UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Provably Correct Code Generation of Real-Time Controllers

Mémoire présenté en vue de l'obtention
du grade de Licencié en Informatique

Nicolas MAQUET
Année académique 2005–2006

Remerciements

Ce sujet de mémoire m'a été proposé par M. Jean-François Raskin. Je tiens à le remercier pour son aide apportée tout au long de la réalisation de ce travail. Son enthousiasme pour la recherche et son intransigeance technique sont pour moi des exemples.

Je tiens également à remercier M. Laurent Doyen et M. Martin De Wulf, et tout particulièrement ce dernier pour avoir relu et commenté de nombreuses versions de ce travail, et pour avoir répondu avec bonne humeur à mes myriades de questions.

Je remercie tout le corps académique du Département d'Informatique, pour leur travail de tous les jours à la réalisation d'un enseignement scientifique de qualité.

Un grand merci à tous mes amis, pour leur chaleureux soutien dans les moments difficiles et leur présence de tous les jours. Un remerciement tout particulier à Thierri de Vos pour son intérêt enthousiaste à mes travaux, et pour avoir accepté de m'aider à la réalisation d'un chapitre de ce mémoire mais qui n'a malheureusement pas abouti (il s'agissait d'une étude de cas supplémentaire impliquant des automates hybrides). Ton travail ne restera pas vain, ce n'est que partie remise !

Enfin, je tiens à témoigner ma gratitude envers toute ma famille pour leur amour et leur soutien tout au long de mes études. Je ne peux pas vous remercier tous personnellement ici, mais je ne peux m'empêcher de citer ceux et celles que je vois et qui me soutiennent quotidiennement (et qui donc, en particulier, ont du me supporter jour après jour pendant la rédaction de cet ouvrage ...) : Maman, Chloé, Roland, Martine, Maxime, Philippe, Catherine, Étienne, et Amandine. Merci !

À mon frère, Laurent Maquet
À mon père, Philippe Maquet

Contents

Chapter 1

Introduction

This work studies a methodology for creating provably correct real-time software controllers. A real-time software is a program for which the correctness depends not only on the soundness of its outputs, but also on the time at which those outputs are produced. For instance, a nuclear power-plant emergency shutdown sequence might involve numerous steps such as opening valves, raising robotic arms, triggering sound alerts, etc. Each of those steps must be accomplished in a timely fashion in order to prevent a catastrophe. When the possible damage caused by software failure reaches a certain point, it is more and more considered necessary to have some kind of *formal assurance* that the control software will always operate as required, that is by keeping the controlled environment in a *safe* state (the plant has not exploded, the aircraft has not crashed, etc.). To obtain this formal assurance, we need a way of reasoning on control software such that a proof of correctness can be constructed. This is done by (1) constructing a formal model $\mathcal{M}$ of the system (including environment and control software), (2) by specifying a formal property ψ that the model needs to satisfy, and (3) by proving that $\mathcal{M} \models \psi$. When such a proof can be obtained, we say that the control software is *provably correct*.

1.1 Formal Methods for Software Verification

Several formal methods for creating provably correct software have been studied in computer science, each having their own particularities. These methods can be classified in two distinct categories, which we introduce in turn.

1.1.1 Bottom-Up Approach

One way of reasoning about software that yields correctness proofs works by proceeding *bottom-up*. Based on informal specifications, a software solution is created, using standard software design techniques. After this first step, the

control software and its environment are then *abstracted*, to provide a formal model. Great care has to be put into this abstraction phase, to make sure that the model is a sound *over-approximation* of the real system.

The notion of over-approximation is a key concept in software verification. In short, a formal model correctly over-approximates a real-world system if we have that every property which can be proven true on the model will also be true for the real system. This is done by ensuring that every possible behavior of the real system is *included* in the set of behaviors that the model contains. The model can contain *more* behaviors however, hence the term *over-approximation*.

Once the formal model has been created, it is possible to reason on that model in a formal way, in order to obtain formal assurance that the real system is indeed correct (i.e. it satisfies some formal specification).

1.1.2 Top-Down Approach

Another approach to software verification (and the one we will use in this study) works the other way around. Using the informal specifications of the system to implement, a model is created *before* the beginning of the implementation process. The same care has to be put into the model design than in the former approach, in order to reason on a sound over-approximation of the target system. This approach however, has one considerable advantage : after having verified that the model is correct, it is possible to construct the implementation of the software in a systematic way. This of course requires that the model be designed in such a way that a systematic implementation is possible. Another considerable advantage of the top-down approach is that it can detect design flaws *before* the implementation process. If severe errors are found after the implementation has been created, part of the programming effort will have been wasted.

1.2 Provably Correct Real-Time Controllers

In this work, we will use a top-down verification approach and apply it in the context of real-time embedded controllers. In this study, a *controller* is a real-time software interacting with an environment, and which has the responsibility of keeping its environment in a safe state (or set of states). Such controllers are often *embedded* physically in their environments, as for instance the ABS which is found in cars, or the auto-pilot software aboard airplanes. These embedded applications are often found in safety-critical environments, justifying the need for formal verification.

The verification process that we will use to verify software controllers will focus on the soundness of its *output commands*. We will not care so much about the secondary computations that the controller must make to take decisions,

but rather focus on verifying that the controller is able to issue the appropriate commands at the right time, in all situations.

For example, consider a controller interacting with a valve, which needs to be operated to regulate the air-pressure of a tank. The software controller is aware of the current air-pressure by using an input-sensor and is linked to the valve by an output port. The valve control chip periodically reads an integer value on its input port and changes the valve opening angle accordingly, in real-time. In this situation, the control software operates by monitoring an integer input value and changing an integer output value accordingly. Our verification methodology will not manipulate these values directly, but rather abstract them with *events*. In this case, the output command could be abstracted by a single ChangeValveAperture event, and our verification process will be able to verify that it is issued by the controller appropriately. The rationale behind this abstraction is that (1) it *greatly* simplifies the design of the controllers and (2) that once it has been proven that the right command is always emitted at the right time, it is always possible to prove that the actual *contents* of that command is also appropriate, if need be. This latter verification would need other techniques which are beyond the scope of this work, so we will not discuss them further. The abstraction of controller-environment communication is illustrated at figure 1.1.

By abstracting the inputs and outputs of our software controllers with events, it is possible to design and analyze them in a natural way by using language theory. Hence, we will model the software control-logic with automata. Likewise, we will construct models for the environments using automata, and study the system as a whole, using the *composition* of the controller and environment automata.

1.3 Modeling Real-Time Controllers with Timed Automata

Since we need to design and verify control logics with real-time requirements, finite-state machines (regular languages) do not suffice in our context. We will need a more powerful computation model to analyze real-time software controllers, namely the *timed automaton*.

Timed automata have been introduced in [AD94], as an extension to regular automata incorporating the notion of time, and have been widely adopted in the research community as a tool for modeling real-time systems. A timed automaton is a finite-state machine augmented with a finite number of *clocks*. These clocks are real-valued variables which are used to count time and thus have a first derivative equal to one. They are used to restrict the possible behaviors of the automaton, by adding timing constraints to its nodes and edges. For example it is possible to express behaviors such that “stay in that location while clock x is

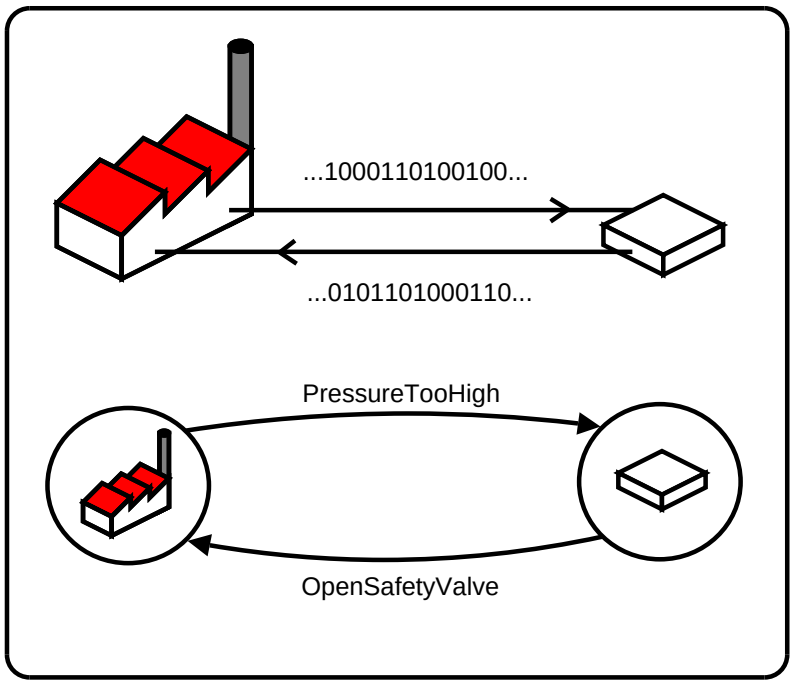

Figure 1.1: Abstraction of real-world communications using events. The top part of the figure represents the real-time data exchange between the controller and its environment, as it really occurs. For our verification purposes, we will study these communications using abstract events, as shown on the bottom part of the figure.

smaller than 3" or "take that edge as soon as clock y is equal to 2", etc. Timed automata are defined formally in the next chapter.

The popularity of timed automata is such that a number of model-checking tools exist to facilitate their analysis. The most notable ones include UPPAAL [BLL+98] and KRONOS [DOTY95].

1.4 Modeling Environments With Timed or Hybrid Automata

Our verification approach is applicable to environments modeled with timed or hybrid automata.

Hybrid automata [Hen96] are more sophisticated computation models than timed automata, as they allow for more complex continuous behaviors. A hybrid automaton can not only model time, but also any continuous dynamics such as temperature, voltage, height, pressure, etc. Like the timed automaton, the hybrid automaton is equipped with a finite number of variables, but their first derivative is not necessarily equal to one; in fact, it does not even need to be constant, as

hybrid automata define their continuous dynamics with *differential equations*.

The tank environment of the previous section for example, could be modeled elegantly with a hybrid automaton. It would have a variable representing the pressure of the tank, with each discrete location enforcing a particular pressure dynamics (indicating how fast the pressure rises when the valve is closed, for instance). With an environment modeled this way, it is possible to verify formally that the pressure variable will never reach some critical threshold for example.

Unlike with timed automata, the analysis of hybrid automata turns out to be undecidable in the general case[1]. This is immediate, because we do not know how to solve differential equations in general. To work around this undecidability, several subclasses of hybrid automata have been studied, most notably the *rectangular hybrid automaton*. The study of these subclasses has made it possible to analyze hybrid automata with the help of automated tools. These tools include for example HyTech [HHW97] and Phaver [Fre05].

In this work we will restrict ourselves to the use of timed automata. However, our methodology is completely applicable to environments modeled with hybrid automata.

1.5 Synchrony Hypothesis

Many approaches to the software verification problem use what is usually referred to as the *synchrony hypothesis*. When making that hypothesis we assume that the hardware on which our software controller will run is *perfect* in the following sense :

1. The synchrony hypothesis assumes an infinitely fast hardware. This implies that (1) any computation can be done in zero time units, and (2) that the controller can read input data and emit output commands instantaneously.

2. It also assumes that clock rounding errors never happen; this is essentially the same as requiring hardware clocks of infinite precision.

3. Finally, it assumes that the hardware clocks do not drift. A drifting clock is a digital clock which sometimes "looses" a tick, yielding inaccurate time readings. No real-world clock has a drift of zero, even the most sophisticated atomic clocks do drift (albeit *very* slowly).

The reason why such an unrealistic hypothesis is used is that (1) it *greatly* simplifies the design and analysis of software systems and (2) that in many situations the execution times and rounding errors are just too small compared to the scale of the environment. Indeed, if every delay or time lapse is of several orders

[1]By *analysis* of timed or hybrid automata, we mean *reachability analysis* unless otherwise stated.

of magnitude smaller than any time constraint required by the environment, they can be safely ignored.

However, clearly there are cases where making the synchrony hypothesis is just too unrealistic. In those circumstances, we have to make sure that the software will run appropriately, even on hardware with limited speed and relatively unreliable clocks.

One way of achieving this verification is to study the target hardware platform, and incorporate its limitations *into the controller model*. This can be quite difficult and time consuming, and the result might only be valid for one platform.

Another approach has been studied by Raskin et al. in [DDR05b], which works by using the synchrony hypothesis for the design of the controller, and by formally validating that hypothesis during the verification phase. This is achieved by using a special semantics for the controller automaton, called the *Almost ASAP semantics*. This approach is studied in detail in chapter 3.

1.6 Syntax and Semantics

The theoretical aspects of this work will rely heavily on the dual notions of syntax and semantics. In this work, what we call *syntax* is a formal definition of a *set of legal objects*. The syntax definition indicates which are legal objects and which are not. For example, the syntax of regular expressions is such that $(a + b(cd)^*)$ and $((ab)^+c)$ are considered legal, while $((ab)^a)$ and $((acd)+))$ are not.

A syntax definition alone is meaningless without an associated semantics. The semantics definition creates a formal object, the form of which depends on the syntax. To continue with regular expression, the semantics of $(a + b(cd)^*)$, which we note $[\![(a + b(cd)^*)]\!]$ is the set of strings : $\{a, b, bcd, bcdcd, bcdcdcd, \ldots\}$.

The advantage of distinguishing syntax and semantics is clear : it is much more convenient to write $(ab)^*(cd)^* + e^+f$, than $\{\epsilon, ab, abab, ababab, \ldots, cd, cdcd, cdcdcd, \ldots, abcd, ababcd, abababcd, \ldots, abcdcd, abcdcdcd, \ldots, ef, eef, eeef, \ldots\}$.

Moreover, the latter description is not only heavy and difficult to understand, it is also *ambiguous*[2]. This is why we use syntax definitions; to represent complex (and possibly infinite) mathematical objects (which are ultimately described by the semantics) in a concise and unambiguous manner.

In this work, we will not use regular expression to reason on software controllers, but rather *timed automata* and *timed transition systems*. Timed automata will provide syntax to our models, and we will define their semantics in the form of timed transition systems. As with timed automata, timed transition systems will be discussed in the next chapter.

[2]This example is a bit overdone to stress the point. It is of course possible the express the language of the regular expression above in a totally unambiguous fashion.

1.7 AASAP Semantics and Implementability

As stated previously, the Almost ASAP semantics provides a means to formally validate the synchrony hypothesis. This approach works as follows :

1. A model for the controller is designed, using timed automata. At this point, we do not worry about the limitations of the hardware yet. This allows the designer to focus on control logic rather than implementation details, yielding simpler and more elegant models.

2. When the controller automaton and environment automata have been created, they are analyzed in the classical way by composing both automata and applying model-checking techniques. This analysis is made using the standard semantics for timed automata, which does not forbid *unimplementable* behaviors. A sequence of actions is considered unimplementable in our context, if it cannot be executed by hardware, no matter how fast it is or how precise its digital clocks are. These behaviors include for example : taking several discrete transitions consecutively, without letting time elapse; or taking transitions only at fixed points in time (this is unimplementable because it requires infinite clock-precision). Thus, this preliminary analysis can yield "correctness proofs" for controllers which cannot be used in practice.

3. To make sure that the control strategy can be be executed by hardware, we analyze the system a second time, but with this time using the Almost ASAP semantics for the controller. Using this semantics, it is be possible verify (with the help of automated tools) the correctness *and* implementability of control strategies expressed with timed automata[3].

The reader might wonder why the second step is necessary. In fact, it is not. However, as analyzing the classical semantics of timed automata is much faster than with the Almost ASAP semantics, that second step is very useful in practice (because if it fails, the next step will certainly fail as well).

1.8 Implementing Controllers Systematically

We have stated earlier that one of the advantages of the top-down verification approach is that it is possible to create implementations in a systematic way. With the Almost ASAP semantics approach, this is achieved using an *implementation semantics*.

[3]Actually, the Almost ASAP semantics is not exactly applied on timed automata but rather on one of its subclass, called ELASTIC controllers. The difference is quite small however, and will be detailed in the following chapters.

Once the control-logic of the controller has been proven correct and implementable, we give a third semantics to the controller's timed automaton, which does not only describe *what* the controller does, but also contains details on *how* this is achieved in practice. The purpose of the three semantics we have mentioned can be summarized as follows :

- The classical semantics enables us to ensure that a control strategy is sound in the context of the synchrony hypothesis.
- The Almost ASAP semantics enables us to verify that a control strategy remains sound, even when executed on hardware with limited speed and precision.
- The implementation semantics describes *how* the control strategy is executed by hardware and enables us to construct implementations in a systematic way.

In [DDR05b], Raskin et al. have demonstrated how their methodology can be used to create implementations in a systematic manner, by using a proof-of-concept implementation semantics, which they called *program semantics*. They successfully applied this methodology in practice, by creating a code generating tool[4] for the LEGO MINDSTORMS™ platform.

1.9 Goal and Structure of this Work

The goal of this work is to study how the Almost ASAP semantics approach can be applied to more realistic embedded real-time platforms. The implementation semantics described in [DDR05b] has been voluntarily kept simple, and thus makes almost no hypothesis about the target run-time environment. As we will see in the following, the Almost ASAP semantics approach can be improved if we assume that the generated code will run on a hard real-time operating system. Our work will be structured as follows :

- In chapter 2, we define timed automata and their classical semantics. We will see how these automata are suitable for the design and analysis of real-time software and how this is done in practice.
- In chapter 3, we define the Almost ASAP semantics and the various definitions needed for its use. We will keep a to-the-point approach, by reviewing only the essential aspects which will be needed in this work.

[4]The tool is available at http://www.ulb.ac.be/di/ssd/madewulf/aasap/

- In chapter 4, we define the Program Semantics, as described in [DDR05b]. This definition will be followed by a discussion analyzing how it can be improved in the context of real-time operating systems. We will then define a new implementation semantics, called unsurprisingly *Real-Time Semantics*, which takes advantage of the capabilities of an RTOS. To validate this new semantics formally, we provide a complete simulation proof with the Almost ASAP Semantics.

- In chapter 5, we present a modern hard real-time operating system, namely RTAI[5]. We briefly describe its architecture and a small subset of its API (this information is needed in the following chapter).

- In chapter 6, we present SPECTRE, a tool for generating provably correct real-time code which has been created to demonstrate the practical applicability of our work. We describe how a SPECTRE-generated controller uses the RTAI run-time environment to behave consistently with the Real-Time Semantics. We also describe the tool's input language, along with details illustrating how it generates code and in what respect this code can be considered provably correct.

- In chapter 7, we illustrate the use of SPECTRE with a case-study.

- In chapter 8, we conclude this work by a discussion reviewing the progress made in this work, and pointing out some suggestions as to how it could be improved.

- Appendix A contains the grammar of the SPECTRE input language.

- Appendix B contains an example of a SPECTRE-generated controller.

[5]RTAI is freely available for download at http://www.rtai.org

Chapter 2

Timed Automata

This chapter introduces the theoretical background needed for the following chapters. Definitions are given for timed automata, as well as a number of their useful properties. In the final part of the chapter, we show how these computation models can be used in practice with a couple examples.

2.1 Syntax of Timed Automata

Timed automata are finite state automata extended with a finite number of *clocks*. When a timed automaton (TA) stays in one particular location, each of his clocks' value increases continuously with first derivative equal to one. As a discrete transition occurs, the TA can either reset a clock to zero, or leave it untouched. Before the formal definition, here is a small example of a timed automaton :

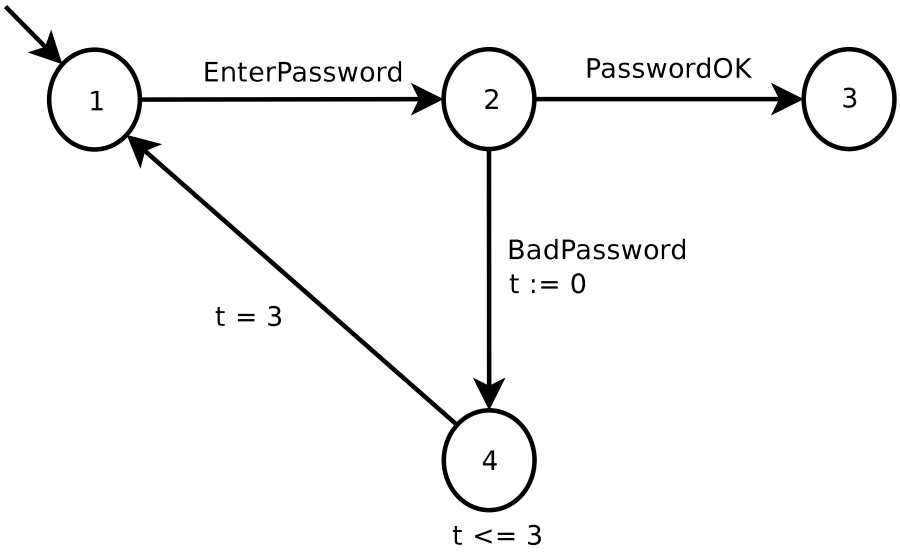

Figure 2.1: Simple example of a timed automaton.

This simple timed automaton has four locations and one clock : t. As indicated by the small arrow in the upper left part of the figure, location 1 is the *initial location* of this automaton. Three of the four transitions are labeled with a string; they materialize events that occur in the system. Labels and events will be treated in depth later in this chapter. Transition $2 \rightarrow 4$ assigns the clock t to the value zero. Thus, while in location 4, the value of clock t increases continuously from zero onward. Exactly three time units later, t will hold the value 3 and the transition $4 \rightarrow 1$ will be *enabled*, because the predicate attached to it becomes true. When one of its transition becomes enabled a timed automaton can take that transition as long as it stays enabled. It is important to note that a TA does not have to take an enabled transition, and this often leads to non-deterministic behaviors. In this particular case, the automaton is fully deterministic thanks to the predicate attached to location 4. The automaton cannot refuse to fire transition $4 \rightarrow 1$ when t equals 3 without violating the predicate $t \leq 3$.

Synchronization labels

The edges of timed automata are sometimes tagged with strings. Their purpose can be purely informational (as in the example of figure 2.1), or they can be used to synchronize events with other automata. When modeling complex interacting systems, it is often convenient to model each system separately with its own TA. Then, a special operation called *synchronized product* (also called *parallel composition* in the literature) makes one big timed automaton with all the small ones. The synchronization labels provide a means of communication between the automata within the composition. This is best illustrated by an example.

The example of figure 2.2 is a much simplified version of the ICMP[1] protocol. One station sends a "ping" packet to the other, which in turn responds with an "echo" reply. Observe that not all labels are used to synchronize the sender with the receiver : the label **Timeout** is not seen by the receiver. Throughout this work, we will use the popular convention of adding an exclamation mark after output messages, and a question mark after input messages. As illustrated by the figure, the synchronized product is a somewhat natural operation.

Guards and rate conditions

The control flow in timed automata is coerced by the predicates attached to its locations and edges. The edge predicates are often called *guards* in the literature. When a guard becomes true, the corresponding edge is said to be *enabled* and the automaton can follow that edge as long as it stays enabled. Location predicates are also often called *invariant predicates* in the literature. A timed automaton cannot be in a location unless its invariant predicate is true, and must thus leave its location whenever it becomes false. In the example of figure 2.2, the receiver

[1]ICMP stands for Internet Control Message Protocol.

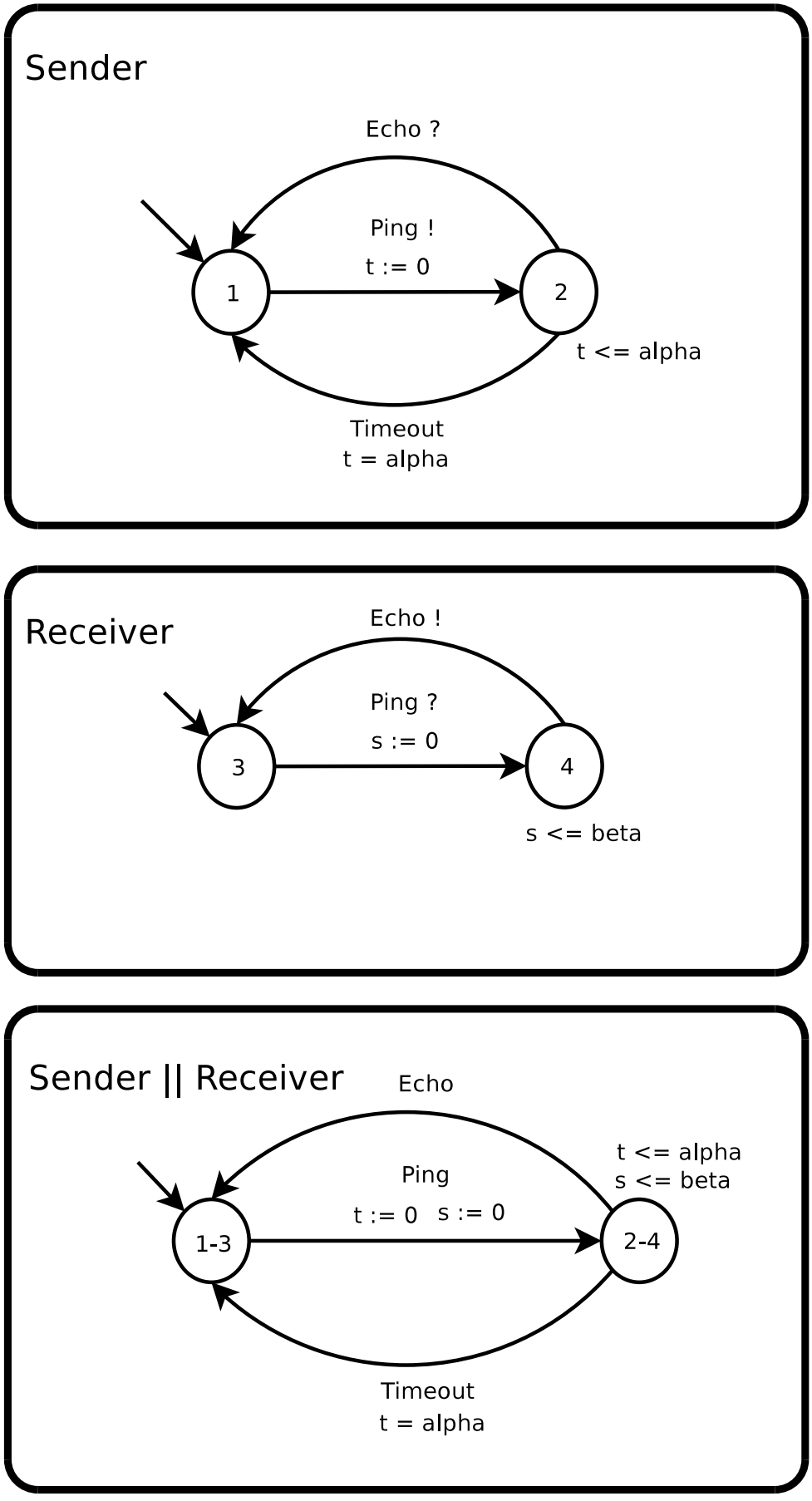

Figure 2.2: Illustration of synchronization labels and parallel composition.

automaton can delay the "echo" up to beta time units, as it can take transition $4 \rightarrow 3$ without restriction (it is always enabled because its predicate is *true*[2]). It may not stay longer, however, without violating the invariant predicate of location 4.

Syntax definitions

Before defining the timed automaton, we need to precise what kind of predicates can be used in their definition. We will use a notion similar to the one introduced in [AD94], namely the rectangular clock predicate. Basically, with these predicates it is only possible to compare a clock to a positive rational value, or to a rational interval (possibly infinite). It is not permitted to compare two clocks directly for example. Here is the formal definition.

Definition 1 [Rectangular Clock Constraint / Predicate] A rectangular clock constraint over a set of clocks X is a formula of the form $x \in I$, where I is a rational interval, open or closed and possibly infinite. Formally, I is of the form $(a, b), [a, b), (a, b], or [a, b]$ with $a, b \in \mathbb{Q}^{\geq 0} \cup \{+\infty\}$ and $a \leq b$.

A rectangular clock predicate is a finite set of rectangular clock constraints. For a rectangular clock predicate p and a clock valuation v, we write $v \models p$ if $v(x) \in I$ for every "$x \in I$" appearing in p. Finally, the set of all rectangular clock predicates over a clock set X is noted $\mathsf{Rect}(X)$, and that same set but with closed intervals is noted $\mathsf{Rect}_{\mathsf{c}}(X)$. □

Note that this definition is quite restrictive. It is not possible to express a disjunction directly with one predicate for instance. These restrictions are made to reduce the complexity of timed automata analysis. This phenomenon is well known : models that are too expressive quickly become undecidable. It is shown in [AD94], for instance that allowing to compare a rational value to the sum of two clocks leads to undecidability of timed automata analysis.

