Implémentation rigoureuse des systèmes temps-réels

Tesnim Abdellatif

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Tesnim Abdellatif

Thèse dirigée par **Prof. Joseph Sifakis**
et codirigée par **Dr. Jacques Combaz**

préparée au sein **du laboratoire VERIMAG**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique.**

Rigorous Implementation of Real-time Systems

Thèse soutenue publiquement le **Juin 2012** ,
devant le jury composé de :

Mr. Roland Groz
Professor, INPG, Président
Mr. Sanjoy Baruah
Professor, University of North Carolina, Rapporteur
Mr. Eugene Asarin
Professor, Université Paris Diderot–Paris 7, Rapporteur
Mr. Félix Ingrand
Doctor, LAAS/CNRS, Examinateur
Mr. Wang Yi
Professor, Uppsala University, Examinateur
Mr. Joseph Sifakis
Professor, CNRS, Directeur de thèse
Mr. Jacques Combaz
Doctor, CNRS, Co-Directeur de thèse

Abstract

Real-time systems are systems that are subject to "real-time constraints"— e.g. operational deadlines from event to system response. Often real-time response times are understood to be in the order of milliseconds and sometimes microseconds. Building real-time systems requires the use of design and implementation methodologies that ensure the property of meeting timing constraints e.g. a system has to react within user-defined bounds such as deadlines and periodicity. A missed deadline in hard real-time systems is catastrophic, like for example in automotive systems, for example if an airbag is triggered too late in a car accident, even one ms too late leads to serious repercussions. In soft real-time systems it can lead to a significant loss of performance and QoS like for example in networked multimedia systems.

We provide a rigorous design and implementation method for the implementation of real-time systems. The implementation is generated from a given real-time application software and a target platform by using two models:

- An abstract model representing the behavior of real-time software as a timed automaton. The latter describes user-defined platform-independent timing constraints. Its transitions are timeless and correspond to the execution of statements of the real-time software.
- A physical model representing the behavior of the real-time software running on a given platform. It is obtained by assigning execution times to the transitions of the abstract model.

A necessary condition for implementability is time-safety, that is, any (timed) execution sequence of the physical model is also an execution sequence of the abstract model. Time-safety means that the platform is fast enough to meet the timing requirements. As execution times of actions are not known exactly, time-safety is checked for worst-case execution times of actions by making an assumption of time-robustness: time-safety is preserved when speed of the execution platform increases. For given real-time software and execution platform corresponding to a time-robust model, we define an execution engine that coordinates the execution of the application software so as to meet its timing constraints. Furthermore, in case of non-robustness, the execution engine can detect violations of time-safety and stop execution. We have implemented the execution Engine for BIP programs with real-time constraints. We have validated the method for the design and implementation of the Dala rover robot. We show the benefits obtained in terms of CPU utilization and amelioration in the latency of reaction.

Acknowledgements

I would like to thank the jury members, especially Prof Sanjoy Baruah and Prof Eugene Asarin for evaluating my dissertation and for their feedbacks. I also thank Prof Wang Yi and Dr Felix Ingrand for examining my work. I thank Prof Rolang Groz for accepting to chair the defense session.

I express all my gratitude to my advisor Prof Joseph Sifakis, who has given me the chance to do my PhD in his team. I learnt a lot from his knowledge and his feedbacks, and I thank him for his support and advices. I am thankful to Dr Jacques Combaz with his precious help. He has given me a great support to accomplish the work, in implementation and writing. I thank him for the time he spend to give me helpful explanations and discussions.

In addition, I thank Prof Saddek Ben Salem from Verimag for giving me the opportunity to work with him on the autonomous systems design project. I thank him for his encouragements and the trust he gave me to fulfill my ideas. I also thank Dr Marc Poulhies for his help in technical aspects, and Prof Marius Bozga and Dr Ananda Basu for their help with the BIP framework. I also thank again Dr Felix Ingrand and Lavindra De Silva from the LAAS laboratory for the interactions we had. I am thankful to the Magillem team, especially Dr. Guillaume Godet Bar for his support. I am also thankful for the implementation effort carried out by the intern Melek Charfi. Furthermore, It was a real pleasure to work in the Verimag laboratory in the DCS team, a very special "thank you" goes to my friends Jean, Emmanuel, Paris, Vasso and Borzoo for their support and for the special times we were talking, taking coffee breaks and even the share of everyday life moments. I also thank Ahlem, Ayoub, Wiem and Wajdi for the moments we spend together talking in the caffet.

I am happy to have great friends that motivated and encouraged me all these wonderful years. I thank the Salsa Team for their joy of living, Audrey, Sarah, Remi, Karen, Dushi, Alex, Olivier and Marion. I also thank Mona, Dali, Jihene, Maro, Marwen, Safe, Imene and all the other members of the nice group for their support and their presence in my life. I thank Lydia for her friendship during all the years I spent in Grenoble. A special thank to Greg for his support and his friendship since the beginning of our studiest It is also a pleasure to thank my friend of always Charbel through all our university studies. Thank you for being here for me, for the great and even the difficult moments. I also thank my wonderful friend Karima, that shared with me the joys of childhood, for her help and her presence in my life. I would like to thank some special friends who contributed in my happiness, Imene, Amani, Layla, Meriem, and many other friends behind the sea.

Finally, I express my gratitude to my family for all what they gave to me. My dear parents for their prayers, the education they gave me and their encouragements to fulfill my ambitions. I thank my brother, Nadir, for making me happy every holidays and my sisters for their help and encouragements. A special thank to takoua, who encouraged me to do a PhD, and Tahia with Antoine for their support, comfort and advise.

Contents

Part I

Context

Chapter 1

Introduction

Computer systems are evolving in all aspects of human life and have become ubiquitous. We can find them in different types of applications from automotive to aeronautic, military, telecommunication, medical and even home automation systems that may include centralized control of lighting, HVAC (heating, ventilation and air conditioning) to provide improved convenience, comfort, energy efficiency and security. We also call these systems embedded systems. An embedded system is some combination of computer hardware and software, either fixed in capability or programmable, that is specifically designed for a particular function. It also continuously interacts with other systems and the physical world. Embedded systems constitute a domain where there is a special need for rigorous design methods, since a failure in a part of the system may have catastrophic consequences on systems performance, security, safety, availability etc. Moreover, systems become more and more complex and their development is increasing exponentially. Although different techniques in software engineering exist for ensuring correctness such as formal verification, simulation, and testing, building correct and reliable systems is still a time-consuming and hardly predictive task. Such methods require formal frameworks to model the system at different design stages, from specification to implementation, and formal techniques to assess its correctness and performance.

Real-time systems are systems that are subject to a "real-time constraint"— e.g. operational deadlines from event to system response. Often real-time response times are understood to be in the order of milliseconds and sometimes microseconds. Building real-time systems requires the use of design and implementation methodologies that ensure the property of meeting timing constraints e.g. a system has to react within user-defined bounds such as deadlines and periodicity. A missed deadline in hard real-time systems is catastrophic, like for example in air traffic control systems or automotive systems. Imagine a car accident and what happens when the airbag is triggered too late, even one ms too late leads to serious repercussions. In soft real-time systems it can lead to a significant loss of performance and QoS like for example in networked multimedia systems. The satisfaction of timing constraints depends on features of the execution platform, in particular its speed and the occurence of stimuli from the external environment.

The component-based design has been also established as an important paradigm for the development of embedded systems. The main principle is that complex systems can be obtained by assembling components (building blocks). Components are systems characterized

by their interface, an abstraction that is adequate for composition and reuse. Composition is used to build complex components by "gluing" together simpler ones. "Gluing" can be seen as an operation that takes in components and their integration constraints, and from these, it provides the description of a new, more complex component.

In this thesis, we provide a rigorous design and implementation methodology, applicable to component-based design, in order to build real-time systems.

1 Challenges for the Design and Implementation of Real-Time Systems

The building process of real-time systems includes in general two essential steps, design and implementation. Rigorous design methodologies are model-based, that is, they explicitly or implicitly associate with a real-time application software an abstract model.

1.1 Modeling

Modeling plays a central role in systems engineering. Usually, modeling techniques are applied at early phases of the system development and at high abstraction layer. Modeling provides advantages such as the ease of construction, the possibility of integrating models of heterogeneous components, generality by using abstraction and behavior nondeterminism, the avoidance of probe effect or disturbances due to experimentation and finally, the possibility of analysis by application of formal methods. Nevertheless, building models that faithfully represent real-time systems is not a trivial task.

The model of a given real-time system is a platform-independent abstraction of the real-time system, expressing timing constraints to be met by the implementation. The model is based on an abstract notion of time. In particular, it assumes that actions, corresponding to the application software computational steps of the system, are atomic and have zero execution times.

An application software is usually written in some high-level programming language. To cope with the complexity of applications, the software is decomposed into components. Conceptually, the programmer reasons in terms of model of computation while developing the software. This model is either explicit in languages with formal semantics or implicitly assumed. This high-level model is based on abstractions about the behavior and interaction of components. Such abstractions include concurrent execution, instantaneous computation, zero delay, and perfect communication between components and/or between components and the external environment, atomicity of actions, and so on. Building faithful models requires a clear understanding of the implementation process and the possibility of relating the application software with its run-time behavior. It is a key issue to discuss the problem of establishing a connection between the application software and its implementation.

1.2 Implementation

Application software must be implemented on a particular platform. Therfore, implementation involves resolving a number of issues not always resolved at application software level, such as ressource allocation (e.g. distribution of tasks, scheduling policies) or task communication and synchronization (e.g. shared memory, semaphores, queues). Implementation compromises the abstractions of the high-level programming model, especially computation

and communication take time. Implementation theory allows deciding if a given application software, i.e. its associated model, can be implemented on a given platform, that is, for particular execution times of actions. Usually, implementability is checked for worst-case execution times by making the assumption that timing constraints will also be met for smaller execution times. This robustness assumption that increasing the speed of the execution platform preserves satisfaction of timing constraints does not always hold as it will be explained in Chapter 4.

1.3 Challenge : from Modeling to Implementation

The gap between application software model and implementation resides in the fact that high-level model of computation is, in general, different from the low-level model of computation. Software is immaterial and, ideally, platform independent; therefore, the high level model often uses a logical time axis. The implementation runs on a platform and interacts with its environment in real-time; thus, the low-level model uses a physical time axis. Since abstractions may collapse during implementation, it is not insured that a real-time system preserves the timing properties of its application software. For example, we may have verified abscence of deadlocks using a high-level model of the application software which assumes actions take zero-time. Nevertheless, the real-time system may have deadlock due to the presence of non zero execution time. Therefore, the challenge is to check that a real-time system implementation is correct with respect to its application software model [83, 84]. This is the correctness problem. To check correctness formally, there are several methods. In general, we first build models of both the application software and the real-time system. Since these models have different time axis (logical versus physical), a correct mapping method should be used to ensure their implementability. A framework must also be developed that relates the two and encompasses a notion of correctness.

2 Our Contribution

We provide a rigorous design and implementation methodology in order to build real-time systems "correct-by-construction". For a given application and a target platform, the principle is as follows.

- We consider that the application software is represented by an abstract model based on timed automata [7]. The model takes into account only platform-independent timing constraints expressing user-dependent requirements. The actions of the model represent statements of the application software and are assumed to be timeless. Using timed automata allows more general timing constraints than logical execution time (LET) (e.g. lower bounds, upper bounds, time non-determinism). The abstract model describes the dynamic behavior of the application software as a set of interacting tasks without restriction on their type (i.e. periodic, sporadic, etc.).
- We introduce a notion of physical model. This model describes the behavior of the abstract model (and thus of the application software) when it is executed on a target platform. It is obtained from the abstract model by assigning to its actions execution times which are upper bounds of the actual execution times for the target platform.
- We provide a rigorous implementation method which from a given physical model (abstract model and given worst case exection times (WCET) for the target platform)

leads under some robustness assumption, to a correct implementation. The method is implemented by a real-time execution engine which respects the semantics of the abstract model (see Figure 1.1). Furthermore, if robustness of models cannot be guaranteed, it checks online if the execution is correct, that is, if timing constraints of the model are met. In addition, it checks violation of essential properties of the abstract model such as deadlock-freedom and consistency of the timing constraints.

More formally, a physical model M_φ is an abstract model M equipped with a function φ assigning execution times to its actions. It represents the behavior of the application software running on a platform. The physical model M_φ is time-safe if all its timed traces are also timed traces of the abstract model. We show that a time-safe physical model may not be time-robust: reducing execution times does not preserve time-safety. A physical model M_φ is called *time-robust* if any physical model $M_{\varphi'}$ is time-safe for all φ' such that $\varphi' \leq \varphi$. We show that non-deterministic models are not time-robust, in general. The implementation of an application software on an execution platform is correct and safe, if the WCET for its actions define a time-robust physical model.

The method also considers open real-time systems, where the behavior of the system depends not only on its internal computations but also on the behavior of the environment. Interactions with the environment are modeled using Input/Output automata, in which actions correspond either to internal computations, or to communications with the environment. Thus, an implementation of an application software depends also on the actual arrival of inputs from the external environment. The application software consists of a set of components modeled as timed input/output automata interacting by rendezvous. An interaction is a set of actions belonging to distinct components that must be synchronized. It can be executed from a given state only if all the involved actions are enabled. We define a real-time execution engine which ensures components coordination by executing interactions. The real-time execution engine proceeds by steps. Each step is the sequential composition of three functions:

- Computing time intervals in which each interaction is enabled, by applying semantics of the abstract model. Time intervals are specified by using a global abstract time variable t.
- Updating the abstract time t by the real time t_r provided by the execution platform, if t_r does not exceed the earliest deadline of the enabled interactions. Otherwise, a time-safety violation is detected and execution stops.
- Scheduling amongst the possible interactions by executing one amongst the most urgent.

We show that our implementation method is correct for time-robust execution time assignments. That is, for time-robust execution time assignments φ, the set of the timed traces computed by the Real-Time Execution engine is contained in the set of the timed traces of M if the execution times of the actions are less than or equal to the execution times defined by φ. If time-safety cannot be guaranteed for some φ, then the Real-Time Execution engine will stop, that is, a deadline is violated by the physical system.

To encompass heterogeneity of execution we need to rely on a component-based framework which provides rigorous semantics. BIP (Behavior, Interaction, Priority) is such a formalism for modeling heterogeneous component-based systems [12], developed at Verimag. It describes systems as the composition of generic atomic components characterized by their

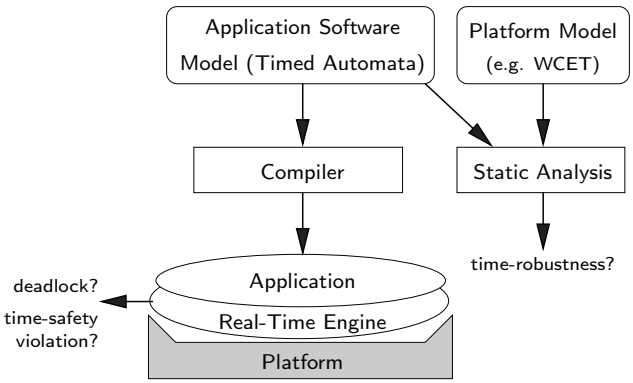


Figure 1.1: Toolset overview.

behavior and their interfaces. It supports a system construction methodology based on the use of two families of composition operators: interactions and priorities. Interactions are used to specify multiparty synchronization between components as the combination of two protocols: rendezvous (strong symmetric synchronization) and broadcast (weak asymmetric synchronizations). Priorities between interactions are used to restrict non determinism inherent to parallel systems. They are particularly useful to model scheduling polices. In contrast to existing formal frameworks, BIP is expressive enough to directly model any coordination mechanism between components [23]. It has been successfully used to model complex systems including mixed hardware/software systems and complex software applications. BIP can be used as a unifying semantic model for structural representation of different, domain specific languages and programming models.

3 Organization of the Thesis

This document is composed of four parts, the first presenting the context of the work (Chapters 2 and 3), the second describing the contribution of the thesis (Chapters 4, 5 and 6), the third presenting autonomous systems design and implementation (Chapters 7 and 8) applying the methodology presented in the contribution, and the last part (Chapter 9) drawing the conclusion and future work. The details of all chapters are as follows:

- Chapter 2 presents the state of the art for building real-time systems. It includes design methodologies for timed systems.
- Chapter 3 presents the basic ideas about component-based methodology, the basic notions about components, their composition using glue operators, and the necessary properties for component-based construction of systems. It introduces the BIP component framework, describing its model-based semantics and architecture. The BIP framework has the property to build correct-by-construction systems.
- Chapter 4 introduces the concepts of time-safety and time-robustness to build correct real-time systems. We present a model-based implementation method that relies on implementation theories that allow deciding if a given application software, i.e. its associated model, can be implemented on a given platform. The method is based

on the use of an *abstract model* representing the abstract behavior of the real-time application with user-defined platform-independent timing constraints, and a *physical model* representing the execution of the abstract model on a specific platform. We show that a physical model is time-safe if its execution sequences are contained in the set of the execution sequences of the corresponding abstract model. A physical model is time-robust if its execution on a faster machine preserves the time-safety property.

- Chapter 5 proposes a concrete implementation method using a real-time execution engine which faithfully implements physical models. That is, if a physical model defined from an abstract model and a target platform is time-robust, then the execution engine coordinates the execution of the application software so as to meet the real-time constraints. We also present some extension in the BIP framework to build real-time component-based systems.

- Chapter 6 extends the concepts presented in Chapters 4 and 5 by not only considering closed systems but also taking into account communications with an external environment.

- Chapters 7 and 8 present interesting results on the design and implementation of the autonomous robot, Dala. We improved the functional and decisional levels design and implementation and we show the benefits obtained in terms of CPU utilization and design simplifications.

- We conclude the thesis in Chapter 9, with an overview of the work and its future perspectives.

Chapter 2

Implementation of Real-Time Systems

In this chapter we summarize a brief description of the current state-of-the-art in real-time systems design and implementation. When developing real-time systems, it is important to make a clear distinction between physical and logical time. The notion of time serves two purposes. Firstly, it is used to specify the order of execution of individual actions of systems applications. Secondly, it can be used to specify durations. On one hand, the logical time can be used in both cases, especially in the design phase. On the other hand specifying the order of execution based on the physical time leads to the non-determinism of execution, since physical time is not known at the design stage. Existing rigorous implementation techniques use specific programming models. Current practice in real-time systems design follows two well-established paradigms, namely synchronous and asynchronous. The time triggered architecture considers more general programming models, relying on a notion of logical execution time (LET). It somehow combines the synchronous and asynchronous paradigm. Finally component-based design is also an important paradigm. Component-based design techniques are used to cope with the complexity of the systems. The idea is that complex systems can be obtained by assembling components. This is essential for the development of large-scale, evolvable systems in a timely and affordable manner.

In Section 1, we present synchronous systems design and a brief description of the Lustre approach. In Section 2, we present the asynchronous approach. In Section 3, we present the time triggered architecture and the OASIS approach. In Section 4, we give a brief presentation of the component-based approach. Finally, in Section 5 we discuss the different approaches for the implementation of real-time systems.

1 Synchronous Systems

Synchronous programming is a design method for modeling, specifying, validating and implementing safety critical applications [22, 54, 56, 57].

1.1 Presentation

The synchronous paradigm provides a set of primitives which allows a program to be considered as instantaneously reacting to external events. It assures that a system interacts

with its environment by performing global computation steps. In a step, the system reacts to environment stimuli by propagating their effects through its components in a well-defined order (causality order). The synchrony assumption states that the system's reaction is fast enough with respect to the environment. Responsiveness and precision are limited by step duration. Synchronous programs can be considered as a network of strongly synchronized components. Their execution is a sequence of non-interruptible steps that define a logical notion of time. In a step each component performs a quantum of computation.

One of the fundamental characteristics of synchronous languages resides in the use of clocks for the specification of the synchronization points between components. A clock is an infinite subset of the natural numbers. The simplest example of a synchronous system consists of a number of components all referring to the same periodic clock, defined by its initial date and its period. The operational semantics of such a system is defined as a sequence of cycles executed at each tick of the clock. A cycle consists of three phases: acquisition of inputs, computation and publication of outputs. The physical duration of computations is then irrelevant. In the synchronous hypothesis: computations are assumed to have no duration or, in other words, the computation duration is negligible compared to that of communication among components. This leads to computation being divisible in steps and execution being well-behaved. Real-time performance is then evaluated by first computing the bounds or estimates of worst-case execution time (WCET) of individual computations, then performing an end-to-end delay analysis of the entire system. An implementation is correct if the worst-case execution times (WCET) for steps are less than the requested response time for the system.

