \mathsf{obs}_k) \quad \textit{for } k \geq 0 \end{cases}$$

Thus, the knowledge *after an history $(\overline{\mathsf{obs}}_k, \overline{\sigma}_k)$ is the set of states the controller can be sure the game is in after this history.*

We define similarly $K_\lambda : (\cup_{0 \leq k} \mathsf{Obs}^k) \to 2^S$, the knowledge for a strategy λ after a sequence of observation $\overline{\mathsf{obs}}$:

$$\begin{cases} K_\lambda(\cdot) &= \mathrm{E} \\ K_\lambda(\overline{\mathsf{obs}}_k) &= \mathsf{post}_{\lambda(\overline{\mathsf{obs}}_k)}(K_\lambda(\overline{\mathsf{obs}}_{k-1}) \cap \mathsf{obs}_k) \quad \textit{for } k \geq 0 \end{cases}$$

Observe that:

- *a strategy is* safe *iff $\forall \overline{\mathsf{obs}} \in \mathsf{Obs}^+ : K_\lambda(\overline{\mathsf{obs}}) \subseteq \overline{F}$;*
- *a strategy is* reaching *iff $\exists i \in \mathbb{N} \cdot \forall \overline{\mathsf{obs}}_i \in \mathsf{Obs}^+ : K_\lambda(\overline{\mathsf{obs}}_i) \subseteq F$.*

Example For the game of Figure 6.2, the knowledge after the history $(\mathsf{obs}_1\mathsf{obs}_2, ab)$ is defined recursively as follows:

$$\begin{cases} K(\cdot,\cdot) &= \mathrm{E} = \{2,3\} \\ K(\mathsf{obs}_1, a) &= \{1\} \\ K(\mathsf{obs}_1\mathsf{obs}_2, ab) &= \{2,3\} \end{cases}$$

Thus, the set $\{2,3\}$ is the set of states the controller can be sure the game is in after the history made of the two sequences $\mathsf{obs}_1\mathsf{obs}_2$, and ab. We will define a strategy for this game in Section 6.4.2.

The *imperfect (resp. incomplete, perfect) information safety problem* is defined as follows for a class $\mathcal{C}$ of games of imperfect (resp. incomplete, perfect) information: given a game $G \in \mathcal{C}$, determine whether there exists a safe observation based strategy for G.

The *imperfect (resp. incomplete, perfect) information reachability problem* is defined as follows for a class $\mathcal{C}$ of games of imperfect (resp. incomplete, perfect) information: given a game $G \in \mathcal{C}$, determine whether there exists a reaching observation based strategy for G.

6.3.2 Controllable Predecessors on the Lattice of Antichains

We show how the lattice of antichains that we have introduced in Section 6.2 can be used to solve games of imperfect information by iterating a predecessor operator.

Definition 6.7 (Controllable predecessors)
For $q \in L$, define the set of controllable predecessors *of q as follows:*

$$\mathsf{CPre}(q) = \lceil \{ s \subseteq S \mid \forall \mathsf{obs} \in \mathsf{Obs} \cdot \exists \sigma \in \Sigma \cdot \exists s' \in q : \mathsf{Post}_\sigma(s \cap \mathsf{obs}) \subseteq s' \} \rceil$$

Let us consider an antichain $q = \{s'_0, s'_1, \dots\}$. A set s belongs to $\mathsf{CPre}(q)$ iff

1. For all observations[1] of a state in s, there exists a move σ such that the next state reached by the game (after the environment has resolved nondeterminism[2]) is surely in a set of q, and

[1] The quantification over obs is universal since for observations that are incompatible with the current state, the condition holds trivially since $\mathsf{Post}_\sigma(\varnothing) = \varnothing \subseteq s'$.

[2] This resolution of nondeterminism is represented by the use of Post: we consider the set of all possible successors.

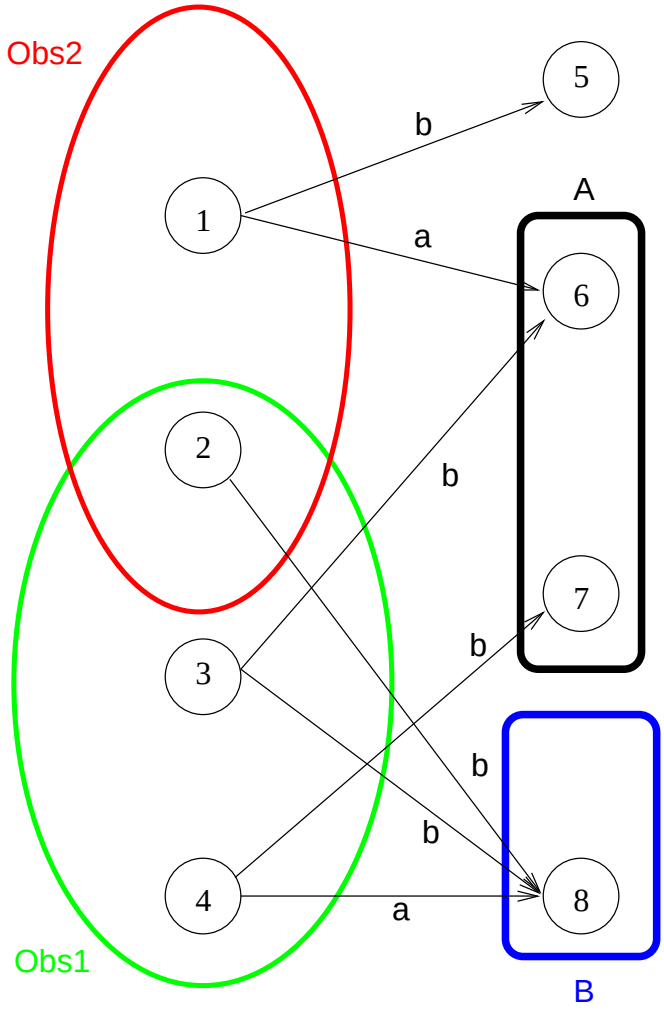

Figure 6.3: $\{\{1,4\},\{2\}\} = \mathsf{CPre}(\mathsf{A},\mathsf{B})$

2. s is maximal.

Before proving the usefulness of this operator, we try to give the reader some intuition through a small example pictured in Figure 6.3. In this example, there are 8 states numbered from 1 to 8 and three observations : $\mathsf{Obs}1 = \{1, 2\}$, $\mathsf{Obs}2 = \{2, 3, 4\}$ and $\mathsf{Obs}3 = \{5, 6, 7, 8\}$ (this last observation is not pictured for the sake of readability of the figure). We want to compute the controllable predecessors of the antichain $\{A, B\}$ where $A = \{6, 7\}$ and $B = \{8\}$. Observe that on the figure, the transition relation is not total, for the sake of readability. This could be simply solved by adding edges leading to a bad state for each unforeseen label.

Let us now look state by state which one is controllable:

- State 1 is controllable since playing an a in this state leads surely in state 6 which belongs to the controllable set A;
- State 2 is controllable too since playing a b in this state leads to 8, which belongs to the controllable set B;
- State 3 is not controllable since when the controller plays the only possible move, a b, he does not know for sure if the following state is in A or B.
- State 4 is controllable, since it offers two different moves, which both lead to controllable sets. In this case, the fact that these two sets are different is not a problem, since the controller can choose, through the choice of a label, in which set the game will go.
- States 5 to 8 are not in $\mathsf{CPre}(A, B)$ since they simply have no successors either in A or B.

Now, let us look for sets of states that are not singleton, and that are also controllable. There is in fact only one such set: $\{1, 4\}$. It is controllable since for all observations, there exists a move and a controllable set such that if the controller plays the move, the next state is in the controllable set. In fact, for the set $\{1, 4\}$, the observations allow to tell if the current state is either 1 or 4 and as we have seen before, there is a winning move from any of those two states.

It is also interesting to check why no other set is controllable. First, observe that no controllable set can contain 3, since there is no winning move from it.

Second, observe that no controllable set can hold both 1 and 2, since for those two states, we may have the same observation $\mathsf{Obs}1$ and there does not exist a common winning move. Finally, observe that no controllable set can hold both 2 and 4 neither, since they belong both to a common observation $\mathsf{Obs}2$ and there does not exist a common controllable set that would contain all their successors for a given move.

Finally, we can say that sets $\{1\}$, $\{2\}$, $\{4\}$ and $\{1,4\}$ are controllable but since we keep an antichain only, we will only keep the sets $\{1,4\}$ and $\{2\}$ as element of $\mathsf{CPre}(A)$.

We now state one crucial property that will be needed to ensure that a fixed point of this CPre operator is computable.

Lemma 6.6

The operator $\mathsf{CPre} : L \to L$ *is monotone for the partial ordering* $\sqsubseteq$.

Proof

Let $q, q' \in L$ *with* $q \sqsubseteq q'$, *it suffices to prove that* $\mathsf{CPre}(q) \subseteq \rceil\mathsf{CPre}(q')\lceil$, *from Lemma 6.1 and Lemma 6.2.*

By definition of CPre *we know that for* $s \in \mathsf{CPre}(q)$, *for all* $\mathsf{obs} \in \mathsf{Obs}$ *there exists* $\sigma \in \Sigma$ *such that there exists* $s' \in q$ *such that* $\mathsf{Post}_\sigma(s \cap \mathsf{obs}) \subseteq s'$. *Since* $q \sqsubseteq q'$, *we know that for all* $s' \in q$, *there exists* $s'' \in q'$ *such that* $s' \subseteq s''$. *Therefore, we have that for any* $\mathsf{obs} \in \mathsf{Obs}$, *there exists* $\sigma \in \Sigma$ *such that there exists* $s'' \in q'$ *such that* $\mathsf{Post}_\sigma(s \cap \mathsf{obs}) \subseteq s''$. *And so we have* $s \in \rceil\mathsf{CPre}(q')\lceil$.

Remark The controllable predecessor operator is also *monotone w.r.t. the set of observations* in the following sense: given a two-player game G, let CPre_1 (resp. CPre_2) be the operator defined on the set of observations Obs_1 (resp. Obs_2). If $\mathsf{Obs}_2 \sqsubseteq \mathsf{Obs}_1$, then for any $q \in L$ we have $\mathsf{CPre}_1(q) \sqsubseteq \mathsf{CPre}_2(q)$. That corresponds to the informal statement that it is easier to control a system with more precise observations.

In the following two sections, we will show how the CPre operator can be used to solve either *safety* or *reachability* games by computing two different, albeit similar, fixed points. Observe that on the contrary of games of perfect information, finding a solution to the reachability question for a game of imperfect information gives no information about the safety question defined using the same set of state, standing as the goal states in the first case and the bad states in the second case.

6.3.3 Solving Safety Games on the Lattice of Antichains

Theorem 6.1

[Safety condition] Let $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs}\rangle$ be a game of imperfect information. There exists a safe observation based strategy on G if and only if

$$\{\mathrm{E}\} \sqsubseteq \bigsqcup\{q \mid q = \mathsf{CPre}(q) \bigsqcap \{\overline{\mathrm{F}}\}\}. \tag{6.1}$$

Before proving this theorem, we give some intuition. We denote by Safe the set $\bigsqcup\{q \mid q = \mathsf{CPre}(q) \bigsqcap \{\overline{\mathrm{F}}\}\}$ which is the greatest fixed point of $\mathsf{CPre}(\cdot) \bigsqcap \{\overline{\mathrm{F}}\}$ (observe that this operator is monotone for $\sqsubseteq$, since $\mathsf{CPre}(\cdot)$ is monotone and forall $q, q' \in 2^S$ we have that $q \sqsubseteq q' \implies \forall q'' \in 2^S : q \bigsqcap q'' \sqsubseteq q' \bigsqcap q''$). Condition (6.1) states that the initial set is included in a set of Safe. Since Safe is a fixed point of the controllable predecessor operator, we know that in each set s of Safe, whatever the observation obs, we have a controllable action that can be played by the controller in every state $x \in s \cap \mathsf{obs}$ such that the state y reached after the move of the environment lies in one of the sets s' of Safe. Since for any $s \in \mathsf{Safe}$, we have that $s \cap \mathrm{F} = \varnothing$, those controllable actions define possible safe strategies.

Following this, there exists a winning strategy if Condition (6.1) holds. The other direction of the theorem is a direct consequence of Tarski's Theorem.

Proof

First, we give an effective construction of a winning strategy for G, in the form of a finite automaton. From the greatest fixed point Safe *of* CPre*, we define the finite automaton $A = \langle S_A, \mathrm{E}_A, \mathrm{F}_A, \Sigma_A, \rightarrow_A\rangle$ where*

- $S_A = \mathsf{Safe}$,
- $\mathrm{E}_A = \{s_0\}$ *where s_0 is picked in the set $\{s \in \mathsf{Safe} \mid \mathrm{E} \subseteq s\}$. The initial set is made of one of the sets including the initial set E of the game G. We know by 6.1 that at least one such set s exists, so there is always an initial state for the finite automaton.*
- $\mathrm{F}_A = \varnothing$.
- $\Sigma_A : \mathsf{Obs} \times \Sigma$. *Each transition will be labeled by an observation and a label of the game G.*

- $\rightarrow_A$ *is defined as follows : for every pair* $(s, \mathsf{obs}) \in \mathsf{Safe} \times \mathsf{Obs}$ *we pick a label* $\sigma \in \Sigma$ *such that* $\exists s' \in \mathsf{Safe} : \mathsf{post}_\sigma(s \cap \mathsf{obs}) \subseteq s'$ *and we fix that* $s \xrightarrow{(\mathsf{obs},\sigma)}_A s'$. *Since* Safe *is a fixed point of* $\mathsf{CPre}(\cdot) \sqcap \{\overline{\mathrm{F}}\}$, *we know that such a label* σ *and such a set* s' *always exist. Notice that we obtain a deterministic finite automaton.*

In this automaton, states are labeled with a set of states of the game and transitions are labeled with one observation and one action. Intuitively, each set s *of* Safe *is a known maximal controllable set and the transitions going out of* s *indicate which action to choose for each observation to ensure that the next state is also controllable.*

Let $\hat{K} : \mathsf{Obs}^+ \rightarrow \mathsf{Safe}$ *be the function that gives the set* $s \in \mathsf{Safe}$ *reached by the automaton* A *after a sequence of observation* $\overline{\mathsf{obs}}$. *The function* $\hat{K}$ *is defined recursively as follows:*

- $\hat{K}(\cdot) = s_0$
- $\hat{K}(\overline{\mathsf{obs}} \cdot \mathsf{obs}) \in \mathsf{Safe}$ *is such that* $\exists \sigma \in \Sigma : \hat{K}(\overline{\mathsf{obs}}) \xrightarrow{(\mathsf{obs},\sigma)}_A \hat{K}(\overline{\mathsf{obs}} \cdot \mathsf{obs})$. *By definition of* $\rightarrow_A$ *there exist one and only one such* σ.