We are now ready to define the timed automaton formally.

Definition 2 [Timed Automaton - Syntax] A timed automaton is a tuple $\langle \mathsf{Loc}, l_0, \mathsf{Var}, \mathsf{Inv}, \mathsf{Lab}^{\mathsf{in}}, \mathsf{Lab}^{\mathsf{out}}, \mathsf{Lab}^{\tau}, \mathsf{Edg} \rangle$ where :

- Loc is a finite set of locations.
- $l_0 \in \mathsf{Loc}$ is the initial location.
- Var is a finite set of clocks, which are positive real-valued variables.
- $\mathsf{Lab} = \mathsf{Lab}^{\mathsf{in}} \cup \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^{\tau}$ is a structured, finite alphabet of synchronization labels. It is partitioned into input output and internal labels, respectively.

[2]Predicates that evaluate to *true* are omitted in the figures.

- $\mathsf{Edg} \subseteq \mathsf{Loc} \times \mathsf{Loc} \times \mathsf{Rect}(\mathsf{Var}) \times \mathsf{Lab} \times 2^{\mathsf{Var}}$ is a finite set of edges, also called transitions. The edge (l_1, l_2, g, σ, R) goes from location l_1 to l_2, resets the clocks contained in R to zero and is enabled only when the clock guard g is true. Depending on the label σ attached to the edge, the transition represents the acceptance of an input ($\sigma \in \mathsf{Lab}^{\mathsf{in}}$), the emission of output($\sigma \in \mathsf{Lab}^{\mathsf{out}}$), or an internal event ($\sigma \in \mathsf{Lab}^{\tau}$).

□

2.2 Classical Semantics of Timed Automata

As mentioned in the introduction, the above syntax definition does not mean much without an associated semantics. As we will define several semantics for timed automata in this work (four in total), we will give a name to each to avoid ambiguity. We start by defining the *classical* semantics of timed automata.

Timed transition systems

To formalize the semantics of timed automata we will use *timed transition systems* (TTS). The TTS associated to a timed automaton captures two things : (1) the complete state space of the TA, and (2) for each state of that space, the TTS tells which states are reachable in one move[3]. The state space of a timed automaton is the set of all configurations it can have, i.e. its current location and the value of its clocks. As the clocks take their values from a dense set, the state space is almost always infinite[4].

Definition 3 [Timed Transition System] A *timed transition system* $\mathcal{T}$ is a tuple $\langle S, \iota, \Sigma, \rightarrow \rangle$ where :

- S is a (possibly infinite) set of states.
- $\iota \in S$ is the initial state.
- Σ is a finite set of labels.
- $\rightarrow \subseteq S \times \Sigma \cup \mathbb{R}^{\geq 0} \times S$ is the transition relation with $\mathbb{R}^{\geq 0}$ being the set $\{x \in \mathbb{R} \mid x \geq 0\}$ of all non-negative real numbers.

□

[3] In this context, a *move* is understood as either a discrete jump or an arbitrarily small time increment.

[4] One could make a timed automaton which never lets time elapse, by taking an infinite number of transitions at time zero. Then indeed, the state space of that automaton would be finite.

Timed transition systems are very useful because they contain in a somewhat concise manner all the *possible behaviors* of a timed system. This notion of possible behavior is formalized by a *path*, and is defined as follows :

Definition 4 [Path in TTS] A *finite path* in the timed transition system $\mathcal{T} = \langle S, \iota, \Sigma, \rightarrow \rangle$ is a finite sequence alternating between state and transition labels, and is noted λ. Let $\lambda = (s_0, \tau_0, s_1, \tau_1, \ldots, \tau_{n-1}, s_n)$. λ is a finite path of $\mathcal{T}$ if (1) for every $i \in [0, n], s_i \in S$ and (2) for every $i \in [0, n), (s_i, \tau_i, s_{i+1}) \in \rightarrow$. The *length* of λ is $n + 1$ and is denoted $\mid \lambda \mid$. This definition is extended to infinite paths in the obvious way and the length of such a path is $+\infty$. A path λ is *initial* if its first state is the initial state. We write $\mathsf{Path}_{\mathsf{F}}(\mathcal{T})$ the set of all *finite initial paths* of $\mathcal{T}$ and $\mathsf{Path}_{\infty}(\mathcal{T})$ the set of all *infinite initial paths* of $\mathcal{T}$. The set of states which *appear* in λ is noted $\mathsf{State}(\lambda)$ □

Now we can define the useful notion of reachability in TTS :

Definition 5 [state-reachability in TTS] Let $\mathcal{T} = \langle S, \iota, \Sigma, \rightarrow \rangle$ be a timed transition system and a state $s \in S$. The state s is *reachable* in $\mathcal{T}$ if there exists a finite initial path $\lambda = (s_0, \tau_0, s_1, \tau_1, \ldots, \tau_{n-1}, s_n)$ such that $s_n = s$. The set of all reachable states in $\mathcal{T}$ is noted $\mathsf{Reach}(\mathcal{T})$. □

As stated in the introduction, the verification process we employ requires a composition of the interacting system models. We cannot perform this composition at the syntax level because the environment and controller semantics will not be the same. Hence, we need a way of composing timed transition systems. This composition requires a partitioning of the synchronization labels into three sets, so we refine our definition of TTS with a definition of *structured* timed transitions systems.

Definition 6 [Structured Timed Transition System] A *structured timed transition system* (STTS) $\mathcal{T}$ is a tuple $\langle S, \iota, \Sigma^{\mathsf{in}}, \Sigma^{\mathsf{out}}, \Sigma^{\tau}, \rightarrow \rangle$ where :

- S is a (possibly infinite) set of states.
- $\iota \in S$ is the initial state.
- The set of labels is partitioned into input (Σ^{in}) output (Σ^{out}) and internal labels (Σ^{τ}).
- $\rightarrow \subseteq S \times \Sigma^{\mathsf{in}} \cup \Sigma^{\mathsf{out}} \cup \Sigma^{\tau} \cup \mathbb{R}^{\geq 0} \times S$ is the transition relation.

□

Before formalizing the notion of composition mentioned earlier, we need to take a few precautions. We have seen that timed automata communicate through synchronization on common labels. This communication is blocking and we need

to make sure that (1) no automaton deliberately refuses an input from another and (2) no automaton sends an output to one that cannot receive it. These issues are resolved by imposing *input enabledness* of the STTS.

Definition 7 [Input Enabled STTS] A STTS $\mathcal{T} = \langle S, \iota, \Sigma^{\mathsf{in}}, \Sigma^{\mathsf{out}}, \Sigma^{\tau}, \rightarrow \rangle$ is *input enabled* if it can accept any input symbol, at any time. Formally, $\mathcal{T}$ must be such that for all $\sigma \in \Sigma^{\mathsf{in}}$, for all $s_1 \in S$ there exists $s_2 \in S$ such that $(s_1, \sigma, s_2) \in \rightarrow$. □

We are now ready to define the composition of two STTS. Note that this definition can be extended to the composition of any number of STTS.

Definition 8 [Composition of STTS] Consider two input enabled STTS $\mathcal{T}_1 = \langle S_1, \iota_1, \Sigma_{2\rightarrow 1}, \Sigma_{1\rightarrow 2}, \Sigma_1^{\tau}, \rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle S_2, \iota_2, \Sigma_{1\rightarrow 2}, \Sigma_{2\rightarrow 1}, \Sigma_2^{\tau}, \rightarrow_2 \rangle$. The parallel composition of $\mathcal{T}_1$ and $\mathcal{T}_2$, noted $\mathcal{T}_1 || \mathcal{T}_2$ is the TTS $\mathcal{T} = \langle S, \iota, \Sigma, \rightarrow \rangle$ such that :

1. $S = \{(s_1, s_2) \mid s_1 \in S_1 \text{ and } s_2 \in S_2\}$ and $\iota = (\iota_1, \iota_2)$.
2. $\Sigma = \Sigma_{1\rightarrow 2} \cup \Sigma_{2\rightarrow 1} \cup \Sigma_1^{\tau} \cup \Sigma_2^{\tau}$.
3. For every $\sigma \in \Sigma \cup \mathbb{R}^{\geq 0}$, we have that $((s_1, s_2), \sigma, (s_1', s_2') \in \rightarrow$ iff one of the following assertions holds :
   - [Internal move of $\mathcal{T}_1$] $\sigma \in \Sigma_1^{\tau}$ and $(s_1, \sigma, s_1') \in \rightarrow_1$ and $s_2' = s_2$.
   - [Internal move of $\mathcal{T}_2$] $\sigma \in \Sigma_2^{\tau}$ and $(s_2, \sigma, s_2') \in \rightarrow_2$ and $s_1' = s_1$.
   - [Synchronized move] $\sigma \in \Sigma_{1\rightarrow 2} \cup \Sigma_{2\rightarrow 1} \cup \mathbb{R}^{\geq 0}$ and $(s_1, \sigma, s_1') \in \rightarrow_1$ and $(s_2, \sigma, s_2') \in \rightarrow_2$.

□

Definition 8 is rather straightforward. In the composition, a STTS can fire internal events without disturbing the other and vice versa. The third assertion in the definition of $\rightarrow$ ensures that (1) synchronization on common labels does happen correctly, and (2) that time elapses identically on both sides.

Classical semantics of timed automata

We are now ready to give a semantics definition to timed automata. Bear in mind that several other semantics will be used in the following chapters, here we define what is called the *classical* semantics of timed automata.

Definition 9 [Timed Automata - Classical Semantics] The classical semantics of the timed automaton $A = \langle \mathsf{Loc}, l_0, \mathsf{Var}, \mathsf{Inv}, \mathsf{Lab}^{\mathsf{in}}, \mathsf{Lab}^{\mathsf{out}}, \mathsf{Lab}^{\tau}, \mathsf{Edg} \rangle$, noted $[\![A]\!]$, is the STTS $\langle S, \iota, \Sigma^{\mathsf{in}}, \Sigma^{\mathsf{out}}, \Sigma^{\tau}, \rightarrow \rangle$ where :

- $S = \{(l, v) \mid l \in \mathsf{Loc}$ and v is a valuation of the clocks in Var such that $v \models \mathsf{Inv}(l)\}$.
- $\iota = (l_0, v_0)$ with v_0 being a valuation of the clocks in Var assigning the value zero to each clock.
- $\Sigma^{\mathsf{in}} = \mathsf{Lab}^{\mathsf{in}}, \Sigma^{\mathsf{out}} = \mathsf{Lab}^{\mathsf{out}}, \Sigma^{\tau} = \mathsf{Lab}^{\tau}$.
- For every $\sigma \in \Sigma^{\mathsf{in}} \cup \Sigma^{\mathsf{out}} \cup \Sigma^{\tau} \cup \mathbb{R}^{\geq 0}$, the transition $((l, v), \sigma, (l', v')) \in \rightarrow$ iff one of the following assertions holds :
  - $\sigma \in \Sigma^{\mathsf{in}} \cup \Sigma^{\tau} \cup \Sigma^{\mathsf{out}}$ and there exists an edge $(l, l', g, \sigma, R) \in \mathsf{Edg}$ with $v \models g$ and v' being the same valuation than v but assigning the value zero to the clocks in R.
  - The edge mentioned above doesn't exist, $\sigma \in \Sigma^{\mathsf{in}}$ and $(l', v') = (l, v)$.
  - $\sigma \in \mathbb{R}^{\geq 0}$, $l' = l$ and for each clock $x \in \mathsf{Var}$ we have that (1) $v'(x) = v(x) + t$ and (2) $\forall t' \in [0, t] : v + t' \models \mathsf{Inv}(l)$.

□

Notice how definition 9 formalizes the intuitions given previously. The semantic state space of the timed automaton (S in the definition) is the set of all configurations (i.e. location and clock valuation) which satisfy the invariant predicates. Transitions can occur along enabled edges, which is only the case when their guard is satisfied by the current clock valuation. The syntactic component R of an edge indicates which clocks need to be reset after taking that edge. A timed automaton can stay in (or enter) a location if and only if the corresponding location predicate is satisfied. Finally, observe that our definition of the classical semantics of timed automata is input-enabled : if an input arrives and no edge specifies what to do, an implicit self-loop occurs (as stated by the second assertion in the definition of $\rightarrow$).

Illustration of the semantics definition

This definition is essential to the following chapters so we illustrate with an example. Let A be the timed automaton depicted in figure 2.1. The syntax of that automaton has not been given formally but it is straightforward. Its classical semantics is the STTS $[\![A]\!] = \langle S, \iota, \Sigma^{\mathsf{in}}, \Sigma^{\mathsf{out}}, \Sigma^{\tau}, \rightarrow \rangle$ where :

- $S = \{(l, v) \mid l \in \{1, 2, 3\}, v(t) \in [0, +\infty)\} \cup \{(4, v) \mid v(t) \in [0, 3]\}$
- $\iota = (1, v_0)$ with $v_0(t) = 0$
- $\Sigma^{\mathsf{in}} = \Sigma^{\mathsf{out}} = \phi$, $\Sigma^{\tau} = \{\epsilon, \mathsf{EnterPassword}, \mathsf{BadPassword}, \mathsf{PasswordOK}\}$

- $\rightarrow =$ $\{((1,v), \mathsf{EnterPassword}, (2,v)) \mid v(t) \in [0,+\infty)\}$
  $\cup$ $\{((2,v), \mathsf{PasswordOK}, (2,v)) \mid v(t) \in [0,+\infty)\}$
  $\cup$ $\{((2,v), \mathsf{BadPassword}, (4,v')) \mid v(t) \in [0,+\infty), v'(t) = 0\}$
  $\cup$ $\{((4,v), \epsilon, (1,v)) \mid v(t) = 3\}$
  $\cup$ $\{((l,v), \delta, (l, v+\delta)) \mid \delta \in \mathbb{R}^{\geq 0}, l \in \{1,2,3\}, v(t) \in [0,+\infty)\}$
  $\cup$ $\{((4,v), \delta, (l, v+\delta)) \mid \delta \in \mathbb{R}^{\geq 0}, v+\delta(t) \in [0,3]\}$

2.3 Timed Systems Analysis

When using timed automata to analyze a timed system, we are generally interested not only by the model itself but by the formal properties that can be inferred from it. For example, an interesting property of the automaton of figure 2.1 could be phrased as "A brute-force attack of n password attempts requires at least $3n$ time units". Of course, we usually want to be more formal and this section will show how properties are defined and used in practice.

Formal specifications have many kind of properties and this work will mainly focus on one kind : *safety properties*. A safety property requires a formal system to stay in a predefined subset of the state space, which is usually referred as the safe states or good states. These kind of properties are very useful in practice and suffice for many applications. One notable exception where safety properties are not enough is the liveness (or fairness) type of property, which requires the model to actually *do* what it is supposed to, in a finite amount of time, and to have no deadlock[5].

To define safety properties formally, we need to extend the notion of reachability of definition 5 to *regions*. A region of a timed transition system is a subset of its state space. A region R is *reachable* in the TTS $\mathcal{T}$ if $R \cap \mathsf{Reach}(\mathcal{T}) \neq \phi$.

Definition 10 [Safety Property in TTS] Let $\mathcal{T} = \langle S, \iota, \sigma, \rightarrow \rangle$ and let $R \subseteq S$ be a region representing a set of good states. $\mathcal{T}$ is safe for R iff $\mathsf{Reach}(\mathcal{T}) \subseteq R$. □

In practice, we use algorithms which compute $\mathsf{Reach}(\mathcal{T})$, and determine if that set has an intersection with the complement of R in S. The set $\bar{R} = S \setminus R$ is often called the set of *bad* states.

To formalize properties in a natural way, we usually use what is called a *monitor* automaton. This automaton usually consists of two states : Good and Bad with one edge going from the first to the second. On that edge we attach a predicate corresponding to the negation of the property we want to verify. That monitor automaton is then included in the composition, along with the other agents in the system and we compute a reachability analysis to verify if indeed

[5]Indeed, a model which sits forever in its initial state (provided that state is safe) trivially satisfies any safety property. This is usually easily detected, however, and does not always need to be verified formally.

Bad is never reachable. Of course, some safety properties will be complex enough so that they cannot be expressed with only one clock predicate, and in that case a bigger monitor automaton will be needed.

For example, to formalize the property mentioned previously, the monitor automaton of figure 2.3 would work. To express the property accurately, it needs three states.

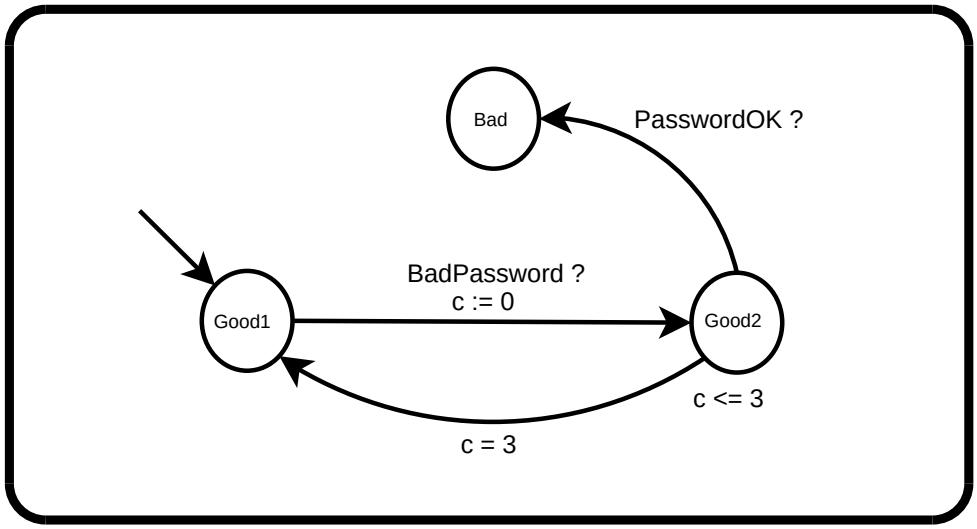

Figure 2.3: Example monitor automaton.

2.4 Timed Automata in the Literature

This chapter only scratched the surface of the theory of timed automata. To keep this work at a reasonable size, we have only given the few definitions needed for the understanding of the following chapters. We have not described, for instance, how timed automata can be analyzed effectively using automated tools. This information, along with decidability results, can be found for example in [BY04] and [AD94].

Chapter 3

Implementability of Timed Controllers

The final goal of this work is to develop a methodology for building real-time embedded software with formal assurance that it will work as intended. In the previous chapter, we have seen that timed automata can be used to provide a model for the controller and its environment and that it is possible to obtain formal assurance that this model is *correct*, in the sense that it satisfies some formal requirement.

However, as stated in the introduction, this proof of correctness is only valid in the context of the synchrony hypothesis. This is because the classical semantics of timed automata allows unimplementable behaviors. Consider the following controller :

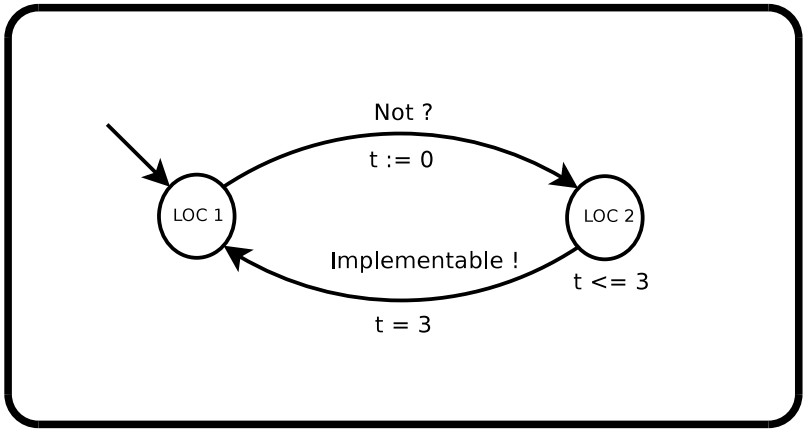

Figure 3.1: This timed automaton has a classical semantics that is not implementable because it requires an infinite clock precision, which any hardware is incapable of.

This controller is obviously not implementable and it illustrates one reason

for this : any computer hardware has clocks with *finite precision*. The clocks of timed automata take their values from a dense set, whereas hardware clocks are always discrete and thus have limited precision. If the correctness of a control strategy relies on infinite time precision, then it will not be implementable. We need to make sure that clock-rounding errors will not induce bad behaviors of the controller.

Infinite clock precision is not the only cause of unimplementability, as illustrated by this controller :

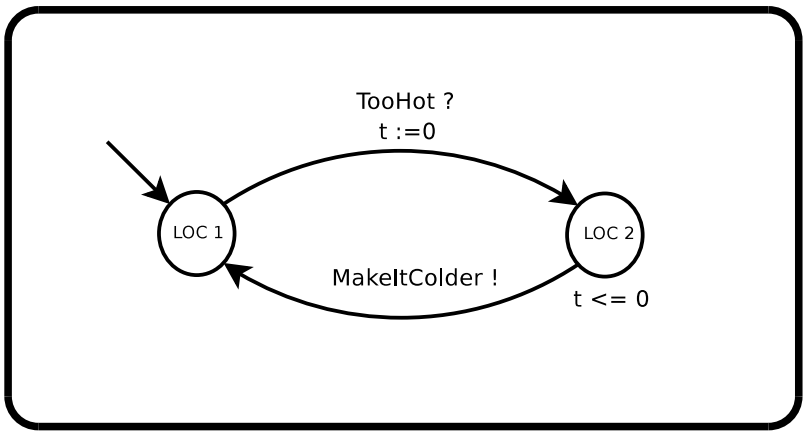

Figure 3.2: This second example illustrates the unimplementability caused by the instantaneity required by the classical semantics. The classical semantics of timed automata requires that these two transitions be taken in zero time units which is impossible on any hardware.

This last example does not require infinite clock-precision (even digital clocks can hold an exact null value) and yet it is not implementable. This is because until now we have considered that the communication between the interacting components of the system was *instantaneous*. This is very convenient at the modeling level, but again it needs to be formally validated. Communications in real implementations will always introduce delays and we need to make sure that the control strategy remains correct when they happen.

The synchrony hypothesis can be seen as an undesirable "side-effect" of the classical semantics of timed automata. This problem could be solved by taking the hardware limitation into account in the controller design process. This amounts to make *syntactic* changes to the controller automaton, by adding enough nodes, clocks, predicates and edges in order to obtain a implementable model.

The Almost ASAP semantics approach suggests a different path; it incorporates the hardware limitations the target platform by the means of a *semantics* change. This is very convenient because it keeps the controller automaton free of implementation details.

3.1 Overview of the Almost ASAP Approach

Recall the classical semantics of timed automata defined in the previous chapter. This semantics interprets the syntax in the most obvious manner; an edge guarded by the predicate $t = 3$ can only be taken when t is exactly 3. Similarly, the classical semantics makes the emission of an output event correspond to its arrival to the receiver *at the exact same time*. The idea behind the Almost ASAP semantics is to make an interpretation of the syntax that is more relaxed; by allowing some bounded imprecision on the guard evaluation mechanism, and some bounded delay during synchronization on common labels, the almost ASAP semantics makes the control strategy something which can be executed by hardware.

The Almost ASAP semantics is parametric, it takes a real-valued parameter noted Δ, which represents a superior bound on both the time-imprecision of the hardware and communication delay. This parameter is used throughout the semantics, extending its possible behaviors to a wider range. For example, with the AASAP semantics the interpretation of every guard of the automaton is enlarged by the parameter Δ. An edge with a guard predicate $t = 3$ will be traversable when $t \in [3 - \Delta, 3 + \Delta]$. So, if the time imprecision of the hardware is bounded by Δ, the controller implementation will have a chance to take the edge appropriately. The AASAP semantics also makes a distinction between the emission of an event and its arrival. This is done by duplicating event labels : one set corresponds to the emissions, and the other to the actual treatments by the receivers. When an event is emitted by the environment, the almost ASAP semantics allows the controller to wait *up to* Δ time units before taking that event into account. This represents the time needed for the hardware to detect the event appropriately.

Suppose we have a controller modeled with a timed automaton A. We can prove that $[\![A]\!]$ satisfies some properties, but we cannot do so directly for its almost ASAP semantics : $[\![A]\!]_\Delta^{\mathsf{AAsap}}$. To be able to analyze the almost ASAP semantics effectively with the tools and theories available, we need a way of transforming A into some A' such that $[\![A']\!]$ is somehow *similar* to $[\![A]\!]_\Delta^{\mathsf{AAsap}}$. In this work, we will use simulations - and the dual notion of refinement - to formalize this similarity.

3.2 Simulation and Controller Refinement

In the previous chapter, we have defined the structured timed transition system, which in our context is used to represent the timed state space of the controllers and environments we want to model. The STTS describes for each state, which states are reachable in one transition - discrete or continuous. The possible behaviors of an STTS is represented by the set of all its initial paths. The concept of simulation is closely related to initial paths; intuitively, an STTS A

can simulate another STTS B if every initial path of B is also an initial path of A. In short, in order for A to simulate B it must have *at least* as much possible behaviors as B. The notion simulation is defined formally by constructing a relation between the state spaces of the two STTS :

Definition 11 [Simulation relation for STTS] Given two STTS, $\mathcal{T}_1 = \langle S_1, \iota_1, \Sigma_1^{\mathsf{in}}, \Sigma_1^{\mathsf{out}}, \Sigma_1^{\tau}, \rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle S_2, \iota_2, \Sigma_2^{\mathsf{in}}, \Sigma_2^{\mathsf{out}}, \Sigma_2^{\tau}, \rightarrow_2 \rangle$, let $\Sigma = \Sigma_1^{\mathsf{out}} \cup \Sigma_1^{\mathsf{in}} \cup \Sigma_1^{\tau}$, we say that $\mathcal{T}_2$ is *simulable* by $\mathcal{T}_1$, noted $\mathcal{T}_2 \sqsubseteq \mathcal{T}_1$, if there exists a relation $R \subseteq S_2 \times S_1$ (called a *simulation relation*) such that:

- $(\iota_2, \iota_1) \in R$;
- for any $(s_2, s_1) \in R$, for any $\sigma \in \Sigma \cup \mathbb{R}^{\geq 0}$, for any s_2' such that $(s_2, \sigma, s_2') \in \rightarrow_2$, there exists $s_1' \in S_1$ such that $(s_1, \sigma, s_1') \in \rightarrow_1$ and $(s_2', s_1') \in R$.

□

When two STTS can simulate each other, we say that they are *mutually similar*, but this will not be needed in the following. The similarity we have defined is sufficient in our context because it preserves safety properties.

To reason on preservation of safety properties, it is convenient to think in terms of *refinement*. The notion of refinement is dual to that of simulation; if $B \sqsubseteq A$, i.e. A can simulate B, then we say that B *refines* A, in the sense that B has less possible behaviors than A. Safety properties which are satisfied by an STTS are also satisfied by *any refined version* of that STTS. In fact, simulation relations even preserve stronger properties such as the ones that can be expressed with LTL formulas.

Recall from definition 10 that an STTS $\mathcal{T}$ is safe for a region R iff $\mathsf{Reach}(\mathcal{T}) \subseteq R$. The preservation of safety properties is formalized as follows :

Theorem 1 Let $\mathcal{T}_1 = \langle S_1, \iota_1, \Sigma_1^{\mathsf{in}}, \Sigma_1^{\mathsf{out}}, \Sigma_1^{\tau}, \rightarrow_1 \rangle$, $\mathcal{T}_2 = \langle S_2, \iota_2, \Sigma_2^{\mathsf{in}}, \Sigma_2^{\mathsf{out}}, \Sigma_2^{\tau}, \rightarrow_2 \rangle$ be two STTS such that $\mathcal{T}_1 \sqsubseteq \mathcal{T}2$. If $\mathcal{T}_2$ is safe for a region $R \subseteq S_2 \subseteq S_1$ then $\mathcal{T}_1$ is also safe for R.

This theorem will be extremely useful in the following. Recall that our ultimate goal is to generate provably correct code in a systematic fashion. As we have seen in the introduction, this will require the use of an implementation semantics, which details how the hardware executes the control strategy described by the corresponding timed automaton. To verify the soundness of the implementation semantics, we will use simulation proofs. Indeed, if we can prove that an implementation semantics is simulable by the AASAP, then we will not need to verify the safety of that implementation semantics, thanks to the above theorem. Such a simulation proof will be given for the Real-Time Semantics in the next chapter.