Hardware description languages such as Esterel [23], Lustre [54] and Signal [22], adopt the synchronous paradigm. These languages are used, among others, in signal processing and automatic control applications.

1.2 The Lustre Approach

In this subsection we present the Lustre Approach [80]. Lustre is a data-flow synchronous language for programming reactive systems. Lustre programs operate on flows of values, that are infinite sequences (x_0, x_1, .. , x_n) of values at logical time instants. An abstract syntax for Lustre programs is shown Figure 2.1. A Lustre program is structured as a set of nodes. Each node computes output flows from input flows. In (resp. Out) denotes the set of inputs (resp. outputs) of a node. Symbol N represents the node names, x represent the flows of the program where E represent expressions. Expressions can be constant values v or boolean values b.

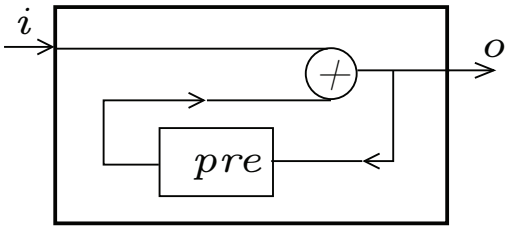
Each flow (and expression) is associated with a logical clock. Implicitly, there always exists a unique, fastest, basic clock which defines the step (or basic clock cycle) of the program. Depending on this clock, other slower clocks can be defined as the sequences of time instants where boolean flow values take the value true. Luste has only few elementary basic types: boolean, integer and one type constructor: tuple. Complex types can be imported from a host language. Usual operators over basic types are available such as arithmetic, boolean, relational and conditional. Functions can be imported from the host language. These are combinatorial operators (op) and the unit delay **pre** operator known as single-clock operators. They operate on operands that share the same clock. Besides these operators, Lustre has operators which operate on multiple clocks. These are the **when** and the **current** operator known as multi-clock operators.

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{node}^+ \\
\textit{node} & ::= & \textbf{node}\ N\ (\textit{In})\ (\textit{Out})\ \textit{equation}^+ \\
\textit{equation} & ::= & x = E \ \mid \\
 & & x, \cdots, x = N(E, \cdots, E) \\
E & ::= & x \ \mid\ v \ \mid\ op(E, \cdots, E) \ \mid\ \textbf{pre}(E,v) \ \mid \\
 & & E\ \textbf{when}\ b \ \mid\ \textbf{current}\ E
\end{array}
$$

Figure 2.1: Abstract syntax for Lustre programs.

Clock Operators

In this subsection, we illustrate the Lustre concepts described above. Single-clock operators contain constants, basic combinatorial operators and the unit delay operator. Flows of values that correspond to constants are constant sequences of values. Combinatorial operators include usual boolean, arithmetic and relational operators. The unit delay **pre** operator gives access to the value of its argument at the previous time instant. For example, the expression $E = pre(E, v)$ means that for an inital value $E_0 = v$ of E, the value of E at instant i is $E_i = E_{i+1}$ for all $i > 0$,.




```
node Integrator(i:  int)
     returns o: int;
let  o = i + pre(o,0); tel;
```

Figure 2.2: An integrator described in LUSTRE

Example 1 *Figure 2.2 shows a discrete integrator written in Lustre (rignt). It uses the single-clock operator* "+" *and* **pre**. *The integrator has an input flow i and an output flow o. The expression* $pre(o, 0)$ *gives access to the value of o at the previous time instant and is initialized to zero. The output flow o is obtained by adding to its previous value* $pre(o, 0)$ *the input flow i. The equation of the integrator is the arithmetic operation "+" between a flow and the expression pre. The instants of a possible execution are shown in figure 2.3.*

basic clock	*1*	*2*	*3*	*4*	*5*	*6*	*7*	$\cdots$
i	2	5	-7	0	3	9	1	$\cdots$
pre	0	2	7	0	0	3	12	$\cdots$
o	2	7	0	0	3	12	13	$\cdots$

Figure 2.3: Execution instants for the Integrator node.

In order to define and manipulate flows operating on slower clocks, Lustre provides two additional operators. The sampling operator **when**, samples a flow depending on a boolean flow. The expression $E' = E\ when\ b$, is the sequence of values E when the boolean flow b

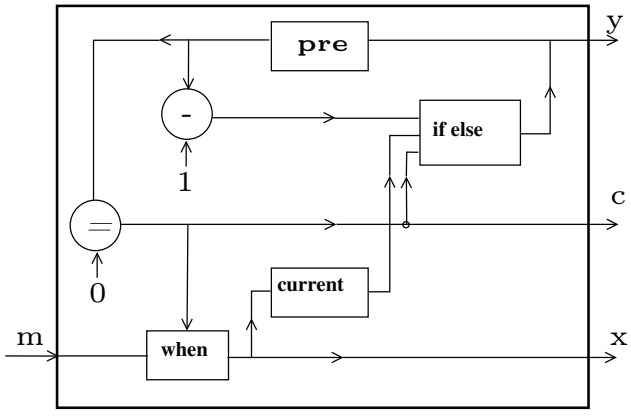
is true. The expression E and the boolean flow b have the same clock, while the expression E' operates on a slower clock defined by the instants at which b is true. The interpolation operator **current**, interpolates an expression on the clock which is faster than its own clock. The expression $E' = current\ E$, takes the value of E at the last instant when b was true, where b is the boolean flow defining the slower clock of E.

Example 2 *An example of using sampling and interpolating operators is shown in Figure 2.4. The basic clock defines six clock cycles. The flows b and x operate on the basic clock. Flow b defines a slower clock, operating at the cycles 3,5 and 6 of the basic clock where the value of b is true. The sampling operator* **when** *defines the flow y that operates on the slower clock b. Flow y is evaluated when b is defined. The interpolation operator* **current** *produces the flow zon the basic clock. Flow z has the same clock with b. For the first two instances the value of z is undefined because yis evaluated for the first time at the clock cycle 3. For any other instant, if b is true, the value of z is evaluated to y. Otherwise, it takes the value of y at the last instant when b was true.*

basic clock	*1*	*2*	*3*	*4*	*5*	*6*	$\cdots$
b	*false*	*false*	*true*	*false*	*true*	*true*	$\cdots$
x	x_1	x_2	x_3	x_4	x_5	x_6	$\cdots$
$y = x$ **when** b			x_3		x_5	x_6	$\cdots$
$z =$ **current** y	*nil*	*nil*	x_3	x_3	x_5	x_6	$\cdots$

Figure 2.4: Example of use of *when* and *current multi-clock operators*

A Lustre program can be composed of multiple nodes using multi-clock operators. Figure 2.5 shows a multiplier (mux) Lustre node. It reads a variable m at each clock cycle and produces three outputs. Outputs y and c are produced at each cycle of the basic clock and x when c is true. If the boolean variable c is false, y decreases its value by one. When y is evaluated to zero, the value of c becomes true and x produces the current value m.




```
node mux(m:  int)
        returns (c: bool;
                 x: int when c; y: int);
let
    y = if c then current x else pre y-1;
    c = true - > pre y=0;
    x = m when c;
tel.
```

Figure 2.5: A *mux* LUSTRE node

Execution of Programs

The Lustre compiler guarantees that the system under design is deterministic and conforms to the properties defined by the synchronous hypothesis. It accomplishes this task thanks to static verification which is summarized in the following steps:

- Definition checking: every local and output variable should have one and only one definition;
- Clock consistency: every operator is applied to operands on suitable clocks;
- Absence of cycles: any cycle should use at least one pre operator.

Lustre is a synchronous language with formal semantics developed at the VERIMAG laboratory for the past 25 years. On the top of the language, there are a number of tools, like code generator, model checker, tool for simulation of the system on design, etc., that constitute the Lustre platform. Lustre has been industrialized by Esterel Technologies in the SCADE tool. SCADE has been used from several companies in the area of aeronautics and automotive. It has been recently used for the development of the latest project of Airbus, the A380 carrier airplane. For the mono-processor and mono-thread implementation, the compiler generates monolithic endless single loop C code. The body of this code implements the inputs to outputs transformations at one clock cycle. The generation of C code is done in two steps. First, introducing variables for implementing the memory needed by the pre operators and then sorting the equations in order to respect data-dependencies.

2 Asynchronous Systems

The asynchronous paradigm does not impose any notion of global execution step. The components (threads, tasks or processes) proceed each at their own pace and usually communicate by message passing. Therefore, this paradigm is particularly suitable for distributed systems. For asynchronous real-time programs e.g. ADA programs [39],C and Java, there is no notion of execution step, and concurrency operators are provided by thread libraries when they are not explicit. Implementation for asynchronous languages relies on an operating system. The latter is responsible for scheduling and tasks are driven by events. Scheduling policies are used for sharing resources between tasks. Scheduling theory also allows to estimate system response times for tasks with known period and time budget. It tries to guarantee satisfaction of simple time constraints, such as deadlines.

One of a known asynchronous programming language is SDL [15,40]. The SDL(Specification and Description Language) language is a CCITT standard specification language. It is based on an extended finite-state machine (EFSM) model. SDL specification can be written in two different syntaxes: graphical (SDL/GR) and textual (SDL/PR); one-to-one mapping is available for the two forms. It has the following concepts:

- The system is described hierarchically by elements called systems, blocks, channels, processes, services, signal routes and signals.
- Behavior is described using the EFSM concepts.
- Data is described using abstract data types (program variables and data structures).

- Communication is asynchronous via channels that can have infinite queues.

An SDL system consists of functional blocks and each block can be decomposed into sub-blocks and so on. The lower block level consists of one or more processes that are described as state machines (see Figure 2.6).

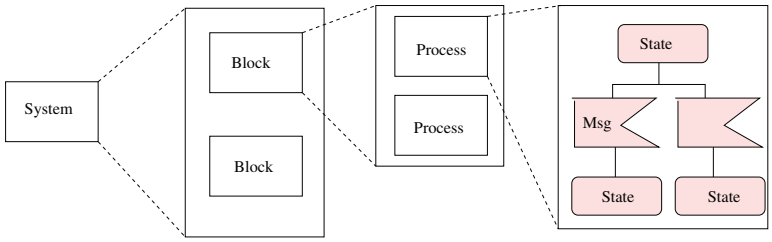


Figure 2.6: Example of an SDL program architecture.

Timing is associated to the occurrence of signals (like interrupts). In the following example, in Figure 2.7, we give an example for using timers by considering a connection protocol mechanism. We declare the variable *Connection*, which is the only local variable in the process. The first transition is the transition *start* that initializes the local variable. A message is sent for connection request with signal *conReq*. A timer of 5 seconds is started with the command $SET(5, ConReqTimer)$, and the process goes in state *Connecting* (a). When the process is at state *Connecting*, when the timer signal is available, the process sends 10 times a request for connection (b). When it receives a confirmation of connection with signal $ConConf$, the process goes in state *Connected* (c). This is a typical scenario in telecommunication protocols.

3 Time Triggered Architecture

For real-time applications, it is desirable to combine the synchronous and asynchronous paradigm for both application software and implementation. We need programming and specification languages combining the two description styles. Recent implementation techniques consider more general programming models.

3.1 Presentation

The proposed approaches rely on a notion of logical execution time (LET) [8, 51, 58] which corresponds to the difference between the release time and the due time of an action, defined in the program using an abstract notion of time. To cope with uncertainty of the underlying platform, a program behaves as if its actions consume exactly their LET: even if they start after their release time and complete before their due time, their effect is visible exactly at these times. This is achieved by reading for each action its input exactly at its release time and its output exactly at its due time. Time-safety is violated if an action takes more than its LET to execute.

The time-triggered approach consists in assigning to each action its desired execution time guaranteeing by construction the end-to-end constraints. The system is then implemented over an execution kernel running on a platform and ensuring that:

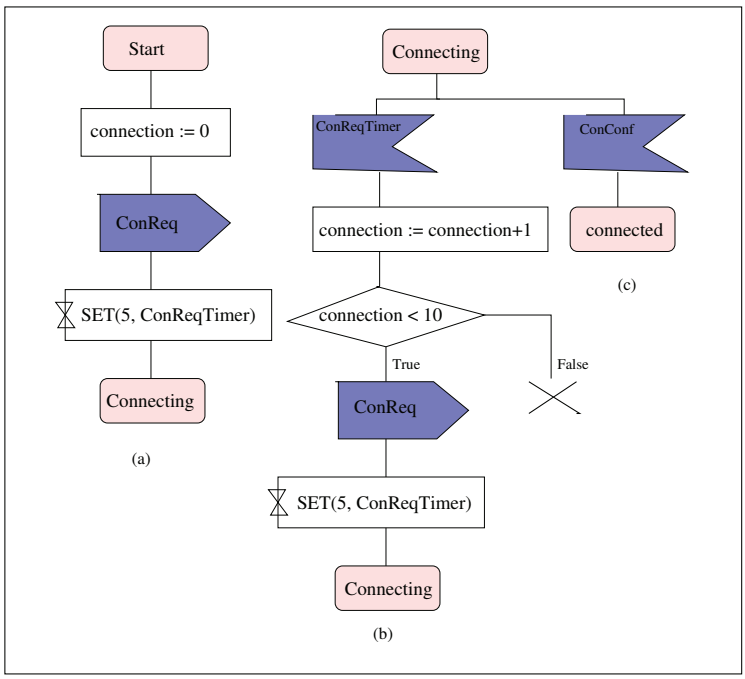


Figure 2.7: Example of an SDL program using Timers.

1. the action currently being executed does not exceed the duration it is allocated, and

2. the execution of the following action is triggered at the appropriate real-time.

Various approaches can be taken in the case when an action violates a deadline: execution can be aborted, a recovery mechanism or a degraded mode can be initialized. Alternatively, WCET analysis can be performed in order to verify whether execution time requirements are respected and select an appropriate target architecture.

3.2 The Oasis Approach

Oasis is a framework for safety-critical real-time systems, based on a time-triggered architecture [41, 44, 68, 69] . The main objective of Oasis is a framework encompassing models, methodologies, and tools for the development of embedded critical software exhibiting completely deterministic temporal behavior. In the Oasis approach, an application is viewed as a set of communicating tasks (or agents) that interact to achieve their functionality and real time is managed by a time-trigged architecture. The processing of each task is synchronous at a particular time. At the end of the logical execution time, variables are made visible to other tasks. The decomposition of tasks corresponds exactly to the processing that should be executed in parallel. In a time triggered approach, a system observes the environment and initiates its processing at recurring predetermined instants. With each task T in the system, a real-time clock H_T represents the set of physical instants where the input and output data are or can be observed. The joining of al clocks H_T for each task of the system S, is the definition of the system real-time clock H_S. The H_S clock includes all the observable instants of the system and is global and regular. It represents the smallest regular

clock from which all tasks clocks are deductible, i.e. H_S exhibits a factor K_T and an offset δ_T such as $H_T = K_t * H_S + \delta_T$.

Example 3 *Figure 2.8 represents a global clock HGlobal and two clocks H1 and H2 deriving from HGlobal as follows:* $H1 = 2 * HGlobal$ *and* $H2 = 3 * HGlobal + 1$.

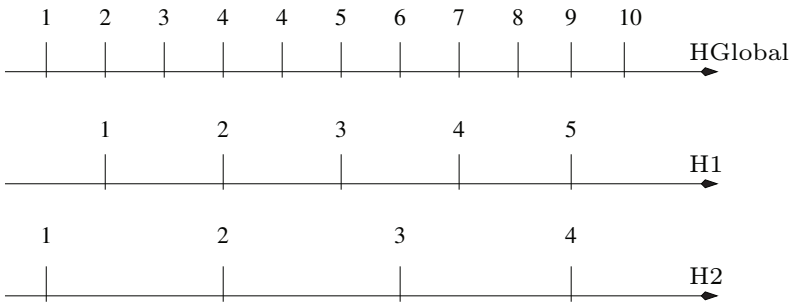
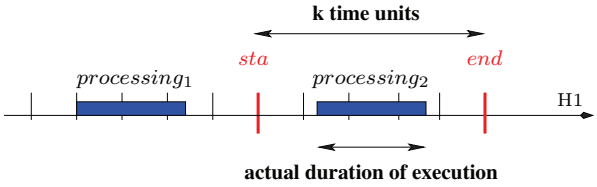


Figure 2.8: Clocks definition in OASIS.

Expressing time constraints

Time can be manipulated through the advance instruction (*Advance*). The instruction *Advance(k)* sets a deadline for the processing operation. It indicates that the next activation instant will be its current instant plus $k * H_T$. The semantics of *Advance(k)* is completely defined, independently of the physical time of its execution. The instruction can simultaneously express a deadline and an activation time, thus encouraging developers to incorporate a complete description of the timing behavior at the design stage. An *Advance* instruction splits the task code into two parts: the part of code (a processing) before and the part of code after the *Advance* instruction. A task is viewed as series of processing steps that have precedence relationships. Thus, we can declare the task's future instants (i.e. activation instants) and both earliest start date (*sta*) and latest end date (*end*) of each processing (see figure 2.10).




```
clock HGlobal = gtc1 (0, 500)
clock H1 = HGlobal
agent process (startime=sta) with H1
   body start {
      processing1
      Advance(x) /* sta = x + ..  */
      processing2
      Advance(K) /* end = sta + k */
}
```

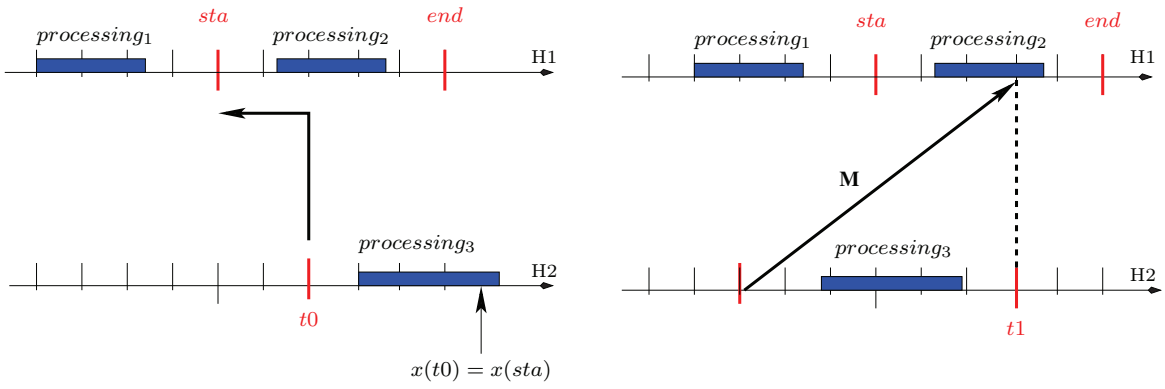
Figure 2.9: Elementary processing and associated time interval in OASIS.

Communication mechanisms

There are two modes of communication between tasks. The first mode uses the exported variables, also called temporal data flow, the second is the sending of messages from a

sender task to a recipient task. The new values of a temporal variable are made visible at every synchronization point of the producer task, while messages require explicit definition of visibility dates.

In the temporal data flow mechanism, each temporal variable defines a real-time data flow: its values, available to any task that consults them, are stored and updated by a single write at a predetermined temporal cadence. Thiscadence is expressed in the source code as a regular time period parameter and allows computation of a periodic updating date. Based on the previous example, between the two dates *sta* and *end*, that is, when the task is executing, it is logically at *sta* date. Consequently, assuming that the task has a temporal variable x, regardless of the values between dates *sta* and *end* (i.e. however the value of x is modified by $processing_2$), and if another task "observes" the value of x at date t_0 (see Figure 2.10 (a)), the x value "observed" is always its past value $x(sta)$. In the sending message mechanism, a message has a visilibility date (date beyond which a message can be seen and extracted by the recipient agent) that is specified by the sender. For example, from [41], if message M is sent by the second task with a visibility date $t1$, it cannot be viewed (or consumed) before the *end* date, since $t1 > sta$ (see Figure 2.10 (b)). A recipient task has queues for receiving purposes. To achieve determinism, the sequence of messages sorted by visibility is made to be total.



(a) Value of a temporal variable

(b) Sending a message

Figure 2.10: Communication mechanisms in OASIS.