The strategy defined by A *is then also defined recursively:*

- $\lambda(\mathsf{obs}) = \sigma$ *such that* $s_0 \xrightarrow{(\mathsf{obs},\sigma)}_A \hat{K}(\mathsf{obs})$
- $\lambda(\overline{\mathsf{obs}} \cdot \mathsf{obs}) = \sigma$ *such that* $\hat{K}(\overline{\mathsf{obs}}) \xrightarrow{(\mathsf{obs},\sigma)}_A \hat{K}(\overline{\mathsf{obs}} \cdot \mathsf{obs})$

Now we proceed with the proof of the theorem.

1. *Let us assume that equation* (6.1) *holds. We show that the strategy* λ *defined by* A *is such that for all* $\overline{x} = x_0 x_1 x_2 \cdots \in \mathsf{Outcome}_\lambda(G)$, *we have* $\forall 0 \leq i : x_i \notin \mathrm{F}$. *To belong to* $\mathsf{Outcome}_\lambda(G)$, $\overline{x} = x_0 x_1 \ldots$ *is such that there exists a sequence of observations* $\overline{\mathsf{obs}} = \mathsf{obs}_0 \mathsf{obs}_1 \ldots$ *such that for all* $0 \leq i$, $x_i \in \mathsf{obs}_i \wedge x_i \xrightarrow{\lambda(\overline{\mathsf{obs}}_i)} x_{i+1}$

   Now we prove that $\forall 0 \leq i : x_i \in \hat{K}(\overline{\mathsf{obs}}_{i-1})$ *which shows that* λ *is safe since the function* $\hat{K}$ *outputs elements of* Safe, *which are sets where no final state is present (by the definition of* Safe*).*

   We show this by induction on index i.

- *The base case consists in proving that* $x_0 \in \hat{K}(\cdot)$. *This is the case since as* $\overline{x}$ *is a play,* $x_0 \in \mathrm{E}$ *while* $\hat{K}(\cdot) = s_0$ *and* s_0 *is as set chosen so that* $\mathrm{E} \subseteq s_0$.
- *The inductive case consists in proving that* $x_{i+1} \in \hat{K}(\overline{\mathsf{obs}}_i)$ *while knowing (by induction) that* $x_i \in \hat{K}(\overline{\mathsf{obs}}_{i-1})$. *We know that* $\lambda(\overline{\mathsf{obs}}_{i-1} \cdot \mathsf{obs}_i) = \sigma$ *such that* $\hat{K}(\overline{\mathsf{obs}}_{i-1}) \xrightarrow{(\mathsf{obs}_i, \sigma)}_A \hat{K}(\overline{\mathsf{obs}}_i)$. *From state* x_{i-1}, *with observation* obs_i *the strategy thus proposes the action* σ. *As* $\hat{K}(\overline{\mathsf{obs}}_{i-1})$ *is an element of* Safe, *we know, by definition of* CPre *and* A *that* $\mathsf{post}_\sigma(\hat{K}(\overline{\mathsf{obs}}_{i-1}) \cap \mathsf{obs}_i) \subseteq \hat{K}(\overline{\mathsf{obs}}_i)$, *which implies that* x_i *is an element of* $\hat{K}(\overline{\mathsf{obs}}_i)$

2. *Let us assume that* λ *is an observation based strategy that is winning on* G. *We must show that* (6.1) *holds. Let* $W_\lambda = \{K_\lambda(\overline{\mathsf{obs}}) \mid \overline{\mathsf{obs}} \in \mathsf{Obs}^+\}$. *Obviously* $\{\mathrm{E}\} \sqsubseteq W_\lambda$ *since* $K_\lambda(\cdot) = E$ *for all* λ. *We now prove that* $W_\lambda \sqsubseteq \mathsf{Safe}$. *This will imply, since* $\sqsubseteq$ *is a partial order that* $\{\mathrm{E}\} \sqsubseteq \bigsqcup\{q \mid q = \mathsf{CPre}(q) \bigsqcap \{\overline{\mathrm{F}}\}\} = \mathsf{Safe}$. *This is exactly equation (6.1) that we wanted to prove.*

   So, we prove that $W_\lambda \sqsubseteq \mathsf{Safe}$. *We just need to prove that* $W_\lambda \sqsubseteq \mathsf{CPre}(W_\lambda) \bigsqcap \{\overline{\mathrm{F}}\}$ *since* Safe *could be equivalently redefined as* $\bigsqcup\{q \mid q \sqsubseteq \mathsf{CPre}(q) \bigsqcap \{\overline{\mathrm{F}}\}\}$. *Thanks to Lemma 6.1, it suffices to show that* $W_\lambda \subseteq \rceil \mathsf{CPre}(W_\lambda) \bigsqcap \{\overline{\mathrm{F}}\} \rceil$.

   Let $s \in W_\lambda$. *We know that* $\exists \overline{\mathsf{obs}} \in \mathsf{Obs}^+ : s = K_\lambda(\overline{\mathsf{obs}})$. *Now, we know that* $\forall \mathsf{obs} \in \mathsf{Obs} : K_\lambda(\overline{\mathsf{obs}} \cdot \mathsf{obs}) \in W_\lambda$, *that is,* $\mathsf{post}_{\lambda(\overline{\mathsf{obs}} \cdot \mathsf{obs})}(K_\lambda(\overline{\mathsf{obs}}) \cap \mathsf{obs}) \in W_\lambda$ *and thus* $\forall \mathsf{obs} \cdot \exists \sigma \in \Sigma \cdot \exists s' \in W_\lambda : \mathsf{post}_\sigma(s) \subseteq s'$ *where* $\sigma = \lambda(\overline{\mathsf{obs}} \cdot \mathsf{obs})$ *and* $s' = K_\lambda(\overline{\mathsf{obs}} \cdot \mathsf{obs})$ *(as* $s = K_\lambda(\overline{\mathsf{obs}})$*).*

   So, we have $s \in \mathsf{CPre}(W_\lambda)$. *Now, as* λ *is a winning strategy* $K_\lambda(\overline{\mathsf{obs}}) \subseteq \overline{\mathrm{F}}$, *for all* $\overline{\mathsf{obs}} \in \mathsf{Obs}^+$, *which means that* $\forall s \in W_\lambda : s \subseteq \overline{\mathrm{F}}$. *This means that, by definition of* $\bigsqcap$, $s \in \rceil \mathsf{CPre}(W_\lambda) \bigsqcap \{\overline{\mathrm{F}}\} \rceil$

6.3.4 Solving Reachability Games on the Lattice of Antichains

Theorem 6.2

[Reachability condition] Let $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs} \rangle$ *be a game of imperfect information. There exists a reaching observation based strategy on* G *if and only if*

$$\{\mathrm{E}\} \sqsubseteq \bigsqcap \{q \mid q = \mathsf{CPre}(q) \bigsqcup \{\mathrm{F}\}\}. \tag{6.2}$$

Proof

Let us call Reaching *the set* $\bigsqcap\{q \mid q = \mathsf{CPre}(q) \bigsqcup \{\mathrm{F}\}\}$*, which is the least fixed point of the operator* $\mathsf{CPre}(q) \bigsqcup \{\mathrm{F}\}$ *(observe that this operator is monotone for* $\sqsubseteq$*, since* $\mathsf{CPre}(\cdot)$ *is monotone and forall* $q, q' \in 2^S$ *we have that* $q \sqsubseteq q' \implies \forall q'' \in 2^S : q \bigsqcup q'' \sqsubseteq q' \bigsqcup q''$*).*

1. *We first prove that the existence of a reaching strategy* λ *implies that* $\{\mathrm{E}\} \sqsubseteq$ Reaching.

   We first define recursively a family of sets $q_i \subseteq 2^S \times \mathsf{Obs}^i$ *(for* $i \in \mathbb{N}$*):*

   - $q_0 = \{(\mathrm{E}, \cdot)\}$*;*
   - $q_i = \left\{ \left(\mathsf{post}_{\lambda(\overline{\mathsf{obs}}_{i-1} \cdot \mathsf{obs}_i)}(s \cap \mathsf{obs}_i), \overline{\mathsf{obs}}_i \right) \mid \mathsf{obs}_i \in \mathsf{Obs} \wedge (s, \overline{\mathsf{obs}}_{i-1}) \in q_{i-1} \right\}$*;*

   Furthermore, let us define $q_i^S = \{s \subseteq S \mid \exists \overline{\mathsf{obs}} \in \mathsf{Obs}^+ : (s, \overline{\mathsf{obs}}) \in q_i\}$.

   Since λ *is a reaching strategy we know that there exists a* k *such that* $q_k^S \sqsubseteq \{\mathrm{F}\}$*. Let us prove by induction that* $q_{k-j}^S \sqsubseteq$ Reaching *for all* $0 \leq j \leq k$ *, which will give us that* $\{\mathrm{E}\} \sqsubseteq$ Reaching.

   (a) *The base case is* $j = 0$*. In this case, we have that* $q_{k-j}^S \sqsubseteq \{\mathrm{F}\}$*, which implies that* $q_{k-j}^S \sqsubseteq$ Reaching.

   (b) *For the inductive case, we assume that* $q_{k-j}^S \sqsubseteq$ Reaching *and we prove that* $q_{k-(j+1)}^S \sqsubseteq$ Reaching*. This comes easily as*

   $$
   \begin{aligned}
   & q_{k-(j+1)}^S \sqsubseteq \mathsf{CPre}(q_{k-j}) \\
   & \quad \textit{(By def. of } \mathsf{CPre} \textit{ and } q_i^S \textit{)} \\
   \implies & q_{k-(j+1)}^S \sqsubseteq \mathsf{CPre}(\mathsf{Reaching}) \\
   & \quad \textit{(By induction hypothesis and monotony of } \mathsf{CPre} \textit{)} \\
   \implies & q_{k-(j+1)}^S \sqsubseteq \mathsf{CPre}(\mathsf{Reaching}) \bigsqcup \{\mathrm{F}\} \\
   \implies & q_{k-(j+1)}^S \sqsubseteq \mathsf{Reaching} \\
   & \quad \textit{(As } \mathsf{Reaching} \textit{ is a fixed point)}
   \end{aligned}
   $$

2. *Second, we prove that* $\{\mathrm{E}\} \sqsubseteq$ Reaching *implies the existence of a reaching strategy* λ*.*

   We define a family of antichains p_i*:*

- $p_0 = \{\mathrm{F}\}$;
- $p_i = \mathsf{CPre}(p_{i-1}) \bigsqcup \{\mathrm{F}\}$.

By Tarski's fixed point theorem, we know that $\mathsf{Reaching} = p_n$ *for some* $n \in \mathbb{N}$.

We now define a strategy λ *as follows: fix* $\lambda(\overline{\mathsf{obs}}_i)$ *to be* σ *such that* $\exists s \in p_{n-i}$ *such that* $\mathsf{post}_\sigma(K_\lambda(\overline{\mathsf{obs}}_{i-1}) \cap \mathsf{obs}_i) \subseteq s$. *If we assume that* $\{K_\lambda(\overline{\mathsf{obs}}_{i-1})\} \sqsubseteq p_{n-i}$, *this definition is meaningful, since we know that there always exist such a* σ *by definition of* CPre *and by the totality of the transition relation.*

Proving that $\forall \overline{\mathsf{obs}}_{i-1}$ *such that* $i > 0$,$\{K_\lambda(\overline{\mathsf{obs}}_{i-1})\} \sqsubseteq p_{n-i}$ *is easy to do by induction on* i*:*

- *For* $i = 0$, *we must prove that* $\{K_\lambda(\cdot)\} \sqsubseteq p_n$, *which is true since* $K_\lambda(\cdot) = \mathrm{E}$ *and* $\{E\} \sqsubseteq \mathsf{Reaching} = p_n$;
- *Assuming* $\{K_\lambda(\overline{\mathsf{obs}}_{i-1}\} \sqsubseteq p_{n-i}$, *the fact that* $\{K_\lambda(\overline{\mathsf{obs}}_i\} \sqsubseteq p_{n-(i+1)}$ *comes directly from the definition of* $\lambda(\overline{\mathsf{obs}}_i)$.

The strategy is reaching since $K_\lambda(\overline{\mathsf{obs}}_{n-1}) \subseteq \mathrm{F}$ *for all* $\overline{\mathsf{obs}}_{n-1} \in \mathsf{Obs}^+$, *since* $\{K_\lambda(\overline{\mathsf{obs}}_{n-1})\} \sqsubseteq p_0 = \{\mathrm{F}\}$ *for all* $\overline{\mathsf{obs}}_{n-1} \in \mathsf{Obs}^+$

6.4 Games with Finite State Space

In this section we show that computing a fixed point of $\mathsf{CPre}(\cdot)$ for finite state games can be done in EXPTIME. We also compare our algorithm based on the lattice of antichains with the classical technique of [Rei84].

6.4.1 Fixed Point Algorithm

To compute the greatest fixed point of $\mathsf{CPre}(\cdot) \bigsqcap \{\overline{\mathrm{F}}\}$, we iteratively repeat Algorithm 4, starting from the antichain $\{S\}$ and to compute the least fixed point of $\mathsf{CPre}(\cdot) \bigsqcup \{\mathrm{F}\}$, we iteratively repeat Algorithm 4, starting from the antichain $\varnothing$. Algorithm 4 constructs systematically all subsets of S and checks at line (9) whether they belong to $\mathsf{CPre}(q)$. This is done by treating all subsets of size i before the subsets of size $i-1$, so we avoid to treat the subsets of the already included subsets and the result is in reduced form. Therefore, Algorithm 4 uses

Algorithm 4: Algorithm for CPre.