3.3 ELASTIC Controllers and ASAP Semantics

As seen in the previous chapter, invariant predicates can be attached to the nodes of a timed automaton to force it to leave its location. This was necessary in order to obtain a controller which does not idle constantly. A simpler way to work with timed automata is to remove invariants completely and assume an ASAP behavior. This ASAP semantics makes the automaton take every action *as soon as possible*, thus removing the need for invariant predicates completely. In [DDR05b], controllers are modeled with this subclass of timed automata, and are called ELASTIC controllers.

Definition 12 [ELASTIC Controller] An Elastic controller A is a tuple $\langle \mathsf{Loc}, l_0, \mathsf{Var}, \mathsf{Lab}^{\mathsf{in}}, \mathsf{Lab}^{\mathsf{out}}, \mathsf{Lab}^{\tau}, \mathsf{Edg} \rangle$ where:

- Loc is a finite set of locations;
- $l_0 \in \mathsf{Loc}$ is the initial location;
- $\mathsf{Var} = \{x_1, \ldots, x_n\}$ is a finite set of clocks;
- $\mathsf{Lab} = \mathsf{Lab}^{\mathsf{in}} \cup \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^{\tau}$ 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 (l, l', g, σ, R) where $l, l' \in \mathsf{Loc}$ are locations, $\sigma \in \mathsf{Lab}$ is a label, $g \in \mathsf{Rect}_{\mathsf{c}}(\mathsf{Var})$ is a guard and $R \subseteq \mathsf{Var}$ is a set of clocks to be reset.

□

Note that ELASTIC controllers only use *closed* rectangular predicates. This is because in our context of finite clock-precision, comparing a clock to a non-closed rational interval does not make sense.

3.4 Almost ASAP Semantics

Before defining the AASAP semantics we need a couple more definitions:

Definition 13 [True Since] We define the function "True Since", noted $\mathsf{TS} : [\mathsf{Var} \to \mathbb{R}^{\geq 0}] \times \mathsf{Rect}_{\mathsf{c}}(\mathsf{Var}) \to \mathbb{R}^{\geq 0} \cup \{-\infty\}$, 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 TS function will be used in the AASAP semantics definition to express that the controller *must* take action when the predicate of an outgoing edge has been true long enough. Recall that the AASAP semantics allows a delay of up to Δ time units before taking a transition.

To represent the clock imprecision, the guards attached to the edges of the elastic controller are enlarged. This is formalized as follows :

Definition 14 [Guard Enlargement] Let $g(x)$ be the rectangular constraint "$x \in [a, b]$", the rectangular constraint ${}_{\Delta}g(x)_{\Delta}$ with $\Delta \in \mathbb{Q}^{\geq 0}$ is the formula "$x \in [a - \Delta, b + \Delta]$" if $a - \Delta \geq 0$ and "$x \in [0, b + \Delta]$" otherwise. If g is a closed rectangular predicate then ${}_{\Delta}g_{\Delta}$ is the set of closed rectangular constraints $\{{}_{\Delta}g(x)_{\Delta} \mid g(x) \in g\}$. □

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

Definition 15 [AASAP semantics] Given an ELASTIC controller

$$A = \langle \mathsf{Loc}, l_0, \mathsf{Var}, \mathsf{Lab}^{\mathsf{in}}, \mathsf{Lab}^{\mathsf{out}}, \mathsf{Lab}^{\tau}, \mathsf{Edg} \rangle$$

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

$$\mathcal{T} = \langle S, \iota, \Sigma^{\mathsf{in}}, \Sigma^{\mathsf{out}}, \Sigma^{\tau}, \rightarrow \rangle$$

where:

($A1$) S is the set of tuples (l, v, I, d) where $l \in \mathsf{Loc}$, $v \in [\mathsf{Var} \rightarrow \mathbb{R}^{\geq 0}]$, $I \in [\Sigma^{\mathsf{in}} \rightarrow \mathbb{R}^{\geq 0} \cup \{\bot\}]$ and $d \in \mathbb{R}^{\geq 0}$;

($A2$) $\iota = (l_0, v, I, 0)$ where v is such that for any $x \in \mathsf{Var} : v(x) = 0$, and I is such that for any $\sigma \in \Sigma^{\mathsf{in}}$, $I(\sigma) = \bot$;

($A3$) $\Sigma^{\mathsf{in}} = \mathsf{Lab}^{\mathsf{in}}$, $\Sigma^{\mathsf{out}} = \mathsf{Lab}^{\mathsf{out}}$, and $\Sigma^{\tau} = \mathsf{Lab}^{\tau} \cup \overline{\mathsf{Lab}^{\mathsf{in}}} \cup \{\epsilon\}$;

($A4$) The transition relation is defined as follows:

- for the discrete transitions, we distinguish five cases:

($A4.1$) let $\sigma \in \mathsf{Lab}^{\mathsf{out}}$. We have $((l, v, I, d), \sigma, (l', v', I, 0)) \in \rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$ such that $v \models {}_{\Delta}g_{\Delta}$ and $v' = v[R := 0]$;

($A4.2$) let $\sigma \in \mathsf{Lab}^{\mathsf{in}}$. We have $((l, v, I, d), \sigma, (l, v, I', d)) \in \rightarrow$ iff
  · either $I(\sigma) = \bot$ and $I' = I[\sigma := 0]$;
  · or $I(\sigma) \neq \bot$ and $I' = I$.

($A4.3$) let $\bar{\sigma} \in \overline{\mathsf{Lab}^{\mathsf{in}}}$. We have $((l, v, I, d), \bar{\sigma}, (l', v', I', 0)) \in \rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$, $v \models {}_{\Delta}g_{\Delta}$, $I(\sigma) \neq \bot$, $v' = v[R := 0]$ and $I' = I[\sigma := \bot]$;

($A4.4$) let $\sigma \in \mathsf{Lab}^\tau$. We have $((l, v, I, d), \sigma, (l', v', I, 0)) \in \longrightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$, $v \models_\Delta g_\Delta$, and $v' = v[R := 0]$;

($A4.5$) let $\sigma = \epsilon$. We have for any $(l, v, I, d) \in S : ((l, v, I, d), \epsilon, (l, v, I, d)) \in \longrightarrow$.

– for the continuous transitions:

($A4.6$) for any $t \in \mathbb{R}^{\geq 0}$, we have $((l, v, I, d), t, (l, v + t, I + t, d + t)) \in \longrightarrow$ iff the two following conditions are satisfied:

· for any edge $(l, l', g, \sigma, R) \in \mathsf{Edg}$ 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 $(l, l', g, \sigma, R) \in \mathsf{Edg}$ 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 definition

($A1$) The state space is made of 4-tuples (l, v, I, d). l and v are the location and clock-valuation as in the classical semantics. I is vector assigning a real value to every input label, representing the time elapsed since the last *untreated* occurrence of that label, or $\perp$ if there is no pending occurrence of that label. The AASAP semantics does not react instantaneously to input events, so it needs to remember how long it has delayed an input. The value d holds the time elapsed since the last location change occurred. That value will be used to delay edge crossings; the controller can ignore an enabled edge if d is smaller than Δ.

($A2$) The initial location definition is straightforward.

($A3$) As mentioned previously, the AASAP semantics duplicates input labels. A distinction is made between the emission of an output label σ and its reception as an input label $\overline{\sigma}$ by the controller. When the input is actually received, it is treated as an internal event, hence $\overline{\mathsf{Lab}^{\mathsf{in}}}$ is added to Σ^τ.

($A4.1$) To cross an edge tagged with an output label, the current clock valuation must satisfy the corresponding guard predicate, enlarged by Δ. The notation $v' = v[R := 0]$ means that v' is the same valuation than v but assigning the value 0 to the clocks in R. This behavior is exactly the same as in the usual semantics, except for the guard enlargement.

($A4.2$) This rule indicates what happens when a label is emitted by the environment. In the normal case, the corresponding value in I is set to zero. If

more than one input of the same kind is received before the controller had the chance to treat the first one, the semantics simply ignores the others and the "older" value in I is kept. Note that no edge is crossed at this point, the controller stays in its location for now. Also, this rule alone ensures the input enabledness of the controller.

($A4.3, 4, 5$) These rules should be clear with the previous comments.

($A4.6$) The previous rules defined when the controller *can* take action, this rule states when it *must* do so. It can let t time units elapse and not take a transition only if (1) the controller made a location change less than Δ time units ago : $d + t \leq \Delta$; or (2) there are no enabled outgoing edge, or if there are enabled outgoing edges, they have been enabled for less than Δ time units : $\mathsf{TS}(v + t, g) \leq \Delta$. Also, the controller cannot delay the treatment of an input event more than Δ time units : $(I + t')(\sigma) \leq \Delta$.

Summary of the Almost ASAP semantics

Let us summarize the previous formal definition. In our methodology, control strategies are expressed using a syntax that is free of invariant predicates. The controller always tries to take a transition as soon as it can thus, almost as soon as possible. The "almost" is quantified by the parameter Δ, which is the upper bound on three different delays or time imprecisions :

- The controller can wait up to Δ time units between two location changes, no matter the outgoing edge guards. This represents the fact that CPUs are not infinitely fast; a "location change" in the real world takes up a few processor cycles. It would be unreasonable to qualify a semantics allowing an infinite number of location changes in a finite amount of time as implementable.

- Since digital clocks aboard real computers have finite precision, every edge of the controller is enlarged by the value Δ. If clock-rounding errors are guaranteed to be always smaller than Δ, then then controller implementation will be able to cross the edge appropriately in all situations, despite the imprecision of its clocks.

- Finally, the almost ASAP semantics models communication delays that can be as large as Δ. Each time an input arrives, the controller can wait up to Δ time units before taking the corresponding edge, thus taking the input into account.

Example

As with the Classical Semantics, we illustrate the Almost AASAP Semantics definition with an example. Again, we use the timed automaton of figure 2.1 but with a minor modification : consider EnterPassword as an input label and BadPassword and PasswordOK as output labels. Also, as we work with an ELASTIC controller this time, the predicate attached to location 4 is removed. To simplify the notations, consider that the clock valuation v and the input delay vector I are real values - I can still have the value $\perp$. There is only one clock and input label in this example so this works fine. Finally, in the definition of $\rightarrow$, assume that the free variables which are not quantified on the right hand side of the set definitions are simply non-negative real values. The Almost ASAP semantics of that ELASTIC controller is the structured timed transition system $[\![A]\!]_{\Delta}^{\mathsf{AAsap}} = \langle S, \iota, \Sigma^{\mathsf{in}}, \Sigma^{\mathsf{out}}, \Sigma^{\tau}, \rightarrow \rangle$ where :

- $S = \{(l, v, I, d) \mid l \in \{1,2,3,4\}, v, d \in \mathbb{R}^{\geq 0}, I \in \{\mathbb{R}^{\geq 0} \cup \perp\}\}$
- $\iota = (1, 0, \perp, 0)$
- $\Sigma^{\mathsf{in}} = \{\mathsf{EnterPassword}\}, \Sigma^{\mathsf{out}} = \{\mathsf{EnterPassword}, \mathsf{BadPassword}\}, \Sigma^{\tau} = \{\epsilon\}$
- $$\begin{aligned}
\rightarrow = \quad & \{((l,v,\perp,d), \mathsf{EnterPassword}, (l,v,0,d)) \mid l \in \{1,2,3,4\}\} && (1)\\
\cup \quad & \{((l,v,I,d), \mathsf{EnterPassword}, (l,v,I,d)) \mid l \in \{1,2,3,4\}\} && (2)\\
\cup \quad & \{((1,v,I,d), \overline{\mathsf{EnterPassword}}, (2,v,\perp,0))\} && (3)\\
\cup \quad & \{((2,v,I,d), \mathsf{PasswordOK}, (3,v,I,0)) \mid I \in \{\mathbb{R}^{\geq 0} \cup \perp\}\} && (4)\\
\cup \quad & \{((2,v,I,d), \mathsf{BadPassword}, (4,0,I,0)) \mid I \in \{\mathbb{R}^{\geq 0} \cup \perp\}\} && (5)\\
\cup \quad & \{((4,v,I,d), \epsilon, (1,v,I,0)) \mid v \in [3-\Delta, 3+\Delta], I \in \{\mathbb{R}^{\geq 0} \cup \perp\}\} && (6)\\
\cup \quad & \{((1,v,\perp,d), t, (1,v+t,\perp,d+t))\} && (7)\\
\cup \quad & \{((1,v,I,d), t, (1,v+t,I+t,d+t)) \mid I \in \{\mathbb{R}^{\geq 0} \cup \perp\}, I+t \leq \Delta\} && (8)\\
\cup \quad & \{((2,v,I,d), t, (2,v+t,I+t,d+t)) \mid I \in \{\mathbb{R}^{\geq 0} \cup \perp\}, d+t \leq \Delta\} && (9)\\
\cup \quad & \{((3,v,I,d), t, (3,v+t,I+t,d+t)) \mid I \in \{\mathbb{R}^{\geq 0} \cup \perp\}\} && (10)\\
\cup \quad & \{((4,v,I,d), t, (4,v+t,I+t,d+t)) \mid I \in \{\mathbb{R}^{\geq 0} \cup \perp\}, v+t \leq 3\} && (11)\\
\cup \quad & \{((4,v,I,d), t, (4,v+t,I+t,d+t)) \mid I \in \{\mathbb{R}^{\geq 0} \cup \perp\}, d+t \leq \Delta\} && (12)
\end{aligned}$$

Comments on the discrete transitions

- (1) The controller waits for the input event EnterPassword. When it arrives, I is set to zero but no location change occurs yet. Notice that there is no restriction on the location here; the controller can receive the input EnterPassword at any time, ensuring input-enabledness.
- (2) As stated by the semantics, the controller ignores the subsequent emissions of an untreated input symbol; I and d are left untouched (We know that there is an untreated occurrence of the input symbol because I is a real value and not $\perp$).

- (3) This set of transitions represent what happens when the controller has taken the input into account. I is reset back to $\perp$, d is reset to zero, and the location changes from 1 to 2.
- (4) There is no guard attached to the edge $2 \rightarrow 3$, so this transition can be taken for any valuation v. It is important to note however, that this does not mean that the controller can delay the emission indefinitely; at some point it will *have* to make that transition when the transition relation $\rightarrow$ does not allow it to let time pass anymore.
- (5) This works just the same as the previous one, except that we reset the clock valuation as required by the edge $2 \rightarrow 4$.
- (6) The edge $4 \rightarrow 1$ illustrates how the guard enlargement works. This edge is enabled as long as the clock valuation satisfies the enlarged guard predicate.

Comments on the continuous transitions

- (7) Since the only outgoing edge of location 1 corresponds to an input, the controller can wait there indefinitely, as long as no input has arrived, i.e. as long as $I = \perp$.
- (8) When in location 1, the controller can delay an untreated input for as long as Δ time units but not longer.
- (9) When in location 2, the controller faces two outgoing edges that are always enabled. The classical semantics would require it to fire one of the two immediately. Here, we allow the controller to stay in the location 2 for as long as d does not grow larger than Δ.
- (10) As there is no outgoing edge in location 3, the Almost ASAP controller can of course let time pass indefinitely.
- (11) This one is a little more subtle. The controller can delay the transition $4 \rightarrow 1$ for as long as the enlarged guard has not been true for longer than Δ time units. Formally, using the semantics definition, we know that the controller can wait t time units if $\mathsf{TS}(v + t, v + t \in [3 - \Delta, 3 + \Delta]) \leq \Delta$, which is equivalent to $v + t \leq 3$.
- (12) This is the same as (9), but for location 4.

3.5 Almost ASAP Semantics Analysis

We have defined the Almost ASAP Semantics formally and given some intuitions and examples about why it is useful to ensure implementability. However, as stated previously, the AASAP semantics is not known to model-checking tools and thus cannot be analyzed "as is". To work around this difficulty, it is shown in [DDR05b] how to construct a timed automaton with a Classical Semantics that is simulable by the Almost ASAP semantics of another automaton which can be constructed effectively. This is formalized by the following theorem :

Theorem 2 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 $[\![A]\!]_{\Delta}^{\mathsf{AAsap}} \sqsubseteq [\![\mathcal{A}^{\Delta}]\!]$ and $[\![\mathcal{A}^{\Delta}]\!] \sqsubseteq [\![A]\!]_{\Delta}^{\mathsf{AAsap}}$.

The interested reader will find all the details of this construction, with full proofs, in [DDR05b].

Thanks to this theorem, we are able to analyze the Almost ASAP Semantics effectively. We explain briefly how this is achieved in practice. Suppose we want to control an environment E to avoid some bad region B, subset of the state space of the STTS $[\![E]\!]$. Using the methods introduced in the previous chapter, it is possible to design an elastic controller A and verify formally that $[\![A]\!]$ is safe[1] and controls E to avoid B. Now we want to to know if the control strategy A is *implementable*. To achieve this, we need to find if there exists some $\Delta > 0$ such that $[\![A]\!]_{\Delta}^{\mathsf{AAsap}}$ controls $[\![E]\!]$ to avoid B as well. Hence, we construct the timed automaton $\mathcal{A}^{\Delta}$ and analyze it parametrically to verify if there exists a positive rational Δ such that the control strategy A controls its environment E in a safe *and* implementable manner. This parametric analysis can for example be computed with the help of HYTECH. If no parametric tool is available or if its computation does not terminate, it always possible to obtain an approximation of the best Δ by doing a binary search in the range of possible values. This way, the best Δ value can be approximated up to any precision.

Our work focuses on the satisfaction of safety properties, but in fact the above construction could be used to prove the satisfaction of more complex properties, such as LTL (Linear Temporal Logic) formulas or even real-time properties such as those expressed in MITL (Metric Interval Temporal Logic) or TCTL (Timed Computational Tree Logic).

[1] It must be mentioned that in this case, $[\![A]\!]$ is understood as the classical semantics of A, but with an ASAP behavior. The classical semantics as defined in the previous chapter cannot be used "as is" with ELASTIC controllers.

Chapter 4

Implementation Semantics

The previous chapter introduced a methodology with which we are able to design and validate implementable strategies, using the Almost ASAP Semantics. But as explained in the introduction, the Almost ASAP Semantics only describes *what* is to be done to keep the environment in a safe state, but contains little information about *how* a software controller is supposed to achieve this. We provide this additional information in an *implementation* semantics.

We will require the implementation semantics to have the following characteristics :

- **Parametricity.** As the AASAP semantics, the implementation semantics will need to be parametric. Its parameters will represent the run-time limitations that the implementation will have to face. This includes clock imprecision, communication delays, CPU speed, etc.
- **Almost ASAP Simulability.** We will require that the implementation semantics be simulable by the Almost ASAP semantics. As both semantics are parametric, the existence of such a simulation relation will depend on the value of the parameters. As we have seen, STTS refinements preserve safety properties. Thus, if a controller has been proven safe for a positive Almost ASAP semantics parameter Δ, then there will exist a set of parameters (constrained by Δ) such that the implementation semantics will be safe for those parameters.
- **Systematic Implementability.** This is probably the trickiest part. The implementation semantics will need to contain details about how exactly the software controller will function. We will require it to contain "enough details" in order to be able to create a fully-functional implementation in a systematic fashion. This will be hard to prove formally, so we will need to justify this with informal arguments.

The rest of this chapter will be organized as follows. First, we will review the proof-of-concept implementation semantics which was introduced in [DDR05b].

We will see how this semantics satisfies the above requirements, and then identify certain aspects which could be improved. This discussion will lead to a new implementation semantics, specially designed to take advantage of the features of real-time operating systems.

4.1 Program Semantics

In [DDR05b], Raskin et al. have demonstrated the practical applicability of the AASAP semantics with a simple, yet functional, approach. The resulting program is composed of an infinite loop which is executed as fast as possible. The body of that loop is called an *execution round* and it is composed of the following steps :

1. The clock register is read and stored in a global variable.
2. The input sensors are checked and stored in a vector.
3. The outgoing edges of the current location are checked. If one of them is enabled, the edge is taken and the input vector, current location and clock values are updated accordingly.
4. The next execution round starts immediately.

This simple implementation scheme has been chosen because it is obviously implementable. Also, this can formalized quite naturally if we make a couple assumptions; (1) the length of an execution round is bounded by a finite rational value, called Δ_L, and (2) the clock register of the CPU (or whichever clock value is read by the program) is updated every Δ_P time units. Using these two values, it is possible to create a formal semantics which is a refined version of the Almost ASAP semantics.

Clock rounding

Before giving a formal definition of the Program Semantics, we need a way to formalize the time-imprecision induced by digital clocks. This is done by using the following clock-rounding function, which converts an exact time value into its corresponding digital value, depending on the clock granularity Δ_P :

Definition 16 [Clock rounding] Let $T \in \mathbb{R}^{\geq 0}$ and $\Delta \in \mathbb{Q}^{>0}$.
$\lfloor T \rfloor_\Delta = \lfloor \frac{T}{\Delta} \rfloor \Delta$, where $\lfloor x \rfloor$ is the greatest integer k such that $k \leq x$.
Likewise, $\lceil T \rceil_\Delta = \lceil \frac{T}{\Delta} \rceil \Delta$, where $\lceil x \rceil$ is the smallest integer k such that $k \geq x$. □

From this definition, we obtain the following lemma, which will be used extensively in the following.

Lemma 1 For any $T \in \mathbb{R}^{\geq 0}$, any $\Delta \in \mathbb{Q}^{>0}$, we have that :

$T - \Delta < \lfloor T \rfloor_\Delta \leq T$, and

$T \leq \lceil T \rceil_\Delta < T + \Delta$.

4.1.1 Formalization of the Program Semantics

In order to prove that this simple implementation strategy is indeed simulable by the Almost ASAP semantics, we need a formal semantics definition for this strategy; this is done as follows :

Definition 17 [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 $\mathcal{T} = \langle S, \iota, \Sigma^{\mathsf{in}}, \Sigma^{\mathsf{out}}, \Sigma^\tau, \rightarrow \rangle$ where:

($P1$) 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 $f \in \{\bot, \top\}$;

($P2$) $\iota = (l_0, r, 0, I, 0, 0, \bot)$ where r is such that for any $x \in \mathsf{Var}$, $r(x) = 0$, I is such that for any $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, $I(\sigma) = \bot$;

($P3$) $\Sigma^{\mathsf{in}} = \mathsf{Lab}^{\mathsf{in}}$, $\Sigma^{\mathsf{out}} = \mathsf{Lab}^{\mathsf{out}}$, $\Sigma^\tau = \mathsf{Lab}^\tau \cup \overline{\mathsf{Lab}^{\mathsf{in}}} \cup \{\epsilon\}$;

($P4$) the transition relation $\rightarrow$ is defined as follows:

- for the discrete transitions:

($P4.1$) let $\sigma \in \mathsf{Lab}^{\mathsf{out}}$. $((l, r, T, I, u, d, \bot), \sigma, (l', r', T, I, u, 0, \top)) \in \rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$ such that $\lfloor T \rfloor_{\Delta_P} - r \models_{\Delta_S} g_{\Delta_S}$ and $r' = r[R := \lfloor T \rfloor_{\Delta_P}]$.

($P4.2$) let $\sigma \in \mathsf{Lab}^{\mathsf{in}}$. $((l, r, T, I, u, d, f), \sigma, (l, r, T, I', u, d, f)) \in \rightarrow$ iff
  · either $I(\sigma) = \bot$ and $I' = I[\sigma := 0]$;
  · or $I(\sigma) \neq \bot$ and $I' = I$.

($P4.3$) let $\bar{\sigma} \in \overline{\mathsf{Lab}^{\mathsf{in}}}$. $((l, r, T, I, u, d, \bot), \bar{\sigma}, (l', r', T, I', u, 0, \top)) \in \rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$ 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 $I' = I[\sigma := \bot]$;

($P4.4$) let $\sigma \in \mathsf{Lab}^\tau$. $((l, r, T, I, u, d, \bot), \sigma, (l', r', T, I, u, 0, \top)) \in \rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$ such that $\lfloor T \rfloor_{\Delta_P} - r \models_{\Delta_S} g_{\Delta_S}$ and $r' = r[R := \lfloor T \rfloor_{\Delta_P}]$.

($P4.5$) $((l, r, T, I, u, d, f), \epsilon, (l, r, T + u, I, 0, d, \bot)) \in \rightarrow$ iff either $f = \top$ or the two following conditions hold:

- for any $\bar{\sigma} \in \overline{\mathsf{Lab}^{\mathsf{in}}}$, for any $(l, l', g, \sigma, R) \in \mathsf{Edg}$, 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 $(l, l', g, \sigma, R) \in \mathsf{Edg}$, we have that $\lfloor T \rfloor_{\Delta_P} - r \not\models {}_{\Delta_S} g_{\Delta_S}$

– for the continuous transitions:

$(P4.6)$ $((l, r, T, I, u, d, f), t, (l, r, T, I+t, u+t, d+t, f)) \in \rightarrow$ iff $u+t \leq \Delta_L$.

□

4.1.2 Comments on the Program Semantics

The previous definition is clearly not obvious so we comment the most important parts.

State space

The state space of this new STTS is a bit more complicated than the one used with the AASAP semantics; it is composed of tuples of the form (l, r, T, I, u, d, f).

- l, I, and d have the same meaning they had in the AASAP semantics.
- The value T holds the exact time at which the current execution round was started.
- The function r represents the clock-valuation but it works differently from the valuation v of the AASAP semantics. Timed automata can have any number of clocks, all containing a different value and increasing simultaneously. In practice however, there is usually only one clock used, so the "clocks" used by the real-time program are materialized by integer variables which, when reset, are assigned to the current clock value (i.e. $\lfloor T \rfloor_{\Delta_P}$). To obtain a clock's value[1], the value stored in r must be substracted from T. The values stored in r are *digital* values (as opposed to T) and are thus always a multiple of Δ_P.
- The variable u holds the exact time elapsed since the beginning of the current execution round. Thus, $T + u$ is always the exact present time.
- The flag f is set to $\top$ when the current execution round ends. It holds the value $\bot$ at all other times.

[1]By current clock value, we mean the value of the clock at the beginning of the current execution round.

Transition relation

Remember that this Program Semantics is meant to be a *refined* version of the Almost ASAP semantics. Basically, this implies two things : (1) when the Program Semantics makes a discrete jump, the AASAP semantics must be able to follow and (2) when the Program Semantics lets time elapse without making any transition, the AASAP semantics must be able to do the same. We will not give the full simulation proof here (as another lengthy simulation proof will be detailed later), so while explaining the rules of the transition relation, we also give insights on how this makes the Program Semantics simulable by the Almost ASAP. The rules which are simple enough are not mentioned.

($P4.1$) In the Program Semantics, the guards are enlarged with the value $\Delta_S = \lceil \Delta_L + \Delta_P \rceil_{\Delta_P}$. It is rounded with Δ_P because this constant is intended to be written in real code - to enlarge the edge guards - so it needs to be expressible in a non-fractional amount of time units Δ_P. Observe that an edge guarded by the predicate g is enabled when $\lfloor T \rfloor_{\Delta_P} - r \models_{\Delta_S} g_{\Delta_S}$. $\lfloor T \rfloor_{\Delta_P}$ is the value held by the global variable we mentioned previously, and as r, g and Δ_S are readily available by the program, this is a computation that can be made by the software controller.

($P4.3$) This rule is self-explanatory, except for the $I(\sigma) > u$ requirement. Remember that inputs are checked only at the beginning of the execution round, so the program cannot treat an input which has arrived *after* the beginning of the current round, and must wait until the next one. The value u represents the time spent into the current round, so it must be strictly smaller than $I(\sigma)$ for the program to take the arrival of σ into account in the current round.

($P4.5$) This transition is taken when a new execution round is started. If the program has already made an action this round (i.e. $f = \top$) then a new round can start immediately. If not, a new round can start, but only if there are no enabled outgoing edges at the current location.

4.1.3 AASAP Simulability of the Program Semantics

In [DDR05b], a simulation proof is given, asserting that the Program Semantics is simulable by the Almost ASAP semantics, requiring only that Δ_P and Δ_L are small enough compared to Δ. This is stated by the following theorem :

Theorem 3 Let A be an ELASTIC controller, for any rationals Δ, Δ_L, $\Delta_P \in \mathbb{Q}^{>0}$ such that $\Delta > 3\Delta_L + 4\Delta_P$, we have $[\![A]\!]^{\mathsf{Prg}}_{\Delta_L,\Delta_P} \sqsubseteq [\![A]\!]^{\mathsf{AAsap}}_{\Delta}$.

We omit the proof of this theorem here, but it can be found in [DDR05b]. To clarify things a bit, we illustrate the existence of the bound $\Delta > 3\Delta_L + 4\Delta_P$ with an example.