Temporal correctness

Temporal correctness is verified if and only if all the critical tasks processing are always executed in a timely manner. This property, always verified in the design phase, has to be ensured while the application is executing (some tasks could miss their processing deadlines). In the Oasis approach, all timing constraints, such as deadlines, are clearly expressed in the design level. All these constraints are calculated on the same global real-time-clock. Processing upper bounds (i.e WCET) are known for each real-time task. To schedule and execute these tasks on the target platform, they use the results of the deadline-driven scheduling techniques. It is necessary to calculate processor load and to make an off-line analytic verification of linear inequalities. It allows us to calculate necessary and sufficient

conditions to guarantee that task on-line scheduling still ensures timing properties. The approach is based on an off-line analysis of all interactions between all the tasks, to guarantee that the deadline-driven online scheduling ensures the temporal correctness.

For Oasis-based applications, execution involves two layers: the microkernel and the system layer. The microkernel manages real-time and processor sharing. Its execution cannot be interrupted. A timer is necessary and sufficient to afford real time. The system layer is generic, safety-oriented and size-limited. It manages the data exchanges required for communication and ensures their logical and temporal consistency. By using the state-transition diagram extracted at the compilation stage, the system layer controls the logical and temporal behavior of each task.

4 Component Based Design

In this section we present a set of component-based approaches [62]. This set is not exhaustive but illustrates the existing landscape for embedded systems design and implementation. We can find either methods that are focused on the abstraction level point of view with very little to do with the implementation, or very close to the implementation level. As discussed later in the thesis, our work can be seen as a bridge between the two.

4.1 Presentation

Component-based design techniques are used to cope with the complexity of the systems. A model is obtained by assembling components (see Figure 2.11). The interfaces of the components are bound together by connections to form an architecture. The idea is that complex systems can be obtained by assembling components. This is essential for the development of large-scale, evolvable systems in a timely and affordable manner. It offers flexibility in the construction phase of systems by supporting the addition, removal or modification of components without any or very little impact on other components. Components are usually characterized by abstractions that ignore implementation details and describe relevant properties to their composition, e.g. transfer functions and interfaces. The main feature of component-based design frameworks is allowing composition. This composition is used to build complex components from simpler ones. It can be formalized as an operation that takes, as input, a set of components and their integration constraints and provides, as output, the description of a new more complex component. This approach allows to cope with the complexity of systems by offering incrementality in the construction phase. There exists a large body of literature dealing with component-based design.

4.2 Examples

PtolemyII [46, 67] is a software framework, developed at Electrical Engineering and Computer Sciences (EECS) University of California Berkeley University. PtolemyII focuses on component-based heterogeneous modeling. The basic building block of a system is called an actor. A model of the system is obtained by an hierarchical interconnection of actors. Actors are software components that run concurrently and communicate through interfaces called ports. An actor can be atomic or composite. An atomic component must be at the bottom of the hierarchy, whereas a composite component contains other actors. The semantics of a model is not determined by the framework, but rather than by a software

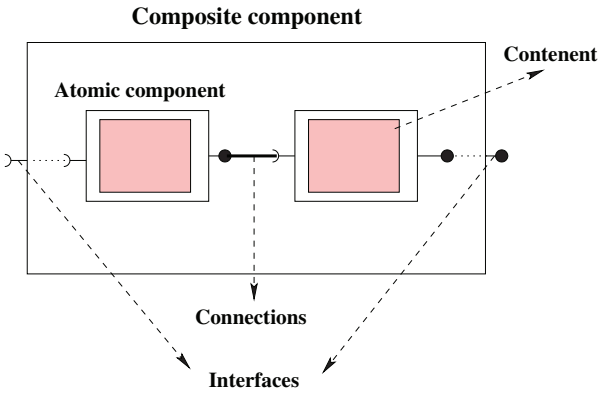


Figure 2.11: Example of components composition

component in the model called director, which implements a model of computation. PtolemyII allows the simulation of models. However, the verification of models is, currently, not possible. Work is underway to add this possibility. It allows the generation of code, to do this, each component must be accompanied by a model ("template") of C code which will be completed by the code generator. Moreover, PtolemyII has no intrinsic notion of mapping between actors or of using declarative specification in the design.

Fractal [24, 37] is a component model, developed at France Telecom and INRIA France. Fractal is a general component model for implementing, deploying and managing complex software systems. It can be understood generally as being composed of a membrane which consists of a set of components (called subcomponents) and one or more interfaces (similar to port in other component models). Interfaces can be of two kinds: server interfaces for incoming operation invocations, and client interfaces for outgoing operations invocations. Think [76] is one of the Fractal implementation. However Fractal and Think do not provide tools or analysis techniques, whether for simulation or verification.

5 Discussion

There are different requirements for building efficient and correct implementation for real-time systems.

- We need component-based frameworks. The concept of component and associated composition operators are used for incremental description and correctness by construction. Those frameworks should be expressive enough to directly encompass all types of coordination with well-founded and organized concepts instead of using dispersed coordination mechanisms such as semaphores, monitors, message passing, remote call, protocols etc. It also needs to be abstract enough by providing high-level primitives for modeling behaviors and communications. However, abstraction reduces expressiveness. Thus, the first challenge is to find the best compromise between a high level of abstraction and high expressiveness. The design of complex systems shoud be done in an easy and rigorous manner. Nonetheless, on top of abstraction and expressiveness, etc., other challenges appear, mainly how to automatically derive a correct and efficient implementation.

- Frameworks have to provide a rigorous but not complex semantics, because complexity limits abstraction. When having a rigorous semantics, we can derive a correct and efficient low-level implementation from the high-level models. Moreover, a strong theoretical backing can be defined at the high-level models that allows formal verification of design properties.

- A key issue for design methodologies is meeting timing constraints e.g. a system reacts within user-defined bounds such as deadlines and periodicity. The satisfaction of timing constraints depends on features of the execution platform, in particular its speed. We should be able to express timing constraints in the design level and detect time-safety violations of the implementation.

We have presented existing implementation techniques that use specific programming models. Synchronous programs can be considered as a network of strongly synchronized components. Their execution is a sequence of non-interruptible steps that define a logical notion of time. In a step each component performs a quantum of computation. An implementation is correct if the worst-case execution times (WCET) for steps are less than the requested response time for the system. For asynchronous real-time programs e.g. ADA programs, there is no notion of execution step. Components are driven by events. Fixed priority scheduling policies are used for sharing resources between components. Scheduling theory allows to estimate system response times for components with known period and time budget. Recent implementation techniques consider more general programming models. The proposed approaches rely on a notion of logical execution time (LET) which corresponds to the difference between the release time and the due time of an action, defined in the program using an abstract notion of time. Time-safety is violated if an action takes more than its LET to execute. One of the difficulties in the synchronous paradigm is that the synchrony assumption is not easy to meet, in particular when high responsiveness to the environment is required. Another drawback is that modularity cannot be easily handled; for instance, it is hard to compile synchronous systems separately and then link them together or with non-synchronous implementations. Our work generalizes existing techniques in particular those based on LET. These techniques consider fixed LET for actions, that is, time-deterministic abstract models. In addition, their models are action-deterministic, that is, only one action is enabled at a given state. In our work, we extend the existing timed triggered approaches by considering more general timing constraints leading to undeterminitic systems.

There also exist many component frameworks without rigorous semantics. They use ad-hoc mechanisms for building systems from components and offer syntax level concepts only. In this case, using ad-hoc transformations, may easily lead to inconsistencies e.g. transformations may not be correct. On exploring the current state of the art we have not seen a component framework that meets the requirements above. Generally speaking, we can divide them into two categories. The first category provides high-level design and modeling, however it is still unclear how to derive correct and efficient implementation from the high-level models. In contrast, the second category provides efficient implementation, however the design process is either based on low-level primitives, or not expressive enough.

In the next chapter, we present the BIP framework for which we added timing features. It enables describing all software and systems according to a single semantic model, guaranteeing that the physical model meets essential properties of the description at the design level. It is a also a component-based framework, since it provides operators for building

composite components from simpler components. Moreover, it guarantees correctness by construction and thereby avoids monolithic a posteriori verification as much as possible.

Chapter 3

The BIP Framework

1 Presentation

In this chapter, we present the BIP (Behavior Interaction Priorities) framework [11, 14, 82]. BIP is a framework for building real-time systems consisting of heterogeneous components. A component has only local data, and its interface is given by a set of communication ports. The behavior of a component is given by an automaton whose transitions are labelled by ports and can execute C++ code (i.e. local data transformations). Connectors between communication ports of components define a set of enabled interactions which are synchronizations between components. Interactions are obtained by combining two types of synchronization: rendez-vous and broadcast. The execution of interactions may involve transfer of data between the synchronizing components. Priority is a mechanism for conflict resolution that allows direct expression of scheduling policies between interactions.

BIP models can be compiled to C++ code. The generated code is intended to be executed by a dedicated Engine implementing the semantics of BIP.

The BIP framework is :

- **Model-based**. Describing all software and systems according to a single semantic model. This maintains the flow's overall coherency by guaranteeing that a description at step n+1 meets essential properties of a description at step n.
- **Component-based**. Providing a family of operators for building composite components from simpler components. This overcomes the poor expressiveness of theoretical frameworks based on a single operator, such as the product of automata or a function call.
- **Correct-by-construction**. Guaranteeing correctness by construction and thereby avoiding monolithic a posteriori verification as much as possible.

This chapter is structured as follows. The BIP model-based framework is described in section 2. It gives an abstract formalization for the concepts of *Behavior*, *Interactions* and *Priorities*. Section 3 describes the BIP component-based framework, the concrete model of BIP with data extensions. It introduces the concepts of *Components* and *Connectors* to build systems models. The BIP tool-chain is described in section 4. Finally, in section 5, we present a timing model of BIP. It shows how timing features are modeled using the current version of BIP in terms of global synchronizations.

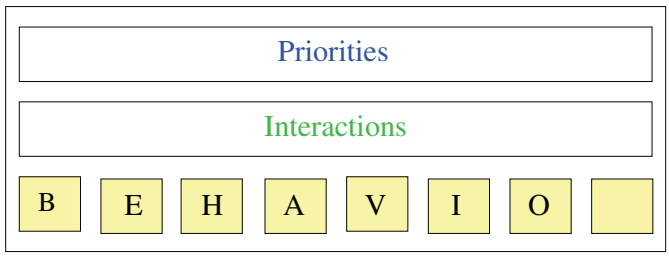


Figure 3.1: Structure of a BIP Model.

2 The BIP Model-based Framework

We provide a formalization of the BIP component model focusing on the individual layers of behavior, interaction and priority glue (see Figure 3.1). In this section, we provide for each layer its abstract model.

2.1 Modeling Behavior

An *atomic component* is the most basic BIP component which represents behavior. A formal definition for the behavior of an atomic BIP component is given below:

Definition 1 (Behavior) *A* behavior B *is a labeled transition system represented by a triple* $(\mathsf{Q}, \mathsf{P}, \longrightarrow)$*, where:*

- Q *is a finite set of* control states*,*
- P *is a set of* communication ports*,*
- $\longrightarrow \subseteq (\mathsf{Q} \times \mathsf{P} \times \mathsf{Q})$ *is a set of transitions, each labeled by a port.*

For a pair of states $q, q' \in \mathsf{Q}$ and a port $p \in \mathsf{P}$, we write $q \xrightarrow{p} q'$, iff $(q, p, q') \in \longrightarrow$ and we say that p is enabled at q. If such q' does not exist, we say that p is disabled at q.

2.2 Modeling Interactions

We compose a set of n atomic component behaviors $\{B_i = (Q_i, P_i, \longrightarrow_i)\}_{i=1}^{n}$, by using interactions. We assume that their respective sets of ports and sets of states are pairwise disjoint, i.e., for all $i \neq j$, we have $P_i \cap P_j = \emptyset$ and $Q_i \cap Q_j = \emptyset$. We define the set $P = \bigcup_{i=1}^{n} P_i$ of all ports in the system.

Definition 2 (Interaction) *An* interaction a *is a non-empty subset* $a \subseteq \mathsf{P}$ *of ports. When we write* $a = \{p_i\}_{i \in I'}$*,* $I' \subseteq [1, n]$*. For each* $i \in I'$*,* $p_i \in P_i$*.*

The interaction model is specified by a set of interactions $\gamma \subseteq 2^P$. Interactions of γ can be enabled or disabled. An interaction a is enabled iff, for all $i \in [1, n]$, the port $a \cap P_i$ is enabled in B_i. That is, an interaction is enabled if each port that is involved in this interaction is enabled. An interaction is disabled if there exist $i \in [1, n]$ for which the port $a \cap P_i$ is disabled in B_i. That is, an interaction is disabled if there exists at least a port, involved in this interaction, that is disabled.

2.3 Modeling Priorities

In a behavior, more than one interaction can be enabled at the same time, introducing a degree of non-determinism. This can be restricted with priorities by filtering the possible interactions based on the current global state of the system.

We compose a set of n atomic components behaviors $\{B_i = (Q_i, P_i, \longrightarrow_i)\}_{i=1}^n$.

Definition 3 (Priority) *A* priority *is a relation* $\prec \subseteq \gamma \times \mathsf{Q} \times \gamma$, *where:*

- γ *is the set of interactions,*
- $\mathsf{Q} = Q_1 \times ... \times Q_n$ *is the global set of states.*

For $a \in \gamma, q \in \mathsf{Q}$ and $a' \in \gamma$, the priority $(a, q, a') \in \prec$ is denoted as $a \prec_q a'$. That is, interaction a has less priority than a' in state q.

2.4 Composition of Abstract models

For a set of components $\{B_i = (Q_i, P_i, \longrightarrow_i)\}_{i=1}^n$, an interaction model γ and a priority model π, the compound component is obtained by application of a glue GL.

The glue GL is composed of the two previous models γ and π and defined as $GL = \pi\gamma$, where the interaction model γ is a set of interactions and the priority model π is a set of priorities.

Definition 4 (Composition for Interactions Model) *The composition of a set of atomic components* $\{B_i\}_{i=1}^n$, *parameterized by a set of interactions* $\gamma \subseteq 2^P$, *is a transition system* $B = (\mathsf{Q}, \gamma, \longrightarrow_\gamma)$, *where:*

- $\mathsf{Q} = \bigotimes_{i=1}^n Q_i$,
- γ *is the set of interactions* $\gamma \subseteq 2^P$ *where* $\mathsf{P} = \bigcup_{i=1}^n P_i$,
- *For* $a = \{p_i\}_{i \in I} \in \gamma$, *we have* $(q_1, ..., q_n) \xrightarrow{a}_\gamma (q'_1, ..., q'_n)$ *in* B *if and only if,* $q_i \xrightarrow{p_i}_i q'_i$ *in* B_i *for all* $i \in I$, *and* $q'_i = q_i$ *for all* $i \notin I$.

The obtained behavior B can execute a transition $a = \{p_i\}_{i \in I} \in \gamma$, if and only if, for each $i \in I$, port p_i is enabled in B_i.

Definition 5 (Composition restricted from the Priority Model) *Given a behavior* $B = (\mathsf{Q}, \gamma, \longrightarrow_\gamma)$, *its restriction by the priority model* π *is the behavior* $B' = (\mathsf{Q}, \gamma, \longrightarrow_\pi)$, *where for* $a \in \gamma$, *we have* $q \xrightarrow{a}_\pi q'$ *in* B' *if and only if,* $q \xrightarrow{a}_\gamma q'$ *in* B *and for all* $a' \in \gamma$ *and* $a \prec_q a'$, a' *is disabled at* q.

The obtained behavior B' can execute a transition $a \in \gamma$ if and only if, each transition $a' \in \gamma$, with higher priority than a is state q, is disabled.

3 The BIP Component-based framework

For each abstract model of the BIP layer, we provide its concrete model. The behavior Layer is modeled with atomic components. The interaction layer is modeled with connectors and finally Priorities is a mechanism for scheduling interactions.

3.1 Atomic Components

An atomic component is a unit of behavior with an interface consisting of ports, and behavior encapsulated as set of transitions. Ports are particular names defining communication points for components. As we shall see later, they are used to establish interactions between components by using connectors. We assume that every port has an associated distinct data variable x. This variable is used to exchange data with other components, when interactions take place.

Here is a definition of a port :

Definition 6 (Port) *A* port $p[x] \in \mathsf{P}$ *is defined by:*

- p *the port identifier,*
- x *the data variable associated with the port.*

The formal definition of a BIP atomic component is given below:

Definition 7 (Atomic component) *An* atomic component *represents behavior* B *as a transition system, extended with variables and functions, represented by* $(\mathsf{V}, \mathsf{P}, \mathsf{Q}, \longrightarrow)$*, where:*

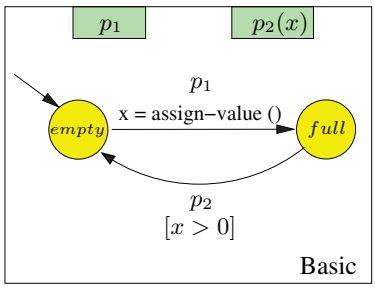
- Q *is a set of* control states $\mathsf{Q} = \{Q_1 ... Q_n\}$*, Control States denote places at which the components wait for synchronization.*
- P *is a set of* communication ports $P = \{p_1 ... p_n\}$,
- V *is a set of* variables *used to store (local) data. Variables may be associated to ports.*
- $\longrightarrow$ *is a set of transitions modeling computation steps of components. Each transition is a tuple of the form* (q_1, p, g_p, f_p, q_2)*, representing a step from control state* q_1 *to* q_2*, denoted as* $q_1 \overset{p,g_p,f_p}{\longrightarrow} q_2$.

g_p is a pre-condition for interaction through p, also known as the guard of the transition, it is a boolean condition on V. f_p is a computation step consisting of data transformations. The transition can be executed if the guard is true.

Example 4 *Figure 3.2 shows an example of a* ***Basic*** *atomic component representation (a) and its corresponding BIP code declaration (b). It has two ports* p_1 *and* p_2*, a variable* $\boldsymbol{x}$*, and two control states* ***empty*** *and* ***full****. The automaton is initialized to control state* ***empty****, from which it can move to* ***full*** *by transition labeled by* p_1*. When this transition is executed, a value is assigned for variable* x *with function* ***assign-value()****. The value of* $\boldsymbol{x}$ *is exported by port* p_2*. From state* ***empty*** *to* ***full****, the transition labeled by* ***out*** *can occur if the guard* $\boldsymbol{x > 0}$ *is true.*

Note that the omission of a guard or a function from a transition means that the associated guard is true and the internal computation step is empty. We have declared in the

code two types of ports. A pure event port **ePort** that does not have any associated variables, and provides the mechanism for event synchronization only. The port **out** is of type **IntPort**, which associates an integer variable with a port. A port in an atomic component is not visible to its environment unless it is exported explicitly. In the above example, both port p_1 and p_2 are exported. It is necessary to export a port if it has to be used in some connector for synchronization purposes.



(a) An atomic component.

```
port type IntPort (int x)
port type ePort ()

atomic type Basic
   data int x = 0
   export port ePort p1() is p1
   export port intPort p2(x) is p2

   place empty
   place full

   initial to empty

   on p1 from empty to full
      do { x = assign-value();}
   on p2 provided [x > 0]
   from full to empty
end
```

(b) BIP code of the atomic component.

Figure 3.2: An example of an open atomic component in BIP.

3.2 Connectors

Composition of components allows to build a system as a set of components that interact by respecting constraints of an interaction model. Connectors are used to specify possible interaction patterns between the ports of components.

Definition 8 (Connector) *A* connector γ *defines sets of ports of atomic components* B_i *wich can be involved in an interaction. It is formalized by* $\gamma = (P_\gamma, A_\gamma, p[x])$ *where:*

- P_γ *is the support set of* γ*, that is the set of ports that* γ *may synchronize.*
- $A_\gamma \subseteq 2^{P_\gamma}$ *is a set of interactions a each labeled by the triple* (P_a, G_a, F_a) *where:*
  - P_a *is the set of ports* $p_{i i \in I}$*,* $I \subseteq [1, n]$ *that take part of an interaction a,*
  - G_a *is the guard of a, a predicate defined on variables* $\bigcup_{p_i \in a} V_{p_i}$*,*
  - F_a *is the data transfer function of a, defined on variables* $\bigcup_{p_i \in a} V_{p_i}$*.*
- p *is the exported port of the connector* γ*.*

A connector is macro notation for representing sets of related interactions in a compact manner. Two types of ports are defined, in order to specify the feasible interactions of a connector:

- A *trigger* (represented by a triangle) is an active port. It can initiate an interaction without synchronizing with other ports. It is represented graphically by a triangle.
- A *synchron* (represented by a circle) is a passive port. It needs synchronization with other ports. It is represented graphically by a circle.