Data : A game of imperfect information $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs} \rangle$ and a set $q \in L$.

Result: The set $Z = \mathsf{CPre}(q)$.

begin

1 $Z \leftarrow \varnothing$;

2 $\mathsf{Level} \leftarrow \{\bigcup_{\sigma \in \Sigma} \mathsf{pre}_\sigma(\bigcup_{s \in q} s)\}$;

3 **while** $\mathsf{Level} \neq \varnothing$ **do**

4 $\mathsf{Current} \leftarrow \mathsf{Level}$;

5 $\mathsf{Level} \leftarrow \varnothing$;

6 **while** $\mathsf{Current} \neq \varnothing$ **do**

7 Pick $s \in \mathsf{Current}$;

8 $\mathsf{Current} \leftarrow \mathsf{Current} \backslash \{s\}$;

9 **if** *for all* $\mathsf{obs} \in \mathsf{Obs}$ *we have :*
for some $\sigma \in \Sigma$*, there exists* $s' \in q$ *such that* $\mathsf{post}_\sigma(s \cap \mathsf{obs}) \subseteq s'$
then

10 $Z \leftarrow Z \cup \{s\}$;

else

11 $\mathsf{Level} \leftarrow \mathsf{Level} \cup \{s' \mid s' \in \mathsf{Children}(s) \wedge \forall s'' \in Z \cup \mathsf{Current} : s' \not\subseteq s''\}$;

12 **return** Z;

end

the following operator $\mathsf{Children}(s) = \{s \backslash \{x\} \mid x \in s\}$ which returns the subsets of s of cardinality $|s| - 1$.

Lemma 6.7
Algorithm 4 computes CPre *in EXPTIME in the size of the game.*

Proof

- **Correctness** *We show that* $s \in \mathsf{CPre}(q)$ *iff* $s \in Z$. *First, let us define* Init *as* $\bigcup_{\sigma \in \Sigma} \mathsf{pre}_\sigma(\bigcup_{s \in q} s)$, *that is the set of predecessors of the states appearing in* q. *Obviously, any set belonging to* $\mathsf{CPre}(q)$ *must be included in* Init, *since to be included in the controllable predecessors of a set* s', *a set* s *must first be included in the* predecessors *of* s'. *Now, the proof is based on an invariant. The* (i)*th time the algorithm reaches line (3) the following assertion is true:*

$$\begin{cases} Z = \{s \mid |s| > m - i + 1 \wedge s \in \mathsf{CPre}(q)\} \\ \mathsf{Level} = \{s \mid |s| = m - i + 1 \wedge \{s\} \not\sqsubseteq Z \wedge s \subseteq \mathsf{Init}\} \end{cases}$$

where $m \leq n$ *is the size of* Init. *Observe that by definition* Z *is an antichain. We prove that this invariant holds by induction for every* i.

  - *If* $i = 1$, *the invariant is true as* $\mathsf{Level} = \{\mathsf{Init}\}$ *and* $Z = \varnothing$. *The latter is correct, since no set of size greater than the size of* Init *can be an element of* $\mathsf{CPre}(q)$;
  - *Suppose the invariant is true for* i. *Let us prove that then, it is still true for* $i + 1$. *For every* s *picked at line* (7), *there are two possibilities:*
    - $s \in \mathsf{CPre}(q)$. *In this case,* s *satisfies the test of line (9) and is added to* Z, *which is good, since by induction hypothesis,* $\{s\} \not\sqsubseteq Z$, *and thus* Z *remains an antichain.* s *is thus maximal regarding* Z *and no greater set can be added later since only sets of the same or smaller size could still be added.*
    - $s \notin \mathsf{CPre}(q)$. *In this case,* s *does not satisfy the test of line (9) and its children are added to the next level to be treated, as long as they are not dominated by* Z *(in which case they do not need to be tested for belonging to* $\mathsf{CPre}(q)$*), or by elements of* Current *(in which case, they will be added later if no element of* Current *belongs to* $\mathsf{CPre}(q)$.

We end the proof by stating that for the first i such that $\mathsf{Level} = \varnothing$*, the invariant implies that* $Z = \mathsf{CPre}(q)$*.*

- **Complexity** *For each s, the body of the inner loop is executed in time $O(|\mathsf{Obs}|.|\Sigma|.|q|.|S^2|)$, which is essentially the complexity of the test of line (4) ($O(|S^2|)$ is a superior bound for the computation of $\mathsf{Post}_\sigma(\cdot)$). The loop is executed at most 2^n times since all the subsets of S picked at line (7) are different. The complexity is thus $O(2^n.|\mathsf{Obs}|.|\Sigma|.|q|.|S^2|)$*

So we have an algorithm to compute CPre in exponential time. We now prove a lemma that will allow us to state that we will not have to iterate the Algorithm 4 more than an exponential number of times (in the size of the game) before obtaining a fixed point.

Lemma 6.8
An ascending (or descending) chain in $\langle L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top\rangle$ has at most $2^n + 1$ elements, where $n = |S|$.

Proof
For any finite set P, let $\mathbb{L}(P)$ be the complete lattice $\langle 2^P, \subseteq, \bigcup, \bigcap, \varnothing, \{P\}\rangle$. An ascending chain in $\mathbb{L}(P)$ has at most $|P| + 1$ elements. Notice that $q \sqsubseteq q'$ if and only if $\rceil q \lceil \subseteq \rceil q' \lceil$ (Lemma 6.2). For $Y \subseteq L$, define $\rceil Y \lceil = \{\, \rceil q \lceil \mid q \in Y\}$. If $Y \subseteq L$ is an ascending chain in the lattice $\langle L, \sqsubseteq, \bigsqcup, \bigsqcap, \varnothing, \{S\}\rangle$, then $\rceil Y \lceil$ is an ascending chain in $\langle \rceil L \lceil, \sqsubseteq, \bigsqcup, \bigsqcap, \varnothing, 2^S\rangle$ and is also an ascending chain in $\mathbb{L}(2^S)$. Observe that if $q \neq q'$ then $\rceil q \lceil \neq \rceil q \lceil'$. Therefore $|Y| \leq |\rceil Y \lceil| \leq 2^n + 1$ since an ascending chain in $\mathbb{L}(2^S)$ has at most $2^n + 1$ elements.

Theorem 6.3
The imperfect information safety and reachability problems are EXPTIME-complete for finite games.

Proof
We first prove the upper bound. From Lemma 6.8 and since CPre *is monotone, we reach the greatest fixed points* Safe *or* Reaching *after at most $O(2^n)$ iterations of* CPre*. From Lemma 6.7 computing* CPre *can be done in EXPTIME. The conclusion follows. For the lower bound, we solve a more general problem than Reif [Rei84] which gives us the EXPTIME-hardness.*

6.4.2 Example of Safety Game

Consider again the two-player game G_1 on Figure 6.2 with state space $S = \{1, 2, 3, \text{Bad}\}$, set of initial states $\text{E} = \{2, 3\}$ and actions $\Sigma = \{a, b\}$. The observation set is $\mathsf{Obs} = \{\mathsf{obs}_1, \mathsf{obs}_2\}$ with $\mathsf{obs}_1 = \{1, 2, \text{Bad}\}$ and $\mathsf{obs}_2 = \{1, 3, \text{Bad}\}$.

For the controller, the goal is to avoid the final states $\text{F} = \{\text{Bad}\}$ in which there is no controllable action. So the controller must play a b in state 1, an a in state 2, and a c in state 3. However the controller cannot distinguish 1 from 2, or 1 from 3 using only the current observation. Thus, to discriminate those states, the controller has to rely on its memory of the past observations.

We show below the iterations of the fixed point algorithm and the construction of the strategy. The fixed point computation starts from $\top = \{S\}$.

$$\begin{array}{rcll} S_1 & = & \mathsf{CPre}(\{S\}) \bigsqcap \{\overline{\text{F}}\} & = \{\{1, 2, 3\}\} \\ S_2 & = & \mathsf{CPre}(S_1) \bigsqcap \{\overline{\text{F}}\} & = \{\{1\}, \{2, 3\}\} \\ S_3 & = & \mathsf{CPre}(S_3) \bigsqcap \{\overline{\text{F}}\} & = S_2 \end{array}$$

Since $S_3 = S_2$, we have $\mathsf{Safe} = S_3 = \{\{1\}, \{2, 3\}\}$. The existence of a winning strategy is established by condition (6.1) of Theorem 6.1 since the set $\text{E} = \{2, 3\}$ is dominated in Safe.

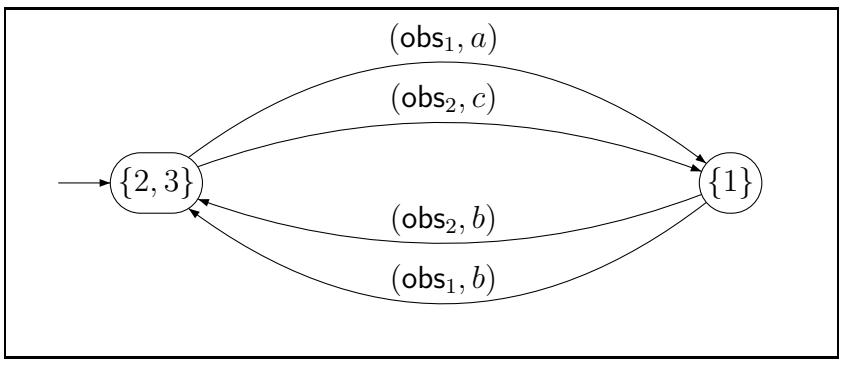

Figure 6.4: An automaton giving a winning strategy for the two-player game G_1 of Figure 6.2

From the fixed point, using the construction given in the proof of Theorem 6.1, we construct the automaton of Figure 6.4 which encodes a winning strategy. Indeed, when the game starts the control is either in state 2 (if the given observation

is obs_1) or in state 3 (if the given observation is obs_2). In the first case, the controller plays a and in the second case, it plays c. Then the game lies in state 1. According to the strategy automaton, the controller plays a b whatever the observation to get back to the initial situation. From there, the controller can clearly iterate this strategy, avoiding state Bad forever.

6.4.3 Comparison with the Classical Technique of Reif

In [Rei84], John H. Reif gives an algorithm to transform a game of incomplete information G into a game G' of perfect information on the histories of G. This construction is such that there exists a reaching or safe strategy for G iff there exists a reaching or safe strategy for G'.

The idea of the construction can be expressed as follows :

given a game of incomplete information $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow_1, \mathsf{Obs}\rangle$ define a perfect information game $G' = \langle S', \mathrm{E}', \mathrm{F}', \Sigma, \rightarrow_2, \mathsf{Obs}\rangle$ as follows:

- S' is the set of knowledges $K(\overline{\mathsf{obs}}_k, \overline{\sigma}_k)$ such that $(\overline{\mathsf{obs}}_k, \overline{\sigma}_k)$ is an history of G.
- E' is the set E.
- $\mathrm{F}' = \begin{cases} \{s \in S' \mid s \cap \overline{F} \neq \varnothing\} & \text{if the original game is a safety game} \\ \{s \in S' \mid s \subseteq F\} & \text{if the original game is a reachability game} \end{cases}$
- $\rightarrow_2$ is defined as follows: $K(\overline{\mathsf{obs}}_k, \overline{\sigma}_k) \xrightarrow{\sigma_{k+1}}_2 K(\overline{\mathsf{obs}}_{k+1}, \overline{\sigma}_{k+1}), \forall \mathsf{obs}_{k+1} \in \mathsf{Obs}$.
- $\mathsf{Obs} = \{\{x\} \mid x \in S'\}$ as G' is a game of perfect information

Solving the resulting games (reachability and safety) of perfect information G' requires linear time in the size of S' but there exist games of incomplete information G requiring the construction of a game of perfect information of size exponentially larger than the size of G.

As our algorithm does not require this determinization and is goal oriented, it is easy to find infinite families of games where our method is exponentially faster than Reif's algorithm. This is formalized in the next theorem.

Theorem 6.4
There exist infinitely many finite state games of incomplete information for which the algorithm of [Rei84] requires an exponential time where our algorithm needs only polynomial time.

Proof
The infinite family of games of Figure 6.5 are such games. For any of those games, $\Sigma = \{a, b\}$*, there is only one possible observation* obs_1 *on every state and* $\mathrm{F} = \{k\}$*. For this family of games, the technique of [Rei84] constructs a game of perfect information* G' *with* $O(2^k)$ *states.*

Let us denote $\mathcal{S}_i$ *the set of sets of the form* $\{1, s_1, \ldots, s_j\}$ *such that* $\forall m : s_m \in \{2, \ldots, i\}$*. We can easily show by induction on* $i \leq k$ *that, for each set* s *of* $\mathcal{S}_i$*, there exists an history* $(\overline{\mathsf{obs}}_{i-1}, \overline{\sigma}_{i-1})$ *such that* $K(\overline{\mathsf{obs}}_{i-1}, \overline{\sigma}_{i-1}) = s$*. This is obviously the case for* $i < 2$*. Now, the induction step : given a set* $s \in \mathcal{S}_i$ *we want to construct an history* $(\overline{\mathsf{obs}}_{i-1}, \overline{\sigma}_{i-1})$ *such that* $K(\overline{\mathsf{obs}}_{i-1}, \overline{\sigma}_{i-1}) = s$*. Let* $s = \{1, s_1, s_2, \ldots, s_j\}$*. We know by induction hypothesis that we can construct an history* $s' = (\overline{\mathsf{obs}}_{i-2}, \overline{\sigma}_{i-2})$ *such that* $K(\overline{\mathsf{obs}}_{i-2}, \overline{\sigma}_{i-2}) = \{1\} \cup \{s_1 - 1, \ldots, s_j - 1\}$ *since this last set clearly belongs to* $\mathcal{S}_{i-1}$*. Now we can construct* $(\overline{\mathsf{obs}}_{i-1}, \overline{\sigma}_{i-1})$ *as follows : it is* $(\overline{\mathsf{obs}}_{i-1}, \overline{\sigma}_{i-2}.a)$ *if* $2 \in s$ *or* $(\overline{\mathsf{obs}}_{i-1}, \overline{\sigma}_{i-2}.b)$ *if* $2 \notin s$*.*

On the other hand, if this game is considered as a safety game, the execution time of our algorithm is polynomial in k *for this family of examples since it stops after* k *iterations of the* CPre *operator, which is computed in* $O(k^2)$ *(which is a maximal bound on the computation of either* pre_σ *or* post_σ*). Indeed, during the computation of the fixed point, the* (i)*th antichain computed will be of the form* $\{\{1, 2, \ldots, k - i\}\}$*, which is exactly the set to which* Level *is initialized in Algorithm 4. Computation of* CPre *in this case requires only the time for checking that the initial guess is the good result.*

6.4.4 An Application: Universality of Finite Automata

In this section, we show how to decide if a nondeterministic finite automaton is universal by solving a game of imperfect information.