Insights on the bound $3\Delta_L + 4\Delta_P$

The following scenario should make the value of this bound on Δ more clear. Consider an outgoing edge tagged with an output event, and guarded with the predicate $t = 3$. In this context, a worst-case scenario is to have $\lfloor T \rfloor_{\Delta_P} - r(t) = 3 - \Delta_S - \Delta_P$, which is the case when the guard has been missed by exactly one clock tick. Then, at worst, the edge will be taken at the end of the next round, when $\mathsf{TS}(t = 3) = 3\Delta_L + 4\Delta_P$. The $4\Delta_P$ term in the expression is the worst rounding-error possible in the guard-evaluation mechanism. The $3\Delta_L$ term is explained by (1) the fact the edge is fired $2\Delta_L$ time units after it is "enabled" and (2) that the the maximum error between $r(t)$ and the exact value of t can be as large as Δ_L (not considering the rounding error which has already been counted). Remember that the Almost ASAP semantics *must* take an edge which has been enabled for Δ time units, so this example shows that a simulation relation cannot exist between the two semantics unless $\Delta > 3\Delta_L + 4\Delta_P$. The interested reader will find the complete proof in [DDR05b].

4.1.4 Implementability of the Program Semantics

At this point it should be clear that, using the Program Semantics, it is possible to construct effectively a piece of software which can be considered provably correct. Again, this has been demonstrated successfully in [DDR05a].

4.1.5 Limitations of the Program Semantics

The implementation strategy that we have previously described was meant to demonstrate the usefulness of the Almost ASAP semantics in a practical manner. However, there is still space for improvements left and one of the main objectives of this work is to create an implementation strategy that could be even more useful.

The big downside of the Program Semantics just described is that it imposes a CPU usage of 100%. This is inconvenient for at least three reasons, which we detail in turn.

1. Multitasking

Clearly, a real-time task requiring a usage of 1 cannot be executed along-side other tasks. This is unfortunate because there are many cases where real-time infrastructures are best designed using many different independent tasks in parallel. The advantages of a multitask design are numerous :

- **Modularity.** Compared to monolithic designs, multitasking offers the benefit of a greater modularity. Tasks can be added, changed or removed with-

out affecting the other tasks (of course, we still need to verify that the real-time instance stays schedulable when adding and/or changing tasks).

- **Simplicity.** In many cases, using concurrency can lead to a simpler design, especially if the system needs to perform various, unrelated actions with different levels of importance.

- **Fault tolerance.** Usually, not all of the code is completely safety-critical and since most of the attention will have been put on assuring that the critical part works, the quality of the less important code might be lower. If a non-critical task crashes, it can be restarted without causing a catastrophe; monolithic systems however, can be brought down by the silliest bug in some relatively unimportant code.

- **Performance.** If one CPU is not fast enough, make it 2, or even 128 if need be ! As the processors' performance have begun to stall in the last few years, the only way to faster software lies in multi-cores, SMP and other multi-tasking hardware technologies. Of course, these require that the software be split into parallel tasks.

2. User interaction

The next inconvenience of having a real-time task with a 100% CPU usage is that it will certainly be very limited in terms of user interaction. The user interface code could be built-in, but this is hardly possible in our context of automatic code generation. Moreover, even if there is only one real-time task running in the system, it is very useful to be able to run a *background server*, which can execute tasks which are non-critical and not necessarily real-time.

For example, the real-time platform that was chosen to experiment with the code generation works this way. RTAI, the Real-Time Application Interface, is a *nano-kernel* which contains a real-time scheduler that runs a modified version of the LINUX operating system as its idle loop. The real-time system is operated from within LINUX. This setup is a very convenient environment; it has all the advantages of a modern hard real-time OS, along with all the applications and tools available to LINUX. However, the architecture of RTAI requires *not* to launch tasks which take up all CPU resources because otherwise LINUX won't be executed, leading to a unrecoverable freeze. The architecture of RTAI is described in chapter 5.

3. Power consumption

In our context of embedded controllers, the power consumption induced by the software execution is not to be overlooked. Embedded applications will often run on batteries, so we need to make sure that the controller does not waste

CPU power unnecessarily. In that respect, the Program Semantics can be clearly improved.

4.2 Real-Time Semantics

In this section, we introduce a new implementation semantics, which mimics the behavior of a periodic hard real-time task. It is quite similar to the Program Semantics of the previous section, with the notable difference that the beginning of each execution round are evenly spaced in a periodic fashion. This will be the period of the real-time task and will be called Δ_T. Additionally we require that every execution round ends within a fixed time amount, called Δ_D, representing the deadline of the periodic task. Finally, we require that $\Delta_D \leq \Delta_T$. This is illustrated at figure 4.1.

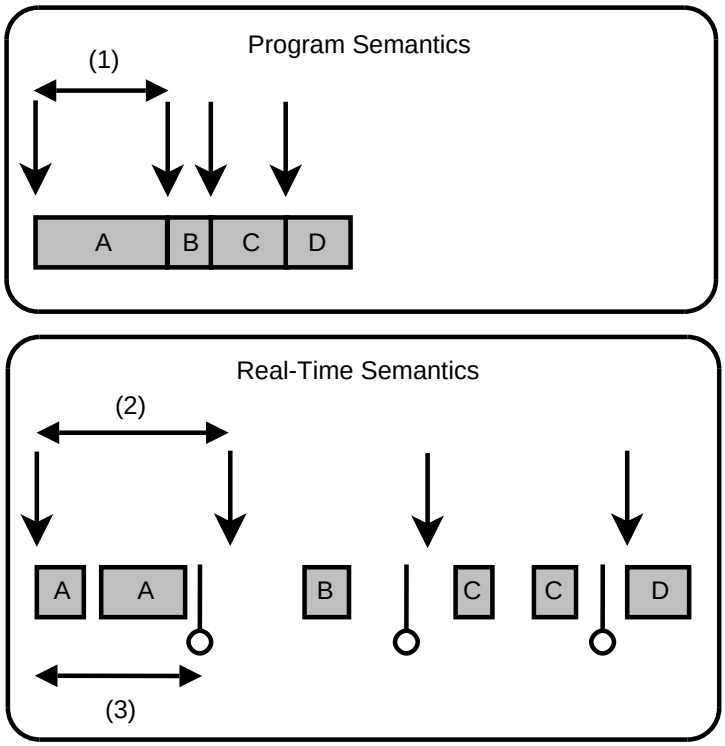

Figure 4.1: Comparison of the Program Semantics and Real-Time Semantics executing four consecutive execution rounds.

The above figure illustrates the respective behaviors of the two implementation semantics when executing four consecutive execution rounds (A, B, C and D). In the Program Semantics, the length of an execution round is bounded by Δ_L (1). The Real-Time Semantics has much tighter requirements and imposes that every execution round be started at periodic time intervals Δ_T (2). It also

requires that every execution round completes within Δ_D time units (3). Observe from the figures that the Real-Time Semantics allows the execution round to be *preempted*. This has no consequence, as long as the deadline is always met.

At a first glance, it might seem unnatural to improve a semantics by forcing it to *slow down*, so we motivate this choice with a few arguments.

Why make the controller idle ?

The answer to that question is simple; the environment does not always need to be watched over a 100% of the time. Consider the following example. An Almost ASAP controller has been proven safe, and controls its environement for all values of $\Delta < 2$ seconds. After implementing the controller, the worst case execution time of the execution round is estimated to 10ms and the platform it is running on has a clock granularity of 1ms. Using the traditional implementation strategy, we would obtain a program running the execution round at least 100 times per second, which in this case is about 200 times more than what was really needed.

True, the previous argument is not really a strong one, because if we wanted to design controllers with such slow (2 seconds) reaction times, we would not have needed to part from the synchrony hypothesis in the first place, and the AASAP methodology would not be needed. Still, even if we will probably not bring the CPU usage of our controllers from 100% to 0,5% like in the previous example, the gain will lie somewhere in the middle, providing all the advantages we have previously mentioned (the possibility of running other tasks and / or running a background server, consuming less power, etc.).

4.2.1 Overview of the Real-Time Semantics

In practice, the implementation of this new semantics will be quite similar to what was done with the Program Semantics, except that it explicitly uses some features of a real-time OS. This new implementation strategy is as follows :

- At startup, the controller initializes its variables, and asks the RTOS to create a new periodic task with a period of $\lfloor \Delta_T \rfloor_{\Delta_P}$ (the period must be an integral number of clock ticks). Usually, the task-creation process in an RTOS is just a matter of indicating which procedure to call when the task is to be run, and which priority it has. We will call this procedure **rt_main**.
- Every time the task is scheduled, the procedure **rt_main** is invoked[2] and performs the same actions than the body of the execution loop in the Program Semantics (clock-reading, sensor-checking and edge-crossing, in that order).

[2]Actually, in practice it is not really a procedure call in the traditional sense. Rather, the context of the RT task is restored to where it was right before its preemption.

- At the end of its execution, **rt_main** indicates to the RTOS that it has finished, and wishes to be rescheduled on the next period.

In practice, this implementation scheme would resemble to the C-like code fragment of figure 4.2.

```
rt_main() {
    running = true;
    while(running) {
        ... // read the clock value
        ... // check sensors
        ... // take enabled edge if, any
        ... // running gets false if current state is a sink
        wait_next_period();
    }
}

init_vars() {
    ... // initialize variables
}

cleanup() {
    ... // free all used resources
}

main() {
    init_vars();
    tid = create_rt_task(rt_main, priority, period);
    wait_task_completion(tid);
    cleanup();
}
```

Figure 4.2: Basic structure of a C implementation following the Real-Time Semantics.

4.2.2 Formalization of the Real-Time Semantics

To obtain formal assurance of correctness for our real-time implementations, we will proceed in the same way than in [DDR05b], i.e. formalize the implementation strategy with an STTS-construction definition and make a simulation proof with the Almost ASAP semantics. We start by defining the Real-Time Semantics of an ELASTIC controller (comments follow the definition).

Definition 18 [Realtime Task Semantics] Let A be an ELASTIC controller and $\Delta_T,\ \Delta_D,\ \Delta_P \in \mathbb{Q}^{>0}$. Let $\Delta_S = \lceil \Delta_T + \Delta_P \rceil_{\Delta_P}$. The $(\Delta_T, \Delta_D, \Delta_P)$ real-time task semantics of A, noted $[\![A]\!]^{\mathsf{RT}}_{\Delta_T,\Delta_D,\Delta_P}$ is the structured timed transition system $\mathcal{T} = \langle S, \iota, \Sigma^{\mathsf{in}}, \Sigma^{\mathsf{out}}, \Sigma^{\tau}, \rightarrow \rangle$ where:

($RT1$) 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 $f \in \{\bot, \top\}$;

($RT2$) $\iota = (l_0, r, 0, I, 0, 0, \bot)$ where r is such that for any $x \in \mathsf{Var}$, $r(x) = 0$, I is such that for any $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, $I(\sigma) = \bot$;

($RT3$) $\Sigma^{\mathsf{in}} = \mathsf{Lab}^{\mathsf{in}}$, $\Sigma^{\mathsf{out}} = \mathsf{Lab}^{\mathsf{out}}$, $\Sigma^{\tau} = \mathsf{Lab}^{\tau} \cup \overline{\mathsf{Lab}^{\mathsf{in}}} \cup \{\epsilon\}$;

($RT4$) the transition relation $\rightarrow$ is defined as follows:

– for the discrete transitions:

($RT4.1$) let $\sigma \in \mathsf{Lab}^{\mathsf{out}}$. $((l, r, T, I, u, d, \bot), \sigma, (l', r', T, I, u, 0, \top)) \in \rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$ such that $\lfloor T \rfloor_{\Delta_P} - r \models_{\Delta_S} g_{\Delta_S}$ and $r' = r[R := \lfloor T \rfloor_{\Delta_P}]$.

($RT4.2$) let $\sigma \in \mathsf{Lab}^{\mathsf{in}}$. $((l, r, T, I, u, d, f), \sigma, (l, r, T, I', u, d, f)) \in \rightarrow$ iff

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

($RT4.3$) let $\bar{\sigma} \in \overline{\mathsf{Lab}^{\mathsf{in}}}$. $((l, r, T, I, u, d, \bot), \bar{\sigma}, (l', r', T, I', u, 0, \top)) \in \rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$ 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 $I' = I[\sigma := \bot]$;

($RT4.4$) let $\sigma \in \mathsf{Lab}^{\tau}$. $((l, r, T, I, u, d, \bot), \sigma, (l', r', T, I, u, 0, \top)) \in \rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$ such that $\lfloor T \rfloor_{\Delta_P} - r \models_{\Delta_S} g_{\Delta_S}$ and $r' = r[R := \lfloor T \rfloor_{\Delta_P}]$.

($RT4.5$) $((l, r, T, I, u, d, \bot), \epsilon, (l, r, T, I, u, d, \top)) \in \rightarrow$ iff the two following conditions hold:

- for any $\bar{\sigma} \in \overline{\mathsf{Lab}^{\mathsf{in}}}$, for any $(l, l', g, \sigma, R) \in \mathsf{Edg}$, 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 $(l, l', g, \sigma, R) \in \mathsf{Edg}$, we have that $\lfloor T \rfloor_{\Delta_P} - r \not\models_{\Delta_S} g_{\Delta_S}$

($RT4.6$) $((l, r, T, I, u, d, \top), \epsilon, (l, r, T + u, I, 0, d, \bot)) \in \rightarrow$ iff $u = \Delta_T$.

– for the continuous transitions:

($RT4.7$) $((l, r, T, I, u, d, \bot), t, (l, r, T, I+t, u+t, d+t, \bot)) \in \rightarrow$ iff $u+t \leq \Delta_D$.

($RT4.8$) $((l, r, T, I, u, d, \top), t, (l, r, T, I+t, u+t, d+t, \top)) \in \rightarrow$ iff $u+t \leq \Delta_T$.

□

Quite not surprisingly, this definition is very similar to the Program Semantics definition. The few differences are detailed in turn.

Comments on the Real-Time Semantics definition

- The worst-case execution time of the execution round Δ_L has been replaced with two distinct parameters Δ_T (the period of the real-time task) and Δ_D (its deadline).
- In this semantics, the flag f indicates whether the task is currently idle ($\top$) or not idle ($\bot$). More precisely, $\top$ indicates that the task is waiting to be rescheduled at the beginning of the next period. When $f = \bot$ the task is *active*, but this not necessarily means that it has the CPU; its scheduling state can be either of *ready to run*, *preempted*, or *running*.
- Rule ($RT4.7$) says that the task can let time elapse while not finished, until the deadline is reached.
- Rules ($RT4.8$) and ($RT4.9$) make the task idle until the next period begins.

Real-Time semantics and periodic tasks

The purpose of the Real-Time Semantics is to mimic the behavior of *schedulable* hard-real time periodic task. We justify this by analyzing the semantics rules :

- **Periodicity.** Rule ($RT4.6$) ensures the exact periodicity of the real-time task.
- **Schedulability.** The RT semantics behaves as a periodic task which never misses a deadline. This is enforced by rule ($RT4.7$); the flag $\bot$ indicates that the task has not reached completion yet, which can never be the case when $u > \Delta_D$
- **Preemptability.** As said earlier, the Real-Time Semantics allows the task to be preempted. This is not explicitly indicated in the semantics, because the only thing it cares about is that the deadline is reached. When running other higher-priority tasks, the only thing we have to make sure is that this periodic task remains schedulable.

4.2.3 AASAP Simulability of the Real-Time Semantics

The real time semantics is simulable by the Almost ASAP semantics, provided that their respective parameters satisfy the following inequality :

$$\Delta > \Delta_T + 2\Delta_D + 4\Delta_P$$

This is formalized by the following theorem :

Theorem 4 (Simulation) Let A be an ELASTIC controller. For any rationals $\Delta, \Delta_T, \Delta_D, \Delta_P \in \mathbb{Q}^{>0}$ such that $\Delta_T + 2\Delta_D + 4\Delta_P < \Delta$ we have $[\![A]\!]^{\mathsf{RT}}_{\Delta_T,\Delta_D,\Delta_P} \sqsubseteq [\![A]\!]^{\mathsf{AAsap}}_{\Delta}$.

Notation Reminder

$$(l_1, r_1, T_1, I_1, u_1, d_1, f_1) \in S1$$

- $l_1 \in \mathsf{Loc}$ is the current location.
- $r_1 : \mathsf{Var} \to \mathbb{R}^{\geq 0}$ contains the *discrete* time at which each clock has been last reset.
- $T_1 \in \mathbb{R}^{\geq 0}$ contains the *exact* time at which the current period has started.
- $I_1 : \mathsf{Lab}^{\mathsf{in}} \to \mathbb{R}^{\geq 0} \cup \{\bot\}$ is a function used to remember the time elapsed since the oldest untreated occurrence of each input symbol. $I_1(\sigma) = \bot$ means that there is no untreated occurrence of σ.
- $u_1 \in \mathbb{R}^{\geq 0}$ holds the *exact* time elapsed since T_1 (The exact current is thus always $T_1 + u_1$).
- $d_1 \in \mathbb{R}^{\geq 0}$ contains the *exact* time elapsed since the last discrete jump occurred.
- $f_1 = \bot$ means that the task is currently active, and when $f_1 = \top$ the task is idle.

$$(l_2, v_2, I_2, d_2) \in S2$$

- l_2, I_2, and d_2 have a meaning that is identical to l_1, I_1, and d_1, respectively.
- $v_2 : \mathsf{Var} \to \mathbb{R}^{\geq 0}$ contains the *exact* time value of each clock.

Real-Time semantics simulation proof

Proof. Let $[\![A]\!]^{\mathsf{RT}}_{\Delta_T,\Delta_D,\Delta_P} = (S_1, \iota_1, \Sigma_1^{\mathsf{in}}, \Sigma_1^{\mathsf{out}}, \Sigma_1^{\tau}, \rightarrow_1)$ and $[\![A]\!]^{\mathsf{AAsap}}_{\Delta} = (S_2, \iota_2, \Sigma_2^{\mathsf{in}}, \Sigma_2^{\mathsf{out}}, \Sigma_2^{\tau}, \rightarrow_2)$.
Consider the relation $R \subseteq S_1 \times S_2$ that contains all the pairs:

$$(s_1, s_2) = ((l_1, r_1, T_1, I_1, u_1, d_1, f_1), (l_2, v_2, I_2, d_2))$$

such that the following conditions hold:

$(R1)$ $l_1 = l_2$;

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

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

$(R4)$ $d_1 = d_2$;

$(R5)$ there exists $(l_2'', v_2'', I_2'', d_2'')$ such that:

- if $f_1 = \bot$ then $((l_2, v_2, I_2, d_2), \Delta_D - u_1, (l_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.
- if $f_1 = \top$ then $((l_2, v_2, I_2, d_2), \Delta_T - u_1, (l_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.

Let us show that R is a simulation relation.

1. $(\iota_1, \iota_2) \in R$. We have to check the 5 rules of the simulation relation.

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

   $(R5)$ To establish this property, we first note that $d_2 = 0$ and so $d_2 + \Delta_D < \Delta$ which implies $\forall t' \leq \Delta_D : d_2 + t' < \Delta$. Hence the two conditions of rule $(A4.6)$ are verified.

2. Let us assume that $(s_1, s_2) = ((l_1, r_1, T_1, I_1, u_1, d_1, f_1), (l_2, v_2, I_2, d_2)) \in R$ and that $(s_1, \sigma, s_1') \in \rightarrow_1$ (with $s_1' = (l_1', r_1', T_1', I_1', u_1', d_1', f_1')$). 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:

   $(H1)$ $s_2 = (l_1, v_2, I_1, d_1)$

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

   $(H3)$ there exists $s_2'' = (l_2'', v_2'', I_2'', d_2'') \in S_2$ such that:
   - either $f_1 = \bot$ and $((l_2, v_2, I_2, d_2), \Delta_D - u_1, (l_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.
   - or $f_1 = \top$ and $((l_2, v_2, I_2, d_2), \Delta_T - u_1, (l_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.

   Note : $(H1)$ is obtained by $(R1)$, $(R3)$, $(R4)$; $(H2)$ by $(R2)$, and $(H3)$ by $(R5)$.

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

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

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

$$s_1' = \begin{cases} (l_1, r_1, T_1, I_1[\{\sigma\} := 0], u_1, d_1, f_1) & \text{if} \quad I_1(\sigma) = \bot \\ (l_1, r_1, T_1, I_1, u_1, d_1, f_1) & \text{if} \quad I_1(\sigma) \neq \bot \end{cases}$$

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:

$$s_2' = \begin{cases} (l_1, v_2, I_1[\{\sigma\} := 0], d_1) & \text{if} \quad I_1(\sigma) = \bot \\ (l_1, v_2, I_1, d_1) & \text{if} \quad I_1(\sigma) \neq \bot \end{cases}$$

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 by $RT4.1$ that:

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

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 real time 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 $\forall x \in \mathsf{Var}$:

$$\begin{array}{lll}
 & a_x - \Delta_S \leq \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) \leq b_x + \Delta_S & (1) \\
\rightarrow & a_x - \Delta_S - \Delta_P \leq T_1 - r_1(x) \leq b_x + \Delta_S + \Delta_P & (2) \\
\rightarrow & a_x - \lceil \Delta_T + \Delta_P \rceil_{\Delta_P} - \Delta_P \leq T_1 - r_1(x) \leq b_x + \lceil \Delta_T + \Delta_P \rceil_{\Delta_P} + \Delta_P & (3) \\
\rightarrow & a_x - \Delta_T - 3\Delta_P \leq T_1 - r_1(x) \leq b_x + \Delta_T + 3\Delta_P & (4) \\
\rightarrow & a_x - \Delta_T - \Delta_D - 4\Delta_P + u_1 \leq T_1 - r_1(x) - \Delta_D - \Delta_P + u_1 \wedge & \\
 & T_1 - r_1(x) + \Delta_D + \Delta_P + u_1 \leq b_x + \Delta_T + \Delta_D + 4\Delta_P + u_1 & \\
\rightarrow & a_x - \Delta_T - \Delta_D - 4\Delta_P + u_1 \leq v_2(x) \leq b_x + \Delta_T + \Delta_D + 4\Delta_P + u_1 & (5) \\
\rightarrow & a_x - \Delta_T - \Delta_D - 4\Delta_P \leq v_2(x) \leq b_x + \Delta_T + 2\Delta_D + 4\Delta_P & (6) \\
\rightarrow & a_x - \Delta \leq v_2(x) \leq b_x + \Delta & (7) \\
\rightarrow & v_2(x) \models_{\Delta} g_{\Delta} &
\end{array}$$

(1) Hypothesis $(J1)$
(2) Lemma 1
(3) $\Delta_S = \lceil \Delta_T + \Delta_P \rceil_{\Delta_P}$
(4) Lemma 1
(5) Hypothesis $(H2)$
(6) $f_1 = \bot \rightarrow 0 \leq u_1 \leq \Delta_D$
(7) $\Delta > \Delta_T + 2\Delta_D + 4\Delta_P$

Now that it is established that $\exists s_2' \in S_2 : (s_1', \sigma, s_2') \in \rightarrow_2$, we know that $s_2' = (l_1', v_2[R := 0], I_1, 0)$.

It remains to prove that $(s_1', s_2') \in R$ which means we must check the five rules of the simulation relation. $(R1)$,$(R3)$,$(R4)$ are clearly true. $(R5)$ is seen easily using $(A4.6)$ and the fact that $u_2 = 0$.

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

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

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

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

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 $f_1 = \bot$ and thus $u_1 - \Delta_D \leq 0$. This is a consequence of Lemma 1. This establishes $(R2)$.

case (c) let $\sigma \in \Sigma^\tau$. $\Sigma^\tau = \mathsf{Lab}^\tau \cup \overline{\mathsf{Lab}^{\mathsf{in}}} \cup \{\epsilon\}$. The proof for the first two sets is similar to the previous case. Let $\sigma = \epsilon$. Depending on the value of the flag f_1, the transition taken will be different (see $(RT4.5)$ and $(RT4.6)$). We treat these two cases in turn with $f_1 = \bot$ in (c.1) and $f_1 = \top$ in (c.2).

(c.1) Since $(s_1, \epsilon, s_1') \in \rightarrow_1$ and $f_1 = \bot$ we know by $(RT4.5)$ that

$(K1)$ $s_1' = (l_1, r_1, T_1, I_1, u_1, d_1, \top)$

$(K2)$
- for any $\bar{\sigma}$ such that $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, for any $(l_1, l', g, \sigma, R) \in \mathsf{Edg}$, 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 $\sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^\tau$, for any $(l_1, l', g, \sigma, R) \in \mathsf{Edg}$, 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, ϵ, s_2') and

$(K3)$ $s_2' = s_2 = (l_1, v_2, I_1, d_1)$.

Now we have to prove that $(s_1', s_2') \in R$. $(R1), (R2), (R3)$ and $(R4)$ are clearly true. Let us now prove $(R5)$. In this case, we must show that there exists s_2'' s.t. $((l_1, v_2, I_1, d_1), \Delta_T - u_1, s_2'') \in \rightarrow_2$. According to rule $(A4.6)$, it amounts to prove that

$(L1)$ for any edge $(l_1, l', g, \sigma, R) \in \mathsf{Edg}$ with $\sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^\tau$, we have that:

$$\forall t' : 0 \leq t' \leq \Delta_T - u_1 : (d_1 + t' \leq \Delta \vee \mathsf{TS}(v_2 + t', g) \leq \Delta)$$

$(L2)$ for any edge $(l_1, l', g, \sigma, R) \in \mathsf{Edg}$ with $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, we have that:

$$\forall t' : 0 \leq t' \leq \Delta_T - u_1 : (d_1 + t' \leq \Delta \vee \mathsf{TS}(v_2 + t', g) \leq \Delta \vee (I_1 + t')(\sigma) \leq \Delta)$$

We first make a proof for labels of $(L1)$.

$\forall (l_1, l', g, \sigma, R) \in \mathsf{Edg}$ 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

$$\begin{aligned}
& \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) < a_x - \Delta_S \\
\rightarrow\ & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) < a_x - \lceil \Delta_T + \Delta_P \rceil_{\Delta_P} && (1) \\
\rightarrow\ & T_1 - r_1(x) < a_x - \Delta_T && (2) \\
\rightarrow\ & T_1 - r_1(x) + u_1 + \Delta_D + \Delta_P < a_x - \Delta_T + u_1 + \Delta_D + \Delta_P \\
\rightarrow\ & v_2(x) < a_x - \Delta_T + u_1 + \Delta_D + \Delta_P && (3) \\
\rightarrow\ & \forall t' : 0 \leq t' \leq \Delta_T - u_1 : v_2(x) + t' \leq a_x + \Delta_D + \Delta_P \\
\rightarrow\ & \forall t' : 0 \leq t' \leq \Delta_T - u_1 : \mathsf{TS}(v_2(x) + t', g(x)) \leq \Delta_D + \Delta_P \\
\rightarrow\ & \forall t' : 0 \leq t' \leq \Delta_T - u_1 : \mathsf{TS}(v_2(x) + t', g(x)) \leq \Delta && (4) \\
\rightarrow\ & \forall t' : 0 \leq t' \leq \Delta_T - u_1 : \mathsf{TS}(v_2 + t', g) \leq \Delta
\end{aligned}$$

(1) $\Delta_S = \lceil \Delta_T + \Delta_P \rceil_{\Delta_P}$ (3) Hypothesis $(H2)$
(2) Lemma 1 (4) $\Delta > \Delta_T + 2\Delta_D + \Delta_P$

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

$$
\begin{array}{ll}
 & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) > b_x + \Delta_S \\
\rightarrow & \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) > b_x + \lceil \Delta_T + \Delta_P \rceil_{\Delta_P} \quad (1) \\
\rightarrow & T_1 - r_1(x) > b_x + \Delta_T + \Delta_P \quad (2) \\
\rightarrow & T_1 - r_1(x) + u_1 - \Delta_D - \Delta_P > b_x + \Delta_T + u_1 - \Delta_D \\
\rightarrow & v_2(x) > b_x + \Delta_T + u_1 - \Delta_D \quad (3) \\
\rightarrow & v_2(x) > b_x \quad (4) \\
\rightarrow & \forall t' : 0 \leq t' \leq \Delta_T - u_1 : v_2(x) + t' > b_x \\
\rightarrow & \forall t' : 0 \leq t' \leq \Delta_T - u_1 : \mathsf{TS}(v_2(x) + t', g(x)) \leq \Delta \quad (5) \\
\rightarrow & \forall t' : 0 \leq t' \leq \Delta_T - u_1 : \mathsf{TS}(v_2 + t', g) \leq \Delta
\end{array}
$$

(1) $\Delta_S = \lceil \Delta_T + \Delta_P \rceil_{\Delta_P}$ (4) $\Delta_D \leq \Delta_T \wedge u_1 \geq 0$
(2) Lemma 1 (5) $v_2(x) + t' \not\models g(x)$
(3) Hypothesis $(H2)$

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

$$
\begin{array}{ll}
 & I_1(\sigma) < u_1 \\
\rightarrow & \forall t' : 0 \leq t' \leq \Delta_T - u_1 : I_1(\sigma) + t' < \Delta_T \\
\rightarrow & \forall t' : 0 \leq t' \leq \Delta_T - u_1 : I_1(\sigma) + t' < \Delta \quad (\Delta_T < \Delta)
\end{array}
$$

(c.2) Since $(s_1, \epsilon, s_1') \in \rightarrow_1$ and $f_1 = \top$ we know by $(RT4.6)$ that

$(M1)$ $s_1' = (l_1, r_1, T_1 + \Delta_T, I_1, 0, d_1, \bot)$

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

$(M2)$ $s_2' = s_2 = (l_1, v_2, I_1, d_1)$.