A feasible interaction of a connector is a subset of its ports such that either it contains some trigger, or it is maximal, i.e., consisting of all the ports. Example of sets of feasible interactions are show Figure 3.3. In (a), the connector consists of the ports s, r_1 and r_2 that are all of type synchron. In this connector, the only feasible interaction is $s.r_1.r_2$. It represents a strong synchronization, meaning that all the actions are necessary for the synchronization. In (b), the connector consists of the ports s of type trigger, r_1 and r_2 of type synchron. In this connector, s can occur alone or synchronize with either or both r_1 and r_2. It represents a *Broadcast*, meaning a weak synchronization between ports

In general, it is possible to represent any arbitrary interaction throught a connector by structured combination of the following two basic synchronization protocols :

- *Strong synchronization*, where the only feasible interaction of γ is the maximal one, i.e., it contains all the ports of γ. All the involved ports are of type synchron. We note $A_\gamma = P_\gamma$.
- *Weak synchronization* or *broadcast*, where all feasible interactions are those containing a trigger port p^{trig} wich initiates the broadcast. We note $A_\gamma = \{a \in \gamma \mid a \cap p^{trig}\}$.

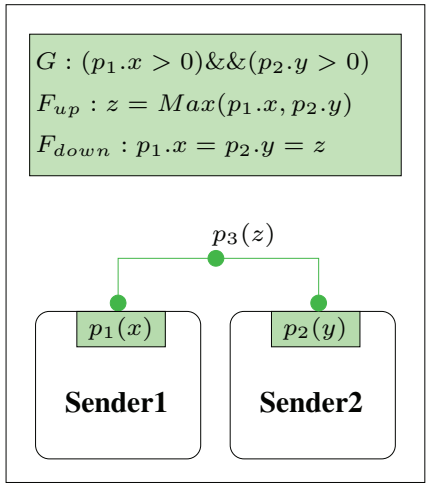
$\gamma = \{s.r_1.r_2\}$ (a) RendezVous

$\gamma = \{s, s.r_1, s.r_2, s.r_1.r_2\}$ (b) Broadcast

Figure 3.3: Connectors and their interactions in BIP.

For every interaction, the data transfer function F_a of an interaction a is specified by an **up** and a **down** action. The action **up** is supposed to update the local variables of the connector, from the values of variables associated with the ports. Conversely, the action **down** is supposed to update the variables associated with the ports, from the values of the connector variables. A guard G_a of the interaction a is a C expression and the up and down actions consist of C statements. Additionnally, a connector type definition may contain C type parameters.

Example 5 *Figure 3.4 shows a connector representation (a) and code definition (b). The connector named* ***Max*** *involves two ports p_1 and p_2 and exporting a port p_3 (allows to define hierarchical connectors) of type* ***Intport****, associating to each port an integer variable. There is only one feasible interaction $\{p_1.p_2\}$ that occurs if its guard $(p_1.x > 0)$ && $(p_2.y > 0)$ is*



(a) Connector.

connector type Max (intPort p_1, intPort p_2)
  data int z
  define [p_1p_2]

  on p_1p_2 **provided** (p_1.x > 0) && (p_2.y > 0)
    up { z = Max (p_1.x , p_2.y);}
    down { p_1.x = p_2.y = z ;}

  export port intPort p_3(z)

end

(b) Connector type definition code in BIP.

Figure 3.4: An example of a connector between two components in BIP

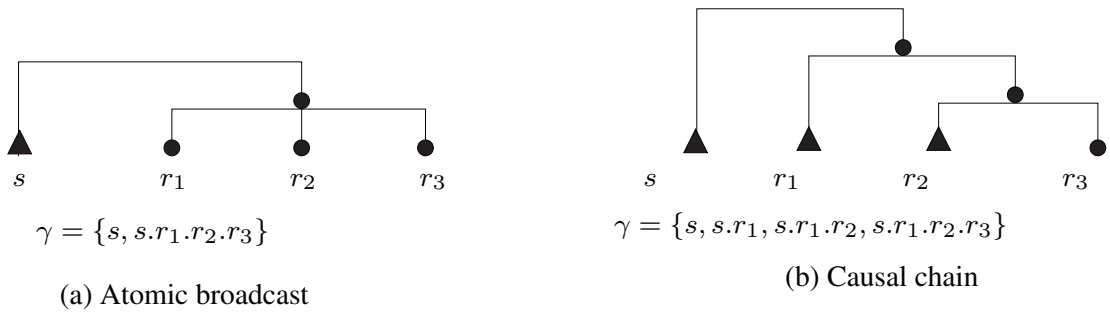
true. In the ***up*** *function, the* ***Max*** *of the variables associated with the ports is calculated and stored in the connector variable* ***z****. As a result of the data transfer, the variables associated with the ports are set to the maximum of their values, through the action* ***down****.*

Hierarchical connectors

We have seen that a connector has an option to define a port and export it. This allows a connector to be used as a port in other connectors, and create structured connectors. The representation of structured connectors require connectors to be treated as expressions with typing and other operations on groups of connectors. This led to a formalization of the algebra of connectors defined in [25, 26]. The Algebra of Connectors is a compact notation for algebric representation and manipulation of connectors and formalizes the concept of connectors supported by the BIP component model.

Figure 3.5 shows two hierarchical connectors :

- The *AtomicBroadcast* (a) involves four ports s, r_1, r_2, r_3. It represents a communication schema between a sender s and multiple reveivers r_i, where either a message is received by all the r_i, or by none. Ports r_i are strongly synchronized and the synchronization with the trigger port s is done via an exported port. This means that either s or interaction $s.r_1.r_2.r_3$ is possible.
- The *CausalChain* (b) involves the same ports with a different structure. It represents a communication schema in which if a message is received by r_i, it has also to be received by r_j, for $j < i$.



(a) Atomic broadcast

(b) Causal chain

Figure 3.5: Hierachical connectors and their interactions in BIP.

3.3 Priority Rule

Given a system of interacting components, priorities are used to filter the enabled interactions. They are given by a set of rules, each consisting of an ordered pair of interactions or connectors.

Definition 9 (Priority Rule) *A priority is a tuple $(C, \prec)$, where C is a state predicate (boolean condition) characterizing the states where the priority applies and $\prec$ gives the priority order on a set of interactions $A = \bigcup A_\gamma$.*

For $a_1 \in A$ and $a_2 \in A$, a priority rule is textually expressed as $C \rightarrow a_1 \prec a_2$. When the state predicate C is true and both interactions a_1 and a_2 specified in the priority rule are enabled, the higher priority interaction, i.e., a_2 is selected for execution.

Example 6 *In this example, we show a priority rule between two connectors **Max1** and **Max2** (see Figure 3.6). Connector Max1 synchronizes ports s_1 and r with interaction $s_1.r$ and connector Max2 synchronizes ports s_2 and r with interaction $s_2.r$. We give a higher priority to Max1 if $s_1.x$ value of port s_1 is greater that value $s_2.x$ of port s_2.*

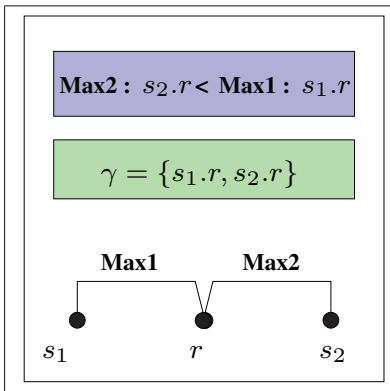
3.4 Composition of Components

In BIP the execution of interactions may involve transfer of data between the synchronizing components. Considering an interaction γ of a connector, G_γ will be its guard (boolean condition over the variables of the interacting components) and F_α its transfer function (data transfer). An enabled interaction is executable if its guard is true. The interaction leads to the execution of the data transfer function F_γ associated to it. As a result, the variables of the synchronizing components are updated.

Here is the formalization of the concept:

Definition 10 (Compound Component) *Consider the composition of n components $\{B_i\}_{i=1}^n$ parameterized by a connector γ. The composed component is a component $B = (\mathsf{V}, \mathsf{P}, \mathsf{Q}, \longrightarrow_\gamma)$ where :*

- V *is a set of variables, which is the the union of the sets of variables of the composed components* $\mathsf{V} = \bigcup_i^n \mathsf{V}_i$.



(a) An Example of a priority relation.

connector `Max1` (s_1, r)
connector `Max2` (s_2, r)

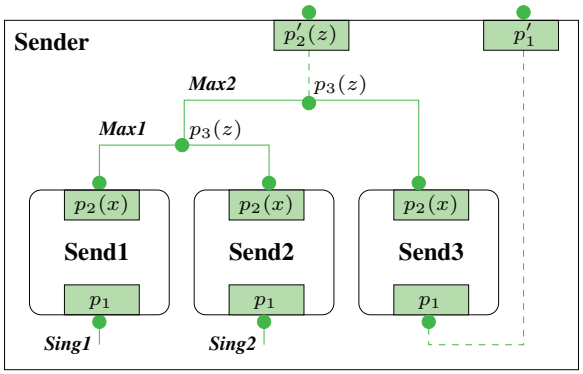
priority `maximal` **if** (s_1`.x >` $s_2.x$)
`Max2 < Max1`

(b) BIP code of the priority relation.

Figure 3.6: An example of priorities in BIP.

- Q *is a set of states, which is the cartesian product of the sets of control states of the composed components* $\mathsf{S} = \bigotimes_{i=1}^{n} \mathsf{S}_i$.
- $\longrightarrow_\gamma$ *is a set of transitions of the form* (q, α, g, f, q')*, where :*
  - $q = (q_1, ..., q_n)$*,* q_i *being a control state of the* i^{th} *component.*
  - α *is a feasible interaction in* γ *associated with a guarded command* (G_α, F_α)*, such that there exists a subset* $J \subseteq \{1, ..., n\}$ *of components with transitions* $\{(q_j, p_j, g_j, f_j, q'_j)\}_{j \in J}$ *and* $\alpha = \{p_j\}_{j \in J}$.
  - $g = \bigwedge_{j \in J} g_j \wedge G_\alpha$.
  - $f = F_\alpha; [f_j]_{j \in J}$*. That is, the computation starts with the execution of* F_α *followed by by the execution of all the functions* f_j *in some arbitrary order. The result is independent of this order as components have disjoint sets of variables.*
- P *is a set of exported ports. Indeed, a connector can be associated with exported ports. This allows a connector to be used as a port in other connectors, and create structured connectors. Those ports allow also to build compound components.*

Example 7 *Figure 3.7(a) shows a compound component* ***Sender*** *consisting of three components,* ***Send1****,* ***Send2*** *and* ***Send3*** *of type Basic (described in Figure 3.2). They interact by using connectors of type Max described in Figure 3.4 to compute the maximal value produced by the components. It exports two ports* p'_2 *and* p'_1*. Port* p'_2 *results from the components synchronizations through their* p_2 *ports. Port* p'_1 *is the exported port* p_1 *of the* ***Send3*** *component. Ports* p_1 *of* ***Send1****,* ***Send2*** *are wrapped into singleton connectors since they are neither exported by the compound component nor involved in any interaction. Figure 3.7(b) presents the corresponding BIP code.*



(a) Composite component representation.

compound type `Sender`

component `Basic Send1`
component `Basic Send2`
component `Basic Send3`

connector `Max Max1(Sender1.`p_2`,Send2.`p_2`)`
connector `Max Max2(Max1.`p_3`,Send3.`p_2`)`
connector `Singleton Sing1 (Send1.`p_1`)`
connector `Singleton Sing2 (Send2.`p_1`)`

export port `Intport` p_2' **is** `Max2.`p_3
export port `Intport` p_1' **is** `Send3.`p_1

end

(b) BIP code of the compound component.

Figure 3.7: An example of compound component in BIP.

4 The BIP Tool-Chain

This section presents the implementation of the BIP framework, formally described in the previous sections, in the form of a tool-chain called the BIP tool-chain. The BIP Tool-chain provides a complete implementation, with a rich set of tools for the modeling, the execution and the verification (both static and on-the-fly) of BIP models.

4.1 General Overview

The overview of the BIP tool-chain is shown in figure 3.8. It includes the following tools:

- *The BIP language.* It is used to build models using components, connectors and priorities and describes components architecture. It is used for the BIP description source.

- *Source-to-source transformation* tools. They are used to transform various programming models, using different languages, into BIP models. The translation of a programming model into a BIP model allows its representation, in a rigorous semantic framework. There exist several translations, including LUSTRE, MATLAB/Simulink, AADL, GenoM applications, NesC/TinyOS applications, C software and DOL systems.

- *The compiler.* It generates a BIP model from the BIP description source. It uses *The BIP meta-model* as the intermediate representation of BIP models and to implement model transformations. It includes :
  - *The BIP meta-model.* It represents a template of the structure of the intermediate model to be generated from a BIP program, using EMF. All the modeling elements, presented in the BIP language, have a representation in the BIP model in the form of the data-structure. Class diagrams are used to define the relations between the different modeling elements, through inheritence and containment.
  - *The parser.* It analyses a BIP description source and generates an intermediate model conforming to the BIP meta-model. It performs syntactic analysis of the input program conforming to the *BIP grammar* and reports the programming errors.
  - *Model-to-model transformation* tools. They are used in order to perform useful static transformations for systems optimizations including run-time. The transformations use a set of correct-by-construction models and preserve functional properties. Moreover, they can take into account extra functional constraints. There exist three types of transformations, architecture optimizations, such as flattening the hierarchy and transforming structured connectors to flat connectors [34], distributed implementation [27, 61], such as the replacement of atomic multiparty interactions by protocols using asychronous message passing (send/receive primitives) and memory management.
  - *The code generator.* It generates C++ code from the model produced by the parser. The code generator has options for generating application code for the single-threaded BIP Engine, the multi-threaded BIP Engine and the distributed BIP implementation.
- *D-Finder.* It is a compositional verification tool for deadlock detection and generation of invariants [16, 20]. Verification is applied only to high level models for checking safety properties such as invariants and deadlock-freedom.
- *The BIP Execution engines.* They are middleware responsible for the coordination of atomic components, that is, they apply the semantics of the interaction and priority layers of BIP. Execution engines are used for execution, simulation, run-time verifications, debug or state-space exploration(i.e. all traces) of BIP models. There are currently two engines available, the single-threaded engine and the multi-threaded engine, and a distributed implementation of BIP.

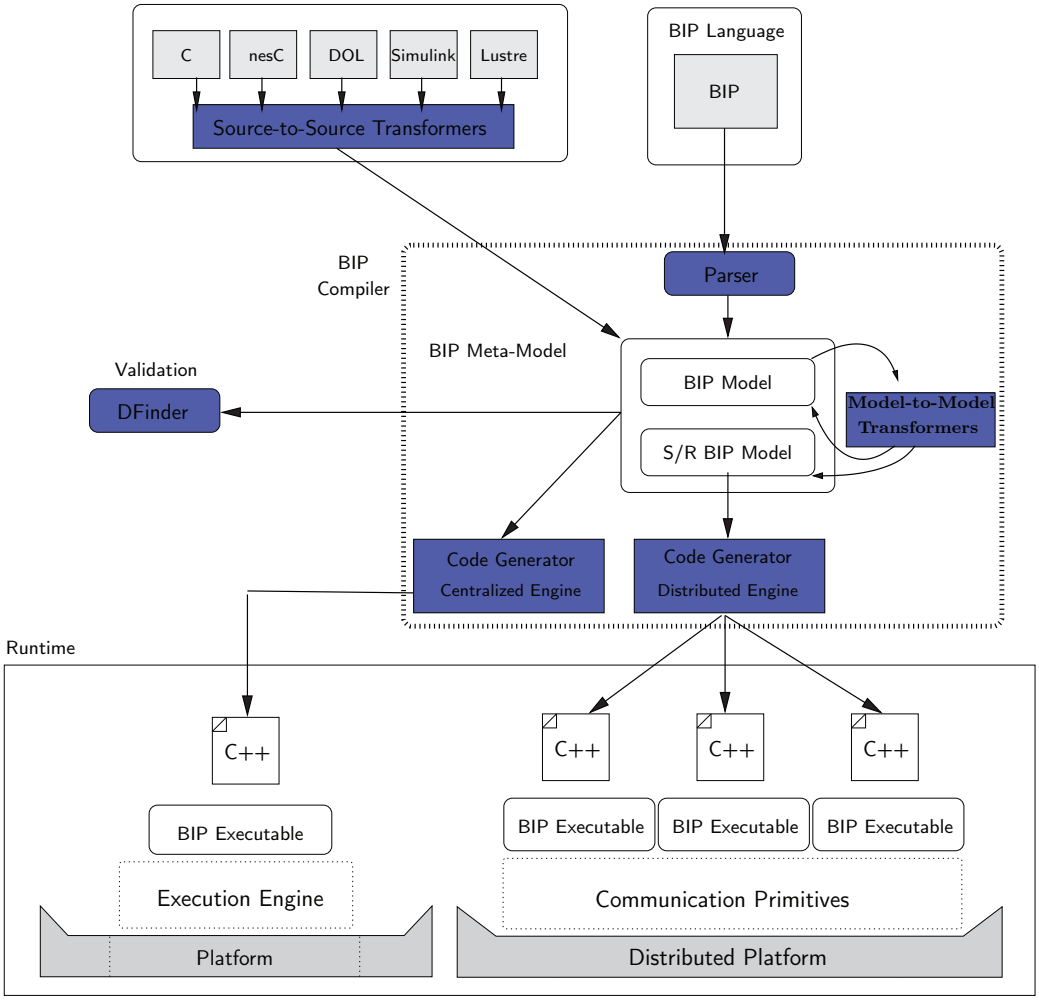


Figure 3.8: The BIP Tool-Chain.

4.2 The BIP Execution Engines

The BIP execution engines and the distributed BIP implementations directly implement the BIP operational semantics. They play the role of the co-coordinator in selecting and executing interactions between the components, taking into account the glue specified in the input component model. It monitors the state of the components and considering the interaction model, finds all the enabled interactions. It then applies the priority rules to eliminate the interactions with low priority, and selects one amongst the maximal enabled, for execution.

Here is the presentation of the current Engines.

The Single-Threaded BIP Engine

From a BIP model, a compiler is used to generate C++ code for atomic components and glue. The code is orchestrated by a sequential engine that interprets the BIP operational semantic rules. The architecture of the sequential implementation is shown in Figure 3.9.

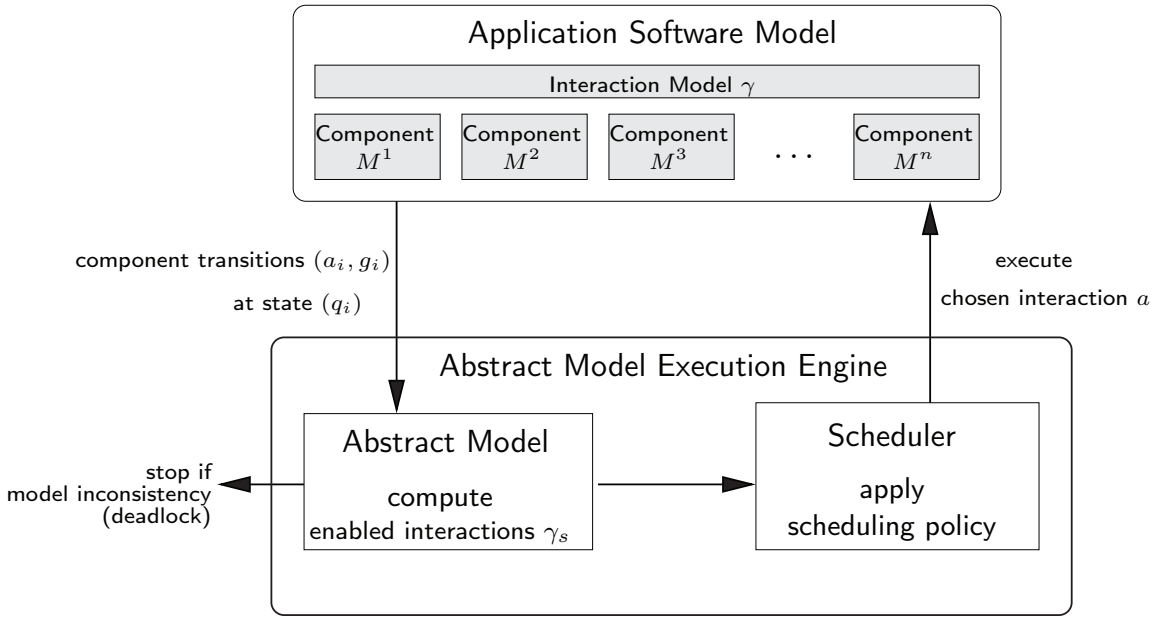


Figure 3.9: BIP model Execution Engine.