Game interpretation of universality Consider the following game played by a protagonist and an antagonist. The protagonist wants to establish that a given

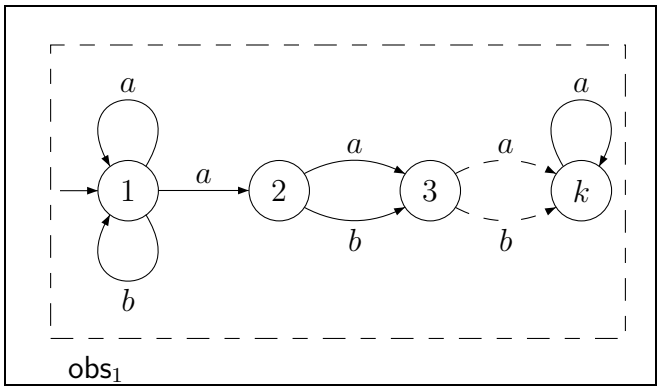

Figure 6.5: Family of games ($k \in \mathbb{N}$, $\mathrm{F} = \{k\}$) requiring exponential time for [Rei84] but polynomial time with our algorithm.

NFA A does not accept the language Σ^*. The protagonist has to provide a finite word w such that, no matter which run of A over w the antagonist chooses, the run does not end in an accepting state. This game is a one-shot game. However, to obtain a fixed point solution to the universality problem, we can consider a multi-round game interpretation of this problem: in each round of the game, the protagonist provides a single letter σ, and the antagonist decides how to update the state of A on input σ according to the nondeterministic transition relation. To be equivalent to the one-shot game, the protagonist must not be able to observe the state of the automaton, which is chosen by the antagonist. So, we have to consider the game $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs}\rangle$ where the protagonist cannot distinguish between states of the automaton, that is $\mathsf{Obs} = \{S\}$. This is a *blind game*, a special case of game of imperfect information. Thus we can solve the universality problem by looking for the existence of reaching strategies for G. Algorithm 5 checks universality of a finite automaton using the CPre^A operator, which simplifies CPre by taking into account that $\mathsf{Obs} = \{S\}$:

Definition 6.8 (Controllable Predecessors for Universality of FA)

$$\mathsf{CPre}^A(q) = \lceil \{s \subseteq S \mid \exists \sigma \in \Sigma \cdot \exists s' \in q : \mathsf{post}_\sigma(s) \subseteq s'\} \rceil$$

The problem of universality of finite automaton is known to be PSpace-complete, so we have a lower bound for the complexity of this fixed point algorithm. For-

Algorithm 5: Antichain algorithm for testing universality of finite automata.

Data : a nondeterministic finite automaton $A = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$.
begin
1 $\mathsf{Start} \leftarrow \{\mathrm{E}\}$;
2 $G \leftarrow \{\mathrm{F}\}$;
3 $\mathsf{Frontier} \leftarrow G$;
4 **while** $(\mathsf{Frontier} \neq \varnothing) \wedge (\mathsf{Start} \not\sqsubseteq \mathsf{Frontier})$ **do**
5 $\quad \mathsf{Frontier} \leftarrow \{s \in \mathsf{CPre}^A(\mathsf{Frontier}) \mid \{s\} \not\sqsubseteq G\}$;
6 $\quad G \leftarrow G \sqcup \mathsf{Frontier}$;
7 **return** $(\mathsf{Start} \not\sqsubseteq \mathsf{Frontier})$;
end

tunately, Algorithm 5 performs well in lots of cases : to compare the performance of our antichain algorithm to the performance of various implementations of subset-construction based algorithms, we used a large set of examples generated in the probabilistic framework by Tabakov and Vardi [TV05]. This framework was proposed with the explicit purpose of comparing the performances of algorithms on finite automata. In their experiments, the authors conclude that explicit determinization, as implemented in [Mø04], outperforms the algorithm of Brzozowski [BL80], as well as newer implementations, which use symbolic methods for the subset construction. Our experimental results show that our implementation of the antichain algorithm is considerably faster, on the whole parameter space of the probabilistic framework, than the most efficient implementation of the standard algorithm. We now explain the details of our implementation and of the random model and we give the experimental results.

Two symbolic implementations of antichains We implemented our new algorithm for testing universality on top of NuSMV [CCGR99] and the BDD library CUDD [Som98]. We considered two encodings of NFAs in NuSMV, and correspondingly, two encodings of antichains of state sets using BDDs.

Fully symbolic encoding In the first encoding, we associate a boolean variable with each state of an NFA. A valuation of the variables corresponds to a state set, and a BDD represents a set of state sets. Two valuations v_1 and v_2 for a set X of variables are incomparable iff there exist $x, y \in X$ such that $v_1(x) > v_2(x)$ and $v_1(y) < v_2(y)$. If the BDD contains only valuations that are incomparable, then it represents an antichain of state sets. We call this encoding *fully symbolic*.

Semi-symbolic encoding In the second encoding, we associate an integer with each state of the automaton. Then a single integer counter is used to encode the current state. A BDD represents a set of integer values and so a set of states. An antichain of state sets is represented by a set[3] of BDDs that are incomparable for valuation inclusion. We call this encoding *semi-symbolic*.

Algorithm For both encodings, we use the Algorithm 5 to check universality. To avoid computing CPre^A twice for the same set, the algorithm computes iteratively CPre^A only on the frontier sets, which are the sets that were added to the approximation G of the least fixed point $\mathcal{F}$ in the previous iteration. When the automaton is not universal, then $\mathcal{F}$ is not fully computed, because we stop the computation as soon as one of the sets in G contains all the initial states.

The randomized model To evaluate the antichain algorithm and compare with the subset algorithm, we use a random model to generate NFAs. This model was recently proposed by Tabakov and Vardi to compare the efficiency of some algorithms for automata [TV05]. In the model, the input alphabet is fixed to $\Sigma = \{0, 1\}$, and for each letter $\sigma \in \Sigma$, a number k_σ of different state pairs $(s, s') \in S \times S$ are chosen uniformly at random before the corresponding transitions (s, σ, s') are added to the automaton. The ratio $r_\sigma = \frac{k_\sigma}{|S|}$ is called the *transition density* for σ. This ratio represents the average out degree of each state for σ. In all experiments, we choose $r_0 = r_1$, and denote the transition density by r. The model contains a second parameter: the *density f of accepting states*. There is only one initial state, and the number m of accepting states is linear in the total number of states, as determined by $f = \frac{m}{|S|}$. The accepting states themselves are chosen uniformly at

[3] In practice implemented by a simply linked list.

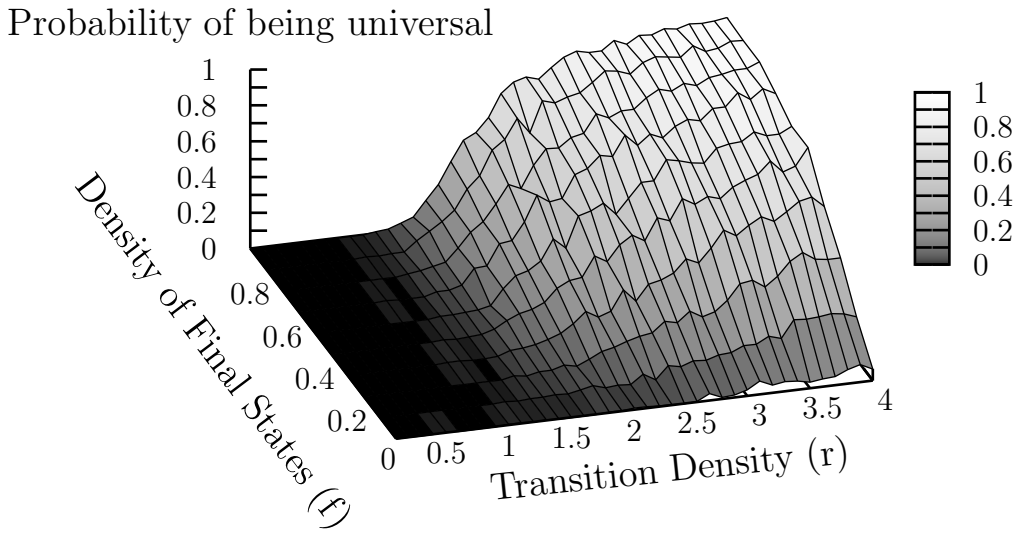

Figure 6.6: Probability of universal automata ($|S| = 30$).

random. Observe that, since the transition relation is not always total, automata with $f = 1$ are not necessarily universal[4].

Tabakov and Vardi have studied the space of parameter values for this model and argue that "interesting" automata are generated by the model when the two parameters r and f vary. They have run large tests to evaluate the probability for an automaton to be universal as a function of the parameters. We reproduced those experiments for a greater space of parameter values and obtained a similar distribution (Figure 6.6). To generate each sample point, we checked the universality of 200 random automata with 30 states.

Performance comparison We compare the performance of our antichain algorithm with the tool `dk.brics.automaton` developed by Møller [Mø04], which implements the forward subset algorithm and stops determinization whenever a rejecting state is encountered. According to the experiments of Tabakov and Vardi, this tool, which uses explicit state representation, is the most efficient one for checking universality [TV05]. For the comparison, we use the semi-symbolic en-

[4]In practice, to match the previous assumption that the transition relation is total,the transition relation can be easily made total by adding one bad states and creating transitions to this state for each unforeseen label.

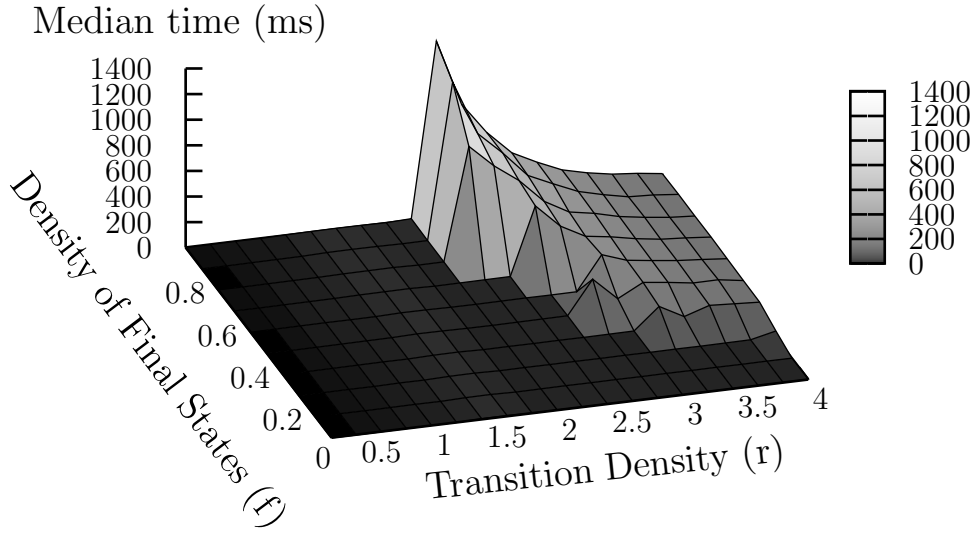

Figure 6.7: Median execution time for the subset algorithm ($|S| = 175$).

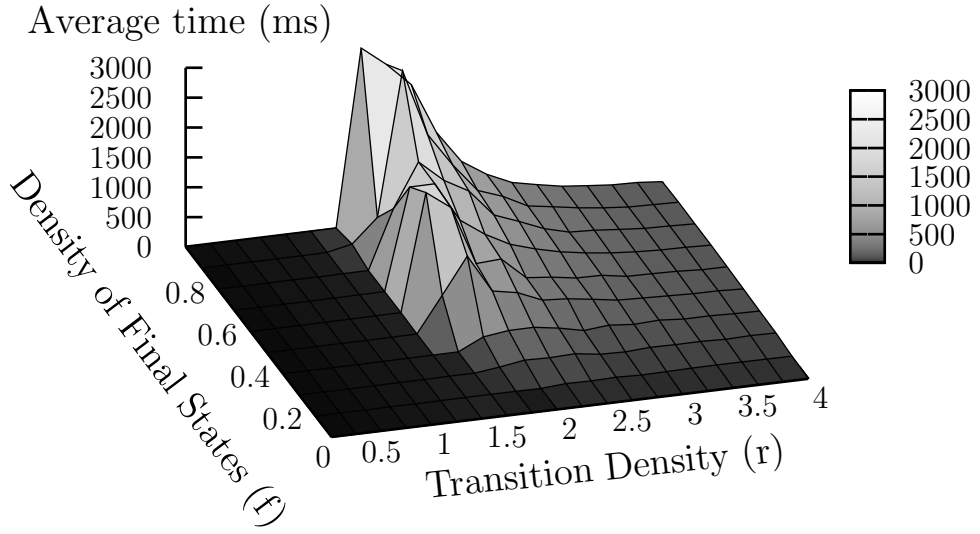

Figure 6.8: Average execution time for the subset algorithm ($|S| = 175$).