Now we have to prove that $(s_1', s_2') \in R$. $(R1), (R2), (R3)$ and $(R4)$ are clearly true. Let us now prove $(R5)$. In this case we have $u_1 = 0$, so we must show that there exists s_2'' s.t. $((l_1, v_2, I_1, d_1), \Delta_D, s_2'') \in \rightarrow_2$. According to rule $(A4.6)$, it amounts to prove that

$(N1)$ for any edge $(l_1, l', g, \sigma, R) \in \mathsf{Edg}$ with $\sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^{\tau}$, we have that:

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

$(N2)$ for any edge $(l_1, l', g, \sigma, R) \in \mathsf{Edg}$ with $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, we have that:

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

First, if a location change occurred during the last period, then $(N1)$ and $(N2)$ are clearly true. Indeed

$d_1 < \Delta_T$
$\rightarrow \quad \forall t' : 0 \leq t' \leq \Delta_D : d_1 + t' < \Delta_T + \Delta_D$
$\rightarrow \quad \forall t' : 0 \leq t' \leq \Delta_D : d_1 + t' < \Delta \qquad (\Delta > \Delta_T + 2\Delta_D + 4\Delta_P)$

Now we have yet to prove that $(N1)$ and $(N2)$ are true in the case where there was no location change during the last period. We proceed in a similar way than in case (c.1). If there was no location change during the last period, we now that

$(M3)$
- for any $\bar{\sigma}$ such that $\sigma \in \mathsf{Lab}^{\mathsf{in}}$, for any $(l_1, l', g, \sigma, R) \in \mathsf{Edg}$, we have that either $\lfloor T_1 \rfloor_{\Delta_P} - r_1 \not\models {}_{\Delta_S} g_{\Delta_S}$ or $I_1(\sigma) \leq \Delta_T$
- for any $\sigma \in \mathsf{Lab}^{\mathsf{out}} \cup \mathsf{Lab}^{\tau}$, for any $(l_1, l', g, \sigma, R) \in \mathsf{Edg}$, we have that $\lfloor T_1 \rfloor_{\Delta_P} - r_1 \not\models {}_{\Delta_S} g_{\Delta_S}$

We first make a proof for labels of $(N1)$.

$\forall (l_1, l', g, \sigma, R) \in \mathsf{Edg}$ with $\sigma \in \mathsf{Lab}^{\mathsf{out}}$ we have $\lfloor T_1 \rfloor_{\Delta_P} \not\models {}_{\Delta_S} g_{\Delta_S}$ by $(M3)$. 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

$\lfloor T_1 \rfloor_{\Delta_P} - r_1(x) < a_x - \Delta_S$
$\rightarrow \quad \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) < a_x - \lceil \Delta_T + \Delta_P \rceil_{\Delta_P}$ (1)
$\rightarrow \quad T_1 - r_1(x) < a_x - \Delta_T$ (2)
$\rightarrow \quad T_1 - r_1(x) + u_1 + \Delta_D + \Delta_P < a_x - \Delta_T + u_1 + \Delta_D + \Delta_P$
$\rightarrow \quad v_2(x) < a_x - \Delta_T + u_1 + \Delta_D + \Delta_P$ (3)
$\rightarrow \quad v_2(x) < a_x + \Delta_D + \Delta_P$ (4)
$\rightarrow \quad \forall t' : 0 \leq t' \leq \Delta_D : v_2(x) + t' \leq a_x + 2\Delta_D + \Delta_P$
$\rightarrow \quad \forall t' : 0 \leq t' \leq \Delta_D : \mathsf{TS}(v_2(x) + t', g(x)) \leq 2\Delta_D + \Delta_P$
$\rightarrow \quad \forall t' : 0 \leq t' \leq \Delta_D : \mathsf{TS}(v_2(x) + t', g(x)) \leq \Delta$ (5)
$\rightarrow \quad \forall t' : 0 \leq t' \leq \Delta_D : \mathsf{TS}(v_2 + t', g) \leq \Delta$

(1) $\Delta_S = \lceil \Delta_T + \Delta_P \rceil_{\Delta_P}$ (4) $u_1 = \Delta_T$
(2) Lemma 1 (5) $\Delta > \Delta_T + 2\Delta_D + 4\Delta_P$
(3) Hypothesis $(H2)$

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

$\lfloor T_1 \rfloor_{\Delta_P} - r_1(x) > b_x + \Delta_S$
$\rightarrow \quad \lfloor T_1 \rfloor_{\Delta_P} - r_1(x) > b_x + \lceil \Delta_T + \Delta_P \rceil_{\Delta_P}$ (1)
$\rightarrow \quad T_1 - r_1(x) > b_x + \Delta_T + \Delta_P$ (2)
$\rightarrow \quad T_1 - r_1(x) + u_1 - \Delta_D - \Delta_P > b_x + \Delta_T + u_1 - \Delta_D$
$\rightarrow \quad v_2(x) > b_x + \Delta_T + u_1 - \Delta_D$ (3)
$\rightarrow \quad v_2(x) > b_x + 2\Delta_T - \Delta_D$ (4)
$\rightarrow \quad v_2(x) > b_x$ (5)
$\rightarrow \quad \forall t' : 0 \leq t' \leq \Delta_D : v_2(x) + t' > b_x$
$\rightarrow \quad \forall t' : 0 \leq t' \leq \Delta_D : \mathsf{TS}(v_2(x) + t', g(x)) \leq \Delta$ (6)
$\rightarrow \quad \forall t' : 0 \leq t' \leq \Delta_D : \mathsf{TS}(v_2 + t', g) \leq \Delta$

(1) $\Delta_S = \lceil \Delta_T + \Delta_P \rceil_{\Delta_P}$ (4) $u_1 = \Delta_T$
(2) Lemma 1 (5) $\Delta > \Delta_T + 2\Delta_D + 4\Delta_P$
(3) Hypothesis $(H2)$ (6) $v_2(x) + t' \not\models g(x)$

Thus, both cases imply that $(N1)$ is true.

The proof for $(N2)$ is the same if we have, by $(M3)$, $\lfloor T_1 \rfloor_{\Delta_P} \not\models {}_{\Delta_S}g_{\Delta_S}$. If not, we have $I_1(\sigma) < \Delta_T$ which also proves $(N2)$. Indeed

$I_1(\sigma) < \Delta_T$
$\rightarrow$ $\forall t' : 0 \leq t' \leq \Delta_D : I_1(\sigma) + t' < \Delta_T + \Delta_D$
$\rightarrow$ $\forall t' : 0 \leq t' \leq \Delta_D : I_1(\sigma) + t' < \Delta$ $(\Delta > \Delta_T + 2\Delta_D + 4\Delta_P)$

case (d) let $\sigma \in \mathbb{R}^{\geq 0}$. For the sake of clarity let us consider that $\sigma = t$. Again, we need to consider two different cases depending on the value of f_1. We treat these two cases in turn with $f_1 = \bot$ in (d.1) and $f_1 = \top$ in (d.2).

(d.1) Since $f_1 = \bot$ and $(s_1, t, s_1') \in \rightarrow_1$ we know by $(RT4.7)$ that

$(P1)$ $s_1' = (l_1, r_1, T_1, I_1 + t, u_1 + t, d_1 + t, \bot)$;

$(P2)$ $u_1 + t \leq \Delta_D$.

With those facts, we know that there exists $s_2' = (l_2', v_2', I_2', d_2') \in S_2$ such that $(s_2, t, s_2') \in \rightarrow_2$ because $(s_2, \Delta_D - u_1, s_2'') \in \rightarrow_2$ by $(H3)$ and $t \leq \Delta_D - u_1$ by $(P2)$.

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

$(P3)$ $s_2' = (l_1, v_2 + t, I_1 + t, d_1 + t)$

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 $(P1)$ to $(P3)$.

For $(R5)$, since by $(H3)$, there exists $s_2'' = (l_2'', v_2'', I_2'', d_2'') \in S_2$ such that $((l_2, v_2, I_2, d_2), \Delta_D - u_1, (l_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$, we have $((l_1, v_2 + t, I_1 + t, d_1 + t), (\Delta_D - u_1 - t), (l_2'', v_2'', I_2'', d_2'')) \in \rightarrow_2$.

(d.2) This case is proven exactly the same way as for (c.1), just replace Δ_D with Δ_T, and use $(RT4.7)$ and the second item of $(H3)$.

■

4.2.4 Implementability of the Real-Time Semantics

We have stated in the beginning of this chapter that an appropriate implementation semantics should contain "enough information" about how the control software must function. This is arguably the case for the Real-Time Semantics : the controller is a periodic real-time task, executing the same sequence of steps

at every period. Moreover, this sequence of steps is detailed thoroughly, it is thus possible to create a code generator which will create the control software automatically. Such a code generator has been constructed and is discussed in detail in chapter 6.

4.2.5 Benefits of the Real-Time Semantics

It should be noted that one does not *need* to use the Real-Time Semantics to create provably correct real-time software in the form of a periodic task. The Program Semantics can also be used for that purpose. Indeed, the latter semantics requires that every execution round be finished within Δ_L time units. Hence, we can implement our controller with a real-time task of period Δ_L, and if it can be proven that the WCRT[3] of that task is smaller than Δ_L, then that implementation will be provably correct provided that $\Delta > 3\Delta_L + 4\Delta_P$.

The benefit of the Real-Time Semantics is that it takes advantage of the knowledge that the WCRT of the task is *smaller* than Δ_L. The Program Semantics requires it, but does nothing with that information. This is why the Real-Time Semantics splits the Δ_L parameter into two distinct parameters Δ_T and Δ_D; the value of Δ_D is used to provide a tighter simulation requirement : $\Delta > \Delta_T + 2\Delta_D + 4\Delta_P$. As Δ_D and Δ_P will be of an order of magnitude smaller than Δ_T in many applications, using the Real-Time Semantics provides a gain of nearly 3 units on the simulation bound.

Observe that when using periodic real-time tasks instead of a program running an infinite loop, we need to measure the task's WCRT, instead of the loop WCET[4]. The worst-case reaction time of a periodic task measures the true worst-case time amount that it needs to reach completion from the moment it has been ready to run. The WCRT may be slightly larger than the WCET because it includes the time spent serving interrupts, the time spent in the scheduler and the time spent serving higher priority tasks.

[3]Worst-Case Reaction Time
[4]Worst-Case Execution Time

Chapter 5

RTAI : GNU/Linux Made RT Capable

In this chapter, we describe RTAI, a modern hard real-time operating system. We start by a brief overview of its architecture, and then detail a small subset of RTAI's API which will be needed for the following chapter.

5.1 Architecture of RTAI

RTAI is not exactly a real-time operating system in itself. Rather, it is an extension of the general-purpose Linux OS, designed to make it hard real-time capable. Because it not a self-contained RTOS, RTAI has a quite exotic architecture. It is essentially composed of three parts :

- A hardware abstraction layer (HAL)
- A patch for the Linux kernel
- A set of real-time services. These include : a real-time scheduler, interprocess communication primitives, real-time threads management primitives, etc.

The purpose of the kernel patch and the HAL is to prevent Linux from handling hardware interrupts directly. To achieve this, a patch for the Linux kernel replaces every interrupt-related instruction contained in the kernel with a macro which transfers the request to the hardware abstraction layer. This in effect *isolates* the kernel from the hardware in a transparent manner. The trick is that the HAL not only listens to the requests from Linux, but also from the RTAI core, to which it will give a higher priority. When an interrupt arrives, it is first treated by an RTAI handler. It is only after this first handling phase that the interrupt is forwarded to Linux.

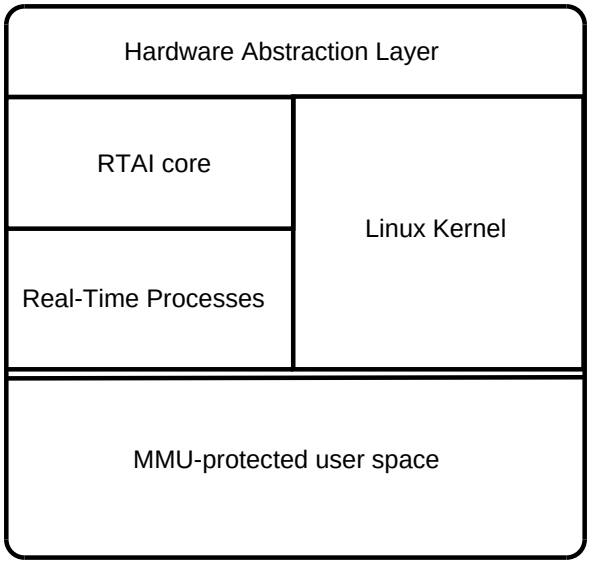

Figure 5.1: This figure represents the components of the RTAI/Linux RTOS. Everything above the double line lies in kernel space.

This design makes the RTAI core in total control of the hardware, which enables it to run hard real-time tasks without any disruption. The RTAI scheduler runs LINUX as its *idle loop*, which makes LINUX a very convenient background server, able to execute every non real-time task that the system might need. It is also from LINUX that the user / administrator interacts with RTAI. A high-level representation of a RTAI / LINUX architecture is depicted at figure 5.1.

RTAI is able to handle real-time processes running either in kernel-space or in user-space. However, it is a bit more of a hassle to use the latter so we will consider that our tasks reside in kernel space. In RTAI, such a process is created by the means of LINUX *kernel modules*. One module represents one RTAI process, and can be started and stopped at will using the commands **insmod** and **rmmod**. Within such a module, the RTAI service primitives are available, provided that its core modules have been loaded.

5.2 RTAI Service Routines

The API of RTAI contains literally hundreds of service routines of all kinds, so we will only detail the ones we will need for this work.

5.2.1 Timer Routines

Probably the most important feature of an RTOS, the time-related primitives must be both reliable and straightforward. We describe briefly how RTAI deals

with time readings.

One-shot and periodic modes

RTAI provides two different ways to get real-time class clock-readings : the *one-shot* and *periodic* modes. On Intel-based machines (we will not consider other hardware platforms in this work but it should only differ slightly), both modes interact with the 8254 Programmable Interval Timer (PIT) chip. This hardware counter can be programmed to :

- Trigger an interrupt after a certain amount of time has passed (this is used by the one-shot mode).
- Trigger an interrupt at fixed periodic intervals (used by the periodic mode).

In the one-shot mode, the 8254 timer needs to be reprogrammed after each interrupt. This is very flexible (a different delay can be specified each time), but it generates a lot of overhead on low-end machines (reprogramming the PIT takes up a few CPU cycles). The periodic mode, however less flexible, has the advantage of only needing to program the PIT once, and is hence less subject to jitter. We will only use the periodic mode in the following, which is done in RTAI by calling the function `rt_set_periodic_mode()`.

Clock-resolution setting

In the periodic mode, it is the interval between two clock interrupts which defines the resolution of RTAI's clocks. This is very convenient, because the 8254 PIT accepts a wide range of frequencies : from 1193180 Hz to about 18 Hz; we will thus bee able to *choose* the appropriate clock resolution. The PIT is programmed by writing a number of *ticks* in a 16-bit register. There are exactly 1193180 ticks in one second, thus writing the value 1 in the register makes the PIT trigger interrupts at a frequency of a little more than a megahertz. This is obviously a *very* bad idea, because it is probably more interrupts than the CPU can handle and would ultimately lead to a crash[1]. The clock-resolution must hence be chosen with care, by finding the best trade-off between interrupt-handling overhead and clock-precision.

To program the PIT in RTAI, simply call the function `start_rt_timer(ticks)`. This call must be made *after* calling `rt_set_periodic_mode()`. Once the PIT starts triggering interrupts, the clocks of RTAI are updated automatically, and we can start using them. When the timer is not needed anymore, it is advised to call `stop_rt_timer()`, which reverts the PIT to its default settings.

[1]This was tried by the author on a 500 MHz Pentium machine, resulting as expected in a complete freeze.

Reading the current time

Once the timer mode and clock-resolution have been set, we can start measuring time by calling the primitive `RTIME rt_get_time()`. On all platforms, this call always returns a 64-bit unsigned integer.

5.2.2 Task-Management Routines

The following describes how real-time tasks are created and managed in RTAI.

Creating a new task

RTAI holds all information about a task into a `RT_TASK` structure. A new task is created using the `rt_task_init` function, which has the following signature :

```
int rt_task_init (RT_TASK *task,
                  void (*thread_main)(int),
                  int data,
                  int stack_size,
                  int priority,
                  int uses_fpu,
                  void(*signal)(void));
```

Calling this function initializes the `task` structure and creates a new task with the indicated parameters. The function pointer `thread_main` is the entry point of the task, and `data` is an integer that will be given as parameter to `thread_main`. The function `signal` will be called each time the scheduler sets the task back in the running state, which enables the task to be aware that it has been preempted (this parameter can be safely set to 0 if this feature is not desired). The `priority` parameter must be an integer between 0 (highest priority) and 255 (lowest priority). The RTAI scheduler does not require that all tasks have different priorities.

Making a task periodic

If we wish to create a periodic task, a subsequent call to the `rt_task_make_periodic` function must be made. Its signature is very straightforward :

```
int rt_task_make_periodic (RT_TASK *task,
                           RTIME offset,
                           RTIME period);
```

If `task` has been correctly initialized by `rt_task_init`, this call will make the task periodic, with the specified `offset` and `period` values. Both `offset` and `period` are expressed in units defined by the clock resolution (set by `start_rt_timer`).

When the task is scheduled for the first time, its entry-point function is called. When the task returns from that function, it is stopped and will not be rescheduled. When a task is preempted, its context is saved, and will be restored when rescheduled. A periodic task does not need to wait for preemption however, it can explicitly yield its current period by calling the `rt_task_wait_period()` function, which takes no parameters. The structure of a periodic task entry-point function in RTAI would typically resemble the following :

```
void task_main (int data)
{
  int running = 1;
  while (running)
  {
      perform_important_things ();
      if (are_we_done_yet ())
         running = 0;
      else
         rt_task_wait_period ();
  }
}
```

Deleting a task

Deleting unused tasks is needed so that the real-time scheduler does not fill up with zombies[2]. This is done by simply calling `rt_task_delete`.

5.3 The RTAI Scheduler

To conclude this short presentation of RTAI, we briefly describe the RTAI scheduler that will be used in the next chapter.

RTAI natively supports the use of 3 built-in schedulers : namely UP, SMP and MUP[3]. Only the first one is meant to be used on uni-processors machines (we will not consider other cases in the following), so we shall use the UP scheduler.

As a standard fixed-priority scheduler, UP will always favor the pending task with highest priority (again, there are 256 unique priorities in RTAI, 0 being the highest). The UP scheduler manages tasks with equal priorities by applying a round-robin strategy. Priority assignments should hence be made using the *Rate Monotonic*, or *Deadline Monotonic* algorithms.

[2]In UNIX terminology, a *zombie* process is a process which has completed its execution but still resides in the process table. While in this state, it is possible to investigate *why* the process was terminated, and retrieve other useful information.

[3]UP, SMP and MUP respectively stand for "uni-processor", "symmetric multi-processor", and "multi uni-processor".

In addition to the schedulers packaged with the RTAI distribution, RTAI also easily supports custom-made schedulers. As every service in RTAI is materialized by kernel modules, using a custom scheduler amounts to simply load a different module. This can be used for instance to implement a dynamic priority scheduler with an *Earliest Deadline First* strategy.

Whichever scheduling strategy is used, this choice is orthogonal to our concerns. Remember that regarding scheduling, the only thing the Real-Time Semantics cares about is never to miss a deadline. How exactly this is guaranteed is beyond the scope of this work.

Chapter 6

Code Generation of Real-Time Controllers

In this chapter we present SPECTRE[1], a tool for generating real-time code from timed-automata. SPECTRE currently generates code for RTAI but should be easily adapted to work with any other RTOS. The following is divided in two main parts : first we describe the tool's input language and discuss the various design choices which had to be made; in the second part, we describe how SPECTRE generates code and discuss the to which extent the generated code can be considered provably correct.

6.1 Input Language

The SPECTRE input language is composed of two sections :

1. The *specification* section describes the model of the controller in the form of a timed automaton.

2. The *decoration* section holds the controller's functionality code. This section maps the timed automaton's abstract behavior to concrete side-effects.

6.1.1 Specification Section

The syntax of this section is quite classic, as it is heavily inspired from the ELASTIC input language [DDR05a], which is very similar to HYTECH's.

A short example of a SPECTRE specification can be found at figure 6.1. This example really does nothing meaningful, it is only meant to present as many syntactic elements as possible in a relatively short space. Most of the syntax should be rather self-explanatory, so we only explain the essential here. More

[1] SPECTRE stands for Spécification de Contrôleurs Temps-Réels Embarqués.

```
specification example

clocks : x;
vars : a, b;

orders : o;
internals : i;
events : e;

initially A, {x := 0, a := 0, b := 0};

location A :
    {x = 2, b = 1}, e, {a := 7}, B;
    {3 <= x, x <= 4, b = 0}, e, {a := a + 4}, B;
    {3 <= x, x <= 4, b = 1}, e, {a := a + 2}, B;

location B :
    {a <= 10}, i, {a := a + 1}, B;
    {a >= 10}, o, {a := 0, x := 0, b := 1}, A;
    {a >= 10}, none, {a := 0, x := 0, b := 0}, A;

end
```

Figure 6.1: Example of a SPECTRE specification section.

information about the SPECTRE input language, along with a complete grammar, can be found in the appendix.

Comments on the specification syntax

The "initially" line indicates which location is the initial location of the automaton. The assignments on the right side of the line specify the initial values of the clocks and variables. There is a key difference here with HYTECH for example, which specifies the initial clock and variables valuations with *constraints* rather than assignments, and thus allows non-deterministic initial valuations. As SPECTRE will ultimately need to generate code, it needs to know the precise initial value of each clock and variable. This should not be a problem in practice, if a controller has been proven correct for a range of possible initial values, the designer can always choose precise values from that range.

Each "location" paragraph defines a location of the automaton and its outgoing edges, if any. The syntax for edges is composed of (in that order) : guard predicates, a label, updates, and the destination location. The guard predicates

are separated by commas, and the guard as a whole is the conjunction of the predicates it contains. Any empty guard is simply interpreted as *True*. The edge label can be either one of the previously declared events, orders or internals[2], or the keyword "none" which represents the silent label ϵ. Just like the initial valuations, the guards updates are expressed using a set of assignments. As expected, any unmentioned clock or variable is left untouched by the update.

A note about discrete variables

The perspicacious reader will have noted that, in the previous chapters, there were no mention of *discrete variables* in the syntax of timed automata. How do these fit with the definition of timed automata given previously ? The answer is simple : discrete variables are just a very convenient syntactic addition to standard timed automata, which allows a designer to represent a timed automaton in a more natural way. To translate a timed automaton with discrete variables into the "classical" syntax, just multiply every location by the number of possible valuations of the discrete variables. As a direct consequence, for the automaton to remain finite-state, there must be a finite number of possible valuations of the discrete variables. The usefulness of discrete variables in timed and hybrid automata is unquestioned in practice. The tools HyTech and Uppaal for example, both allow their use extensively. A simple example showing how to transform a timed automaton with variables into its variable-less equivalent is shown at figure 6.2.

6.1.2 Decoration Section

As stated previously, the decoration section of a Spectre input file contains all the code and information necessary to produce a fully-functional real-time controller, that will be provably correct w.r.t to the specification[3]. The decoration is composed of several different *items*, which can be found at most once in the decoration. We detail each item type in turn.

Global decoration

This item is optional, but it will most likely be present in any implementation. It enables the user to provide global declarations. Everything that will be needed further on can be declared here -functions, global variables, includes, etc. The following code fragment shows how a global decoration item is declared in Spectre.

[2]The Elastic terminology identifies input labels as "events", and output labels as "orders". We have chosen to keep that terminology in Spectre.

[3]This will only hold under certain conditions which will be detailed later on.

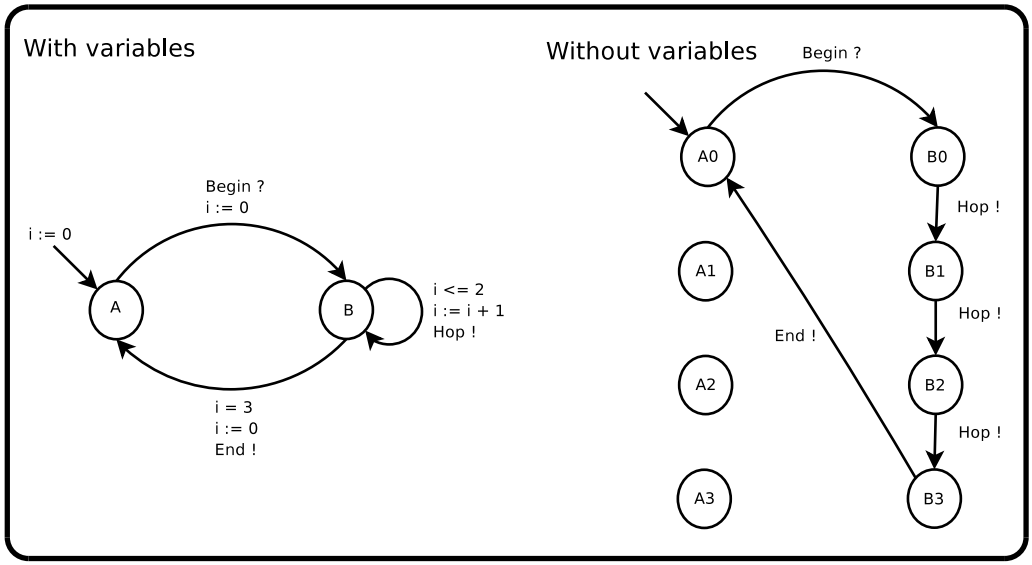

Figure 6.2: Transformation of a timed automaton with discrete variables into a standard timed automaton.

```
global
{%
    #include "complex.h"

    const int length = 8;
    int var_a;
    int array_b[length];
    int b_index;
%}
```

Figure 6.3: Example of a SPECTRE global decoration item.

Variables decoration

Conceptually, the discrete variables are used in two different ways : in the guard predicates they are *read*, and in the updates assignments they are *written to*. For each of those two uses, and for each variable, SPECTRE requires a code fragment which will perform the read or write operation.

Short examples of reading and writing decoration items are shown in figure 6.4. The items involving variable a are the simplest and probably the most common use of this decoration. Other more sophisticated uses are possible, as shown by the decorations of b and c. Note that all identifiers (functions and variables) mentioned in the decoration must be declared at the global level (see above).

```
reading a {% return var_a; %}
writing a (val) {% var_a = val; %}

reading b {% return array_b[b_index]; %}
writing b (val) {% array_b[b_index] = val; %}

reading c {% return something_complex(); %}
writing c (val) {% update_the_complex_thing(val); %}
```

Figure 6.4: Example of SPECTRE reading and writing decoration items.

Labels decoration

SPECTRE takes a quite radical stance regarding labels, because it considers that they are somewhat *context-free*[4]. The tool is designed in such a way that the emission of an output or internal order will produce the same side-effects, regardless of the source and destination locations of the edge to which the label was attached. Likewise, an event label will be treated by the tool the same way in every part of the automaton. While this radical choice may seem unnecessary at first, we believe that it will enforce cleaner and more elegant designs.

Order and internal labels require two code fragments, which we note d_1 and d_2. Upon crossing an edge from location A to location B labeled by σ, the produced code will execute $d_1(\sigma)$ *before* the variables updates and $d_2(\sigma)$ *after* the updates. This enables the designer of a SPECTRE implementation to produce label side-effects which depend on the variable valuations in a fine-controlled manner. As a result, labels in SPECTRE are not totally context-free, they can use a context in the form of variable valuations.