The engine computes from the set of ports for each atomic components and defined by connectors, the set of enabled interactions. It chooses an interaction $a = \{ a_i \mid i \in I \} \in \gamma_s$ enabled at state s. The choice of a depends on the considered scheduling policy. For instance, EDF (Earliest Deadline First) scheduling policy can be used. It executes a that corresponds to the execution of all atomic components involved in the interaction a_i, $i \in I$, followed by the execution of the data transfer function F_a and the update of control locations.

Algorithm 1 gives an implementation of the Execution Engine for the composition of BIP models. It basically consists of an infinite loop that first computes enabled interactions at current state s of the composition (line 3). It stops if no interaction is possible from s (i.e. deadlock) at line 5. Otherwise, it chooses an interaction a (line 7), executes the data transfer function F_a associated to it (line 8) and executes a (line 12). Finally, the state s is updated in order to take into account the execution of a (line 14).

Algorithm 1 Single-Threaded Execution Engine

Require: Model $M^i = (\mathsf{Q}_i, \longrightarrow_i)$, $1 \leq i \leq n$, initial control location $(q_0^1, \ldots, q_0^n)$, set of interactions γ

1: $s = (q_1, \ldots, q_n) \leftarrow (q_0^1, \ldots, q_0^n)$ *// init.*
2: **loop**
3: $\gamma_s = EnabledInteractions(s)$
4:
5: **if** $\exists a \in \gamma_s$ **then**
6: $a = \{\ a_i \mid i \in I\ \} \leftarrow EDFScheduler(\gamma_s)$
7: $ExecuteDataTransfer(F_a)$ *// execute transfer function*
8:
9: **for all** $i \in I$ **do**
10: Execute(a_i) *// execute involved component*
11: $q_i \leftarrow q_i'$ *// update control location*
12: **end for**
13: **else**
14: **exit**(`DEADLOCK`)
15: **end if**
16: **end loop**

The Multi-Threaded BIP Engine

The implementation of the multi-threaded implementation whith centralized engine is based on the notion of partial state semantics where interactions are allowed to fire as soon as only the involved components are stable [12]. Each atomic component is assigned to a different thread (process), the engine being assigned to a thread as well. Algorithm 2 describes the execution of an atomic component. Each atomic component performs its computations locally and then, when it reaches a stable state, it notifies the engine about the ports P_i on wich it is willing to interact (line 3). It waits for the engine to select the port p_i to be executed upon the chosen interaction (line 4).

Algorithm 2 Atomic component Execution

1: $P_i = initialize()$ *// init.*
2: **loop**
3: notify(P_i) *// notifies the engine*
4: wait(p_i) *// waits for execution*
5:
6: $P_i = execute(p_i)$
7: **end loop**

The engine is parametrized by an oracle. Iteratively, the engine computes feasible interactions available on state components. Then, if such interactions exist and the oracle allows them, the engine selects one for execution and notifies the involved components.

Iteratively, the Engine receives the sets of ports and the local states of components

ready to interact (line 2). Depending on this information, the engine computes the feasible interactions (line 3). It chooses a feasible interaction, which is allowed by the oracle O (line 4) . If such an interaction exists, the engine executes it by notifying sequentially, in some arbitrary order, all the involved components (line 12). Otherwise, it's a deadlock (line 15).

Algorithm 3 Multi-Threaded Execution Engine

Require: Model $M^i = (\mathsf{Q}_i, \longrightarrow_i)$, $1 \leq i \leq n$, initial control location $(q_0^1, \ldots, q_0^n)$, set of interactions γ, Oracle O.

```
1:  loop
2:    wait(P_i)                                   // waits for components ready to interact
3:    γ_s = EnabledInteractions(P_i)
4:    γ_o = restriction(γ_s, O)                   // feasible interactions
5:
6:    if ∃a ∈ γ_o then
7:      a = { a_i | i ∈ I } ← Scheduler(γ_s)
8:      ExecuteDataTransfer(F_a)                  // execute transfer function
9:
10:     for all i ∈ I do
11:       notify(M_i, a_i)                        // notifies involved component
12:       q_i ← q_i'                              // update control location
13:     end for
14:   else
15:     exit(DEADLOCK)
16:   end if
17: end loop
```

The Distributed BIP Implementation

Currently, powerful hardware platforms are needed for executing applications on multi-core or many-core platforms. The application code should be optimally distributed over the platforms to take advantage of its computing power. Although distributed systems are widely used nowadays, their implementation is still time-consuming and an error-prone task. Coordination in BIP is achieved through multi-party interactions (i.e., those across multiple components), and scheduling by using dynamic priorities. Transforming the semantics of BIP—which is based on an atomic interaction execution and is defined on a global state model—into a distributed implementation is clearly a nontrivial task.

A generic framework allowing the transformation of high-level BIP models into distributed implementations has been recently developed [27, 61]. The method involves BIP to BIP transformations preserving observational equivalence. It transforms multi-party interactions into asynchronous message passing, that is, send/receive primitives. The target *Send/Receive* BIP model is structured in three layers (see Figure 3.10): *(i)* the *component layer* corresponds to a modified behavior of the components of the original model; *(ii)* the *interaction protocol* consists of a set of components such that each component detects enabledness of a subset of interactions of the original model using partial-state knowledge, and executes them after resolving conflicts (e.g., regarding which interaction to execute when there is more than one involving the same port) either locally or with the help of

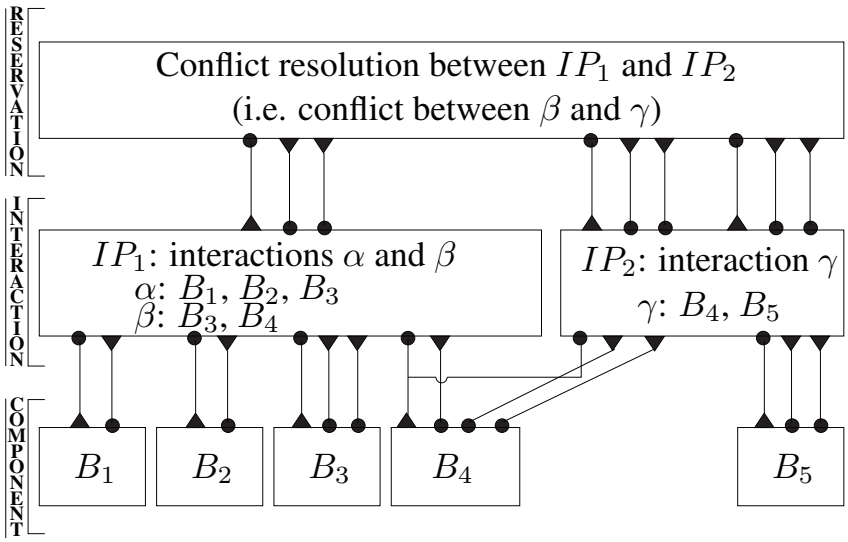


Figure 3.10: Send/Receive BIP model obtained from BIP to BIPtransformations.

the third layer; *(iii)* the *reservation protocol* resolves conflicts between components of the interaction protocol layer using committee coordination algorithms such as the token-ring distributed algorithm or the distributed dining philosophers algorithm. Notice that the obtained Send/Receive BIP model depends on a user-defined partition of the interactions of the original model, associating subsets of interactions to components of the interaction protocol layer.

A C++ code generator has been developped. It performs, given a user-defined mapping of the components of a Send/Receive BIP model, the generation of distributed implementations using communication mechanisms offered by the platform. We have the following backends: Unix processes communicating through TCP sockets, MPI, and threads using semaphores and shared memory. Efficient monolithic code can be produced by merging components using another BIP to BIP transformation, according to the mapping of the components.

The method has been fully implemented in a toolset allowing the automatic generation of distributed implementations from BIP models. It is parameterized by the partitioning of interactions, a committee coordination algorithm, and the mapping of components. The performance of the resulting implementation strongly depends on the choice of these parameters [61].

5 Modeling Time using BIP

In BIP, component behaviors are automata extended with data. There is no explicit notion of time, that is, conditions (guards) for enabledness of interactions between components may only depend on the values of components variables. One possible solution is to enforce timing constraints directly in the components by calling primitives of the platform on transitions. Another solution is to introduce Clocks that are represented by integer variables. Components are then strongly synchronized by a connector Tick for incrementing synchronously all the clocks. Urgency of transitions can be expressed by giving more priority to urgent transitions than to the connector Tick. We now explain timed systems modeling in BIP through the following scheduling tasks example.

Example 8 *The example models two tasks T_1 and T_2 processing events that are produced by an event generators G_1. Both tasks are executing on a shared CPU. The event generator G_1 activates task T_1 by sending an event. It can either process the event alone or ask the task T_2 to make some additional computation before completion. The block diagramm of the system is shown in Figure 3.11. We design a model to measure the total delay of an event, starting from its creation from the event generator till it is processed by T_1.*

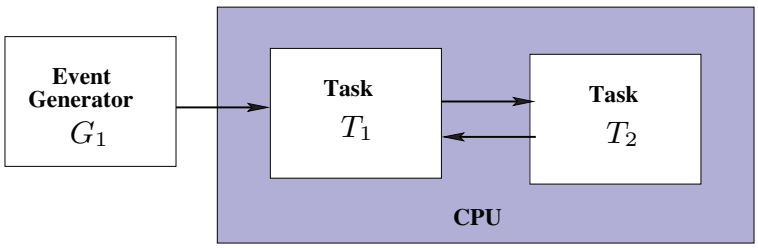


Figure 3.11: Scheduling of timed tasks.

The basic components for the model are Task and EventGenerator. A generic timed model of Task is shown in Figure 3.12, which can be used either as a simple task, or as a collaborative task. It has 5 ports, *start* to start the execution when receiving an event, $finish$ to end the execution, *ask* and *resume* to ask another task to make additional computation when necessary, and *tick* to measure time progress. It has a time variable x representing a clock to keep track of the execution time. Variable x is set to zero at the beginning of the execution, in the transition labelled by port *start*. Transition labelled by port $finish$ has a guard over the timed variable x such as $[x < WCET]$. It means that the task has to finish before its worst case execution time WCET. Time progress is measured in terms of ticks. From each state, either an enabled transition can be executed or time can progress by one time unit.

The model of an EventGenerator is shown in Figure 3.13. It has one state *Run* and two ports *tick* and *go*. It has a timer variable y representing a clock in order to generate periodically an event. From state *Run*, time can progress until y reaches the deadline $y == D$.

The model of the system is represented in Figure 3.14. The system is a serial connection of G_1 instance of the event generator component, and two instances T_1 and T_2 of the task

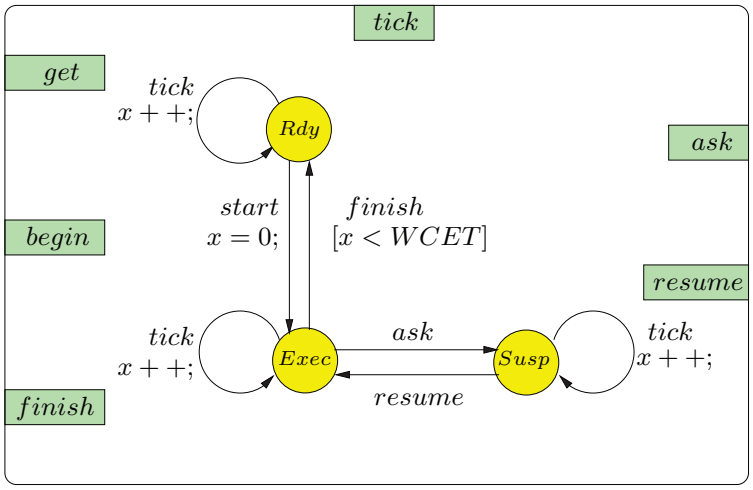


Figure 3.12: Task component.

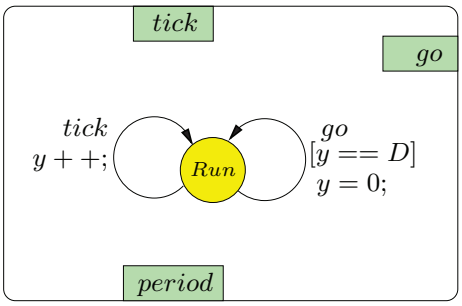


Figure 3.13: Event Generator component.

component. The transmission of an event, i.e. synchronization between G_1 and T_1 is modeled by the connector involving ports $G_1.go$ and $T_1.start$. The synchronization between T_1 and T_2 is modeled by the connector involving ports $T_1.ask$ and $T_2.start$ when asking for additional computation, and the connector involving ports $T_1.resume$ and $T_2.finish$ when T_2 finishes the computation. The three components are strongly synchronized by a Tick connector, that is, all ports of type *tick* are synchronized.

To inforce the urgency criteria of the transitions, we have to use priorities. For example, to enforce the urgency for the periodic transition of G_1, we have the rule :

$$Tick : G_1.tick, T_1.tick, T_2.tick < G_1.go$$

We also have the following rules to enforce the completition of Tasks T_1 and T_2:

$$Tick : G_1.tick, T_1.tick, T_2.tick < T_1.finish$$

$$Tick : G_1.tick, T_1.tick, T_2.tick < T_2.finish$$

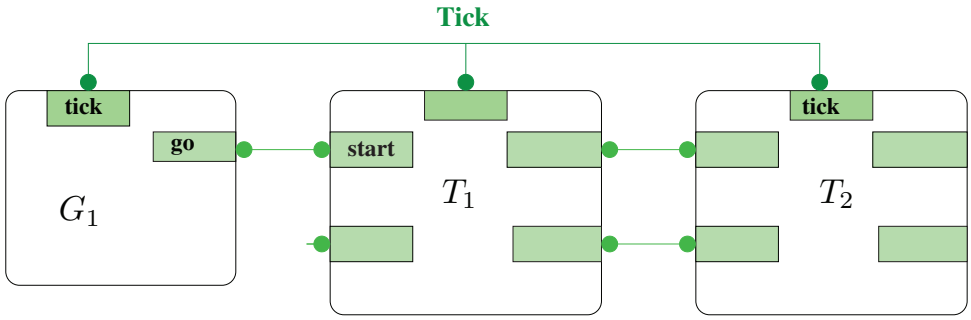


Figure 3.14: Modelization of the scheduling of timed tasks example.

6 Conclusion

We have presented the BIP framework, a component-based framework for modeling heterogeneous systems. The BIP model is the superposition of three layers: the lower layer describes the behavior of a component as a transition system; the intermediate layer consists of the interactions between transitions of the layer underneath; the upper layer describes the priorities characterizing a set of scheduling policies for interactions. Such a layering offers a clear separation between components behaviors and the structure of the system (interactions and priorities).

Component-based approach is aimed to deal with the complexity of systems. It is based on the idea of building a complex system by assembling basic components (blocks). BIP modeling framework allows dealing with complexity of systems by providing incremental composition of heterogeneous components. It also considers correctness-by-construction for a class of essential properties such as deadlock-freedom [53]. The BIP tool-chain has been developed providing automated support for component integration and generation of glue code meeting given requirements. Efficient model transformations and verification methods have also been studied and implemented in the BIP toolchain.

We have seen that global tick synchronizations between the components are used to model time progress, that is, time progresses synchronously. Each component counts the time progress in terms of *tick* in order to execute periodic tasks. This modeling method is not sufficiently effective and should even be avoided. One of the reasons is that the timing behavior of the model will be untrackable and doesn't allow the use of verification techniques on timing features. Moreover, using such tick synchronizations is even less effective due to the periodic execution of ticks. The engine loses in CPU utilization when it executes tick interactions to count time. The method is also inefficient when the period of the execution of the Tick connector is small compared to the actual period of activation of the components, because the engine wakes up to count time even if there are no interactions to execute. This approach requires also for execution times of interactions to be bounded by this period, which is a very strong assumption. It is also difficult to verify if the task completes before its WCET or if an event occurs at its periodic time when executing the model on a real platform. Finally, since Tick strongly synchronizes all components in all states, such models can easily deadlock.

In the following chapters, we extend the BIP framework with timing features according

to a rigorous model-based design. It directly expresses timing constraints in components using clocks that are mapped to the platform clock for implementation, which avoids ticks synchronizations.

Part II

Contribution

Chapter 4

Time-Safety and Time-Robustness

Correct and efficient implementation of general real-time applications remains by far an open problem. A key issue for building real-time systems is meeting timing constraints, that is, a system reacts within user-defined bounds such as deadlines and periodicity. The satisfaction of timing constraints depends on features of the execution platform, in particular its speed.

Rigorous design methodologies are model-based, that is, they explicitly or implicitly associate with a real-time application software an abstract model– a platform-idependent abstraction of the real-time systems– expressing timing constraints to be met by the implementation. We have seen in Chapter 2 , existing rigorous implementation techniques that are only applicable to specific classes of systems. In this chapter, we generalize these approaches by introducing a general model-based implementation method for real-time systems. It relies on implementation theories that allow deciding if a given application software, i.e. its associated model, can be implemented on a given platform, that is, for particular execution times of actions. Our method is based on the use of two models:

- The **abstract model** represents the behavior of a real-time software as a timed automaton. The later describes user-defined platform-independent timing constraints requirements.
- The **physical model** is introduced to represent the behavior of the real-time software running on a given platform. It is obtained by assigning execution times to the transitions of the abstract model.

We also propose the notion of implementability, that is, the notion of correctness of the implementation of real-time systems models. It relies on the properties of *time-safety* and *time-robustness*. Time-safety is a necessary condition for the implementability of physical models. It means that the platform is fast enough to meet the timing constraints described in the abstract model. Time-robustness is a powerful property of a system correctness. It ensures that the time-safety property of a given system is steel achieved when speed of the execution platform increases.

In the first section, we describe the notion of abstract models. In Section 2, we first discuss the correctness issue in the implementation of such abstract models, e.g. how to ensure a correct tracking of physical time by avoiding the drift phenomena between abstract time and physical time. In the second, we give the definition of physical models. Finally, in Section 3, we introduce the notions of time-safety and time-robustness and we show that time-determinism is a sufficient condition for time-robustness.

1 Abstract Models

1.1 Preliminary Definitions

In this chapter, we use *timed automata* [6, 7, 75] to model the behavior of real-time systems requirements over time. Timed automata provide a simple and powerful way to annotate state-transitions automata with *timing constraints* using *clocks*.

Clocks

Clocks are used in order to measure time progress. They are variables increasing synchronously. They can be valued either as integer or as real. A clock can be set to 0 (independly of other clocks) with a transition of the automaton. It keeps track of the time elapsed since its last reset.

We denote by $\mathbb{T}$ the set of clock values. $\mathbb{T}$ can be the set of non-negative integers $\mathbb{N}$ or the set of non-negative reals $\mathbb{R}^+$.

Definition 11 (Clock Valuation) *Given a set of clocks* X*, a* valuation *of the clocks* $v : \mathsf{X} \to \mathbb{T}$ *is a function associating with each clock* x *its value* $v(x)$*. Given a subset of clocks* $\mathsf{X}' \subseteq \mathsf{X}$ *and a clock value* $l \in \mathbb{T}$*, we denote by* $v[\mathsf{X}' \mapsto l]$ *the valuation defined by:*

$$v[\mathsf{X}' \mapsto l](x) = \begin{cases} l \text{ if } x \in \mathsf{X}' \\ v(x) \text{ otherwise.} \end{cases}$$

Timing Constraints

The transitions of the automaton have certain constraints on the clock values. Indeed, a transition may be taken only if the current values of the clocks satisfy the associated constraint. Those constraints are also called *guards*.

Definition 12 (Timing Constraints) *Given a set of clocks* X*, guards over* X *are finite conjunctions of typed intervals. They are expressions used to specify when actions of a system are enabled. They are of the form* $[l \leq x \leq u]^\tau$ *where :*

- x *is a clock,*
- $l \in \mathbb{T}$ *and* $u \in \mathbb{T} \cup \{+\infty\}$*,*
- τ *is an* urgency type*, that is,* $\tau \in \{ \mathsf{l}, \mathsf{d}, \mathsf{e} \}$*, where :*
  - l *is used for* lazy *actions (i.e. non-urgent),*
  - d *is used for* delayable *actions (i.e. urgent just before they become disabled),*
  - e *is used for* eager *actions (i.e. urgent whenever they are enabled).*

We write $[x = l]^\tau$ for $u = l$.