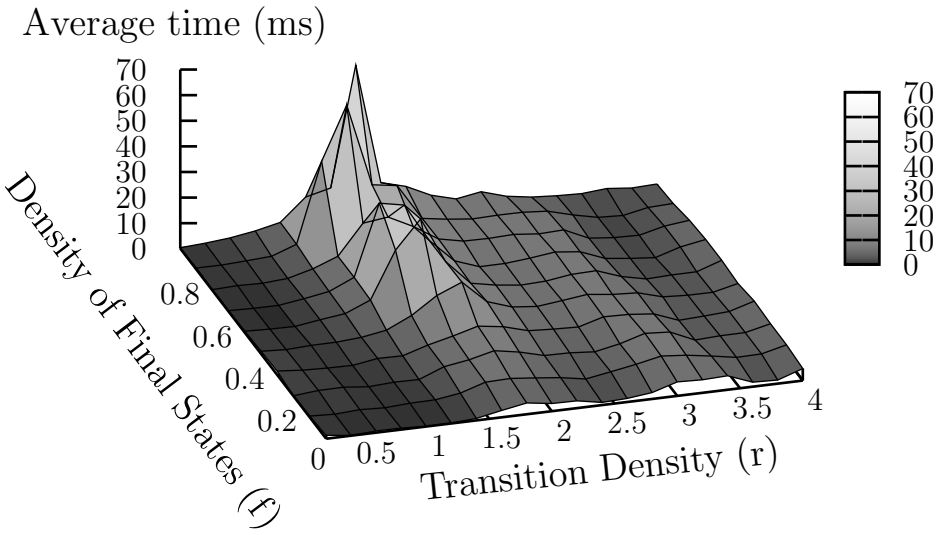

Figure 6.9: Average execution time for the semi-symbolic antichain algorithm ($|S| = 175$).

coding of antichains, which turns out to be much more efficient than the fully symbolic encoding. The comparison is carried out on the whole parameter space of the randomized model. All experiments are conducted on a biprocessor Linux station (two 3.06Ghz Intel Xeons with 4GB of RAM). We only measure the execution times for the universality test in both approaches, not the time for parsing the input files and constructing the initial data structures.

In Figures 6.7, 6.8, and 6.9, we present the execution times for checking universality by the explicit subset algorithm and the semi-symbolic antichain algorithm. To generate each sample point, we check the universality of 100 random automata with $|S| = 175$ (this is roughly the largest size that the subset algorithm is able to handle on the entire parameter space with the available memory). In Figure 6.7, we present the median execution times for testing universality by the subset approach as a function of r (transition density) and f (density of accepting states). The figure shows that the universality test is most difficult when $r = 2$ and $f = 1$. For the same instances, the median execution time of our algorithm is always less than the time unit of the system clock (1ms), and has thus not been depicted.

In Figure 6.8 and Figure 6.9, we present the average execution times for testing

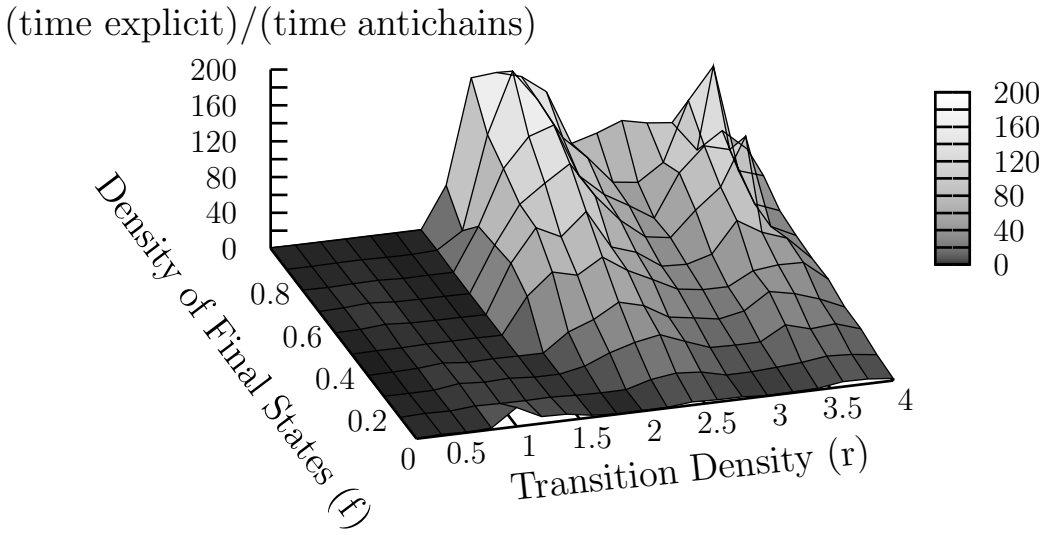

Figure 6.10: Average execution time ratio ($|S| = 175$).

universality by the subset approach and the semi-symbolic antichain approach, respectively. Both figures exhibit similar peaks, showing that the difficult instances are roughly the same for both approaches. However, the antichain algorithm is much faster. For the most difficult parameter values ($r = 2$ and $f = 1$), the antichain algorithm is 165 times faster than the subset algorithm. Intuitively, these instances are difficult for both algorithms for the following two reasons. First, the probability to be universal for these parameter values is around 50 percent, and we believe that most of these instances are neither trivially universal nor trivially non universal. Second, when an automaton is universal, the subset method has to build the entire deterministic automaton, and the antichain method has to complete the computation of the least fixed point.

In Figure 6.10 we present the ratio of the average time for the subset approach and the average time for the antichain approach as a function of the densities. The comparison for $r \leq 1.4$ and $f \leq 0.2$ is not very significant, because the execution times are very close to the precision of the system clock (1ms). For the rest of the parameter space, the antichain algorithm performs always better (up to 200 times better). Finally, in Figure 6.11, we show that the semi-symbolic antichain approach scales well when the size of the automaton increases, in contrast to the

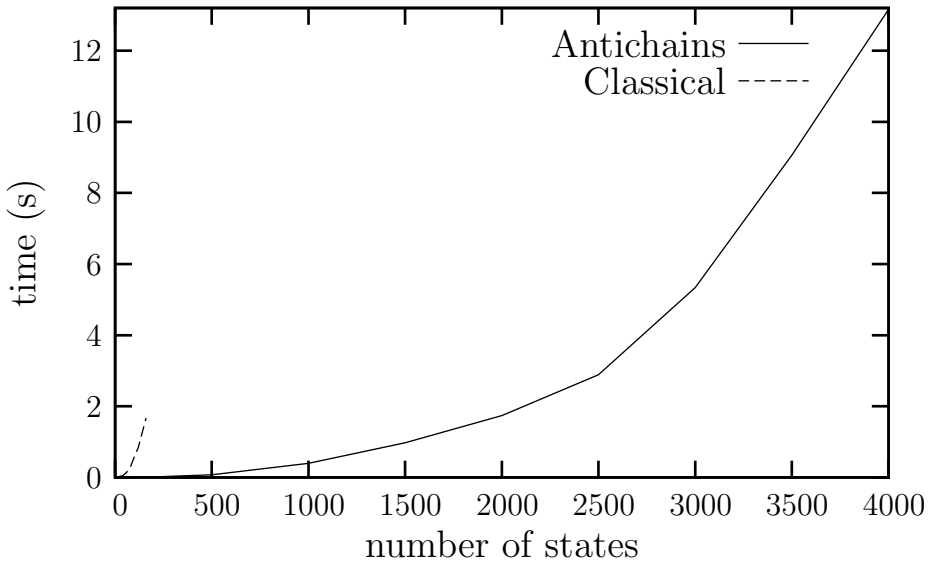

Figure 6.11: Average execution times for the subset and semi-symbolic antichain algorithms in hard cases (transition density 2; accepting-states density 1).

subset approach. For the experiments we generated randomly 100 automata per sample point for automaton sizes under 200 states, and 30 automata per sample point for sizes over 200 states. The densities are again $r = 2$ and $f = 1$, to capture hard cases. The antichain algorithm is able to handle random automata with 4000 states in average time 12s. The average size of the final antichain (for universal automata) is 217 state sets for automata with 4000 states. We did not pursue experiments with larger automata, because we would have had to modify the automaton generator, as it is not designed for such large automaton sizes. The subset algorithm quickly exceeds the memory limit when the number of states gets close to 200, so the curve is quite short in the left corner of Figure 6.11.

As mentioned above, the semi-symbolic antichain encoding gives far better performances on the random model than the fully symbolic encoding, as shown in Table 6.1 for the difficult instances ($r = 2$ and $f = 1$). It also turns out that the fully symbolic encoding does not scale well when the size of the automaton increases. Each sample point is computed on a set of 50 random automata with less than 100 states. For 175 states, the size of the automata sample is 100, and for more states, the sample size is 30. The number of boolean variables of the BDDs

Table 6.1: Average execution times (ms) for checking universality with $r = 2$ and $f = 1$.

number of states	20	40	60	80	100	175	1000	2000	3000	4000
subset algorithm	23	50	141	309	583	2257	-	-	-	-
fully symb. antich.	3	14	70	175	421	6400	-	-	-	-
semi-symb. antich.	1	2	2	3	5	14	400	1741	5341	13160

that encode antichains seems to be the reason for the difference in performances: the number of boolean variables grows linearly with the number of states in the fully symbolic encoding, but logarithmically in the semi-symbolic encoding.

6.5 Games with Infinite State Space

In this section, we tackle game of imperfect informations with an infinite state space. We identified a class of infinite state games for which we can use our lattice approach. We define this class in the section 6.5.1 and show in section 6.5.2 that discrete games on rectangular automata fall into it. By *discrete games*, we mean that the duration of any continuous transition is exactly 1. In this type of games, the action of the controller always take place at evenly spaced time instants.

6.5.1 Games with Finite R-stable Quotient

Here we drop the assumption that S is finite in the games $\langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs}\rangle$ and we consider the case where there exists a finite quotient of S over which the game is *stable*, that is, the important sets of the game can be defined in terms of the quotient. We obtain a general decidability result for games of imperfect information with finite stable quotients.

Definition 6.9 (R-definable,R-stable)
Let $R = \{r_1, r_2, \ldots, r_l\}$ be a finite partition of S. A set $s \subseteq S$ is R-definable *if $s = \bigcup_{r \in Z} r$ for some $Z \subseteq R$. An antichain $q \in L$ is R-definable if for every $s \in q$, s is R-definable.*

A game of imperfect information $\langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs}\rangle$ *is* R-stable *if for every* $\sigma \in \Sigma$ *the following conditions hold:*

(*i*) *for every* $r \in R$, $\mathsf{post}_\sigma(r)$ *is* R*-definable;*

(*ii*) *every* $\mathsf{obs} \in \mathsf{Obs}$ *is* R*-definable;*

(*iii*) E *and* F *are* R*-definable;*

(*iv*) *for all* $r, r' \in R$*, for all* $\sigma \in \Sigma$*:*

$$\begin{gathered}\exists x \in r : \mathsf{post}_\sigma(\{x\}) \cap r' \neq \varnothing \\ \Longrightarrow \\ \forall x \in r : \mathsf{post}_\sigma(\{x\}) \cap r' \neq \varnothing\end{gathered}$$

The next lemma states properties of R-stable games of imperfect information. They are useful for the proof of the next theorem.

Lemma 6.9
Let $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs}\rangle$ *be a* R*-stable game of imperfect information. Let* $s, s', s'' \subseteq S$ *and* $r \in R$ *such that*

(*i*) s' *and* s'' *are* R*-definable*

(*ii*) $s \cap r \neq \varnothing$

(*iii*) $\exists \sigma \in \Sigma : \mathsf{post}_\sigma(s \cap s') \subseteq s''$

then $\mathsf{post}_\sigma((s \cup r) \cap s') \subseteq s''$.

Proof
We prove the theorem ad absurdum. *In order to find a contradiction with the hypothesis we assume that*

$$\mathsf{post}_\sigma((s \cup r) \cap s') \not\subseteq s''.$$

Then :

$$
\begin{aligned}
&\Longrightarrow \exists x \in r : x \notin s \wedge \mathsf{post}_\sigma(\{x\} \cap s') \not\subseteq s'' \\
&\quad (\text{by } (iii)) \\
&\Longrightarrow \exists x \in r \cdot \exists r' \in R : r' \not\subseteq s'' \wedge \mathsf{post}_\sigma(\{x\} \cap s') \cap r' \neq \varnothing \\
&\quad (\text{since } s'' \textit{ is } R\textit{-definable}) \\
&\Longrightarrow \exists x \in r \cdot \exists r' \in R : r' \not\subseteq s'' \wedge x \in s' \wedge \mathsf{post}_\sigma(\{x\}) \cap r' \neq \varnothing \\
&\Longrightarrow \exists r' \in R : r' \not\subseteq s'' \wedge (\exists x \in r : x \in s' \wedge \mathsf{post}_\sigma(\{x\}) \cap r' \neq \varnothing) \\
&\Longrightarrow \exists r' \in R : r' \not\subseteq s'' \wedge (\forall x \in r : x \in s' \wedge \mathsf{post}_\sigma(\{x\}) \cap r' \neq \varnothing) \\
&\quad (\textit{by point } (iv) \textit{ of the definition or a } R\textit{-stable game}) \\
&\Longrightarrow \exists r' \in R : r' \not\subseteq s'' \wedge (\exists x' \in (r \cap s) : x' \in s' \wedge \mathsf{post}_\sigma(\{x'\}) \cap r' \neq \varnothing) \\
&\quad (\textit{by } (ii)) \\
&\Longrightarrow \exists r' \in R : r' \not\subseteq s'' \wedge (\exists x' \in (r \cap s) : \mathsf{post}_\sigma(\{x'\} \cap s') \cap r' \neq \varnothing) \\
&\Longrightarrow \exists x' \in (r \cap s) \cdot \exists r' \in R : r' \not\subseteq s'' \wedge \mathsf{post}_\sigma(\{x'\} \cap s') \cap r' \neq \varnothing) \\
&\Longrightarrow \exists x' \in s : \mathsf{post}_\sigma(\{x'\} \cap s') \not\subseteq s'' \\
&\Longrightarrow \mathsf{post}_\sigma(s \cap s') \not\subseteq s''
\end{aligned}
$$

which clearly contradicts the hypothesis (iii).