Here is a small example of order and internal labels decoration. The keyword "nop" can be used to provide an explicitly empty code fragment.

```
internal i {% printf("before : a = %d", var_a); %}
           {% printf("after  : a = %d", var_a); %}

order o    {% produce_some_side_effect(); %}
           nop
```

Figure 6.5: Example of SPECTRE order and internal decoration items.

Event labels decoration items require three code fragments instead of two. This additional code fragment (which we note $p(\sigma)$) has the responsibility of returning a non-zero value if and only if there is an untreated occurrence of the

[4]This "context-freeness" is not absolute, as will be explained shortly.

input label. The generated real-time task will call $p(\sigma)$ every period, for every event label σ (that is, unless there is already an untreated occurrence of σ, in which case the function need not be called).

Here is an example of an event decoration in SPECTRE. The code fragments d_1 and d_2 were left empty, but this is of course not mandatory.

```
event e    {% return event_e_has_arrived(); %}
           nop
           nop
```

Figure 6.6: Example of a SPECTRE event decoration item.

Transitions decoration

As with label decorations, the design choices behind SPECTRE's way of decorating transitions might seem limiting at first glance. In SPECTRE, the implementation designer cannot associate a piece of code to a specific edge of the automaton. Rather, it is possible to attach code to the following events[5] :

- The controller goes from location A to location B.
- The controller leaves location C.
- The controller enters location D.
- The controller makes a discrete jump, no matter which edge is used.

Just like with label decorations, it is possible to provide two code fragments for each transition decoration item. As a consequence, it is possible to differentiate side-effects on a per-edge fashion to a limited extent. For instance, in the example of figure 6.1, there are two edges from B to A, and the code can tell which edge was taken by testing the value of variable b after the updates have been computed.

The syntax for transitions decorations is straightforward, as illustrated by the example of figure 6.7. The keyword "any" represents the set of all locations. This example also illustrates SPECTRE's comment syntax, which is borrowed from the language Haskell, in which SPECTRE is written (line comments start with two consecutive minus symbols, and block comments are enclosed between "{-" and "-}").

The "any to any" item would probably not be used in a production environment but it is very useful for testing, by generating logs or execution traces for instance.

[5]Here, *event* is meant in the broad sense, not as a type of label.

```
A to B      {% printf("going from A to B !"); %}
            nop

any to B    nop
            {% printf("entering B !"); %}

A to any    {% printf("leaving A !"); %}
            nop

any to any  {% printf("before update : i = %d", var_i); %}
            {% printf("after update  : i = %d", var_i); %}

B to B      nop -- these two lines are,
            nop -- as expected, useless
```

Figure 6.7: Example of SPECTRE transitions decoration items.

Restrictions

The SPECTRE input language uses restriction items in order to produce fully-deterministic implementations out of a non-deterministic model. Controller models are usually non-deterministic because they represent the set of all possible behaviors of the controller, not its *actual* control flow. Restriction items provide a way to specify the desired control flow precisely, without any modification to the model.

Consider the following simple example. Both transitions are always enabled, because they represent a conditional statement that exclusively belongs to the implementation. The two restriction items are used to provide this missing conditional statement.

```
{- ... -}

location A :
    {}, none, {}, B;
    {}, none, {}, C;

{- ... -}

restrict A to B {% return  condition(); %}
restrict A to C {% return !condition(); %}
```

Figure 6.8: Example of SPECTRE restriction items.

As with transitions decorations, it is not possible to restrict the crossing of an edge in particular. It is possible however, to use the keyword "any" in restrictions.

It should be noted that restriction items are dangerous if used poorly, because they can modify the controller's behavior in such a way that correctness is no longer assured. This issue will be treated more in depth later on in this chapter.

It is also worth noting that controller implementations do not *need* to be fully-deterministic. Some network protocols, for instance, take certain decisions at random. However, in that case, the implementation designer will need to provide the code which generates the random numbers and makes the appropriate choice.

Startup and cleanup decorations

These last two remaining decoration items are called just before the controller starts and right after it has completed its execution. It is useful for allocating and freeing up resources, initializing variables, setting up peripherals, etc. The syntax for those items is straightforward, as shown by the following code excerpt.

```
startup {% /* allocate and initialize everything here */  %}
cleanup {% /* free up all used resources here */          %}
```

Figure 6.9: SPECTRE's startup and cleanup decoration items.

6.2 Code Generation

In this section, we will first describe the code generation process, and then discuss the various conditions that need to be met in order to insure correctness.

6.2.1 Architecture of the Generated Code

The way SPECTRE works is quite straightforward, so we will not examine every detail of the code generation process. Instead, we detail a number of key portions of the generated code. Additional information can be found in the appendix, along with complete examples of SPECTRE's output.

Additional defines

After running SPECTRE on an input file, the generated code needs a few small modifications before compiling. These modifications are however very limited, the implementation designer only needs to encode the following numerical constants, which are defined in the first few lines of the generated file :

1. The period of the real-time task.

2. The number of ticks between two clock interrupts.
3. The guard-enlargement constant.
4. The time unit, which maps the automaton time values to real time values.
5. The real-time task priority.
6. The real-time task stack size.

In code these constants are specified as pre-processor defines, as shown by the following code fragment :

```
#define TIME_UNIT        7000
#define WIDENER          1001
#define PERIOD           1000
#define BEAT               11
#define STACK_SIZE      10000
#define TASK_PRIORITY     128 /* 0 = highest, 255 = lowest */
```

Figure 6.10: Pre-processor defines which need to be filled in before compiling.

The code of figure 6.10 specifies a clock interrupt frequency of one interrupt every 11 ticks. This makes for a clock resolution of $\frac{11}{1193180}$ seconds, which is approximately 9,219061667 microseconds. With such a setting, the controller can be safely verified using a Δ_P of 10 microseconds. To distinguish the clock-resolution of the verification phase (Δ_P) and the one that is used in code (which can be smaller), we use the term *beat* for the latter.

Setting the period define to 1000 specifies a period of 1000 beats, thus approximately 9,219061667 milliseconds. Since the Real-Time Semantics has the "faster is better" property, such a setting could safely be guaranteed correct if the controller was verified with a Δ_T of 10 milliseconds. Of course, the schedulability analysis will need to be done using the real period value, not Δ_T.

The widener define specifies a guard enlargement of 1001 beats. This is consistent with the definition of $\Delta_S = \lceil \Delta_T + \Delta_P \rceil_{\Delta_P}$.

A time unit define of 7000 beats means that every numeric constant that is compared or assigned to clocks in the automaton are multiples of 7000 beats. For instance, a guard testing the equality of a clock with the constant 3 in the automaton, will be true if that clock's value lies in the interval $[\frac{11\times((3\times7000)-1001)}{1193180}, \frac{11\times((3\times7000)+1001)}{1193180}]$, thus $[0.184372014, 0.202828576]$, with both intervals being expressed in seconds.

Finally, the remaining two defines are shown with their default values. They need not be changed in normal circumstances, except maybe the task priority if there are other real-time tasks running on the target platform.

Kernel module management and real-time main loop

SPECTRE's output is the C source code of a LINUX kernel module which, when inserted into an RTAI-enabled kernel, launches a single periodic real-time task. The code of figure 6.11 shows how this is achieved in practice.

```
void spectre_task_main (int data) {
  spectre_running = 1;
  spectre_startup ();
  spectre_transition_initially ();
  while (spectre_running == 1) {
      spectre_current_time = rt_get_time ();
      spectre_check_for_events ();
      spectre_current_location ();
      rt_task_wait_period ();
  }
  spectre_cleanup ();
}

int spectre_init_module (void) {
  rt_set_periodic_mode ();
  start_rt_timer (BEAT);
  rt_task_init (&spectre_task, spectre_task_main, 0, STACK_SIZE,
                TASK_PRIORITY, USE_FPU, 0);
  rt_task_make_periodic (&spectre_task, rt_get_time () + PERIOD, PERIOD);
  return 0;
}

void spectre_exit_module (void) {
  stop_rt_timer ();
  rt_task_delete (&spectre_task);
}

module_init (spectre_init_module);
module_exit (spectre_exit_module);
```

Figure 6.11: Main loop and kernel module management routines.

The "init" and "exit" functions should be self-explanatory, all the RTAI primitives used by SPECTRE were detailed in the previous chapter. The first function of the figure is the entry point of the real-time task. Before entering the main loop, the startup decoration is called, followed by a call to the initial transition. This call will initialize the clocks and variables and set the current

location, all according to the "initially" line of the SPECTRE specification section. The current location is remembered by the code with a function pointer, which is called every period (the internals of this function will be detailed further on). The loop body is consistent with the Real-Time Semantics : first the current time is read, then the inputs are checked, and finally the outgoing edges of the current location are examined to see if any of them is enabled. This will result in at most one location change, before the current period is yielded. After the loop ends, the cleanup decoration code is called and the real-time task stops.

Location functions

As stated just before, every location in a SPECTRE-generated controller is represented by a function. This function, when called, examines every outgoing edge of its location and fires the first enabled transition it sees. The generated code for locations is extremely simple, as shown by figure 6.12.

```
void spectre_location_A (void)
{
  if (spectre_check_A_to_B_1 ()) {
    spectre_transition_A_to_B_1 (); return;
  }

  if (spectre_check_A_to_B_2 ()) {
    spectre_transition_A_to_B_2 (); return;
  }
}
```

Figure 6.12: Example of a location function.

Enabledness checking functions

For each edge of the automaton, SPECTRE generates a "check" function which returns a non-zero value if and only if its edge is currently enabled. This is done by evaluating the guard predicates, calling the appropriate restrictions decorations, and if necessary, verifying that there is an untreated occurrence of the attached event label.

The code in figure 6.13 shows an example of a check function. The function calls prefixed by "r" correspond to restrictions decorations. Every restriction that is not mentioned in the SPECTRE input file is given a function which simply returns 1. The internals of the "guard" function in the figure are described hereafter.

```
int spectre_check_A_to_B_1 (void) {
  int spectre_r1, spectre_r2, spectre_r3, spectre_r4, spectre_g;
  spectre_r1 = spectre_r_A_to_B ();
  spectre_r2 = spectre_r_A_to_any ();
  spectre_r3 = spectre_r_any_to_B ();
  spectre_r4 = spectre_r_any_to_any ();
  spectre_g = spectre_guard_A_to_B_1 ();
  return (spectre_g && spectre_r1 && spectre_r2 &&
          spectre_r3 && spectre_r4 && (spectre_e_is_pending == 1));
}
```

Figure 6.13: Example of a transition check function.

Guard evaluation functions

The code at figure 6.14 shows how the generated controller evaluates guards (in this case, it is the guard $\{x = 2, p \geq 7\}$ with x being a clock and p a discrete variable). The `spectre_equals` function provides the enlargement mechanism. As mentioned in the beginning of this chapter, this guard evaluation function uses the "read" decorations to retrieve the variable values.

```
int spectre_guard_One_to_Two_1 (void) {
  RTIME spectre_t1, spectre_t2;
  int spectre_e1, spectre_e2;
  int spectre_res = 1, spectre_tmp;
  spectre_t1 = spectre_read_x ();
  spectre_t2 = (TIME_UNIT * 2);
  spectre_tmp = spectre_equals (spectre_t1, spectre_t2);
  spectre_res = spectre_tmp && spectre_res;
  spectre_e1 = spectre_read_p ();
  spectre_e2 = 7;
  spectre_tmp = spectre_e1 >= spectre_e2;
  spectre_res = spectre_tmp && spectre_res;
  return spectre_res;
}
```

Figure 6.14: Example of a guard evaluation function.

Transitions functions

When a location function (described previously) detects an enabled edge, it calls the corresponding "transition" function. This function calls the appropriate dec-

orations, computes the clocks and variables updates, and updates the current location appropriately. An example of such a function is illustrated at figure 6.15.

```
void spectre_transition_A_to_B_1 (void)
{
  spectre_d1_A_to_B ();
  spectre_d1_A_to_any ();
  spectre_d1_any_to_B ();
  spectre_d1_any_to_any ();
  spectre_d1_e ();
  spectre_update_A_to_B_1 ();
  spectre_current_location = spectre_location_B;
  spectre_d2_e ();
  spectre_e_is_pending = 0;
  spectre_d2_A_to_B ();
  spectre_d2_A_to_any ();
  spectre_d2_any_to_B ();
  spectre_d2_any_to_any ();
}
```

Figure 6.15: Example of a transition function.

The transition function first calls the d_1 transition decorations. Any such decoration that was not present in the input file is replaced by an empty stub (this also applies to "nop" fragments). Right after, the d_1 label decoration is called, followed by the clocks and variables updates, the location update, the d_2 label decoration, and the reset of the event label pending flag (this last action is only performed for event labels, obviously). Finally, the d_2 transitions decorations are called.

These numerous function calls might raise some performance concerns. However, we believe that this should easily be solved if a good optimizing compiler is used, which will "inline"[6] many (if not all) of these calls, and simply remove the empty stubs.

Updates functions

Updates functions are called by the transitions functions to perform the clocks and variables updates. It uses both the "read" and "write" decorations to perform

[6] "Inlining" is a common optimization trick which replaces a function call by a copy of that function's body. This generates faster code (the call / return sequence is saved), at the cost of a larger executable.

the updates. The example of figure 6.16 shows the generated update function for the assignments $\{x := 2, a := a + 4\}$.

```
void spectre_update_A_to_B_2 (void)
{
  spectre_write_x (TIME_UNIT * 2);
  spectre_write_a (spectre_read_a () + 4);
}
```

Figure 6.16: Example of an update function.

6.2.2 Correctness of the Generated Code

In the following, we shall not prove that the code generated by SPECTRE is correct, as that would require a very involved effort that is beyond the scope of this work. Rather, we will enumerate a number of *necessary* conditions that need to be met to insure correctness, and then give informal arguments showing that these conditions should be *sufficient* in practice.

Necessary conditions for correctness

In the following, we will assume that the AASAP semantics of the automaton given in the specification has been proven to satisfy a set of safety properties, for some positive value Δ. We will assume that the designer assigned values to the Real-Time Semantics parameters such that the inequality $\Delta > \Delta_T + 2\Delta_D + 4\Delta_P$ is satisfied. The generated code needs to satisfy the following requirements (details follow) :

($R1$) The clock resolution set by the `BEAT` define must not be greater than Δ_P.

($R2$) The `PERIOD` define must not be greater than Δ_T.

($R3$) The WCRT of the task loop must not be greater than Δ_D.

($R4$) All decorations code fragments must be proven free of run-time errors.

($R5$) All decorations code fragments must be proven to terminate.

($R6$) The `WIDENER` define must be set to `PERIOD + 1`.

($R7$) The restrictions decorations must only reduce non-determinism.

($R8$) The input-checking functions must be proven never to miss any event.

($R9$) The `PRIORITY` define must be set such that the task is proven schedulable.

The requirements ($R1$) through ($R3$) are very natural. If the real-time semantics inequality holds for a set of parameters, it will remain true if the parameters effectively used are smaller.

The requirements ($R4$) and ($R5$) are related to requirement ($R3$). The generated code must be proven to always complete a task loop without errors and within the deadline. Proving this in a formal way requires techniques which are beyond the scope of this work. However, if great care is put into the decoration programming, it might suffice in practice.

Requirement ($R6$) comes from the definition of $\Delta_S = \lceil \Delta_T + \Delta_P \rceil_{\Delta_P}$.

Requirement ($R7$) insures that the generated controller's behavior is somehow a *refined version* of the original automaton. It is easy to prove that reducing nondeterminism does not break safety-properties[7].

Requirement ($R8$) needs to be met because we made the hypothesis that the controller was *input-enabled*. For this hypothesis to hold in practice, we need to make sure that the input-polling functions never miss an event.

Requirement ($R9$) comes from the Real-Time Semantics, which makes the hypothesis that the deadline is always met.

Final notes on code correctness

Because it follows precisely the strategy dictated by the Real-Time Semantics, the code generated by SPECTRE (without the decorations) can be considered correct by construction. Moreover, if great care has been put in the decoration effort, and if all the requirements just described are proven to be satisfied, then it seems reasonable to have great confidence in the generated controller, decorations included.

6.3 Use of the SPECTRE Decoration Language

The decoration language of SPECTRE is simple and straightforward but to some extent somewhat limiting. Indeed, for an arbitrary controller automaton, a SPECTRE decoration file might be hard to write (because of the impossibility to decorate an edge in particular for instance). However, if is known during the modeling phase that SPECTRE will be used for code generation, we believe that it should be rather simple to deal with its few limitations. Here are two guidelines to follow when modeling a controller to use with SPECTRE :

- The semantic weight should be stressed on *locations*. Going from a location A to a location B should have a meaning that is independent of the edge followed.

[7] This is because the existence of a simulation relation is easy to show.

- Synchronization labels should have a meaning of their own, regardless of *where* they are used in the automaton.

Clearly, these guidelines seem reasonable for an automaton that is intended to be used for code generation.

6.4 Implementation of SPECTRE

The current SPECTRE implementation is made of about 3000 lines of Haskell[8]. The parsing code was created using the ALEX lexical analyzer generator, and the HAPPY parser generator. The rest of the SPECTRE source code is composed of a semantic checker and the code generator itself. The entire source code is freely available upon request to the author : nmaquet@ulb.ac.be

As Haskell is completely portable, SPECTRE should run without modification on any platform.

[8]Haskell is a purely functional language. Compilers, interpreters and documentation for Haskell are available at http://www.haskell.org

Chapter 7

Case Study : Philips Audio Control Protocol

This chapter illustrates how our methodology is used in practice, from verification to code generation. For this purpose, we have chosen to implement a simple real-time data transmission protocol, namely the Philips Audio Control Protocol (PACP in the following). This physical-layer protocol was designed by Philips engineers for stereo device equipments, and uses Manchester encoding to send arbitrary long binary sequences on a wire between a single sender and a receiver. We have chosen to implement this particular protocol for the following reasons :

1. Because it is a physical-layer protocol using Manchester encoding, a software implementation must run on a real-time platform to insure correctness.
2. The protocol has already been analyzed with the Almost ASAP semantics in [DDR05a], so we will use their verification results.
3. The protocol is quite subtle, and is arguably difficult to implement by hand. This advocates in favor of code generation.
4. The experimental setup needed for implementation and testing is very limited : only a single low-cost PC is required. Optionally, one could also link two machines with a single cable.

The following is structured as follows : first we describe the data transmission protocol informally, then we present the modeling effort and verification results obtained in [DDR05a], and finally we describe the implementation process using the tool SPECTRE.

Remark : Figures 7.4 through 7.7 are found at the end of this chapter.

7.1 Description of the PACP

As stated just above, the protocol uses Manchester encoding to transmit binary sequences. The binary information is sent by changing the wire voltage in the middle of evenly-spaced *time slots*. Each time slot holds one bit of information; if the voltage goes from a low state to a high state in the middle of the time slot then a "1" is transmitted. If on the contrary, the switch went from high to low voltage then it is considered by the receiver as a "0". The PACP faces the following difficulties :

- Even if the sender and receiver agree on the length of a time slot, they run asynchronously so they do not know where the time slot begins.
- The protocol needs to work for binary sequences of any length, so the receiver needs a way of detecting the beginning and end of a sequence.
- The receiver cannot detect voltage switches from high to low reliably[1] (called DOWN signals from now on), so it will need to rely solely on switches from low to high (UP signals).

The above issues are solved by the PACP as follows : when no transmission takes places, the wire voltage stays low. To initiate a sequence, the sender starts by sending an UP signal, which is understood by the receiver as the middle of the first time-slot. The binary sequence is then sent using the standard Manchester encoding technique. To decode the binary sequence, the receiver simply measures the time elapsed between consecutive UP signals. This works fine except for the end of the sequence, because by relying only on UP signals, the receiver cannot distinguish sequences ending with "1" from those ending with "10". This is illustrated at figure 7.1.

As seen in the figure, the difference between "111" and "1110" is marked only by the time of the last DOWN signal, which the receiver does not see. The PACP solves this problem by requiring that all transmitted sequences be either odd in length or end with "00".

7.2 Modeling the PACP with Timed Automata

In [DDR05a], Raskin et al. have analyzed the PACP using the Almost ASAP semantics. The purpose of the article was to illustrate the practical and theoretical attractiveness of the Almost ASAP semantics compared to other approaches, which do not address the synchrony hypothesis thoroughly. They also briefly described a code generation scheme which targets the LEGO MINDSTORMS™ platform, but the emphasis was clearly on the more theoretical aspects.

[1] This limitation comes from the original target hardware environment of the PACP.

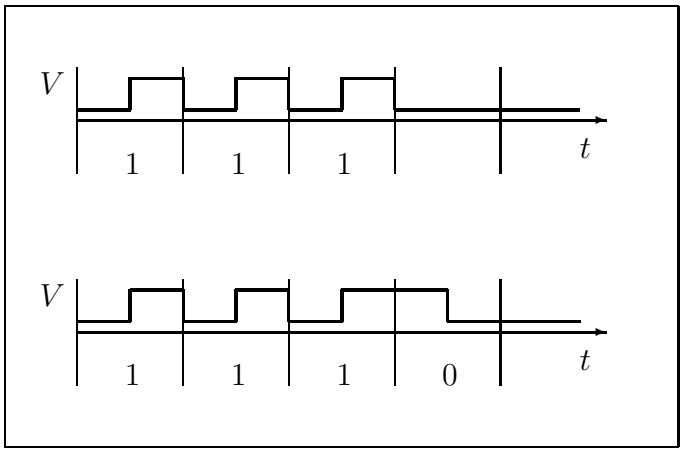

Figure 7.1: Manchester encodings for sequences "111" and "1110".

As our work focuses on code generation, rather than model-checking, we review the verification results obtained in [DDR05a], but without much detail as to *how* they were obtained. The interested reader will find this information in [DDR05a].

7.2.1 Sender Automaton

A timed automaton for the PACP sender controller is depicted at figure 7.2. It has a single clock x, and three discrete variables p, dz and i. Again, these discrete variables could have been encoded in the locations, but this has not been done for the sake of clarity. These variables have the following meaning :

- p is used to remember the parity of the transmitted sequence (0 for even, 1 for odd).
- dz (standing for "double zero") contains 1 if the last two transmitted bits were "00", and 0 otherwise.
- i contains the next bit to be transmitted, and is read by the sender to produce UP and DOWN signals appropriately. During verification, this variable is assigned to a non-deterministic binary value, which can change at any time.

As the first UP signal does not correspond to a transmitted bit, the sequence parity is initialized to 0. Every time a bit is sent, the update `p := 1-p` sets the new parity appropriately. When the automaton goes from `WaitZero` to `ZeroSent`, we know we have just sent two consecutive zeroes, variable dz is thus set to 1.

Observe that the sender automaton non-deterministically chooses to end or continue to transmit, after sending each bit. This makes the sender automaton valid for any binary sequence of finite length.

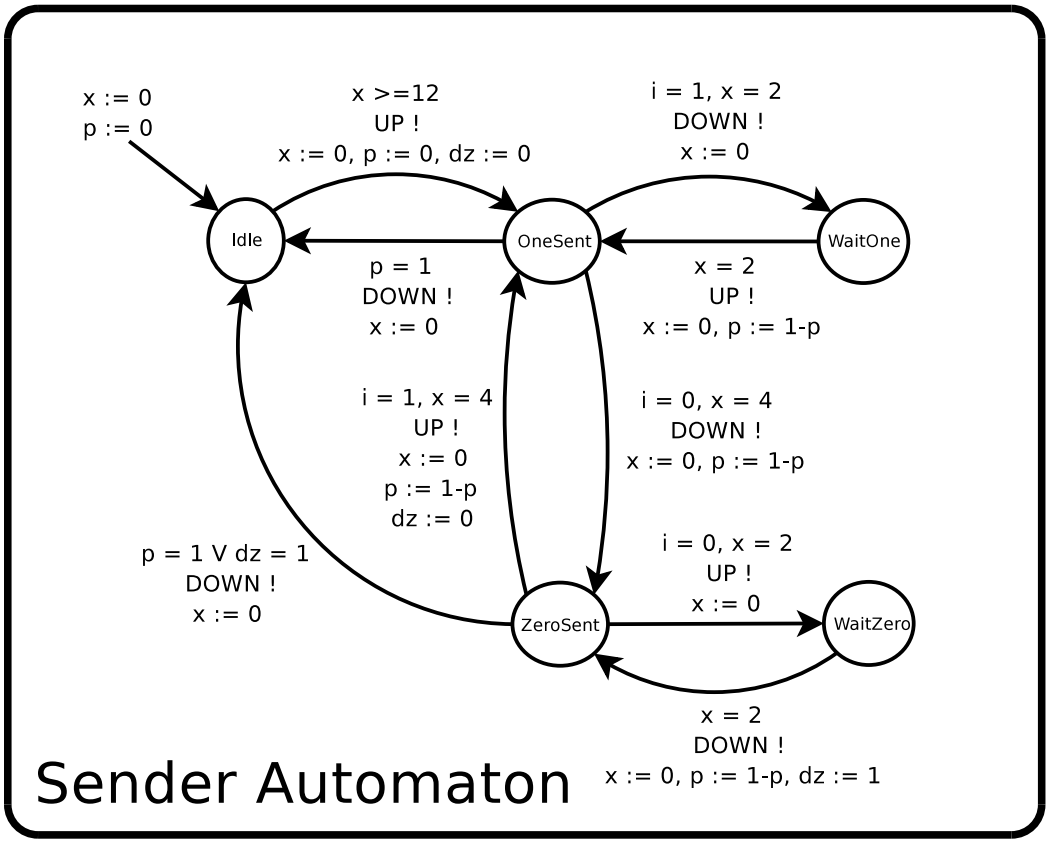

Figure 7.2: Sender automaton for the PACP.

Finally, the predicate $\{x \geq 12\}$ attached to the edge from `Idle` to `OneSent` is there to ensure that the sender stays idle for at least three time slots between two consecutive sequences. This is necessary because the receiver automaton relies on that period of inactivity to detect the end of the sequence.

7.2.2 Receiver Automaton

The receiver automaton is a bit more simple because it has only three location, and it is illustrated at figure 7.3. It has one clock y and two discrete variables m and r. The m variable is equivalent to the p variable of the sender automaton and contains the parity of the received sequence. The receiver uses this variable to determine whether it needs to append a final "0" to the binary stream when the transmission ends. Variable r is assigned to the received bit(s) each time the receiver sees an UP signal (except the for the first UP signal). When r is assigned the value 2, it means that the receiver has decoded "01".

Each time the automaton sees an UP signal, it rounds the reception time to the closest time-slot center. This is what makes the PACP robust : if all accumulated clock-rounding errors never reach more than one quarter of a time-

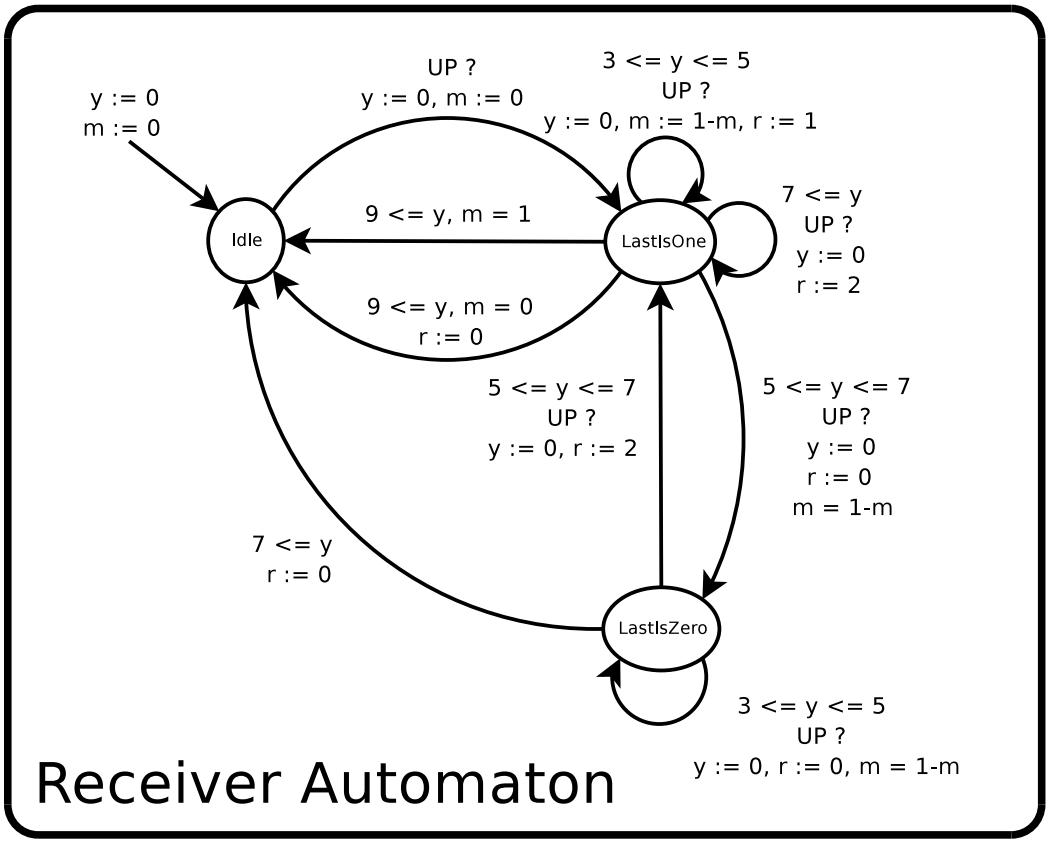

Figure 7.3: Receiver automaton for the PACP.

slot, then the receiver will be able to decode each UP signal appropriately. The Almost ASAP semantics enables us to prove this in a formal fashion.