Simplification rule

Considering that urgency types are ordered as follows: $\mathsf{l} < \mathsf{d} < \mathsf{e}$, we consider the following simplification rule [31]:

$$\begin{aligned} & [l_1 \leq x_1 \leq u_1]^{\tau_1} \wedge [l_2 \leq x_2 \leq u_2]^{\tau_2} \\ \equiv \; & [(l_1 \leq x_1 \leq u_1) \wedge (l_2 \leq x_2 \leq u_2)]^{\mathbf{max}\ \tau_1, \tau_2} \end{aligned}$$

By application of this rule, any guard g can be put into the following form:

$$g = \Big[\bigwedge_{i=1}^{n} l_i \leq x_i \leq u_i \Big]^{\tau}.$$

- The predicate of g characterizes the valuations of the clocks for which g is enabled. This predicate is the expression: $\bigwedge_{i=1}^{n} l_i \leq v(x_i) \leq u_i$.
- The predicate $\mathsf{urg}[g]$ that characterizes the valuations of clocks for which g is urgent is also defined by:

$$\mathsf{urg}[g] \iff \begin{cases} \mathsf{false} & \text{if } g \text{ is lazy} \quad (\text{i.e. } \tau = \mathsf{l}) \\ g \wedge \neg(g_{>}) & \text{if } g \text{ is delayable} \quad (\text{i.e. } \tau = \mathsf{d}) \\ g & \text{if } g \text{ is eager} \quad (\text{i.e. } \tau = \mathsf{e}), \end{cases}$$

where $g_{>}$ is a notation for the predicate defined by :

$$g_{>}(v) \iff \exists \varepsilon > 0 \ . \ \forall \delta \in [0, \varepsilon] \ . \ g(v + \delta).$$

We denote by $\mathsf{G}(\mathsf{X})$ the set of guards over a set of clocks X.

1.2 Definition of Abstract Models

An abstract model describes the abstract behavior of the system based on timed automata. The model takes into account only platform-independent timing constraints expressing user-dependent requirements (e.g. deadlines, periodicity, etc.). The actions of the model represent statements of the application software. Timing constraints are guards of transitions that take into account these requirements. Using timed automata allows more general timing constraints that LET.

We are able to express lower bounds, upper bounds, time non-determinism etc.. We also give the semantics of abstract models that assumes timeless execution of actions.

Definition 13 (abstract model) *An* abstract model *is a timed automaton* $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ *such that:*

- A *is a finite set of* actions*,*
- Q *is a finite set of* control locations*,*
- X *is a finite set of* clocks*,*
- *and* $\longrightarrow \subseteq \mathsf{Q} \times (\mathsf{A} \times \mathsf{G}(\mathsf{X}) \times 2^{\mathsf{X}}) \times \mathsf{Q}$ *is a finite set of labeled transitions. A transition is a tuple* (q, a, g, r, q') *where :*
  - a *is an action executed by the transition.*
  - g *is a* guard *over* X*.*
  - r *is a subset of clocks that are reset by the transition.*

  We write $q \xrightarrow{a,g,r} q'$ *for* $(q, a, g, r, q') \in \longrightarrow$*.*

The semantics of an abstract model is a *Timed Transition System(TTS)*, that consists of two types of transitions: actions and time steps. An execution sequence of the abstract model is a sequence of such transitions with an alternance between actions and time steps. The definition of an abstract model semantics is as follows:

Definition 14 (abstract model semantics) *An abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ defines a transition system TS. States of TS are of the form (q, v), where q is a control location of M and v is a valuation of the clocks X.*

It has two types of transitions:

- **Actions**. It corresponds to the beginning of the execution of the action a. It is triggered by guard g and it resets the set of clocks r.
  We have:
  $(q, v) \xrightarrow{a} (q', v[r \mapsto 0])$ if $q \xrightarrow{a,g,r} q'$ is a transition of the abstract model and $g(v)$ is true.
  We assume that the execution of action a takes zero time.
- **Time steps**. It corresponds to a *waiting time* $\delta \in \mathbb{T}$, $\delta > 0$ transition.
  We have:
  $(q, v) \xrightarrow{\delta} (q, v+\delta)$ if for all transitions $q \xrightarrow{a,g,r} q'$ of M and for all $\delta' \in [0, \delta[$, $\neg \mathsf{urg}[g](v+\delta')$.
  This means that it is not possible to wait δ time if there is an urgency at time $\delta' < \delta$.

In contrast to other models of timed automata [5], for abstract models it is always possible to execute a transition from a state [31]. If no action is possible only time can progress. We call this situation a *deadlock*. Henceforth, we consider abstract models $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ such that any circuit in the graph $\longrightarrow$ has at least a clock that is reset and tested against a positive lower bound, that is, M is structurally non-zeno [30]. This class of abstract models does not have time-locks, that is, time always eventually progresses.

The definition of an execution sequence of an abstract model is as follows:

Definition 15 (execution sequence) *A finite (resp. an infinite)* execution sequence *of M from an* initial *state (q_0, v_0) is a sequence that alterns actions and time-steps:*
$(q_i, v_i) \xrightarrow{\sigma_i} (q_{i+1}, v_{i+1})$ *of M, $\sigma_i \in \mathsf{A} \cup \mathbb{T}$ and $i \in \{ 0, 1, 2, \ldots, n \}$ (resp. $i \in \mathbb{N}$).*

Waiting Time

Following the abstract model semantics in Definition 14, it is not possible for a waiting time transition to wait δ time if there is an urgency at time $\delta' < \delta$. Indeed, given a state, there should be restrictions on the waiting time allowed by the semantics because they influence the date on wich actions are triggered. If the waiting time is too long, the deadlines over actions might be missed. We define a function that gives the maximal waiting time allowed for a waiting transition from a state as follows.

Given an abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$, we denote by $\mathsf{wait}(q, v)$ the *maximal waiting time* allowed at state (q, v), defined by:

$$\mathsf{wait}(q, v) = \mathbf{min}\left(\left\{\ \delta \geq 0 \ \middle|\ \bigvee_{q \xrightarrow{a_i, g_i, r_i} q_i} \mathsf{urg}[g_i](v+\delta)\ \right\} \cup \{\ +\infty\ \}\right).$$

The maximal waiting time in a state (q, v) is then the minimum waiting time of actions of all the transitions from state (q, v).

Notice that we have $\mathsf{wait}(q, v+\delta) = \mathsf{wait}(q, v) - \delta$ for all $\delta \in [0, \mathsf{wait}(q, v)]$. A waiting time $\delta > 0$ is allowed in M at state (q, v), that is, $(q, v) \xrightarrow{\delta} (q, v+\delta)$, if and only if $\delta \leq \mathsf{wait}(q, v)$.

Example 9 *Consider an abstract model* $M = (\mathsf{A}, \{q_0, q_1, q_2\}, \{x\}, \longrightarrow)$ *with a set of actions* $\mathsf{A} = \{a, b, c, i\}$*, a single clock* x *and the following set of transitions (see Figure 4.1):*

$$\begin{aligned} \longrightarrow \; = \{ \;& (q_0, a, [0 \leq x \leq +\infty]^{\mathsf{e}}, \emptyset, q_1), \\ & (q_1, b, [51 \leq x \leq 60]^{\mathsf{d}}, \emptyset, q_2), \\ & (q_1, c, [0 \leq x \leq 50]^{\mathsf{l}}, \emptyset, q_2), \\ & (q_2, i, [100 \leq x \leq 120]^{\mathsf{d}}, \{x\}, q_0) \quad \}. \end{aligned}$$

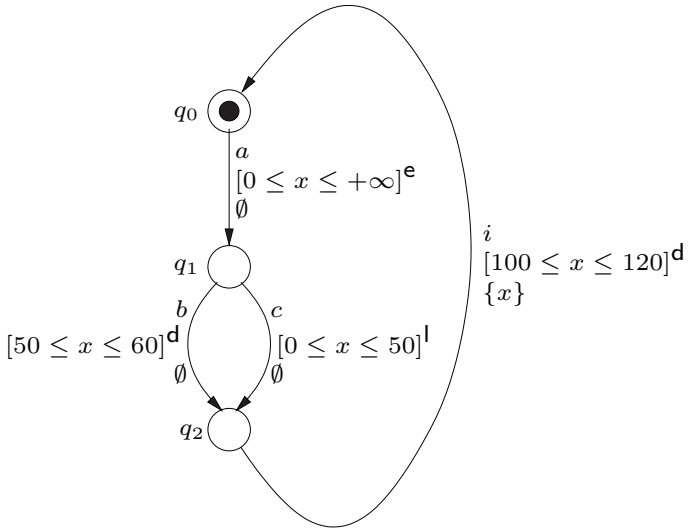


Figure 4.1: Example of abstract model.

Consider execution sequences of M from the initial state $(q_0, 0)$.

- From initial control location q_0, we have: $(q_0, 0) \xrightarrow{a} (q_1, 0)$. Since the only transition issued from initial control location q_0 of M is eager and its guard is always true, only action a is possible from the initial state $(q_0, 0)$.

- At state $(q_1, 0)$, the system cannot wait for more than $\mathsf{wait}(q_1, 0) = 60$ time units due to the delayable guard of b, which is urgent when x reaches 60.

  - If b is executed, the waiting time δ_1 at $(q_1, 0)$ must satisfy $50 \leq \delta_1 \leq 60$.
  - If c is executed, the waiting time δ_1 at $(q_1, 0)$ must satisfy $0 \leq \delta_1 \leq 50$

  The execution of b or c leads to state (q_2, δ_1).

- At state (q_2, δ_1), the system cannot wait for more than $\mathsf{wait}(q_2, \delta_1) = 120$ time units due to the delayable guard of i, which is urgent when x reaches 120. Time must progress by δ_2 time units before executing i, such that $100 - \delta_1 \leq \delta_2 \leq \mathsf{wait}(q_2, \delta_1) = 120 - \delta_1$, that is, $100 \leq \delta_1 + \delta_2 \leq 120$. Then, action i is executed leading back to the initial state $(q_0, 0)$.

This demonstrates that execution sequences of M are infinite repetitions of sequences of two following forms:

1. $(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{\delta_1} (q_1, \delta_1) \xrightarrow{b} (q_2, \delta_1) \xrightarrow{\delta_2} (q_2, \delta_1 + \delta_2) \xrightarrow{i} (q_0, 0)$ where $50 \leq \delta_1 \leq 60$ and $100 - \delta_1 \leq \delta_2 \leq 120 - \delta_1$

2. $(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{\delta_1} (q_1, \delta_1) \xrightarrow{c} (q_2, \delta_1) \xrightarrow{\delta_2} (q_2, \delta_1 + \delta_2) \xrightarrow{i} (q_0, 0)$ where $0 \leq \delta_1 \leq 50$ and $100 - \delta_1 \leq \delta_2 \leq 120 - \delta_1$.

2 Physical Models

In the previous section, we presented the notion of abstract models, based on timed automata. It allows the representation of real-time systems requirements independently of any execution platform. However, we can ask the question of the relevance of the use of these mathematical objects when it comes to real systems implementation. The semantics of these models is indeed very precise and assumes in particular, immediate transitions and clocks infinitely precise. None of the existing execution platforms allows the implementation of such precise automaton, and nothing assumes that the properties verified on the theoretical model will be preserved by the implementation.

In this section, we expose the problem by reasoning on a simple example of a periodic task. We will also present the notion of physical models that allows to take into account the differences between the theoretical and the real behavior of the system. Finally, we introduce the notions of time-safety and time-robustness that allows us to verify if a physical model is a correct implementation of its abstract model.

2.1 Time Tracking

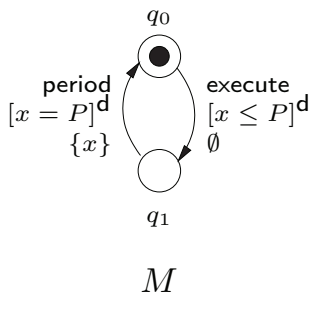
Time and timing features are an important aspect of modeling real-time systems. In the abstract model, time has a logical nature (rather than physical), what we also call the abstract time. The abstract time is controllable and doesn't depend on any execution platform features. A key issue for a correct implementation from an abstract model is the correspondence between abstract time and physical time. The implementation of heteregeneous applications onto architecture platforms amounts to adjust the former abstract time demands onto the latter physical time abilities. The physical time is the actual time, i.e. it is the time that actually flows and that cannot be controlled by the system. The abstract time is the time perceived by the program, it should represent the track of the physical time by the abstract model. It is recognized that it may be differences between physical time and abstract time since there will never be a perfect way to measure the real time. However, the divergence between the two should remain under control.

There are different manners for establishing such correspondence as discussed as follows through a simple example, modeling a periodic task.

Periodic Task Example

Consider an abstract model M of a periodic task (see Figure 4.2 (left)) with period P. It consists of two control locations q_0 and q_1, a single clock x, and two transitions. Its behavior involves a cyclic execution of actions execute and period.

- Action execute corresponds to the execution of the task. It is guarded by the timing constraint $x \leq P$ to enforce this execution before the next activation of the task.




```
void main() {
  Timer x();

  while(true) {
    f();
    x.setTimeout(P);
    x.waitTimeout();
    x.reset();
  }
}
```

Figure 4.2: Simple periodic task model (left) and its naive implementation (right).

- Action period corresponds to the activation of the task, that is, it is executed as x reaches P. Its effect is to reset the clock x so that x measures the time elapsed since the last activation of the task.

We consider a naive implementation of M (see Figure 4.2 (right)) as an infinite loop that first executes sequentially the bloc of code `f()`. It then sets a timeout at `P` for a timer `X`. It waits until the timer `X` reaches this timeout, and it resets the timer to start counting another period.

At initialization, the value of the clock x is 0 and the control location is q_0. In real word technical systems, clocks are specific devices that are used to measure the progress of physical time. We assume that the task is executed with an Operating System (OS) that provides timers and mechanisms for resetting timers and waiting for timeouts. We also assume that timers give an exact value of the physical time. Let's consider the execution of a "wait for a timeout" action. It is classically implemented as follows (see Figure 4.3):

1. After the execution of the task, the CPU is released to the OS by performing a context switch.
2. The OS executes as long as the task is "asleep".
3. When the timer x equals the period `P`, an interruption is triggered.
4. The interruption is handled such as to notify the OS that a timeout occurred.
5. Then, the OS switches the context in order to let the task execute.

Drift Between Physical and Abstract Time

We can see that there is a difference between the abstract time and the physical time whenever executing the transition period. Indeed, the execution sequences of the ideal execution of the model is an iteration of the following sequence:

$$(q_0, 0) \xrightarrow{\delta_1} (q_0, \delta_1) \xrightarrow{execute} (q_1, \delta_1) \xrightarrow{\delta_2} (q_1, P) \xrightarrow{period} (q_0, 0)$$

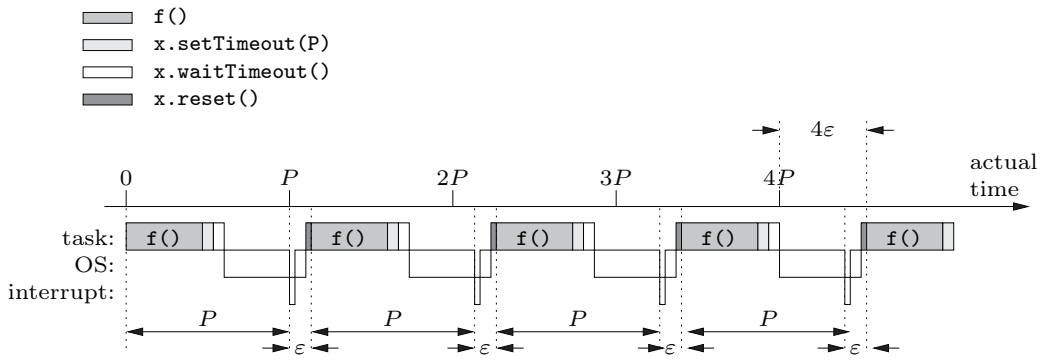


Figure 4.3: Simple periodic task execution.

The abstract time does not take into account any execution times on the target platform. The effect of the reset on the timer `x` is done exactly at P. With the physical execution, although the OS is interrupted exactly when the timer timeouts, i.e after a period of `P`, operations 3, 4 and 5 take time, at least several CPU cycles. In addition, resetting the timer can also take time. This means that the effect of the reset on the timer `x` is delayed by $\varepsilon > 0$ time units. Typically ε is at least few CPU cycles. Assuming that this delay is constant, the execution period of the periodic task considered here becomes $P+\varepsilon$ instead of P (see Figure 4.3). The physical time includes then a drift comparing to the abstract time due to the actual execution times of the operations presented above. Indeed, this drift is given by $t\frac{\varepsilon}{P+\varepsilon}$, where t denotes the global physical time elapsed. This drift can be arbitrarily large as t tends to $+\infty$ (see Figure 4.4).

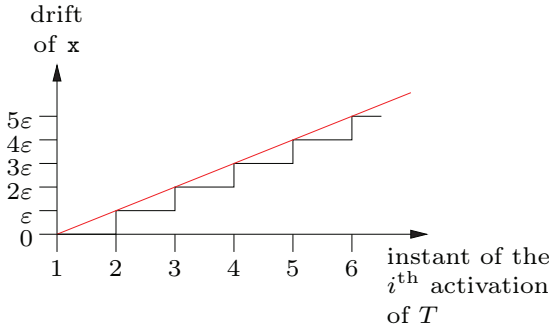


Figure 4.4: Drift Diagramm

Correct Tracking of Time

Let's consider the tracking of the time for the transition labeled by action period of the periodic task example presented above (see Figure 4.2). The transition resets the clock x when x reaches P time units. We consider that action *period* resets a clock x at the global abstract time t. We assume that the reset of x takes $\varepsilon > 0$ time units in the physical model, meaning that the reset of x starts at t and completes at $t + \varepsilon$.

A naive approach is to continuously map the abstract time on the value of the clock x.

Since x is reset at the actual time $t + \varepsilon$ (see Figure 4.5 (left)), using this approach leads to a drift of ε between the abstract model and the physical model. There exist approaches for analyzing how clock drifts may disable properties of an abstract model [4, 45, 91].

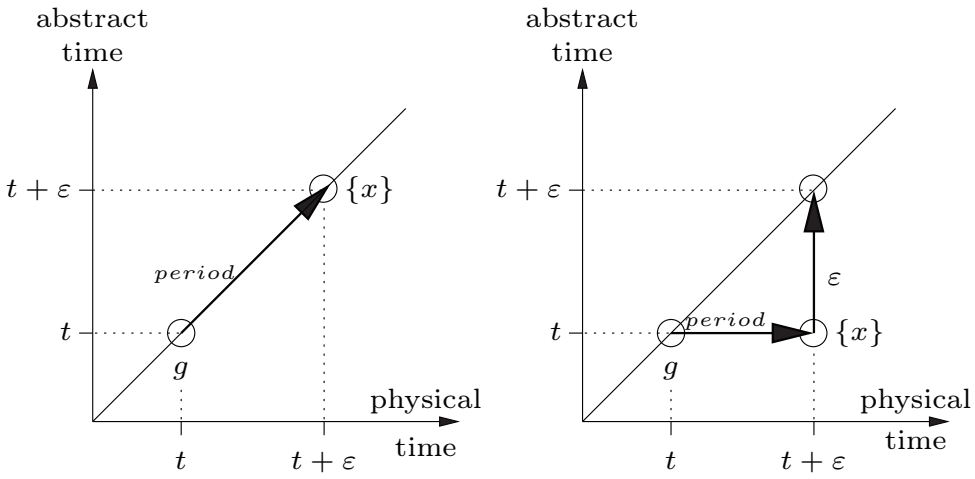


Figure 4.5: Execution based on continuous mapping of the physical time (left) vs frozen clocks (right).

An alternative approach is to ensure a correct tracking of physical time and completely avoid this kind of drift between abstract time and physical time. The proposed semantics for physical models considers that the clock x is reset exactly at model time t. This is implemented by freezing the values of the clocks during the execution of an action, and by updating the clocks after that in order to take into account action execution time.

The clock x is considered to be reset at the model time t even if x is reset at the actual time $t + \varepsilon$. Then abstract time is updated with respect to actual time at $t + \varepsilon$, that is, the current value of x at the actual time $t + \varepsilon$ is ε which complies with the abstract model (see Figure 4.5 (right)).