Theorem 6.5
Let $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs} \rangle$ *be a* R*-stable game of imperfect information. For any* R*-definable antichain* $q \in L$*, the antichain* $\mathsf{CPre}(q)$ *is also* R*-definable.*

Proof
We show that for any R*-definable antichain* $q \in L$*, the antichain* $\mathsf{CPre}(q)$ *is also* R*-definable. Let* $s \in \mathsf{CPre}(q)$*. For any* $r \in R$ *such that* $s \cap r \neq \varnothing$*, we have by Lemma 6.9 that* $s \cup r \in \mathsf{CPre}(q)$*. Since* $s \subseteq s \cup r$ *and* $\mathsf{CPre}(q)$ *is an antichain, we must have* $s = s \cup r$*. This shows that* s *is* R*-definable.*

Corollary 6.1
Let $G = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs} \rangle$ *be a* R*-stable game of imperfect information. The greatest fixed point of* $\mathsf{CPre}(\cdot) \bigsqcap \{\overline{\mathrm{F}}\}$*, and the least fixed point of* $\mathsf{CPre}(\cdot) \bigsqcup \{\mathrm{F}\}$ *are* R*-definable antichains and are computable.*

Proof
Since F *is R-definable,* $\overline{\mathrm{F}}$ *is also R-definable. Furthermore, the number of R-definable antichains is finite, and so, using Tarski's theorem, we can compute those two fixed points of* CPre *in a finite number of iterations.*

6.5.2 An application: Discrete Control with Imperfect Information of Rectangular Automata

We use the notion of infinite games with finite stable quotient to show that the *discrete reachability and safety problems for games of imperfect information defined by rectangular automata* are decidable. This result extends the results in [HK99].

In this section we define a discrete time game semantics for rectangular automata (see Definition 2.19 for a reminder of the definition of rectangular automata). We then recall a result of [HK99] that establishes the existence of a finite bisimulation quotient for this game semantics. In this section, we assume that the constants appearing in the rectangular constraint of the rectangular automata are all integers. This does not restrict the generality of the results, since such an automaton can be obtained from a rectangular automaton with rational constants through scaling (as explained in Section 2.3.2).

Definition 6.10 (Discrete time game semantics of rectangular automata) *The game* semantics *of a rectangular automaton*

$$H = \langle \mathsf{Loc}, \mathsf{Init}, \mathsf{Final}, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg}, \mathsf{Flow}, \mathsf{Jump} \rangle$$

over a set of variables X *is the game* $[\![H]\!] = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$ *where*

- $S = \mathsf{Loc} \times \mathbb{R}^n$ *is the* state space *(with* $n = |X|$*)*
- $\mathrm{E} = \{(\ell, v) \in S \mid v \in [\![\mathsf{Init}(\ell)]\!]\}$ *is the* initial space
- $\mathrm{F} = \{(\ell, v) \in S \mid v \in [\![\mathsf{Final}(\ell)]\!]\}$ *is the* final space
- $\Sigma = \mathsf{Lab}$
- *and* $\rightarrow$ *contains all the tuples* $((\ell, v), \sigma, (\ell', v''))$ *such that there exists*
  - $v' \in \mathbb{R}^n$,

– $e = (\ell, \sigma, \ell') \in \mathsf{Edg}$

– f *a continuously differentiable function* $f : [0,1] \to [\![\mathsf{Inv}(\ell')]\!]$

such that

– $(v, v') \in [\![\mathsf{Jump}(e)]\!]$

– $f(0) = v'$, $f(1) = v''$

– *and for all* $t \in [0,1]$: $\dot{f}(t) \in [\![\mathsf{Flow}(\ell')]\!]$.

Games constructed from rectangular automata are played as follows. The game is started in a location ℓ with a valuation v for the continuous variables such that $v \in [\![\mathsf{Init}(\ell)]\!]$. At each round, the controller decides to take one of the enabled edges (we assume, without loss of generality, that there always is such an edge) and one unit of time elapses. The environment resolves the nondeterminism in choosing both the next location and values of the continuous variables in the range specified by the flow predicates. A new round is started from there. As for the games that we have considered previously, the goal of the controller can be either to avoid to get to some bad states represented by the set F or to ensure that this set F, considered then as a goal, is reached.

To prove that we can use our antichain algorithm for such games we need a finite quotient of the state space of the game. We found such a quotient in a paper by Henzinger and Kopke [HK99] and we recall its definition here.

First we need to introduce the notion of nondecreasing and bounded variables.

Definition 6.11 (Nondecreasing and bounded variables)
Let H be a rectangular automaton, and let $i \in \{1, \ldots, n\}$. The variable x_i of H is nondecreasing *if, for every control mode $\ell \in \mathsf{Loc}$, the invariant interval $[\![\mathsf{Inv}(\ell)]\!]_i$ and the flow interval $[\![\mathsf{Flow}(\ell)]\!]_i$ are subsets of the nonnegative reals. The variable x_i is* bounded *if, for every control mode $\ell \in \mathsf{Loc}$, the invariant interval $[\![\mathsf{Inv}(\ell)]\!]_i$ is a bounded set. The automaton H has* nondecreasing *(resp.* bounded*;* nondecreasing or bounded*)* variables *if all variables of H are nondecreasing (resp. bounded; either nondecreasing or bounded).*

A rectangular automaton is m-bounded if all its rectangular constraints are m-bounded. In the sequel, all the rectangular automata that we consider are assumed to have only nondecreasing or bounded variables.

We then consider the following equivalence relation between states of rectangular automata.

Definition 6.12
Given the game semantics

$$[\![H]\!] = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$$

of a m-bounded rectangular automaton H over a set of variable X, define the equivalence relation $\approx_m$ on S by $(\ell, v) \approx_m (\ell', v')$ iff $\ell = \ell'$ and for all $x \in X$

- *either $\lfloor v_{|x} \rfloor = \lfloor v'_{|x} \rfloor$ and $\lceil v_{|x} \rceil = \lceil v'_{|x} \rceil$*
- *or both $v_{|x}$ and $v'_{|x}$ are strictly greater than m.*

Let us call $R_{\approx_m}$ the set of equivalence classes of $\approx_m$ on S. Observe that a set of state $\{(\ell, v) \mid \exists g \in \mathsf{Rect}(X) : v \in [\![g]\!]\}$ is $R_{\approx_m}$-definable.

The following result, from [HK99], is the key of our decidability result for discrete games of imperfect information for rectangular automata.

Lemma 6.10 ([HK99])
Let H be a m-bounded rectangular automaton over a set of variables X. The equivalence relation $\approx_m$ is the largest bisimulation of the game semantics $[\![H]\!]$. If the number of location of H is k, the number of equivalence classes of $\approx_m$ is $k \cdot (4m+3)^{|X|}$.

6.5.3 Rectangular Automata with Imperfect Information

We are now ready to extend the results of [HK99] to the case of imperfect information.

Given $H = \langle \mathsf{Loc}, \mathsf{Init}, \mathsf{Final}, \mathsf{Lab}, \mathsf{Edg}, \mathsf{Flow}, \mathsf{Jump} \rangle$, a m-bounded rectangular automaton, we say that the observation set Obs is m-bounded if each $\mathsf{obs} \in \mathsf{Obs}$, is definable as a finite union of sets of the form $\{(l, v) \mid v \in g\}$ where g is a m-bounded rectangle.

Theorem 6.6
For any m-bounded rectangular automaton H with game semantics

$$[\![H]\!] = \langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow \rangle$$

, for any m-bounded observation set Obs*, the game of imperfect information*

$$\langle S, \mathrm{E}, \mathrm{F}, \Sigma, \rightarrow, \mathsf{Obs} \rangle$$

is $R_{\approx_m}$*-stable.*

Proof
Let $\mathsf{reg}(x)$ *be the equivalence class of* $x \in S$*. We have to check whether* $[\![H]\!]$ *meets the four requirements of definition 6.9:*

(*i*) *We will prove that* $\forall \sigma \in \Sigma, \forall r \in R_{\approx_m} : \mathsf{post}_\sigma(r)$ *is* $R_{\approx_m}$*-definable. Since* r *is an equivalence class of* $\approx_m$*, it is also a rectangle. Thanks to [HK99], we know that the successor of a rectangle through* $\rightarrow$ *is a rectangle too and that every rectangle is an union of equivalence classes of* $R_{\approx_m}$*.*

(*ii*) *every* $\mathsf{obs} \in \mathsf{Obs}$ *is* $R_{\approx_m}$*-definable.*

(*iii*) E *and* F *are* $R_{\approx_m}$*-definable.*

(*iv*) *We have to prove that for all* $r, r' \in R_{\approx_m}$*, for all* $\sigma \in \Sigma$*:*

$$\exists x \in r : \mathsf{post}_\sigma(\{x\}) \cap r' \neq \varnothing$$
$$\Longrightarrow$$
$$\forall x \in r : \mathsf{post}_\sigma(\{x\}) \cap r' \neq \varnothing$$

This is immediate from the fact that $R_{\approx_m}$ *is a quotient of* S *for the bisimulation* $\approx_m$ *and from the definition of a bisimulation.*

As corollary of Corollary 6.1 and Theorem 6.6, we have that:

Corollary 6.2
Solving safety and reachability games of imperfect information defined by m bounded rectangular automata and m-bounded observation sets is decidable (in $2EXPTIME$*).*

The second exponential arises from the number of equivalence classes for $\approx_m$ (see Lemma 6.10). So far, we do not have a hardness result but we conjecture that the problem is 2EXPTIME-complete. Now, let us illustrate the discrete control problem for games of imperfect information defined by rectangular automata on an example.

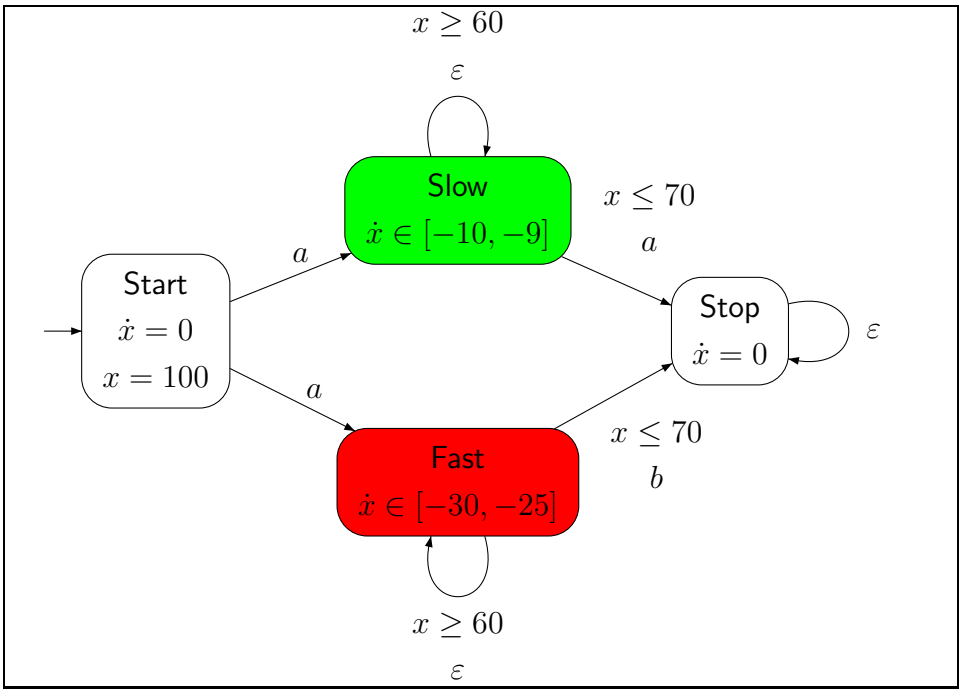

Figure 6.12: A game of imperfect information on a discrete rectangular automaton.

6.5.4 Example of Discrete Safety Game for Rectangular Automata

We have implemented our fixed point algorithm using HyTech and its script language [HHWT95a]. We illustrate the use of the algorithm on a simple example. Figure 6.12 shows a rectangular automaton with four locations and one continuous variable x. Remember that we required that the transition relation of our systems be total. In this example, we only pictured the edges allowing a good move. Any move that should use an edge not pictured would lead to a bad state. This is a safety game : we want to avoid the bad states. The goal is in practice to reach the location Stop, where we have a very simple safe strategy : emit events ε forever.

In this example, the game models a cooling system that controls the temperature x. When requested to start, by $\sigma = a$, the system begins to cool down. There are two modes of cooling, either fast or slow, among which the environment chooses. The controller can only observe the system through two observations: H with $\mathsf{H} = \{(\ell, x) \mid x \geq 80\}$ and L with $\mathsf{L} = \{(\ell, x) \mid x \leq 85\}$. Thus, only the continuous variable x can be observed imperfectly, not the modes. Depending on the mode however, the timing and action to stop the system are different. In both modes, the controller has to issue an action when the temperature is below 70. In the slow mode, it is an a; in the fast mode, it is a b. The difficulty is that the rule of the game imposes to emit a letter at each unit of time and that, when $x < 60$, it is not possible anymore to wait by emitting ε events. The controller has thus to emit an a or a b before $x < 60$ and after $x \leq 70$, while relying only on observations stating if $x \leq 85$ or $x \geq 80$.

Figure 6.13 illustrates the two possible behaviors of the system, either if the mode is fast or slow.

The controller must use its memory of the past observations to make the correct action in time. If the first two observations are "H, H", then the controller knows that the mode is Slow and that it should emit an a after 4 units of time. If the first two observations are "H, L", then the controller knows that the mode is Fast and that it should emit a b after 2 units of time.