7.3 Verification Results

In [DDR05a], it is shown that the above specification is correct in the context of the Almost ASAP semantics iff the sum of the respective Δ parameters of the sender and receiver automata is strictly less than one sixteenth of a time slot. Formally, for an appropriate Observer automaton, and U being one quarter of a time-slot :

$$\mathsf{Reach}([\![\mathsf{Sender}]\!]^{\mathsf{AAsap}}_{\Delta_1} \,||\, [\![\mathsf{Receiver}]\!]^{\mathsf{AAsap}}_{\Delta_2} \,||\, [\![\mathsf{Observer}]\!]) \cap \mathsf{Bad} = \phi \;\text{ iff }\; \Delta_1 + \Delta_2 < \frac{1}{4}U$$

We see from the above formula that the Δ parameters of the automata are related : if one controller is less precise (i.e. has a higher Δ parameter), the other must compensate to retain correctness of the whole (by decreasing its Δ

parameter). If the hardware platforms of both controllers are equal then the best throughput (which is directly related to U, the length of a time-slot) will be obtained by assigning both Δ parameters to the same value.

7.4 Systematic Implementation of the PACP using SPECTRE

Now that we have given a complete specification of the PACP and indicated under which conditions it is proven correct, we can describe how the implementation process is achieved.

7.4.1 Sender Specification

The SPECTRE specification section of the PACP sender controller is given at figure 7.4. This section should be straightforward, as it is simply a textual version of the automaton of figure 7.2.

7.4.2 Sender Decoration

A SPECTRE decoration section for the sender automaton is found at figure 7.5. We comment each decoration item in turn.

Global decoration

This implementation of the PACP uses the parallel port to transmit data. On our target platform (a Intel-based PC), sending data through the parallel port is achieved by writing a byte at the memory address `0x378`. The parallel port can be configured in various modes, which is done by writing the appropriate byte at the memory address `0x37A`.

Variables `var_p` and `var_dz` will be used to store their respective automaton variables. For the i variable, it is a bit more complicated as we do not want our controller to send random bits. Instead, we declare an array `data[]` along with an index, and we will map the i variable to the contents of `data[]` at index `data_index`.

Finally, we declare a small boolean function : `sequence_finished()`, which will be used to make the implementation deterministic.

Labels decorations

The UP order of the sender automaton is decorated by a single call to the `outb` function, which sets all data bits of the parallel port to 1. Similarly, the decoration for the DOWN order clears all bits of the parallel port back to 0.

Startup and cleanup decorations

The only initializations that the sender needs to perform are (1) configure the parallel port for output, and (2) put the initial data bits to 0. No cleanup decoration is needed.

Variables decorations

The decorations for variables p and dz are straightforward. Variable i is bound to the next bit to be sent, thus `data[data_index]`. The array-accessing code is protected by a conditional statement ensuring that we will never access the wrong memory space. As our controller runs in kernel mode and is thus not subject to page fault violations, this might seem unnecessary. We do it anyway for the sake of best practices.

Transitions decorations

The transitions decorations are used to initialize and increment the data index when necessary. When the automaton goes from `Idle` to `OneSent`, we know that the transmission starts so we take this opportunity to initialize `data_index` to 0. When entering `OneSent` from locations `WaitOne` and `ZeroSent`, we know we have just sent a "1" bit so we increment `data_index`. Similarly, when entering location `ZeroSent` from anywhere, we know that a "0" has been sent so we increment `data_index` in this case as well.

Finally, when the automaton goes back to `Idle`, we know that the transmission is complete so we set the `spectre_running` flag to 0. This will cause the real-time task to stop.

Restrictions

The final part of the decoration section uses five restrictions to make the sender controller fully-deterministic. The non-determinism of the sender automaton comes from the fact that it can go back to `Idle`, and thus end the transmission, after each bit is sent. Hence, to make the sender controller deterministic we must make sure that the two following statements are true :

1. The controller does not end the transmission until it is finished.
2. The controller *does* end the transmission when it is finished.

This can be expressed in SPECTRE by using the following restrictions :

1. `restrict any to Idle {% return sequence_finished(); %}`

```
2. restrict any to WaitZero     {% return !sequence_finished();  %}
   restrict any to WaitOne      {% return !sequence_finished();  %}
   restrict any to ZeroSent     {% return !sequence_finished();  %}
   restrict ZeroSent to OneSent {% return !sequence_finished();  %}
```

It is easy to see that the above restrictions have the effect of only reducing non-determinism.

7.4.3 Receiver Specification

The SPECTRE specification section for the PACP receiver controller can be found at figure 7.6. Again, this is simply a textual representation of the receiver automaton given earlier in this chapter. Notice the usage of the "none" keyword, which represents the silent label.

7.4.4 Receiver Decoration

A SPECTRE decoration section for the receiver automaton can be found at figure 7.7. We comment each decoration item in turn.

Global decoration

The global decoration of the PACP receiver controller is very similar to the sender's. The `GOOD_INDEX` define is slightly modified (it requires `data_index` to be at least two positions away from the bottom of `data[]`) because the receiver will write up to two bits at once in the `data[]` array. As the `data[]` array is statically allocated with a length of 128, this implementation of the PACP receiver only supports transmission of up to 128 bits[2].

Labels decorations

The only label of the receiver automaton is the input UP signal. In the SPECTRE input file, we must decorate this event with a function which returns a non-zero value if and only if an untreated occurrence of UP is detected. To achieve this, the receiver controller reads the contents of the parallel port data register every period (stored in `val`), and remembers the value read at the previous period in a global variable `last_val`. The following table shows the various possibilities. A "1" in the table represents a non-zero value.

[2]It might seem silly to waste a whole integer (four bytes on our target platform) to store one bit of information. This is done solely to alleviate the figures.

last_val	val	event up
0	0	0
0	1	1
1	0	0
1	1	0

Admittedly, this way of checking for the UP signal is a bit cheating, because it relies on seeing the DOWN signals reliably (third line in the table). We could have avoided cheating by providing a handler for the parallel port interrupt line (IRQ 7 on Intel-based machines), but we have chosen not to do so to keep the receiver decoration short and simple.

It is easy to see that this input-polling function will never miss an event. Since we have that $\Delta < \frac{1}{8}U$ and $\Delta > \Delta_T + 2\Delta_D + 4\Delta_P$, we know that the period of the real-time task will be *at least* 32 times smaller than the time-slot. It should hence be impossible for the controller to miss an event by using this polling function.

Startup and cleanup decorations

The startup decoration configures the parallel port for input use, and initializes the variables that are not initialized by the `initially` line of the specification.

The cleanup decoration simply prints out the received bits to the kernel log. This output can be consulted in LINUX by using the **dmesg** command, or by reading the file **/var/log/messages**.

Variables decorations

The decoration of variable m is straightforward. Remember that the r variable is written by the receiver automaton when it has decoded a new bit (or sometimes two consecutive bits). Hence, we provide a writing decoration for that variable, which writes to the `data[]` array whatever r is written to. Again, the array accessing code is protected by a conditional statement. If the sender goes mad and sends more than what the receiver can handle, we need to make sure that no buffer overflow will occur. This is especially important because the receiver controller runs in kernel space, where memory corruption is very destructive to say the least.

Transitions decorations

The only transition decoration of the receiver is one that makes the controller stop when the sequence is finished (i.e. when the automaton goes back to `Idle`).

7.4.5 Assigning the RT Semantics Parameters

One last step is needed before our implementation of the PACP is complete. After running each input file through the SPECTRE tool, we need to encode some numerical constants in the `BEAT`, `PERIOD`, `WIDENER` and `TIME_UNIT` defines. To ensure correctness, we must respect the following equations :

$$\begin{aligned}
\Delta &> \Delta_T + 2\Delta_D + 4\Delta_P \\
U &> 8 \times \Delta \\
U &< (1193180)^{-1} \times \texttt{BEAT} \times \texttt{TIME_UNIT} \\
\Delta_T &> (1193180)^{-1} \times \texttt{BEAT} \times \texttt{PERIOD} \\
\Delta_P &> (1193180)^{-1} \times \texttt{BEAT} \\
\Delta_D &> \text{WCRT of the task} \\
\texttt{WIDENER} &= \texttt{PERIOD} + 1
\end{aligned}$$

On our testing machines, a safe setting for the clock resolution is 11 ticks. Also, some very approximate WCET analysis shows that both the sender and receiver controllers will always be able to complete a task loop within a time certainly not larger than 1 millisecond. As a CPU usage of 50% seems sufficiently high, we set the period to 2 milliseconds. These observations suffice to assign a safe value to the remaining parameters, which we do as follows :

$$\begin{aligned}
\texttt{BEAT} = 11 &\rightarrow \Delta_P = 10\ \mu\text{s} \\
\Delta_P = 10\ \mu\text{s} & \\
\Delta_D = 1\ \text{ms} & \\
\Delta_T = 2\ \text{ms} &\rightarrow \Delta = 5\ \text{ms} \\
\Delta = 5\ \text{ms} &\rightarrow U = 45\ \text{ms} \\
U = 45\ \text{ms} &\rightarrow \texttt{TIME_UNIT} = 5000 \\
\Delta_T = 2\ \text{ms} &\rightarrow \texttt{PERIOD} = 200 \\
\texttt{PERIOD} = 200 &\rightarrow \texttt{WIDENER} = 201
\end{aligned}$$

7.5 Execution of the Generated Code

The generated code has been successfully tested on two 500MHz Pentium machines. With the above settings, the actual CPU usage observed in practice is less than 1%. This shows that our WCET analysis was very pessimistic indeed. More fine-grained WCET and schedulability analysis could be used to push the throughput to several hundred bits per second (with the above settings it is a little over 5 bits per second). Peformance-wise, this seems very satisfying considering that (1) everything is done in software, (2) this software is provably correct, and (3) the hardware used is quite modest.

```
specification sender

clocks : x;
vars : i, p, dz;
events : ;
internals : ;
orders : up, down;

initially Idle, {x := 0, p := 0, dz := 0};

location Idle :
    {x >= 12}, up, {x := 0, p := 0, dz := 0}, OneSent;

location OneSent :
    {x = 2, i = 1}, down, {x := 0}, WaitOne;
    {x = 4, i = 0}, down, {x := 0, p := 1-p}, ZeroSent;
    {p = 1}, down, {x := 0}, Idle;

location WaitOne :
    {x = 2}, up, {x := 0, p := 1-p}, OneSent;

location ZeroSent :
    {x = 2, i = 0}, up, {x := 0}, WaitZero;
    {x = 4, i = 1}, up, {x := 0, p := 1-p, dz := 0}, OneSent;
    {p = 1}, down, {x := 0}, Idle;
    {dz = 1}, down, {x := 0}, Idle;

location WaitZero :
    {x = 2}, down, {x := 0, p := 1-p, dz := 1}, ZeroSent;

end
```

Figure 7.4: SPECTRE specification for the PACP sender controller.

```
decoration sender

global
{%
    #define DATA_REGISTER   0x378
    #define CTRL_REGISTER   0x37A
    #define GOOD_INDEX      (0 <= data_index && data_index < data_length)
    int var_p, var_dz, data_index;
    int data[] =
    {   1, 1, 1, 1, 1, 1, 1, 1,    1, 0, 1, 0, 1, 0, 1, 1,
        1, 0, 1, 0, 1, 1, 1, 1,    1, 0, 1, 0, 1, 0, 1, 1,
        1, 0, 1, 1, 1, 0, 1, 1,    1, 0, 1, 0, 1, 0, 1, 1,
        1, 1, 1, 0, 1, 0, 1, 1,    1, 1, 1, 1, 1, 1, 1, 1,    0, 0
    };
    const int data_length = 8*8+2;
    int sequence_finished(void) {return data_index >= data_length;}
%}

order up   {% outb(255, DATA_REGISTER); %} nop
order down {% outb(0,   DATA_REGISTER); %} nop

startup {% outb(0, CTRL_REGISTER); outb(0, DATA_REGISTER); %}
cleanup nop

reading i          {% return GOOD_INDEX ? data[data_index] : 0; %}
reading p          {% return var_p;                              %}
writing p (value)  {% var_p = value;                             %}
reading dz         {% return var_dz;                             %}
writing dz (value) {% var_dz = value;                            %}

Idle     to OneSent   {% data_index = 0;      %}     nop
WaitOne  to OneSent   {% data_index++;        %}     nop
ZeroSent to OneSent   {% data_index++;        %}     nop
any      to ZeroSent  {% data_index++;        %}     nop
any      to Idle      {% spectre_running = 0; %}     nop

restrict any to Idle         {% return  sequence_finished();  %}
restrict any to WaitZero     {% return !sequence_finished();  %}
restrict any to WaitOne      {% return !sequence_finished();  %}
restrict any to ZeroSent     {% return !sequence_finished();  %}
restrict ZeroSent to OneSent {% return !sequence_finished();  %}

end
```

Figure 7.5: SPECTRE decoration for the PACP sender controller.

```
specification receiver

clocks : y;
vars : m, r;
events : up;
internals : ;
orders : ;

initially Idle, {y := 0, m := 0};

location Idle :
    {}, up, {y := 0, m := 0}, LastIsOne;

location LastIsOne :
    {3 <= y, y <= 5}, up, {y := 0, m := 1-m, r := 1}, LastIsOne;
    {7 <= y}, up, {y := 0, r := 2}, LastIsOne;
    {5 <= y, y <= 7}, up, {y := 0, m := 1-m, r := 0}, LastIsZero;
    {9 <= y, m = 0}, none, {r := 0}, Idle;
    {9 <= y, m = 1}, none, {}, Idle;

location LastIsZero :
    {3 <= y, y <= 5}, up, {y := 0, m := 1-m, r := 0}, LastIsZero;
    {5 <= y, y <= 7}, up, {y := 0,r := 2}, LastIsOne;
    {7 <= y}, none, {r := 0}, Idle;

end
```

Figure 7.6: SPECTRE specification for the PACP receiver controller.

```
decoration receiver

global
{%
    #define DATA_REGISTER   0x378
    #define CTRL_REGISTER   0x37A
    #define GOOD_INDEX      (0 <= data_index && data_index < data_length-1)
    const int data_length = 128;
    int var_m, data_index, data[data_length];
    unsigned char last_val;
    void print_received_bits(void) {
        int i;
        if(GOOD_INDEX)
            for(i = 0; i < data_index-1; i++) rt_printk("%d\n", data[i]);
        else
            rt_printk("ERROR : received more bits than buffer length...\n");
    }
%}

event up
{%
    unsigned char val = inb(DATA_REGISTER);
    return val == last_val ? 0 : (last_val = val);
%} nop nop

startup {% outb(32, CTRL_REGISTER); last_val = 0; data_index = 0; %}
cleanup {% print_received_bits(); %}

reading m       {% return var_m; %}
writing m (val) {% var_m = val;  %}

writing r (val)
{%
    if(!GOOD_INDEX) return;
    switch (val) {
        case 0: case 1: data[data_index++] = val; break;
        case 2:         data[data_index++] = 0; data[data_index++] = 1;
    }
%}

any to Idle {% spectre_running = 0; %} nop

end
```

Figure 7.7: SPECTRE decoration for the PACP receiver controller.

Chapter 8

Conclusions

The objective of this work was to develop a methodology for creating provably correct software systematically, in the context of real-time embedded controllers. A controller in our context is a software-operated device which interacts with a physical environment and has the responsibility of keeping that environment in a safe configuration. Our methodology belongs to the so-called "top-down" approach to the creation of provably correct software, which can be summarized as follows : first, a formal model for the system (controller and environment) is created; second, a formal property is specified; third, the model is checked (usually with automated tools) against the property; and finally, the control software is constructed systematically and is guaranteed correct by construction.

In this work, we have put a lot of effort into ensuring that the models we use are *realistic* in the sense that they do take into account the software and hardware limitations that a real-time controller implementation must face. Indeed, many verification approaches to this day make a number of simplifications regarding these limitations, by making the hypothesis that no computation delay or data transmission delay will ever be large enough to cause any harm. This can be justified informally in a number of cases, but clearly there are situations where those simplifications need to be *formally* validated. To this end, we have used recent research from Raskin et al. [DDR05b] which proposes an elegant way of dealing with these simplifications formally, namely the Almost ASAP Semantics.

Experimental testing has revealed that the code generated using the Almost ASAP Semantics approach, when run on a hard real-time platform, behaves better than expected. These experimental results suggested that the model for the control software was somehow too coarse than needed, and that it hence could be refined. Further investigation revealed the possibility to perform this refinement by making a few assumptions about the run-time environment, namely the presence of a hard real-time operating system. This has led to the creation of a new implementation semantics, which we named Real-Time Semantics, that provides a tighter model for the control software. To validate this new semantics, we have constructed a simulation proof with the Almost ASAP Semantics, which

resulted in a non-trivial adaptation of a similar proof found in [DDR05b].

As our work was focused more on code generation than on model-checking, we have only illustrated the use of *safety* properties, which require the environment to avoid a subset region of its state-space. Our methodology could be used with more complex specifications however, such as LTL formulas.

The final part of our work has been to develop a code generator which was meant to illustrate the practical applicability of our approach. This step involved the design of an input language composed of two parts called *specification* and *decoration*. The former is used to encode the controller's timed automaton and is very classic, borrowing heavily from the syntax of the tools HyTech and Elastic. The decoration part of the language has been developed from the ground up however, and much effort has been put into making it as generic and straightforward as possible. The final result is a trade-off between simplicity and expressive power. Much of the involved design choices were made in favor of straightforwardness, which we believe is especially important since the decoration programming effort is the only part of our methodology which is not formally ensured correct. Hence, a simple and effective tool with some limitations seemed more appropriate than one more powerful but intricate and harder to manipulate.

The code generation process itself that we have implemented has also been designed with straightforwardness in mind. This makes it easier to be convinced that the generated code follows the Real-Time Semantics very closely, and can hence be considered correct by construction. This correctness is only ensured within a certain set of requirements however, which we have tried to enumerate in the most exhaustive way possible.

As final note, it is interesting to point out that the case study we have presented in the previous chapter is a fine example of what could barely be possible to verify only ten years ago. In contrast, the model-checking tools available today are able to perform this verification in a matter of seconds, and that is by using the Almost ASAP Semantics which is much coarser than the classical semantics and is thus harder to verify. This clearly illustrates the progress made in the field in recent years, and without which this work would not have been possible.

Appendix A

SPECTRE Grammar

A.1 Typographical Conventions Used

Empty symbol :	ϵ
Non terminals :	$\langle non\_terminal \rangle$
Keyword terminals :	`keyword_terminal`
Non keyword terminals :	non keyword terminal

A.2 Grammar

$\langle spectre\_input \rangle$ :: $\langle specification \rangle$
$\langle decoration \rangle$

$\langle specification \rangle$ :: `specification`
identifier
$\langle declarations \rangle$
$\langle initially \rangle$
$\langle locations \rangle$
`end`

$\langle declarations \rangle$ :: ϵ
| $\langle declaration \rangle$
$\langle declarations \rangle$

$\langle declaration \rangle$ :: $\langle clocks\_declaration \rangle$
| $\langle vars\_declaration \rangle$
| $\langle events\_declaration \rangle$
| $\langle orders\_declaration \rangle$
| $\langle internals\_declaration \rangle$

⟨*clocks_declaration*⟩ :: `clocks` : ⟨*identifier_list*⟩ `;`
⟨*vars_declaration*⟩ :: `vars` : ⟨*identifier_list*⟩ `;`
⟨*events_declaration*⟩ :: `events` : ⟨*identifier_list*⟩ `;`
⟨*orders_declaration*⟩ :: `orders` : ⟨*identifier_list*⟩ `;`
⟨*internals_declaration*⟩ :: `internals` : ⟨*identifier_list*⟩ `;`

⟨*identifier_list*⟩ :: ϵ
| identifier
⟨*identifier_list_tail*⟩

⟨*identifier_list_tail*⟩ :: ϵ
| `,` identifier
⟨*identifier_list_tail*⟩

⟨*initially*⟩ :: `initially` : identifier `,` ⟨*assignments*⟩ `;`

⟨*assignments*⟩ :: `{` ⟨*assignment_list*⟩ `}`

⟨*assignment_list*⟩ :: ϵ
| ⟨*assignment*⟩
⟨*assignment_list_tail*⟩

⟨*assignment_list_tail*⟩ :: ϵ
| `,` ⟨*assignment*⟩
⟨*assignment_list_tail*⟩

⟨*assignment*⟩ :: identifier `:=` ⟨*expression*⟩

⟨*expression*⟩ :: ⟨*expression*⟩ `+` ⟨*term*⟩
| ⟨*expression*⟩ `-` ⟨*term*⟩
| ⟨*factor*⟩

⟨*term*⟩ :: ⟨*term*⟩ `*` ⟨*factor*⟩
| ⟨*term*⟩ `/` ⟨*factor*⟩
| ⟨*factor*⟩

⟨*factor*⟩ :: `(` ⟨*expression*⟩ `)`
| integer
| identifier

⟨*locations*⟩ :: ⟨*location*⟩
⟨*locations_tail*⟩

⟨*locations_tail*⟩ :: ϵ
| ⟨*location*⟩ ⟨*locations_tail*⟩

⟨*location*⟩ :: `location` identifier `:` ⟨*transitions*⟩

⟨*transitions*⟩ :: ϵ
| ⟨*transition*⟩ ⟨*transitions*⟩

⟨*transition*⟩ :: ⟨*predicates*⟩ `,` ⟨*label*⟩ `,` ⟨*assignments*⟩ `,` identifier `;`

⟨*label*⟩ :: `none`
| identifier

⟨*predicates*⟩ :: `{` ⟨*predicate_list*⟩ `}`

⟨*predicate_list*⟩ :: ϵ
| ⟨*predicate*⟩ ⟨*predicate_list_tail*⟩

⟨*predicate_list_tail*⟩ :: ϵ
| `,` ⟨*predicate*⟩ ⟨*predicate_list_tail*⟩

⟨*predicate*⟩ :: ⟨*expression*⟩ ⟨*comparator*⟩ ⟨*expression*⟩

⟨*comparator*⟩ :: `=`
| `<=`
| `>=`

⟨*decoration*⟩ :: `decoration` identifier ⟨*decoration_items*⟩ `end`

⟨*decoration_items*⟩ :: ϵ
| ⟨*decoration_item*⟩ ⟨*decoration_items*⟩

⟨*decoration_item*⟩ :: `global` ⟨*fragment*⟩
| `startup` ⟨*fragment*⟩
| `cleanup` ⟨*fragment*⟩
| `event` identifier ⟨*fragment*⟩ ⟨*fragment*⟩ ⟨*fragment*⟩
| `order` identifier ⟨*fragment*⟩ ⟨*fragment*⟩
| `internal` identifier ⟨*fragment*⟩ ⟨*fragment*⟩
| `reading` identifier ⟨*fragment*⟩
| `writing` identifier `(` identifier `)` ⟨*fragment*⟩
| identifier `to` identifier ⟨*fragment*⟩ ⟨*fragment*⟩
| `any to` identifier ⟨*fragment*⟩ ⟨*fragment*⟩
| identifier `to any` ⟨*fragment*⟩ ⟨*fragment*⟩
| `any to any` ⟨*fragment*⟩ ⟨*fragment*⟩
| `restrict` ⟨*restriction_header*⟩ ⟨*fragment*⟩

⟨*restriction_header*⟩ :: identifier `to` identifier
| `any to` identifier
| identifier `to any`
| `any to any`