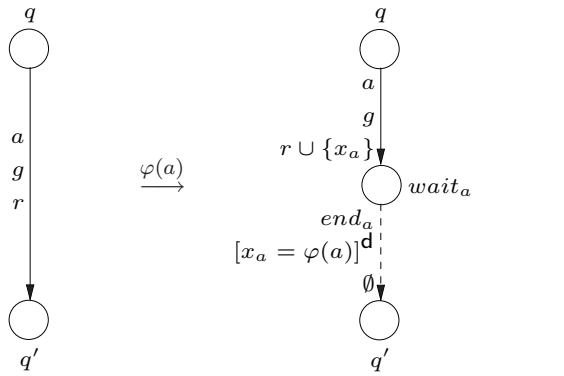
In Chapter (Implementation), we will see that this method allows us detecting violations of timing constraints.

2.2 Definition of Physical Models

Physical models are abstract models modified so as to take into account non-null execution times. They represent the behavior of the application software running on a platform.

Since actions are timeless in abstract models, timing constraints are applied to the instants they occur. In physical model, the start and completion times of an action may not coincide. We consider that timing constraints in physical models apply to start times of the actions. As explained above, we also consider that clock resets associated to each action behave exactly as if they were done at action start time. This allows considering timing constraints that are equalities for non-instantaneous actions. Such constraints are useful for modeling exact synchronization with time, e.g. for describing a periodic execution.

A physical model M_φ corresponds to the abstract model M transformation. by adding actions executions times, that are given by function φ (see Figure 4.6). The physical model definition is formalized as follows.



Transition in M. Corresponding transitions in M_φ.

Figure 4.6: From abstract model to physical model.

Definition 16 (physical model) *Let $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ be an abstract model and $\varphi : \mathsf{A} \to \mathbb{T}$ be an* execution time function *that gives for each action a its execution time $\varphi(a)$.*

The physical model *$M_\varphi = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow, \varphi)$ corresponds to the abstract model M modified so that each transition (q, a, g, r, q') of M is decomposed into two consecutive transitions:*

1. *The first transition $(q, a, g, r \cup \{x_a\}, wait_a)$ corresponds to the beginning of the execution of the action a. It is triggered by guard g and it resets the set of clocks r, exactly as (q, a, g, r, q') in M. It also resets an additional clock x_a used for measuring the execution time of a.*

2. *The second transition $(wait_a, end_a, g_{\varphi(a)}, \emptyset, q')$ corresponds to the completion of a. It is constrained by $g_{\varphi(a)} \equiv [x_a = \varphi(a)]^{\mathsf{d}}$ that enforces waiting time $\varphi(a)$ at control location $wait_a$, which is the time elapsed during the execution of the action a.*

Notice that if (q, v) is a state of the abstract model then (q, v, v') is a state of the physical model such that v' is a valuation of clocks $\{\ x_a \mid a \in \mathsf{A}\ \}$. We compare the behavior of M_φ from initial states of the form $(q_0, v_0, 0)$ with the behavior of M from corresponding initial states (q_0, v_0). In the above definition, an abstract model M and its corresponding physical model M_φ coincide if actions are timeless, that is, if $\varphi = 0$. In a physical model M_φ, every execution of an action a is followed by a wait for $\varphi(a)$ time units as follows:

$$(q, v) \xrightarrow{a} (wait_a, v[r \mapsto 0]) \xrightarrow{\varphi(a)} (wait_a, v + \varphi(a)) \xrightarrow{a_{end}} (q', v + \varphi(a)) \quad (1),$$

which can be abbreviated as

$$(q, v) \xrightarrow{a,\varphi(a)} (q', v[r \mapsto 0] + \varphi(a)) \quad (2).$$

We consider that the execution sequence (1) of M_φ is equivalent to the following execution of the corresponding abstract model M:

$$(q, v) \xrightarrow{a} (q', v[r \mapsto 0]) \xrightarrow{\varphi(a)} (q', v[r \mapsto 0] + \varphi(a)),$$

We note $\equiv$ this type of equivalence relation between the execution sequence of M_φ and M. Notice that a time step $(q', v[r \mapsto 0]) \xrightarrow{\varphi(a)} (q', v[r \mapsto 0] + \varphi(a))$ of M_φ may not be a time step

of M if there exists a transition $q' \xrightarrow{a',g',r'} q''$ such that $\mathsf{urg}[g'](v[r \mapsto 0] + \delta)$ and $\delta \in [0, \varphi(a)[$, that is, the execution time $\varphi(a)$ of a is greater than the maximal waiting time allowed at state $(q', v[r \mapsto 0])$: $\varphi(a) \geq \mathsf{wait}(q', v[r \mapsto 0])$. In this case the physical model violates timing constraints defined in the corresponding abstract model.

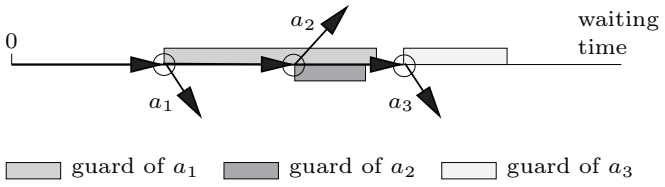


Figure 4.7: Minimal waiting time for action execution.

We consider only execution sequences of physical models M_φ such that the waiting times for the actions are minimal, that is, $(q, v) \xrightarrow{\delta} (q, v + \delta) \xrightarrow{a,\varphi(a)} (q', (v + \delta)[r \mapsto 0] + \varphi(a))$ is an execution sequence of M_φ if $\delta = \mathbf{min}\ \{\ \delta' \geq 0 \mid g(v + \delta')\}$ where g is the guard of the action a at control location q (see Figure 4.7).

3 Time-Safety and Time-Robustness

In the previous sections, we presented the notion of abstract models representing real-time systems requirements independly of any execution platform. Then, we presented the notion of physical models, representing the application software running on a platform. In this section, we present the notions of time-safety and time-robustness that allows to determinate if a physical model is faithful to its abstract model. We consider that a physical model is time-safe if its execution sequences are execution sequences of the corresponding abstract model, that is, execution times are compatible with timing constraints. Furthermore, a physical model is time-robust if reducing the execution times preserves this time-safety property. Execution times reduction may result from the execution of the model on a faster platform.

3.1 Definitions

Definition 17 (time-safety) *A physical model $M_\varphi = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow, \varphi)$ is* time-safe *if for any initial state (q_0, v_0) the set of the execution sequences of M_φ is contained in the set of the execution sequences of M, by considering the equivalence relation $\equiv$ between the execution sequences (see subsection 2.2).*

Definition 18 (time-robustness) *A physical model M_φ is* time-robust *if $M_{\varphi'}$ is time-safe for all execution time functions $\varphi' \leq \varphi$. An abstract model is time-robust if all its time-safe physical models are time-robust.*

Most of the techniques for analyzing the schedulability of real-time systems are based on worst-case estimates of execution times. They rely on the fact that the global worst-case behavior of the system is achieved by assuming local worst-case behavior. Unfortunately, this assumption is not valid for systems that are prone to timing anomalies, that is, a faster local execution may lead to a slower global execution [78]. A time-robust abstract model is

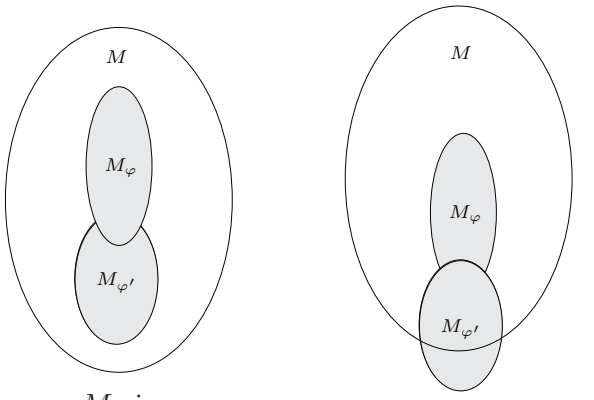


Figure 4.8: Illustration for robustness ($\varphi' < \varphi$).

a system without such timing anomalies, that is, if it is time-safe for execution time function φ, then it is time-safe for execution time functions less than or equal to φ.

Example 10 *We consider the abstract model M given in Example 9 and a family of execution time functions φ such that $\varphi(a) = \varphi(b) = K$, $\varphi(c) = 2K$ and $\varphi(i) = 0$. The behavior of the corresponding physical models M_φ from initial state $(q_0, 0)$ is summarized in Figure 4.9.*

- **Execution sequences of M_φ for $K \leq 40$.**

  For $K \leq 40$, M_φ has execution sequences that are infinite repetitions of the following execution sequences:

  1. $(q_0, 0) \xrightarrow{a,K} (q_1, K) \xrightarrow{c,2K} (q_2, 3K) \xrightarrow{i,0} (q_0, 0)$, and
  2. $(q_0, 0) \xrightarrow{a,K} (q_1, K) \xrightarrow{50-K} (q_1, 50) \xrightarrow{b,K} (q_2, 50 + K) \xrightarrow{50-K} (q_2, 100) \xrightarrow{i,0} (q_0, 0)$.

  These are execution sequences of M (see Example 9) that is, M_φ is time-safe for $K \leq 40$.

- **Execution sequences of M_φ for $K \in [41, 50]$.**

  For $K \in [41, 50]$, M_φ has execution sequences that are repetitions of the following execution sequences:

  1. $(q_0, 0) \xrightarrow{a,K} (q_1, K) \xrightarrow{c,2K} (q_2, 3K)$ leading to a deadlock, and
  2. $(q_0, 0) \xrightarrow{a,K} (q_1, K) \xrightarrow{50-K} (q_1, 50) \xrightarrow{b,K} (q_2, 50 + K) \xrightarrow{50-K} (q_2, 100) \xrightarrow{i,0} (q_0, 0)$.

  Infinite repetitions of the sequence 2 is also execution sequence of M. Other execution sequences of M_φ for $K \in [41, 50]$ are finite and lead to a deadlock. They are not execution sequences of M since M is deadlock-free, that is, M_φ is not time-safe $K \in [41, 50]$.

- **Execution sequences of M_φ for $K \in [51, 60]$.**

  For $K \in [51, 60]$, M_φ has a single execution sequence that is an infinite repetition of the following execution sequence:

  $$(q_0, 0) \xrightarrow{a,K} (q_1, K) \xrightarrow{b,K} (q_2, 2K) \xrightarrow{i,0} (q_0, 0).$$

  This is an execution sequence of M that is, M_φ is time-safe M for $K \in [50, 60]$. However, M_φ is not time-robust since M_φ is not time-safe for $K \in [41, 50]$.

- **Execution sequences of M_φ for $K > 60$.**

  For $K > 60$, M_φ has a single execution sequence $(q_0, 0) \xrightarrow{a,K} (q_1, K)$ leading to a deadlock. This is not an execution sequence of M since M is deadlock-free, that is, M_φ is not time-safe $K > 60$.

We have shown that the abstract model M is not time-robust since it has physical models M_φ, $K \in [51, 60]$, that are time-safe but not time-robust. However, the physical models M_φ for $K \leq 40$ are time-robust (see Figure 4.9).

α	$\varphi(\alpha)$
a	K
b	K
c	2K
i	0

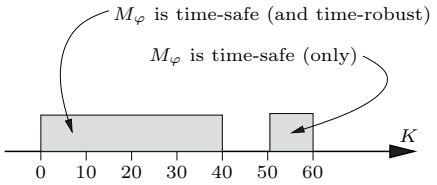


Figure 4.9: Time-safe physical models M_φ.

3.2 Enforcing Time-Robustness

Definition 19 (time-determinism) *An abstract model is* time-deterministic *if all its guards are eager (or delayable) equalities.*

Time-deterministic abstract models are such that if two execution sequences have the same corresponding sequences of actions, then they are identical. That is, time instants for the execution of the actions are the same. Time-deterministic abstract models are time-robust, as shown below.

Proposition 1 *Time-deterministic abstract models are time-robust.*

To prove that time-deterministic abstract models are time-robust we need the following lemma.

Lemma 1 *Given a time-deterministic abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ and a state (q, v) of M, the only waiting time allowed at (q, v) is the maximal waiting time $\mathsf{wait}(q, v)$, that is, for all $\delta \in [0, \mathsf{wait}(q, v)[$ no action is enabled at $(q, v + \delta)$.*

Proof 1 *of lemma Let (q, v) be a state of a time-deterministic abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$. Since M contains only guards that are eager (or delayable) equalities, transitions $q \xrightarrow{a_i, g_i, r_i} q_i$, $1 \leq i \leq n$, issued from q are such that the guard g_i is of the form $g_i \equiv [x_i = l_i]^{\mathsf{e}}$. We have:*

$$\begin{aligned} \bigvee_{1 \leq i \leq n} g_i(v + \delta) &\iff \bigvee_{1 \leq i \leq n} \mathsf{urg}[g_i](v + \delta) \\ &\iff \delta \in \Delta = \{\ \delta_i \geq 0 \mid 1 \leq i \leq n\ \}, \end{aligned}$$

where $\delta_i = l_i - v(x_i)$. By application of the definition of $\mathsf{wait}(q, v)$ (see Section 1.2) we have $\mathsf{wait}(q, v) = \boldsymbol{min}\ \Delta$, and for all $\delta \in [0, \mathsf{wait}(q, v)[$ and actions a_i are not enabled at $(q, v + \delta)$ since $\delta \notin \Delta$.

Notice Lemma 1 also holds for abstract models that contains only eager guards, that is, such that its actions are urgent as they are enabled.

Proof 2 *of proposition 1 Let $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ be a time-deterministic abstract model that is time-safe for an execution time function φ. Consider an execution time function φ' such that $\varphi' \leq \varphi$. We show by induction that each execution sequence of $M_{\varphi'}$ is also an execution sequence of M_φ. By induction hypothesis, we consider a state (q, v) of both $M_{\varphi'}$ and M_φ, and a transition $q \xrightarrow{a,g,r} q'$ executed at (q, v) in $M_{\varphi'}$, that is:*

$$M_{\varphi'} : (q, v) \xrightarrow{a, \varphi'(a)} (q', v' + \varphi'(a)) \xrightarrow{\delta'} (q', v' + \varphi'(a) + \delta').$$

where $v' = v[r \mapsto 0]$ and δ' is the waiting time for the execution of the next action in $M_{\varphi'}$. Since $g(v)$ is true, action a is also enabled in M_φ at (q, v):

$$M_{\varphi} : (q, v) \xrightarrow{a, \varphi(a)} (q', v' + \varphi(a)) \xrightarrow{\delta} (q', v' + \varphi(a) + \delta),$$

where δ is the waiting time for the execution of the next action in M_φ. As M_φ is time-safe and $\varphi'(a) \leq \varphi(a)$, we have $\varphi'(a) \leq \varphi(a) \leq \mathsf{wait}(q', v')$. Using properties of wait (see Section 1.2), we have $\mathsf{wait}(q', v' + \varphi(a)) = \mathsf{wait}(q', v') - \varphi(a)$ and $\mathsf{wait}(q', v' + \varphi'(a)) = \mathsf{wait}(q', v') - \varphi'(a)$. By application of Lemma 1 we obtain $\delta = \mathsf{wait}(q', v') - \varphi(a)$ and $\delta' = \mathsf{wait}(q', v') - \varphi'(a)$, that is, $\varphi(a) + \delta = \varphi'(a) + \delta'$. This demonstrates that the execution of a at state (q, v) leads to the same state $(q', v' + \varphi'(a) + \delta') = (q', v' + \varphi(a) + \delta)$ in M_φ and $M_{\varphi'}$ before executing the next action. By induction, execution sequences of $M_{\varphi'}$ are execution sequences of M_φ.

In [8, 51, 58] execution times of actions have fixed values called logical execution times (LET) specified in the program. LET define the difference between the release time and the due time of the actions. A program behaves as if its actions consume exactly their LET: even if they start after their release time and complete before their due time, their effect is visible exactly at these times. This is achieved by reading for each action its input exactly at its release time and its output exactly at its due time. A program based on LET defines a time-deterministic abstract model which is a timed automaton for which actions occur at fixed times. This ensures time-determinism: if two execution sequences execute the same sequence of actions, then corresponding actions occur at the same time instants. When execution times are less than LET, the abstract model and its corresponding physical model define exactly the same execution sequences, that is, the behavior of the program is independent of the platform.

Example 11 *Consider the time-deterministic abstract model M given in Figure 4.10 obtained from the abstract model of Example 9. Execution sequences of M are infinite repetitions of sequences of the following form:* $(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{50} (q_1, 50) \xrightarrow{c} (q_2, 50) \xrightarrow{70} (q_2, 120) \xrightarrow{i} (q_0, 0)$. *The physical models M_φ corresponding to M are time-safe if and only if $\varphi(a) \leq 50$, $\varphi(c) \leq 70$ and $\varphi(i) = 0$. Notice that for $51 \leq \varphi(a) \leq 60$, $\varphi(b) \leq 60$ and $\varphi(i) = 0$, M_φ remains deadlock-free but it is not time-safe.*

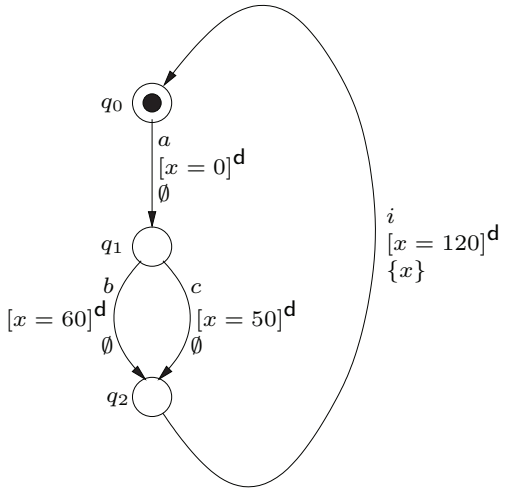


Figure 4.10: Time-deterministic abstract model M.

Definition 20 (action-determinism) *An abstract model is* action-deterministic *if there is at most one transition issued from each control location.*

If a time-deterministic abstract model is also action-deterministic, it has a single execution sequence from a given initial state (q_0, v_0), that is, it is totally deterministic. Such models have been considered in [8,51,58]. Their time-robustness allows checking time-safety only for worst-case execution times. In addition, for these systems checking time-safety boils down to checking deadlock-freedom, as shown below.

Proposition 2 *If M is an abstract model which is action-deterministic, deadlock-free and contains only delayable guards, then a physical model M_φ is time-safe if and only if it is deadlock-free.*

Proof 3 *of proposition Let $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ be a deadlock-free action-deterministic abstract model containing only delayable guards. We demonstrate that M_φ is time-safe if and only if M_φ is deadlock-free.*
*M_φ **is time-safe** $\Rightarrow$ M_φ **is deadlock-free.** If the physical model M_φ is time-safe, then its execution sequences are execution sequences of the deadlock-free abstract model M, that is, they are deadlock-free.*
*M_φ **is deadlock-free** $\Rightarrow$ M_φ **is time-safe.** We prove by contradiction that M_φ is time-safe if M_φ is deadlock-free. Assume that time-safety is violated for an action a at a state (q, v) of an execution sequence of M_φ, that is:*

$$(q, v) \xrightarrow{a,\varphi(a)} (q', v[r \mapsto 0] + \varphi(a))$$

such that a transition $q' \xrightarrow{a',g',r'} q''$ *satisfy* $\mathsf{urg}[g'](v[r \mapsto 0] + \delta)$, $\delta \in [0, \varphi(a)[$ *(i.e.* $\varphi(a) > \mathsf{wait}(q', v[r \mapsto 0])$*). Since* M *is action-deterministic,* $q' \xrightarrow{a',g',r'} q''$ *is the only transition issued from* q'*, and its guard* g' *is a delayable conjunction of intervals, that is,* g' *is of the form:*

$$g' \equiv \Big[\bigwedge_{1 \le i \le n} [l_i \le x_i \le u_i] \Big]^{\mathsf{d}}.$$

As a consequence, $\mathsf{urg}[g'](v[r \mapsto 0] + \delta) \Rightarrow \forall \delta' > \delta \;.\; \neg g'(v[r \mapsto 0] + \delta')$*, that is, no action can be executed from* $(q', v[r \mapsto 0] + \varphi(a))$*. This establishes that* M_φ *has a deadlock at state* $(q', v[r \mapsto 0] + \varphi(a))$*.*

Example 12 *We modify the time-deterministic abstract model given in Example 11 in order to make it also action-deterministic (see Figure 4.11). Its execution sequences remain the same, that is, infinite repetitions of sequences of the following form:* $(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{50} (q_1, 50) \xrightarrow{c} (q_2, 50) \xrightarrow{70} (q_2, 120) \xrightarrow{i} (q_0, 0)$*. The corresponding physical model* M_φ *is time-safe if and only if* $\varphi(a) \le 50$*,* $\varphi(c) \le 70$ *and* $\varphi(i) = 0$*, and deadlocks otherwise.*

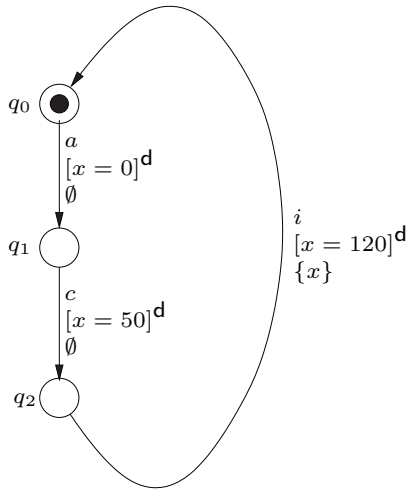


Figure 4.11: Deterministic abstract model M.