We were able to compute the greatest fixed point for this game using HyTech and to extract a deterministic strategy that is pictured in Figure 6.14. The correspondance between state numbers in the figure and elements of the fixed point is the following:

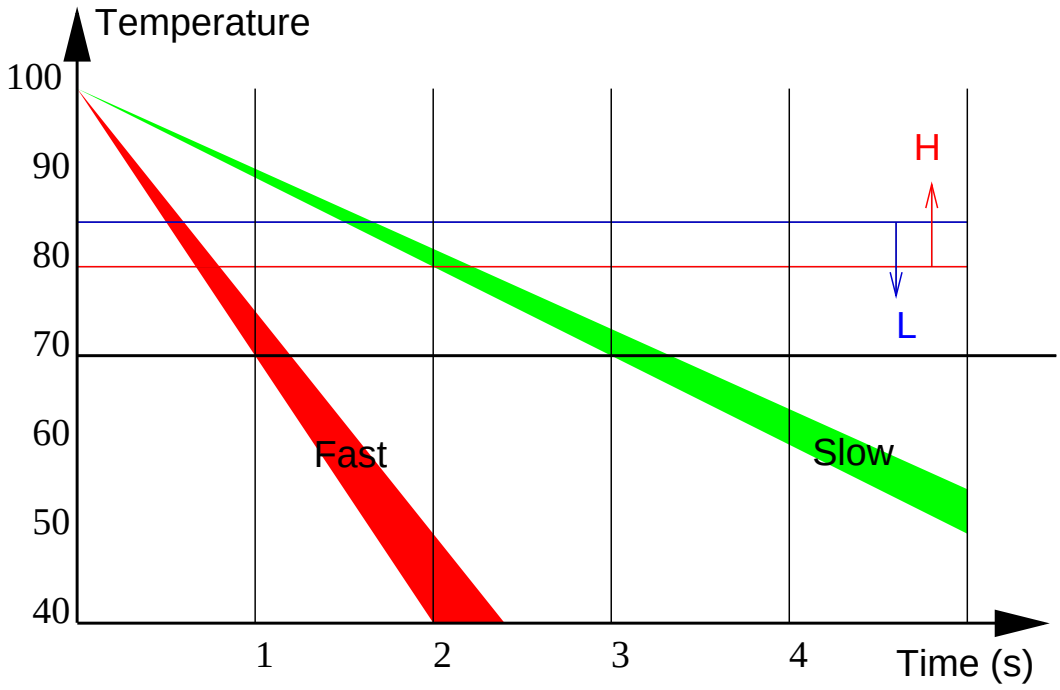

Figure 6.13: Illustration of the game of Figure 6.12.

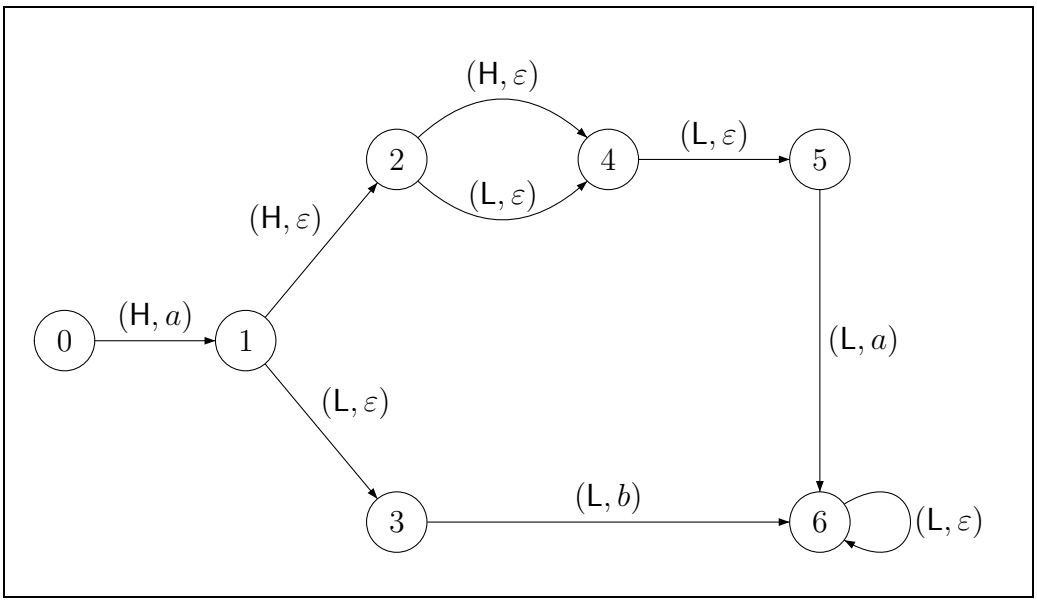

Figure 6.14: A winning strategy for the game of Figure 6.12.

- State $0 \equiv (\mathsf{Start}, x = 100)$
- State $1 \equiv (\mathsf{Slow}, 90 \leq x \leq 91), (\mathsf{Fast}, 70 \leq x \leq 75)$
- State $2 \equiv (\mathsf{Slow}, 80 \leq x \leq 82)$
- State $3 \equiv (\mathsf{Fast}, 40 \leq x \leq 50)$
- State $4 \equiv (\mathsf{Slow}, 70 \leq x \leq 73)$
- State $5 \equiv (\mathsf{Slow}, 60 \leq x \leq 74)$
- State $6 \equiv (\mathsf{Stop}, true)$

As before, the strategy associates an action to each observation in a given set of the fixed point. The strategy 'branches' after state 1 and then keeps in memory the number of observations to play a or b in the good timing.

6.6 Conclusion and Future Works

As future works, we would like to apply our antichain approach to *continuous control for timed or rectangular games*. The first problem we encounter is the problem of defining such games. It is not clear what game semantics we could choose: for example, it could be a semantics in which the controller chooses at the same time a discrete action and the time to wait after this action, like in [ABD$^+$00]. We must be careful since the game semantics we choose must make sense for real problems and at the same time still offer interesting decidability properties. To illustrate the difficulty, observe that a rather natural definition of timed games leads to the undecidability of finding a reaching strategy: if we consider the semantics of a timed automaton $[\![A]\!] = \langle S, \mathrm{E}, \mathrm{F}, \Sigma \uplus \mathbb{R}^{\geq 0}, \rightarrow \rangle$ as a two-player game and equip it with the set of observations $\{S\}$, we obtain a *blind game* for which finding a reaching strategy for the controller amounts to decide if there exists a word not accepted by the automaton. Unfortunately, this *universality problem for timed automata* is known to be undecidable (see [AM04]).

In the future, we would also like to use the technique for the automatic generation of *implementable* strategies where implementability is defined in a similar way to what we have done in the previous chapters.

The use of antichains in the computation of a fixed point for a controllable predecessor operator is of interest by itself, as we have shown through the application to the universality test for finite automata. The idea has already been explored further in [CDHR06] for omega-regular games and there is also a hope that the use of antichains could greatly improve the efficiency of the universality test for Büchi automata.

Chapter 7

Conclusions

7.1 Summary

The central topic of this thesis is the design of *robust* embedded controllers under *real time* constraints. By robust, we mean that the correctness of the controllers should not depend on too simplifying assumptions like the *synchrony hypothesis*, that assume that all computations are instantaneous, or the *perfect information hypothesis*, that assumes that a controller can know *exactly* the value of some variables of its environment, unless they are *formally validated*.

In a first part of this thesis we described a complete design approach, using the formalism of timed and rectangular automata, for the design of embedded controllers. We first underlined the fundamental flaws of those formalisms as a specification for programs: instantaneity of synchronization, possible instantaneity of reaction that can lead to zeno behaviors, i.e. blocking of time, and infinite precision of the clocks. Since those formalisms are nevertheless very intuitive to use for designers, and sustained by numerous tools, we did not discard their use. We proposed instead a methodology which uses the classical semantics verification as an useful first step in the design, but also allows the formal validation of the synchrony hypothesis during a second step. This formal validation is obtained through the definition of a new semantics for controllers specified as timed automata: the AASAP semantics. AASAP is an acronym standing for Almost As Soon As Possible, which describes well the behavior we are expecting from our controllers: they should react in a reasonable delay to events, even if we cannot expect them to react instantaneously. This delay is formalized through the use of a parameter that models the speed of the controller. The two great advantages of the AASAP semantics are that it is on the one side verifiable, i.e. we can check *reachability properties* on it, and on the other side implementable, i.e. if the speed

of the controller is not required to be infinite to ensure correctness we prove that it is possible to provide a practical implementation. We developed prototype tools for the verification of the AASAP semantics and for correct-by-construction code generation from timed automata to a toy real-time platform. This forced us to tackle the methodological problems linked to code generation in practice.

The verification tool uses a compositional construction that profits from the on-the-fly construction of products of automata by a tool like UPPAAL.

In a second part of this thesis, we tackled the problem of controller synthesis for systems relying on *imperfect information* about their environment. The basic example of imperfect information is the temperature sensor: no such sensor is able to give the exact temperature of an environment at any moment, one measurement can match different states and one state can result in different measurements. The controller synthesis problem is often presented as a type of *game*, where we are looking for a *strategy* that will ensure that one of the player wins no matter how the other player behaves. We thoroughly treated the problem of finding strategies for *games of imperfect information*, using a fixed point computation on *the lattice of antichains*. The use of this lattice makes us benefit from the monotonicity of the *controllable predecessor* operators. We proved the decidability of the problem of finding a strategy in finite games and in an interesting class of infinite state games, both for *safety games*, where we want to keep the environment safe from some *bad* sates, and for *reachability games*, where we want the environment to reach some *good* states at some point in the future. These results allowed us to prove that the problem of discrete control of rectangular automata is decidable.

It also happens that the use of the lattice of antichains is very useful for other problems that can be presented as games. We reported in this thesis interesting experimental results for testing the *universality of finite automata*.

7.2 Personal Contributions

Since all this work has been made in collaboration with Laurent Doyen, who was writing his thesis parallelly, I feel the need to properly delimitate what is my work, what is collaboration, and what is Laurent's work.

- In Chapter 3 the definition of the AASAP semantics and the implementation

semantics are collaborative work, and are presented in both our theses, but the main proofs (of simulation) are my work.

- Chapter 4 is essentially my personal work. I defined both transformations to HyTech and Uppaal and studied the Philips Audio Control Protocol. The one thing I did not do is the proof of simulation for the HyTech transformation. In consequence, this proof is not to be found in this thesis but in [Doy06].

- Chapter 5 is my work only, although the implementation scheme was designed by Laurent Doyen and me for his DEA's thesis [Doy03].

- Chapter 6, although it appears only in my thesis and not in Laurent's one, is essentially collaborative work, although everything has been rewritten for more readability, and all the implementations about universality of finite automata are my work.

Bibliography

[ABD+00] E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli. Effective synthesis of switching controllers for linear systems. In *Proceedings of the IEEE*, volume 88, pages 1011–1025, 2000.

[AC05] Rajeev Alur and Arun Chandrashekharapuram. Dispatch sequences for embedded control models. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 508–518, 2005.

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

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: A tool for modelling and implementation of embedded systems. In J.-P. Katoen and P. Stevens, editors, *Proc. Of The 8 Th International Conference On Tools And Algorithms For The Construction And Analysis Of Systems*, number 2280 in Lecture Notes In Computer Science, pages 460–464. Springer–Verlag, 2002.

[AFP+02] Tobias Amnell, Elena Fersman, Paul Pettersson, Hongyan Sun, and Wang Yi. Code synthesis for timed automata. *Nord. J. Comput.*, 9(4):269–300, 2002.

[AFP+03] Tobias Amnell, Elena Fersman, Paul Pettersson, Hongyan Sun, and Wang Yi. Code synthesis for timed automata. *Nordic Journal of Computing(NJC)*, 9(4), 2003.

[AHK02] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49:672–713, 2002.

[AIK+03] Rajeev Alur, Franjo Ivancic, Jesung Kim, Insup Lee, and Oleg Sokolsky. Generating embedded software from hierarchical hybrid models. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 171–182, New York, NY, USA, 2003. ACM Press.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AL94] Martín Abadi and Leslie Lamport. An old-fashined recipe for real-time. *ACM Trans. Program. Lang. Syst.*, 16(5):1543–1571, 1994.

[AM04] Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2004.

[AT05] Karine Altisen and Stavros Tripakis. Implementation of timed automata: An issue of semantics or modeling? In Paul Pettersson and Wang Yi, editors, *FORMATS*, volume 3829 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2005.

[BC84] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, pages 389–448, 1984.

[BCG88] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59:115–131, 1988.

[Ber00] G. Berry. The foundations of esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000.

[BGK+02] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated verification of an audio-control protocol using uppaal. *J. Log. Algebr. Program.*, 52-53:163–181, 2002.

[BL80] Janusz A. Brzozowski and Ernst L. Leiss. On equations for regular languages, finite automata, and sequential networks. *TCS*, 10:19–35, 1980.

[BPV94] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio Control Protocol. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863, pages 170–192, Lübeck, Germany, 1994. Springer-Verlag.

[CBG88] E.M Clarke, M.C. Browne, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499, 1999.

[CDHR06] K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Algorithms for omega-regular games of incomplete information. In *Proceedings of CSL 2006: Computer Science Logic*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[CHR02] F. Cassez, T.A. Henzinger, and J.-F. Raskin. A comparison of control problems for timed and hybrid systems. In *HSCC 02: Hybrid Systems—Computation and Control*, Lecture Notes in Computer Science 2289, pages 134–148. Springer-Verlag, 2002.

[cov06] Coverity: Automated error prevention and source code analysis. http://www.coverity.com, 2006.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[DDMR04] M. De Wulf, L. Doyen, N. Markey, and J.-F. Raskin. Robustness and implementability of timed automata. In *Proceedings of FORMATS-*

FTRTFT 2004, Lecture Notes in Computer Science 3253. Springer-Verlag, 2004.

[DDR04] M. De Wulf, L. Doyen, and J.-F. Raskin. Almost ASAP semantics: From timed models to timed implementations. In *Proc. of HSCC' 04*, volume 2993 of *LNCS*, pages 296–310. Springer-Verlag, 2004.

[DDR05a] M. De Wulf, L. Doyen, and J.-F. Raskin. Almost ASAP semantics: From timed models to timed implementations. *Formal Aspects of Computing*, 17(3):319–341, 2005.

[DDR05b] M. De Wulf, L. Doyen, and J.-F. Raskin. Systematic implementation of real-time models. In *Proceedings of Formal Methods 2005 (FM'2005)*, volume 3582 of *LNCS*, pages 139–156. Springer-Verlag, 2005.

[DDR06a] M. De Wulf, L. Doyen, and J.-F. Raskin. Antichains: a new algorithm to solve universality of finite automata. In Thomas Ball and Robert B. Jones, editors, *Proceedings of CAV 2006: Computer Aided Verification*, volume 4144 of *LNCS*, pages 17–30. Springer-Verlag, 2006.