⟨*fragment*⟩ :: `nop`
| `{%` C code `%}`

Appendix B

Example of SPECTRE Output

The following code is what SPECTRE generates when run on the "receiver" input file given in the previous chapter.

```
/* RTAI includes ---------------------------------------- */
#include <linux/module.h>
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_sem.h>
#include <rtai_fifos.h>

/* user defines ----------------------------------------- */
#define TIME_UNIT   <define this>
#define WIDENER     <define this>
#define PERIOD      <define this>
#define BEAT        <define this>

/* system defines --------------------------------------- */
#define STACK_SIZE 10000
#define USE_FPU 1
#define TASK_PRIORITY 128       /* 0 = highest, 255 = lowest */

/* global decoration ------------------------------------ */

#define DATA_REGISTER   0x378
#define CTRL_REGISTER   0x37A
#define GOOD_INDEX      (0 <= data_index && data_index < data_length-1)
const int data_length = 128;
int var_m, data_index, data[data_length];
unsigned char last_val;
void print_received_bits(void) {
```

```
    int i;
    if (GOOD_INDEX)
        for (i = 0; i < data_index - 1; i++)
            rt_printk("%d\n", data[i]);
    else
        rt_printk("ERROR : received more bits than buffer length...\n");
}


/* spectre gobals ------------------------------------- */
RTIME spectre_clock_y;

int spectre_equals(RTIME a, RTIME b);
int spectre_largerthan(RTIME a, RTIME other);
int spectre_smallerthan(RTIME a, RTIME other);
RT_TASK spectre_task;
void spectre_task_main(int data);
void (*spectre_current_location) (void);
RTIME spectre_current_time;
int spectre_running;
void spectre_check_for_events(void);
int spectre_up_is_pending = 0;


/* variables readings headers ----------------------------- */
int spectre_read_m(void);
int spectre_read_r(void);

/* clocks readings headers -------------------------------- */
RTIME spectre_read_y(void);


/* variables writings headers ----------------------------- */
void spectre_write_m(int value);
void spectre_write_r(int value);

/* clocks writings headers -------------------------------- */
void spectre_write_y(RTIME value);


/* orders headers ----------------------------------------- */

/* internals headers -------------------------------------- */
```

```
/* events headers ---------------------------------------- */
int spectre_poll_up(void);
void spectre_d1_up(void);
void spectre_d2_up(void);

/* location headers -------------------------------------- */
void spectre_location_Idle(void);
void spectre_location_LastIsOne(void);
void spectre_location_LastIsZero(void);

/* transitions headers ----------------------------------- */
void spectre_transition_initially(void);
void spectre_transition_Idle_to_LastIsOne_1(void);
void spectre_transition_LastIsOne_to_LastIsOne_1(void);
void spectre_transition_LastIsOne_to_LastIsOne_2(void);
void spectre_transition_LastIsOne_to_LastIsZero_3(void);
void spectre_transition_LastIsOne_to_Idle_4(void);
void spectre_transition_LastIsOne_to_Idle_5(void);
void spectre_transition_LastIsZero_to_LastIsZero_1(void);
void spectre_transition_LastIsZero_to_LastIsOne_2(void);
void spectre_transition_LastIsZero_to_Idle_3(void);

/* updates headers --------------------------------------- */
void spectre_update_initially(void);
void spectre_update_Idle_to_LastIsOne_1(void);
void spectre_update_LastIsOne_to_LastIsOne_1(void);
void spectre_update_LastIsOne_to_LastIsOne_2(void);
void spectre_update_LastIsOne_to_LastIsZero_3(void);
void spectre_update_LastIsOne_to_Idle_4(void);
void spectre_update_LastIsOne_to_Idle_5(void);
void spectre_update_LastIsZero_to_LastIsZero_1(void);
void spectre_update_LastIsZero_to_LastIsOne_2(void);
void spectre_update_LastIsZero_to_Idle_3(void);

/* guards headers ---------------------------------------- */
int spectre_guard_Idle_to_LastIsOne_1(void);
int spectre_guard_LastIsOne_to_LastIsOne_1(void);
int spectre_guard_LastIsOne_to_LastIsOne_2(void);
int spectre_guard_LastIsOne_to_LastIsZero_3(void);
int spectre_guard_LastIsOne_to_Idle_4(void);
int spectre_guard_LastIsOne_to_Idle_5(void);
int spectre_guard_LastIsZero_to_LastIsZero_1(void);
```

```
int spectre_guard_LastIsZero_to_LastIsOne_2(void);
int spectre_guard_LastIsZero_to_Idle_3(void);

/* checks headers ---------------------------------------- */
int spectre_check_Idle_to_LastIsOne_1(void);
int spectre_check_LastIsOne_to_LastIsOne_1(void);
int spectre_check_LastIsOne_to_LastIsOne_2(void);
int spectre_check_LastIsOne_to_LastIsZero_3(void);
int spectre_check_LastIsOne_to_Idle_4(void);
int spectre_check_LastIsOne_to_Idle_5(void);
int spectre_check_LastIsZero_to_LastIsZero_1(void);
int spectre_check_LastIsZero_to_LastIsOne_2(void);
int spectre_check_LastIsZero_to_Idle_3(void);

/* decorations headers ----------------------------------- */
void spectre_d1_Idle_to_LastIsOne(void);
void spectre_d1_LastIsOne_to_Idle(void);
void spectre_d1_LastIsOne_to_LastIsOne(void);
void spectre_d1_LastIsOne_to_LastIsZero(void);
void spectre_d1_LastIsZero_to_Idle(void);
void spectre_d1_LastIsZero_to_LastIsOne(void);
void spectre_d1_LastIsZero_to_LastIsZero(void);

void spectre_d2_Idle_to_LastIsOne(void);
void spectre_d2_LastIsOne_to_Idle(void);
void spectre_d2_LastIsOne_to_LastIsOne(void);
void spectre_d2_LastIsOne_to_LastIsZero(void);
void spectre_d2_LastIsZero_to_Idle(void);
void spectre_d2_LastIsZero_to_LastIsOne(void);
void spectre_d2_LastIsZero_to_LastIsZero(void);

void spectre_d1_any_to_Idle(void);
void spectre_d1_any_to_LastIsOne(void);
void spectre_d1_any_to_LastIsZero(void);

void spectre_d2_any_to_Idle(void);
void spectre_d2_any_to_LastIsOne(void);
void spectre_d2_any_to_LastIsZero(void);

void spectre_d1_Idle_to_any(void);
void spectre_d1_LastIsOne_to_any(void);
void spectre_d1_LastIsZero_to_any(void);
```

```
void spectre_d2_Idle_to_any(void);
void spectre_d2_LastIsOne_to_any(void);
void spectre_d2_LastIsZero_to_any(void);

void spectre_d1_any_to_any(void);
void spectre_d2_any_to_any(void);


/* restrictions headers ----------------------------------- */
int spectre_r_Idle_to_LastIsOne(void);
int spectre_r_LastIsOne_to_Idle(void);
int spectre_r_LastIsOne_to_LastIsOne(void);
int spectre_r_LastIsOne_to_LastIsZero(void);
int spectre_r_LastIsZero_to_Idle(void);
int spectre_r_LastIsZero_to_LastIsOne(void);
int spectre_r_LastIsZero_to_LastIsZero(void);

int spectre_r_any_to_Idle(void);
int spectre_r_any_to_LastIsOne(void);
int spectre_r_any_to_LastIsZero(void);

int spectre_r_Idle_to_any(void);
int spectre_r_LastIsOne_to_any(void);
int spectre_r_LastIsZero_to_any(void);

int spectre_r_any_to_any(void);

/* variables readings implementation ---------------------- */
int spectre_read_m() { return var_m; }
int spectre_read_r() { return 0; }

/* clocks readings implementation ------------------------- */
RTIME spectre_read_y(void) {
    return spectre_current_time - spectre_clock_y;
}


/* variables writings implementation ---------------------- */
void spectre_write_m(int val) { var_m = val; }
void spectre_write_r(int val) {
    if (!GOOD_INDEX) return;
    switch (val) {
        case 0: case 1: data[data_index++] = val; break;
```

```
        case 2: data[data_index++] = 0; data[data_index++] = 1;
    }
}

/* clocks writings implementation ------------------------- */
void spectre_write_y(RTIME newtime) {
    spectre_clock_y = spectre_current_time - newtime;
}

/* decorations implementation ----------------------------- */
void spectre_d1_Idle_to_LastIsOne() {;}
void spectre_d1_LastIsOne_to_Idle() {;}
void spectre_d1_LastIsOne_to_LastIsOne() {;}
void spectre_d1_LastIsOne_to_LastIsZero() {;}
void spectre_d1_LastIsZero_to_Idle() {;}
void spectre_d1_LastIsZero_to_LastIsOne() {;}
void spectre_d1_LastIsZero_to_LastIsZero() {;}
void spectre_d2_Idle_to_LastIsOne() {;}
void spectre_d2_LastIsOne_to_Idle() {;}
void spectre_d2_LastIsOne_to_LastIsOne() {;}
void spectre_d2_LastIsOne_to_LastIsZero() {;}
void spectre_d2_LastIsZero_to_Idle() {;}
void spectre_d2_LastIsZero_to_LastIsOne() {;}
void spectre_d2_LastIsZero_to_LastIsZero() {;}
void spectre_d1_any_to_Idle(void) { spectre_running = 0; }
void spectre_d1_any_to_LastIsOne(void) {;}
void spectre_d1_any_to_LastIsZero(void) {;}
void spectre_d2_any_to_Idle(void) {;}
void spectre_d2_any_to_LastIsOne(void) {;}
void spectre_d2_any_to_LastIsZero(void) {;}
void spectre_d1_Idle_to_any(void) {;}
void spectre_d1_LastIsOne_to_any(void) {;}
void spectre_d1_LastIsZero_to_any(void) {;}
void spectre_d2_Idle_to_any(void) {;}
void spectre_d2_LastIsOne_to_any(void) {;}
void spectre_d2_LastIsZero_to_any(void) {;}
void spectre_d1_any_to_any(void) {;}
void spectre_d2_any_to_any(void) {;}

/* guards implementation ---------------------------------- */
int spectre_guard_Idle_to_LastIsOne_1(void) {
    RTIME spectre_t1, spectre_t2;
    int spectre_e1, spectre_e2;
```

```
    int spectre_res = 1, spectre_tmp;
    return spectre_res;
}

int spectre_guard_LastIsOne_to_LastIsOne_1(void) {
    RTIME spectre_t1, spectre_t2;
    int spectre_e1, spectre_e2;
    int spectre_res = 1, spectre_tmp;
    spectre_t1 = (TIME_UNIT * 3);
    spectre_t2 = spectre_read_y();
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    spectre_t1 = spectre_read_y();
    spectre_t2 = (TIME_UNIT * 5);
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    return spectre_res;
}

int spectre_guard_LastIsOne_to_LastIsOne_2(void) {
    RTIME spectre_t1, spectre_t2;
    int spectre_e1, spectre_e2;
    int spectre_res = 1, spectre_tmp;
    spectre_t1 = (TIME_UNIT * 7);
    spectre_t2 = spectre_read_y();
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    return spectre_res;
}

int spectre_guard_LastIsOne_to_LastIsZero_3(void) {
    RTIME spectre_t1, spectre_t2;
    int spectre_e1, spectre_e2;
    int spectre_res = 1, spectre_tmp;
    spectre_t1 = (TIME_UNIT * 5);
    spectre_t2 = spectre_read_y();
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    spectre_t1 = spectre_read_y();
    spectre_t2 = (TIME_UNIT * 7);
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    return spectre_res;
```

```
}

int spectre_guard_LastIsOne_to_Idle_4(void) {
    RTIME spectre_t1, spectre_t2;
    int spectre_e1, spectre_e2;
    int spectre_res = 1, spectre_tmp;
    spectre_t1 = (TIME_UNIT * 9);
    spectre_t2 = spectre_read_y();
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    spectre_e1 = spectre_read_m();
    spectre_e2 = 0;
    spectre_tmp = spectre_e1 == spectre_e2;
    spectre_res = spectre_tmp && spectre_res;
    return spectre_res;
}

int spectre_guard_LastIsOne_to_Idle_5(void) {
    RTIME spectre_t1, spectre_t2;
    int spectre_e1, spectre_e2;
    int spectre_res = 1, spectre_tmp;
    spectre_t1 = (TIME_UNIT * 9);
    spectre_t2 = spectre_read_y();
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    spectre_e1 = spectre_read_m();
    spectre_e2 = 1;
    spectre_tmp = spectre_e1 == spectre_e2;
    spectre_res = spectre_tmp && spectre_res;
    return spectre_res;
}

int spectre_guard_LastIsZero_to_LastIsZero_1(void) {
    RTIME spectre_t1, spectre_t2;
    int spectre_e1, spectre_e2;
    int spectre_res = 1, spectre_tmp;
    spectre_t1 = (TIME_UNIT * 3);
    spectre_t2 = spectre_read_y();
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    spectre_t1 = spectre_read_y();
    spectre_t2 = (TIME_UNIT * 5);
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
```

```
    spectre_res = spectre_tmp && spectre_res;
    return spectre_res;
}

int spectre_guard_LastIsZero_to_LastIsOne_2(void) {
    RTIME spectre_t1, spectre_t2;
    int spectre_e1, spectre_e2;
    int spectre_res = 1, spectre_tmp;
    spectre_t1 = (TIME_UNIT * 5);
    spectre_t2 = spectre_read_y();
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    spectre_t1 = spectre_read_y();
    spectre_t2 = (TIME_UNIT * 7);
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    return spectre_res;
}

int spectre_guard_LastIsZero_to_Idle_3(void) {
    RTIME spectre_t1, spectre_t2;
    int spectre_e1, spectre_e2;
    int spectre_res = 1, spectre_tmp;
    spectre_t1 = (TIME_UNIT * 7);
    spectre_t2 = spectre_read_y();
    spectre_tmp = spectre_smallerthan(spectre_t1, spectre_t2);
    spectre_res = spectre_tmp && spectre_res;
    return spectre_res;
}

/* restrictions implementation --------------------------- */
int spectre_r_Idle_to_LastIsOne(void) { return 1; }
int spectre_r_LastIsOne_to_Idle(void) { return 1; }
int spectre_r_LastIsOne_to_LastIsOne(void) { return 1; }
int spectre_r_LastIsOne_to_LastIsZero(void) { return 1; }
int spectre_r_LastIsZero_to_Idle(void) { return 1; }
int spectre_r_LastIsZero_to_LastIsOne(void) { return 1; }
int spectre_r_LastIsZero_to_LastIsZero(void) { return 1; }
int spectre_r_any_to_Idle(void) { return 1; }
int spectre_r_any_to_LastIsOne(void) { return 1; }
int spectre_r_any_to_LastIsZero(void) { return 1; }
int spectre_r_Idle_to_any(void) { return 1; }
int spectre_r_LastIsOne_to_any(void) { return 1; }
```

```
int spectre_r_LastIsZero_to_any(void) { return 1; }
int spectre_r_any_to_any(void) { return 1; }

/* updates implementation ------------------------------- */
void spectre_update_initially(void) {
    spectre_write_y((TIME_UNIT * 0));
    spectre_write_m(0);
}

void spectre_update_Idle_to_LastIsOne_1(void) {
    spectre_write_y((TIME_UNIT * 0));
    spectre_write_m(0);
}

void spectre_update_LastIsOne_to_LastIsOne_1(void) {
    spectre_write_y((TIME_UNIT * 0));
    spectre_write_m(1 - spectre_read_m());
    spectre_write_r(1);
}

void spectre_update_LastIsOne_to_LastIsOne_2(void) {
    spectre_write_y((TIME_UNIT * 0));
    spectre_write_r(2);
}

void spectre_update_LastIsOne_to_LastIsZero_3(void) {
    spectre_write_y((TIME_UNIT * 0));
    spectre_write_m(1 - spectre_read_m());
    spectre_write_r(0);
}

void spectre_update_LastIsOne_to_Idle_4(void) {
    spectre_write_r(0);
}

void spectre_update_LastIsOne_to_Idle_5(void) {
}
void spectre_update_LastIsZero_to_LastIsZero_1(void) {
    spectre_write_y((TIME_UNIT * 0));
    spectre_write_m(1 - spectre_read_m());
    spectre_write_r(0);
}
```

```
void spectre_update_LastIsZero_to_LastIsOne_2(void) {
    spectre_write_y((TIME_UNIT * 0));
    spectre_write_r(2);
}

void spectre_update_LastIsZero_to_Idle_3(void) {
    spectre_write_r(0);
}

/* orders implementation ---------------------------------- */

/* internals implementation ------------------------------- */

/* events implementation ---------------------------------- */
void spectre_d1_up(void) {;}
void spectre_d2_up(void) {;}
int spectre_poll_up(void) {
    unsigned char val = inb(DATA_REGISTER);
    return val == last_val ? 0 : (last_val = val);
}

/* checks implementation ---------------------------------- */
int spectre_check_Idle_to_LastIsOne_1(void) {
    int spectre_r1, spectre_r2, spectre_r3, spectre_r4, spectre_g;
    spectre_r1 = spectre_r_Idle_to_LastIsOne();
    spectre_r2 = spectre_r_Idle_to_any();
    spectre_r3 = spectre_r_any_to_LastIsOne();
    spectre_r4 = spectre_r_any_to_any();
    spectre_g = spectre_guard_Idle_to_LastIsOne_1();
    return (spectre_g && spectre_r1 && spectre_r2 && spectre_r3
            && spectre_r4 && (spectre_up_is_pending == 1));
}

int spectre_check_LastIsOne_to_LastIsOne_1(void) {
    int spectre_r1, spectre_r2, spectre_r3, spectre_r4, spectre_g;
    spectre_r1 = spectre_r_LastIsOne_to_LastIsOne();
    spectre_r2 = spectre_r_LastIsOne_to_any();
    spectre_r3 = spectre_r_any_to_LastIsOne();
    spectre_r4 = spectre_r_any_to_any();
    spectre_g = spectre_guard_LastIsOne_to_LastIsOne_1();
    return (spectre_g && spectre_r1 && spectre_r2 && spectre_r3
            && spectre_r4 && (spectre_up_is_pending == 1));
}
```

```
int spectre_check_LastIsOne_to_LastIsOne_2(void) {
    int spectre_r1, spectre_r2, spectre_r3, spectre_r4, spectre_g;
    spectre_r1 = spectre_r_LastIsOne_to_LastIsOne();
    spectre_r2 = spectre_r_LastIsOne_to_any();
    spectre_r3 = spectre_r_any_to_LastIsOne();
    spectre_r4 = spectre_r_any_to_any();
    spectre_g = spectre_guard_LastIsOne_to_LastIsOne_2();
    return (spectre_g && spectre_r1 && spectre_r2 && spectre_r3
            && spectre_r4 && (spectre_up_is_pending == 1));
}

int spectre_check_LastIsOne_to_LastIsZero_3(void) {
    int spectre_r1, spectre_r2, spectre_r3, spectre_r4, spectre_g;
    spectre_r1 = spectre_r_LastIsOne_to_LastIsZero();
    spectre_r2 = spectre_r_LastIsOne_to_any();
    spectre_r3 = spectre_r_any_to_LastIsZero();
    spectre_r4 = spectre_r_any_to_any();
    spectre_g = spectre_guard_LastIsOne_to_LastIsZero_3();
    return (spectre_g && spectre_r1 && spectre_r2 && spectre_r3
            && spectre_r4 && (spectre_up_is_pending == 1));
}

int spectre_check_LastIsOne_to_Idle_4(void) {
    int spectre_r1, spectre_r2, spectre_r3, spectre_r4, spectre_g;
    spectre_r1 = spectre_r_LastIsOne_to_Idle();
    spectre_r2 = spectre_r_LastIsOne_to_any();
    spectre_r3 = spectre_r_any_to_Idle();
    spectre_r4 = spectre_r_any_to_any();
    spectre_g = spectre_guard_LastIsOne_to_Idle_4();
    return (spectre_g && spectre_r1 && spectre_r2 && spectre_r3
            && spectre_r4);
}

int spectre_check_LastIsOne_to_Idle_5(void) {
    int spectre_r1, spectre_r2, spectre_r3, spectre_r4, spectre_g;
    spectre_r1 = spectre_r_LastIsOne_to_Idle();
    spectre_r2 = spectre_r_LastIsOne_to_any();
    spectre_r3 = spectre_r_any_to_Idle();
    spectre_r4 = spectre_r_any_to_any();
    spectre_g = spectre_guard_LastIsOne_to_Idle_5();
    return (spectre_g && spectre_r1 && spectre_r2 && spectre_r3
            && spectre_r4);
```

```
}

int spectre_check_LastIsZero_to_LastIsZero_1(void) {
    int spectre_r1, spectre_r2, spectre_r3, spectre_r4, spectre_g;
    spectre_r1 = spectre_r_LastIsZero_to_LastIsZero();
    spectre_r2 = spectre_r_LastIsZero_to_any();
    spectre_r3 = spectre_r_any_to_LastIsZero();
    spectre_r4 = spectre_r_any_to_any();
    spectre_g = spectre_guard_LastIsZero_to_LastIsZero_1();
    return (spectre_g && spectre_r1 && spectre_r2 && spectre_r3
            && spectre_r4 && (spectre_up_is_pending == 1));
}

int spectre_check_LastIsZero_to_LastIsOne_2(void) {
    int spectre_r1, spectre_r2, spectre_r3, spectre_r4, spectre_g;
    spectre_r1 = spectre_r_LastIsZero_to_LastIsOne();
    spectre_r2 = spectre_r_LastIsZero_to_any();
    spectre_r3 = spectre_r_any_to_LastIsOne();
    spectre_r4 = spectre_r_any_to_any();
    spectre_g = spectre_guard_LastIsZero_to_LastIsOne_2();
    return (spectre_g && spectre_r1 && spectre_r2 && spectre_r3
            && spectre_r4 && (spectre_up_is_pending == 1));
}

int spectre_check_LastIsZero_to_Idle_3(void) {
    int spectre_r1, spectre_r2, spectre_r3, spectre_r4, spectre_g;
    spectre_r1 = spectre_r_LastIsZero_to_Idle();
    spectre_r2 = spectre_r_LastIsZero_to_any();
    spectre_r3 = spectre_r_any_to_Idle();
    spectre_r4 = spectre_r_any_to_any();
    spectre_g = spectre_guard_LastIsZero_to_Idle_3();
    return (spectre_g && spectre_r1 && spectre_r2 && spectre_r3
            && spectre_r4);
}

/* transitions implementation ----------------------------- */
void spectre_transition_initially(void) {
    spectre_update_initially();
    spectre_current_location = spectre_location_Idle;
}

void spectre_transition_Idle_to_LastIsOne_1(void) {
    spectre_d1_Idle_to_LastIsOne();
```

```
    spectre_d1_Idle_to_any();
    spectre_d1_any_to_LastIsOne();
    spectre_d1_any_to_any();
    spectre_d1_up();
    spectre_update_Idle_to_LastIsOne_1();
    spectre_current_location = spectre_location_LastIsOne;
    spectre_d2_up();
    spectre_up_is_pending = 0;
    spectre_d2_Idle_to_LastIsOne();
    spectre_d2_Idle_to_any();
    spectre_d2_any_to_LastIsOne();
    spectre_d2_any_to_any();
}

void spectre_transition_LastIsOne_to_LastIsOne_1(void) {
    spectre_d1_LastIsOne_to_LastIsOne();
    spectre_d1_LastIsOne_to_any();
    spectre_d1_any_to_LastIsOne();
    spectre_d1_any_to_any();
    spectre_d1_up();
    spectre_update_LastIsOne_to_LastIsOne_1();
    spectre_current_location = spectre_location_LastIsOne;
    spectre_d2_up();
    spectre_up_is_pending = 0;
    spectre_d2_LastIsOne_to_LastIsOne();
    spectre_d2_LastIsOne_to_any();
    spectre_d2_any_to_LastIsOne();
    spectre_d2_any_to_any();
}

void spectre_transition_LastIsOne_to_LastIsOne_2(void) {
    spectre_d1_LastIsOne_to_LastIsOne();
    spectre_d1_LastIsOne_to_any();
    spectre_d1_any_to_LastIsOne();
    spectre_d1_any_to_any();
    spectre_d1_up();
    spectre_update_LastIsOne_to_LastIsOne_2();
    spectre_current_location = spectre_location_LastIsOne;
    spectre_d2_up();
    spectre_up_is_pending = 0;
    spectre_d2_LastIsOne_to_LastIsOne();
    spectre_d2_LastIsOne_to_any();
    spectre_d2_any_to_LastIsOne();
```

```
    spectre_d2_any_to_any();
}

void spectre_transition_LastIsOne_to_LastIsZero_3(void) {
    spectre_d1_LastIsOne_to_LastIsZero();
    spectre_d1_LastIsOne_to_any();
    spectre_d1_any_to_LastIsZero();
    spectre_d1_any_to_any();
    spectre_d1_up();
    spectre_update_LastIsOne_to_LastIsZero_3();
    spectre_current_location = spectre_location_LastIsZero;
    spectre_d2_up();
    spectre_up_is_pending = 0;
    spectre_d2_LastIsOne_to_LastIsZero();
    spectre_d2_LastIsOne_to_any();
    spectre_d2_any_to_LastIsZero();
    spectre_d2_any_to_any();
}

void spectre_transition_LastIsOne_to_Idle_4(void) {
    spectre_d1_LastIsOne_to_Idle();
    spectre_d1_LastIsOne_to_any();
    spectre_d1_any_to_Idle();
    spectre_d1_any_to_any();
    spectre_update_LastIsOne_to_Idle_4();
    spectre_current_location = spectre_location_Idle;
    spectre_d2_LastIsOne_to_Idle();
    spectre_d2_LastIsOne_to_any();
    spectre_d2_any_to_Idle();
    spectre_d2_any_to_any();
}

void spectre_transition_LastIsOne_to_Idle_5(void) {
    spectre_d1_LastIsOne_to_Idle();
    spectre_d1_LastIsOne_to_any();
    spectre_d1_any_to_Idle();
    spectre_d1_any_to_any();
    spectre_update_LastIsOne_to_Idle_5();
    spectre_current_location = spectre_location_Idle;
    spectre_d2_LastIsOne_to_Idle();
    spectre_d2_LastIsOne_to_any();
    spectre_d2_any_to_Idle();
    spectre_d2_any_to_any();
```

```
}

void spectre_transition_LastIsZero_to_LastIsZero_1(void) {
    spectre_d1_LastIsZero_to_LastIsZero();
    spectre_d1_LastIsZero_to_any();
    spectre_d1_any_to_LastIsZero();
    spectre_d1_any_to_any();
    spectre_d1_up();
    spectre_update_LastIsZero_to_LastIsZero_1();
    spectre_current_location = spectre_location_LastIsZero;
    spectre_d2_up();
    spectre_up_is_pending = 0;
    spectre_d2_LastIsZero_to_LastIsZero();
    spectre_d2_LastIsZero_to_any();
    spectre_d2_any_to_LastIsZero();
    spectre_d2_any_to_any();
}

void spectre_transition_LastIsZero_to_LastIsOne_2(void) {
    spectre_d1_LastIsZero_to_LastIsOne();
    spectre_d1_LastIsZero_to_any();
    spectre_d1_any_to_LastIsOne();
    spectre_d1_any_to_any();
    spectre_d1_up();
    spectre_update_LastIsZero_to_LastIsOne_2();
    spectre_current_location = spectre_location_LastIsOne;
    spectre_d2_up();
    spectre_up_is_pending = 0;
    spectre_d2_LastIsZero_to_LastIsOne();
    spectre_d2_LastIsZero_to_any();
    spectre_d2_any_to_LastIsOne();
    spectre_d2_any_to_any();
}

void spectre_transition_LastIsZero_to_Idle_3(void) {
    spectre_d1_LastIsZero_to_Idle();
    spectre_d1_LastIsZero_to_any();
    spectre_d1_any_to_Idle();
    spectre_d1_any_to_any();
    spectre_update_LastIsZero_to_Idle_3();
    spectre_current_location = spectre_location_Idle;
    spectre_d2_LastIsZero_to_Idle();
    spectre_d2_LastIsZero_to_any();
```

```
    spectre_d2_any_to_Idle();
    spectre_d2_any_to_any();
}

/* location implementation ------------------------------- */
void spectre_location_Idle(void) {
    if (spectre_check_Idle_to_LastIsOne_1()) {
        spectre_transition_Idle_to_LastIsOne_1();
        return;
    }
}

void spectre_location_LastIsOne(void)
{
    if (spectre_check_LastIsOne_to_LastIsOne_1()) {
        spectre_transition_LastIsOne_to_LastIsOne_1();
        return;
    }
    if (spectre_check_LastIsOne_to_LastIsOne_2()) {
        spectre_transition_LastIsOne_to_LastIsOne_2();
        return;
    }
    if (spectre_check_LastIsOne_to_LastIsZero_3()) {
        spectre_transition_LastIsOne_to_LastIsZero_3();
        return;
    }
    if (spectre_check_LastIsOne_to_Idle_4()) {
        spectre_transition_LastIsOne_to_Idle_4();
        return;
    }
    if (spectre_check_LastIsOne_to_Idle_5()) {
        spectre_transition_LastIsOne_to_Idle_5();
        return;
    }
}

void spectre_location_LastIsZero(void)
{
    if (spectre_check_LastIsZero_to_LastIsZero_1()) {
        spectre_transition_LastIsZero_to_LastIsZero_1();
        return;
    }
    if (spectre_check_LastIsZero_to_LastIsOne_2()) {
```

```
        spectre_transition_LastIsZero_to_LastIsOne_2();
        return;
    }
    if (spectre_check_LastIsZero_to_Idle_3()) {
        spectre_transition_LastIsZero_to_Idle_3();
        return;
    }
}

void spectre_check_for_events(void)
{
    if ((spectre_up_is_pending == 0) && spectre_poll_up())
        spectre_up_is_pending = 1;
}

void spectre_cleanup(void) {
    print_received_bits();
}

void spectre_startup(void) {
    outb(32, CTRL_REGISTER);
    last_val = 0;
    data_index = 0;
}

int spectre_equals(RTIME a, RTIME b) {
    RTIME lower = b - WIDENER;
    RTIME higher = b + WIDENER;
    return (a > lower && a < higher);
}

int spectre_largerthan(RTIME a, RTIME other) {
    other -= WIDENER;
    return (a > other);
}

int spectre_smallerthan(RTIME a, RTIME other) {
    other += WIDENER;
    return (a < other);
}

/* main function implementation -------------------------- */
void spectre_task_main(int data) {
```

```
    spectre_running = 1;
    spectre_startup();
    spectre_transition_initially();
    while (spectre_running == 1) {
        spectre_current_time = rt_get_time();
        spectre_check_for_events();
        spectre_current_location();
        rt_task_wait_period();
    }
    spectre_cleanup();
}

/* kernel module functions ------------------------------- */
int spectre_init_module(void) {
    rt_set_periodic_mode();
    start_rt_timer(BEAT);
    rt_task_init(&spectre_task, spectre_task_main, 0, STACK_SIZE,
                 TASK_PRIORITY, USE_FPU, 0);
    rt_task_make_periodic(&spectre_task, rt_get_time() + PERIOD, PERIOD);
    return 0;
}

void spectre_cleanup_module(void) {
    stop_rt_timer();
    rt_task_delete(&spectre_task);
}

module_init(spectre_init_module);
module_exit(spectre_cleanup_module);
MODULE_LICENSE("GPL");
```

Bibliography

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

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1986.

[BLL+98] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New Generation of UPPAAL. *Proceedings of the International Workshop on Software Tools for Technology Transfer, Aalborg, Denmark, July*, 1998.

[BY04] J. Bengtsson and W. YI. Timed automata: Semantics, algorithms and tools. *In Lecture Notes on Concurrency and Petri Nets. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag*, 2004.

[DDMR04] M. De Wulf, L. Doyen, N. Markey, and J.F. Raskin. Robustness and implementability of timed automata. *Proceedings of FORMATS-FTRTFT*, pages 118–133, 2004.

[DDR05a] M. De Wulf, L. Doyen, and JF Raskin. Systematic implementation of real-time models (extended version). Technical Report 543, ULB, 2005. http://www.ulb.ac.be/di/publications/.

[DDR05b] Martin De Wulf, Laurent Doyen, and Jean-François Raskin. Almost ASAP semantics: from timed models to timed implementations. *Formal Aspects of Computing*, 17(3):319–341, 2005.

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

[DPB00] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel*. O'Reilly, October 2000.

[Fre05] Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *HSCC*, pages 258–273, 2005.

[Hen96] Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[HHW97] T.A. Henzinger, P.H. Ho, and H. Wong-Toi. HYTECH: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):110–122, 1997.

[HPF99] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98. http://www.haskell.org/tutorial/, 1999.

[LML04] Luca Abeni Luca Marzario and Giuseppe Lipari. Improving the responsiveness of linux applications in rtai. http://www.ocera.net/doc/, 2004.

[MDP00] P. Mantegazza, EL Dozio, and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux Journal*, 2000(72es), 2000.