[DDR06b] M. De Wulf, L. Doyen, and J.-F. Raskin. A lattice theory for solving games of imperfect information. In *Proceedings of HSCC 2006: Hybrid Systems—Computation and Control*, volume 3927 of *LNCS*, pages 153–168. Springer-Verlag, 2006.

[Die01] H. Dierks. PLC-automata: a new class of implementable real-time automata. *Theoretical Computer Science*, 253(1):61–93, 2001.

[Doy03] Laurent Doyen. A systematic implementation of simple timed controllers. Technical Report 504, U.L.B., 2003.

[Doy06] Laurent Doyen. *Algorithmic Analysis of Complex Semantics for Timed and Hybrid Automata*. PhD thesis, Université Libre de Bruxelles, 2006.

[DW02] Martin De Wulf. Distribution automatique d'automates hybrides. DEA's thesis, 2002.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise wcet determination for a real-life processor. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.

[FPY02] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2002.

[Fre05] Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *HSCC*, pages 258–273, 2005.

[Har87] David Harel. Statecharts: A visual formulation for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[Hen91] Thomas A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems*. PhD thesis, Stanford Universtiy, 1991.

[Hen96] T.A. Henzinger. The theory of hybrid automata. In *Proc. of LICS '96*, pages 278–292. IEEE, 1996.

[HHWT95a] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HyTech. In E. Brinksma, W.R. Cleaveland, K.G. Larsen, T. Margaria, and B. Steffen, editors, *TACAS 95: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1019, pages 41–71. Springer-Verlag, 1995.

[HHWT95b] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: The next generation. In *16th Annual Real-Time Systems Symposium (RTSS)*, pages 56–65. IEEE Computer Society Press, 1995.

[Hit] Hitachi. *H8/300L Series, Programming Manual*.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *SPIN*, pages 235–239, 2003.

[HK99] T.A. Henzinger and P.W. Kopke. Discrete-time control for rectangular hybrid automata. *Theoretical Computer Science*, 221:369–392, 1999.

[HKSP03] T.A. Henzinger, C.M. Kirsch, M.A. Sanvido, and W. Pree. From control models to real-time code using GIOTTO. *IEEE Control Systems Magazine*, 23(1):50–64, 2003.

[HL02] Martijn Hendriks and Kim G. Larsen. Exact acceleration of real-time model checking. In *Workshop on Theory and Practice of Timed Systems*, 2002.

[HLS99] Klaus Havelund, Kim Guldstrand Larsen, and Arne Skou. Formal verification of a power controller using the real-time model checker uppaal. In *ARTS*, pages 277–298, 1999.

[HNSY92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science*, pages 394–406. IEEE Computer Society Press, 1992.

[HPR94] Nicolas Halbwachs, Yann-Eric Proy, and Pascal Raymond. Verification of linear hybrid systems by means of convex approximations. In *SAS*, pages 223–237, 1994.

[HS06] Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In *Proceedings of the 14th International Symposium on Formal Methods (FM)*, Lecture Notes in Computer Science. Springer, 2006.

[HWT95] P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 381–394, Liege, Belgium, 1995. Springer Verlag.

[IKL+00] Torsten K. Iversen, Køare J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen. Model-checking real-time control programs. In *Proceedings of ECRTS'2000*, pages 147–155, 2000.

[KMTY04] P. Krčál, L. Mokrushin, P.S. Thiagarajan, and W. Yi. Timed vs time triggered automata. In *Proc. of CONCUR'04*, volume 3170 of *LNCS*, pages 340–354. Springer–Verlag, 2004.

[LNZ05] Denis Lugiez, Peter Niebert, and Sarah Zennou. A partial order semantics approach to the clock explosion problem of timed automata. *Theor. Comput. Sci.*, 345(1):27–59, 2005.

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

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. of the 6 th PODC*, pages 137–151. ACM, 1987.

[Maq06] Nicolas Maquet. Provably correct code generation of real-time controllers. Master's thesis, U.L.B., 2006.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer-Verlag, 1980.

[Min67] N.M. Minsky. *Finite and Infinite Machines*. Prentice-Hall, 1967.

[MMP92] Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In *REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 447–484. Springer, 1992.

[Mø04] A. Møller. dk.brics.automaton. http://www.brics.dk/automaton/, 2004.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS'95*, volume 900 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 1995.

[Nie00] Stig Nielsson. Introduction to the legOS kernel. http://brickos.sourceforge.net, september 2000.

[PL00] Paul Pettersson and Kim G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. of FOCS 1977*, pages 46–57. IEEE, 1977.

[Pro98] Kekoa Proudfoot. RCX internals. http://graphics.stanford.edu/ kekoa/rcx/, 1998.

[Rei84] John H. Reif. The complexity of two-player games of incomplete information. *Journal of Computer and System Sciences*, 29(2):274–301, 1984.

[Som98] F. Somenzi. CUDD: CU Decision Diagram Package Release 2.3.0. University of Colorado at Boulder, 1998.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285309, 1955.

[Tew02] Ashish Tewari. *Modern Control Design With MATLAB and SIMULINK*. John Wiley & Sons, 2002.

[TV05] Deian Tabakov and Moshe Y. Vardi. Experimental evaluation of classical automata constructions. In *LPAR*, volume 3835 of *LNCS*, pages 396–411. Springer-Verlag, 2005.

[Wei99] V. Weispfenning. Mixed real-integer linear quantifier elimination. In *Proc. of ISSAC 99*, pages 129–136. ACM, 1999.

[Yov96] Sergio Yovine. Model checking timed automata. In *European Educational Forum: School on Embedded Systems*, 1996.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.

Appendix

Appendix A

ELASTIC Specification of the Philips Audio Control Protocol

```
-- The Elastic specification of the whole system for the audio control protocol of FM05
-- Hytech parametric analysis only terminates with stronger constraints than delta1 + delta2 <1/2 *-


var

x,y,z  : clock;
i, c,leng, m,p,r,doublezero, endM
: discrete;
automaton random
synclabs : ;
initially ran & i=0 ;

loc ran : while True wait{}
          when True do {i'=1} goto ran;
          when True do {i'=0} goto ran;
end


elastic automaton sender

eventlabs : messReady;
internlabs: ;
orderlabs :  up, down, messSent;

initially waitMessage & x=0 & p=0 & i=0 & leng=0 & c=0 & doublezero=0 & endM=1;

loc waitMessage :
     when get messReady & True goto idle;

loc messFinished :
     when True put messSent goto waitMessage;

loc idle :
        when x>=12 put up do{x'=0,p'=1,  c'=1, leng'=1} goto oneSent;
```

```
loc oneSent :
        when x>=2 & x<=2 & i=1 put  down do{x'=0} goto waitingOne ;
        when x>=4 & x<= 4 & i=0 put down
             do{ x'=0, p'=1-p, leng'=leng+1,c'=2 c,i'>=0,i'<=1,doublezero'=0} goto zeroSent;
        when x>=2 & x<=2 & p=1  & endM=1 put down do{x'=0,p'=0} goto messFinished;

loc waitingOne :
        when x>=2 & x<=2 put up do{x'=0, p'=1-p, leng'=leng+1, c'=2 c +1,i'>=0,i'<=1} goto oneSent;

loc zeroSent :
        when x>=2 & x<=2  & i=0 put  up do {x'=0} goto waitingZero;
        when x>=4 & x <= 4  & i=1 put up
             do{ x'=0, p'=1-p, leng'=leng+1, c'=2 c + 1,i'>=0,i'<=1} goto oneSent;
        when  x>=2 & x<=2 & p=1  & endM=1 do{x'=0,p'=0} goto messFinished;
        when  x>=2 & x<=2 & doublezero=1  & endM=1 do{x'=0, p'=0} goto messFinished;


loc waitingZero :
        when x>=2 & x<=2 put  down
             do{x'=0, p'=1-p,  leng'=leng+1,c'=2 c,i'>=0,i'<=1,doublezero'=1} goto zeroSent;
end

elastic automaton receiver

eventlabs : up;
internlabs : finalZero;
orderlabs : messReceived;
initially idle2 & y=0 & m=0 & r=0   ;

loc idle2:
        when get up & True  do {y'=0,m'=1,r'=1} goto last_is_1;
loc message :
        when True put messReceived goto idle2;
loc last_is_1 :
        when  get up & 3<=y & y<=5  do {y'=0,m'=1-m,r'=1} goto last_is_1;
        when  get up & 5<=y & y<=7  do {y'=0,m'=1-m,r'=0} goto last_is_0;
        when  get up & 7<=y         do {y'=0,r'=2} goto last_is_1;
        when  y>=9 & m=1 put messReceived do {y'=0} goto idle2;
        when  y>=9 & m=0 put finalZero do{y'=0,r'=0,m'=1-m}   goto message;
loc last_is_0 :
        when get up & 3<=y & y<=5  do {y'=0, m'=1-m, r'=0} goto last_is_0;
        when get up & 5<=y         do {y'=0, r'=2} goto last_is_1;
        when y>=7 put finalZero    do {y'=0, r'=0} goto message;

end

automaton checkOutput

synclabs : getup , finalZero;

initially check & z=0 & leng=0 & c=0 & doublezero=0 ;
```

```
loc check : while True wait{}
        when True sync getup do {z'=0} goto treating;
        when True sync finalZero do {z'=0} goto treating;
        when leng>3 goto cerror;
        when leng<0 goto cerror;


loc treating : while z<=0 wait{}
        when r=0 & leng=1 & c=1 do{z'=0} goto cerror;
        when r=0 & leng=1 & c=0 do{leng'=leng-1,z'=0} goto check;
        when r=0 & leng=2 & c>1 do{z'=0}  goto cerror;
        when r=0 & leng=2 & c<=1 do{leng'=leng-1,z'=0} goto check;
        when r=0 & leng=3 & c>3 do{z'=0} goto cerror;
        when r=0 & leng=3 & c<=3 do{leng'=leng-1,z'=0} goto check;

        when r=1 & leng=1 & c=0 do{z'=0} goto cerror;
        when r=1 & leng=1 & c=1 do{leng'=leng-1,c'=c-1,z'=0} goto check;
        when r=1 & leng=2 & c<=1 do{z'=0} goto cerror;
        when r=1 & leng=2 & c>1 do{leng'=leng-1,c'=c-2,z'=0} goto check;
        when r=1 & leng=3 & c<=3 do{z'=0} goto cerror;
        when r=1 & leng=3 & c>3 do{leng'=leng-1,c'=c-4,z'=0} goto check;


        when r=2 & leng=1 do{z'=0} goto cerror;
        when r=2 & leng=2 & c=1 do{leng'=0, c'=0,z'=0} goto check;
        when r=2 & leng=2 & c=0 do{z'=0} goto cerror;
        when r=2 & leng=2 & c>2 do{z'=0} goto cerror;
        when r=2 & leng=3 & c=3 do {leng'=1, c'=1,z'=0} goto check;
        when r=2 & leng=3 & c=2 do {leng'=1,c'=0,z'=0} goto check;
        when r=2 & leng=3 & c>3 do{z'=0} goto cerror;
        when r=2 & leng=3 & c<2 do{z'=0} goto cerror;

loc cerror : while True wait{}

end

init := param[sender]= 0 & param[receiver]=0;

bad  := loc[checkOutput] = cerror ;

view[up]=getup;
```

Appendix B

ELASTIC Code Annotations to the Specification of the Audio Control Protocol

```
platform : brickos ;
unit : 89;
hide : i, c, leng, doublezero, p;

-------------- SENDER ---------------

declarations sender :
%{
#include <dbutton.h>
#define M_SIZE 14
int message[]= {1,0,1,1,0,0,1,1,1,1,0,1,0,0};
int j=0;

bool zeroNext()
{
  return (j<M_SIZE && message[j++]==0);
}

bool oneNext()
{
  return (j<M_SIZE && message[j++]==1);
}

bool finished()
{
  return (j>=M_SIZE);
}
%}

init sender :
%{
  endM=0;
  cputs("HI");
```

```
%}

put up :
%{
  motor_b_dir (brake);
  motor_b_speed (MAX_SPEED);
%}

put down :
%{
  motor_b_dir (brake);
  motor_b_speed (MIN_SPEED);
%}

restrict oneSent to waitingOne
%{
   oneNext
%}

restrict oneSent to zeroSent
%{
   zeroNext
%}

restrict zeroSent to waitingZero
%{
   zeroNext
%}

restrict zeroSent to oneSent
%{
   oneNext
%}

restrict all to idle
%{
   finished
%}

-------------- RECEIVER -------------

declarations receiver :
%{
 #define M_SIZE 14
 int toReceive[M_SIZE] = {1,0,1,1,0,0,1,1,1,1,0,1,0,0};
 int received[M_SIZE];
 int i=0;
 int lastWasUp=0;

bool
checkMsg (int received[], int toReceive[], int t)
```

```
{
  int i;
  for (i = 0; i < M_SIZE && received[i] == toReceive[i]; i++)
    ;
  if (i != M_SIZE)
    {
      cputs ("KO");
    }
  else
    {
      cputs ("OK");

    }
  return true;
}

wakeup_t getup(wakeup_t data)
 {
     if (lastWasUp)
          {
            if (TOUCH_1)
               return false;
            else
               {
                 lastWasUp=false;
                 return false;
               }
           }
        else
          {
            if (TOUCH_1)
              {
               lastWasUp=true;
               return true;
              }
            else
               {
                 return false;
               }
           }
    }
%}

init receiver :
%{
   ds_passive (&SENSOR_1);
   cputs("Rec");
%}

detect up :
%{
```

```
   getup
%}

get up :
%{
if (i<M_SIZE)
{
  switch(r)
  {
     case 0:
       break;
     case 1:
     {
      received[i]=1;
      i++;
      break;
     }
     case 2:
     {
      received[i]=1;
      received[i+1]=0;
      i=i+2;
      break;
     }
     default :
      cputs ("Prob");
  }
}
%}

restrict all to idle
%{
  checkMsg (received,  toReceive, M_SIZE);
%}
```