4 Conclusion

In this chapter we have presented essential notions for correct implementation of real-time applications. The method is new and innovates in several aspects:

- It does not suffer limitations of existing methods regarding the behavior of systems or the type of timing constraints. Considered real-time applications include not only periodic tasks with deadlines but also tasks with non-deterministic behavior and actions subject to interval timing constraints.
- It is based on a formally defined relation between application software written in high level languages with atomic and timeless actions and its execution on a given platform. The relation if formalized by using two models: 1) abstract models which describe the behavior of the application software as well as timing constraints on its actions; 2)

physical models which are abstract models equipped with an execution time function specifying WCET for the actions of the abstract model running on a given platform. Time-safety is the property of physical models guaranteeing that they respect timing constraints. Time-robust physical models have the property to remain time-safe for decreasing execution times of their actions. Non-robustness is a timing anomaly that appears in time non-deterministic systems.

The method generalizes existing techniques in particular those based on LET. These techniques consider fixed LET for actions, that is, time-deterministic abstract models. In addition, their models are action-deterministic, that is, only one action is enabled at a given state. For these models time-robustness boils down to deadlock-freedom for WCET as shown in Proposition 2.

To the best of our knowledge, the concept of time-robustness is new. It can be used to characterize timing anomalies due to time non-determinism. These timing anomalies have in principle different causes from timing anomalies observed for WCET.

Results on time-safety and time-robustness allow a deeper understanding of causes of anomalies. They advocate for time-determinism as a mean for achieving time-robustness.

In the next chapter, we propose a concrete implementation method using a Real-time Execution Engine which faithfully implements physical models. That is, if a physical model defined from an abstract model and a target platform is time-robust then the Engine coordinates the execution of the application software so as to meet the real-time constraints. The Real-time Execution Engine is correct-by-construction. It executes an algorithm which directly implements the operational semantics of the physical model.

Chapter 5

Correct Implementation of Real-Time Systems

In the previous chapter, we defined the concepts and gave definitions for the design of real-time systems. In this chapter, we give an implementation method that guarantees a correct implementation of those systems. A correct implementation is an implementation that satisfies the time-safety property, that is, the timing constraints requirements are met by the execution on a target platform. We also guarantee that if the model is robust for WCET then the implementation is time-safe. Otherwise, the method detects violations of time-safety and stops execution. We consider that the application software is a set of interacting components where each component is represented by an abstract model. Thus the abstract model corresponding to the application is the parallel composition of the timed automata representing the components. Given a physical model corresponding to the abstract model, the implementation method defines a real-time execution engine that takes into account their timing constraints.

We prove that the method is correct in two steps. We first define an execution engine for the abstract model and show that it correctly implements its semantics. Then we define a real-time execution engine and show that it correctly implements the semantics of the physical model. Our implementation method is based on a correct tracking of the physical time, as we have seen in the previous chapter, which allows the detection of timing constraints violations.

The chapter is structured as follows. In Section 1, we present the execution engine for abstract models. We first give a definition of the composition of abstract models, then, we give the execution algorithm that satisfies its semantics. In Section 2, we present the real-time execution engine for physical models. We also give a definition of the composition of physical models, then, we give the execution algorithm that detects time-safety violations. Finally, in Section 3, we present the implementation method for the component-based framework BIP and we present a use case to study time-safety and time-robustness for an adaptative MPEG video encoder.

1 Abstract Models Execution Engine

An abstract model is the representation of an application software based on timed automata. The model takes into account only platform-independent timing constraints expressing user-dependent requirements. The actions of the model represent statements of the application software and are assumed to be timeless. We consider that the application software is a set of interacting components where each component is represented by an abstract model. We explain how the abstract model corresponding to the application is the parallel composition of the timed automata representing the components. We also present the execution engine for the resulting abstract model.

1.1 Composition of Abstract Models

The composition of a set of behaviors gives a restricted behavior, contained in the product of their behavior. The composition $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow_\gamma)$ of abstract models M^i, $1 \leq i \leq n$, corresponds to a general notion of product for the timed automata M^i. Here is a formal definition for the composition of abstract models.

Let $M^i = (\mathsf{A}_i, \mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i)$, $1 \leq i \leq n$, be a set of abstract models with disjoint sets of actions and clocks, that is, for all $i \neq j$ we have $\mathsf{A}_i \cap \mathsf{A}_j = \emptyset$ and $\mathsf{X}_i \cap \mathsf{X}_j = \emptyset$.

Definition 21 (composition of abstract models) *A set of* interactions γ *is a subset of* 2^{A}*, where* $\mathsf{A} = \bigcup_{i=1}^{n} \mathsf{A}_i$*, such that any interaction* $a \in \gamma$ *contains at most one action of each component* M^i*, that is,* $a = \{\ a_i \mid i \in I\ \}$ *where* $a_i \in \mathsf{A}_i$ *and* $I \subseteq \{\ 1, 2, \ldots, n\ \}$*. We define the* composition *of the abstract models* M^i *as the abstract model* $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow_\gamma)$ *over the set of actions* γ *as follows:*

- $\mathsf{Q} = \mathsf{Q}_1 \times \mathsf{Q}_2 \times \ldots \times \mathsf{Q}_n$
- $\mathsf{X} = \mathsf{X}_1 \cup \mathsf{X}_2 \cup \ldots \cup \mathsf{X}_n$
- *For* $a = \{\ a_i \mid i \in I\ \} \in \gamma$ *we have* $(q_1, q_2, \ldots, q_n) \xrightarrow{a,g,r}_\gamma (q'_1, q'_2, \ldots, q'_n)$ *in* M *if and only if* $g = \bigwedge_{i \in I} g_i$*,* $r = \bigcup_{i \in I} r_i$*,* $q_i \xrightarrow{a_i, g_i, r_i} q'_i$ *in* M^i *for all* $i \in I$*, and* $q'_i = q_i$ *for all* $i \notin I$*.*

Example 13 *Figure 6.10 shows two abstract models modeling the behavior of a video encoder and its controller. The abstract model E is the behavior of the encoder encapsulated in component Encoder and the abstract model C is the behavior of the controller encapsulated in component Controller.*

The abstract model $E = (\mathsf{A}, \{q_0, q_1, q_2\}, \{x\}, \longrightarrow)$ is composed of a set of actions $\mathsf{A} = \{get, enc, next\}$, action *get* for receiving a frame, *enc* for encoding it and *end* to indicate the end of encoding, a single clock x and the following set of transitions:

$$\begin{aligned} \longrightarrow \ = \{\ & (q_0, get, [0 \leq x \leq +\infty]^{\mathsf{e}}, \emptyset, q_1), \\ & (q_1, enc, [0 \leq x \leq +\infty]^{\mathsf{d}}, \emptyset, q_2), \\ & (q_2, next, [100 \leq x \leq 120]^{\mathsf{d}}, \{x\}, q_0) \quad \}. \end{aligned}$$

The abstract model $C = (\mathsf{A}, \{q'_0, q'_1\}, \{y\}, \longrightarrow)$ is composed of a set of actions $\mathsf{A} = \{enc_a, enc_b, next\}$, action enc_a to set the encoding quality a, action enc_b to set the encoding quality b and action *next* to move to the next frame encoding, a single clock x and the following set of transitions:

$$\begin{aligned}\longrightarrow \quad = \{ \quad & (q_0', enc_a, [51 \leq y \leq 60]^{\mathsf{d}}, \emptyset, q_1'), \\ & (q_0', enc_b, [0 \leq y \leq 50]^{\mathsf{l}}, \emptyset, q_1'), \\ & (q_1', next, [0 \leq y \leq +\infty]^{\mathsf{d}}, \{y\}, q_0') \quad \}.\end{aligned}$$

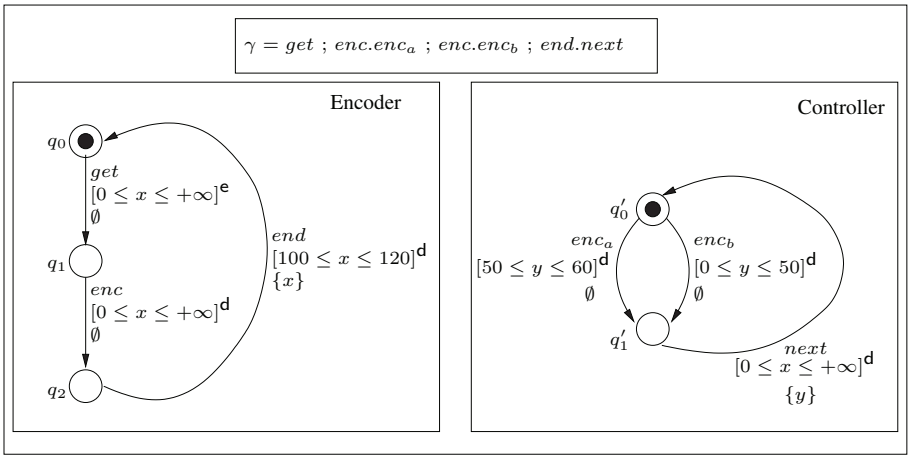


Figure 5.1: Interacting abstract models of an encoder (left) and its controller (right).

The two models interact following this interaction model:
$\gamma = \{get\ ;\ enc.enc_a\ ;\ enc.enc_b\ ;\ end.next\}$.

Interactions $enc_a^{'} = enc.enc_a$ and $enc_b^{'} = enc.enc_b$ synchronize the action enc of the encoder with the action enc_a and enc_b of the controller. Indeed, the encoder can either encode the frame with a quality a or b depending on the duration of action get. Interaction $next^{'} = end.next$ synchoronizes the end of encoding of a frame and the enabless to move to the next frame through actions end and $next$.

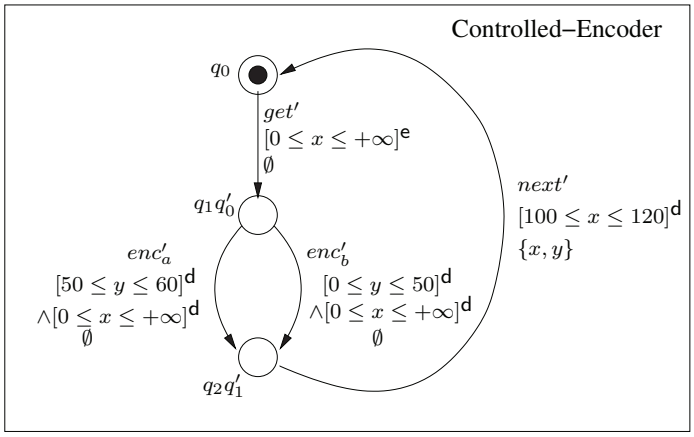


Figure 5.2: Abstract model composition of the encoder and its controller.

The composition of the two models is then $EC = (\mathsf{A}, \{q_0, q_1, q_2\}, \{x\}, \longrightarrow)$ (see Figure 6.11) , composed of a set of actions $\mathsf{A} = \{get, enc_a^{'}, enc_b^{'}, next^{'}\}$, a single clock x and the following set of transitions:

$$
\begin{aligned}
\longrightarrow \quad = \{ \quad & (q_0, get, [0 \leq x \leq +\infty]^{\mathsf{e}}, \emptyset, q_2), \\
& (q_1, enc_a^{'}, [51 \leq x \leq 60]^{\mathsf{d}}, \emptyset, q_2), \\
& (q_1, enc_b^{'}, [0 \leq x \leq 50]^{\mathsf{d}}, \emptyset, q_2), \\
& (q_2, next^{'}, [100 \leq x \leq 120]^{\mathsf{d}}, \{x\}, q_1) \quad \}.
\end{aligned}
$$

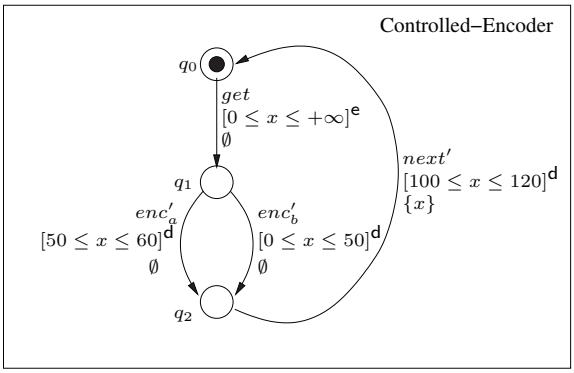


Figure 5.3: Simplification of the abstract model composition of the encoder and its controller.

We notice that the composition of the two models described in the example correspond to the abstract model presented in the previous chapter in example 9.

1.2 Execution Algorithm of Abstract Models

We consider that the application software is a set of interacting components where each component is represented by an abstract model. We have presented the methodology for building the composition of such components. We now define an execution Engine that computes, at run-time, sequences of interactions between components $M^i = (\mathsf{A}_i, \mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i)$, by applying the above operational semantics rule.

Figure 5.4 is a representation of the abstract models execution Engine. For given states (q_i, v_i) of the components M^i with their corresponding lists of transitions $\{ q_i \stackrel{a_j, g_j, r_j}{\longrightarrow} q'_j \}$ issued from q_i and an interaction model γ, the execution engine computes the set of enabled interactions γ_s and stops the execution if the model is inconsistent, that is, when a deadlock is detected. If the model is consistent, the execution Engine chooses one (enabled) interaction using a real-time scheduling policy and executes it. We will first explain how we manage to handle timing features involving components local clocks and timing constraints, then we give the execution algorithm.

Timing Constraints Translation

Every component representing an abstract model can define local clocks. They can be reset at any time and are involved in timing constraints labelling transitions. According to Definition 21, the timing constraint g of an interaction $a = \{a_i, i \in I\}$ of a composition of abstract models M^i is the conjunction of the real-time guards g_i, labelling transitions of the ports a_i involved in the interaction. Since the components clocks are local, in order to compute interactions between components and schedule them correctly, we need to express real-time constraints of interactions in terms of a single scale time, that is, a single global

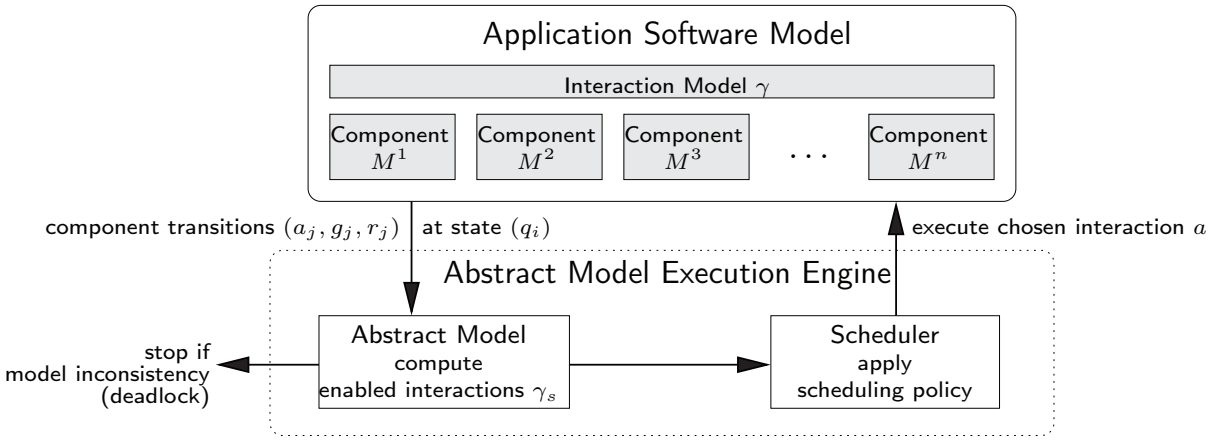


Figure 5.4: Abstract Models Execution Engine.

clock. We have chosen to express all timing constraints according to the global clock t, measuring the absolute time elapsed, i.e. global clock t is never reset.

We first express the timing constraints involving local clocks of components in terms of the single global clock t. Here are the different steps the execution Engine makes to do the timing translation:

1. It first stores the absolute time $w(x)$ of the last reset of each clock x with respect to the clock t. Indeed, since the global clock t is never reset, by storing the time on wich the reset has been done, we are able to compute the time elapsed since the last reset. It represents clock x value in terms of global clock t. For this, we use a valuation $w : \mathsf{X} \to \mathbb{T}$. The valuation v of the clocks X can be computed from the current value of global time t and the last reset value w by using the equality $v = t - w$.

2. Thus, the Execution Engine considers states of the form $s = (q, w, t)$ where $q = (q_1, q_2, \ldots, q_n) \in \mathsf{Q}$ is a control location of M, $w : \mathsf{X} \to \mathbb{T}$ is valuation for clocks representing their reset times, and $t \in \mathbb{T}$ is the value of the current (absolute) time.

3. We rewrite each atomic expression $l \leq x \leq u$ involved in a timing constraint, with a local clock x, by using the global clock t and reset times w. In that purpose, we have to add to the initial lower and upper bounds the last resest value $w(x)$ of x as follows:

$$l \leq x \leq u \;\equiv\; l + w(x) \leq t \leq u + w(x).$$

This allows reducing the conjunction of guards from synchronizing components into a sinple timing constraint such as its lower and upper bounds are respectively the maximal value of the components upper and lower bounds and its urgency is also the maximal urgency of the components. The computation of this conjunction is as follows:

$$\bigwedge_j [l_j \leq t \leq u_j]^{\tau_j} \;=\; \Big[(\mathbf{max}_j\ l_j) \leq t \leq (\mathbf{min}_j\ u_j)\Big]^{\mathbf{max}\ \tau_j}.$$

4. Thus, the guard g associated to an interaction a at a given state $s = (q, w, t)$ can be put in the form $g = [l \leq t \leq u]^{\tau}$.

We associate to an interaction a, such that its timing constraint $g = [l \leq t \leq u]^{\tau}$ statisfies $l \leq u$, its next activation time $\mathsf{next}_s(a)$. It corresponds to the next value of the global time (i.e. maximal value) for which the interaction a is enabled.

We also associate to an interaction a its next deadline $\mathsf{deadline}_s(a)$. It corresponds to the next value of the global time for which interaction a is urgent, so it also depends on the urgency of the timing constraint. If the urgency is delayable and global time t is smaller than the upper bound u then the deadline is u. If the urgency is eager, the deadline is the lower bound l if $t < l$ and the deadline is t if $t \in [l, u]$.

Values $\mathsf{next}_s(a)$ and $\mathsf{deadline}_s(a)$ from guard $g = [l \leq t \leq u]^{\tau}$ can be resumed as follows:

$$\mathsf{next}_s(a) \quad = \quad \begin{cases} \mathbf{max}\ \{\ t, l\ \} \text{ if } t \leq u \\ +\infty \text{ otherwise,} \end{cases}$$

$$\mathsf{deadline}_s(a) \quad = \quad \begin{cases} u \text{ if } t \leq u \ \wedge\ \tau = \mathsf{d} \\ l \text{ if } t < l \ \wedge\ \tau = \mathsf{e} \\ t \text{ if } t \in [l, u] \ \wedge\ \tau = \mathsf{e} \\ +\infty \text{ otherwise.} \end{cases}$$

Notice that we have $\mathsf{next}_s(a) \leq \mathsf{deadline}_s(a)$.

Execution Algorithm

Given a state $s = (q, w, t)$, $q = (q_1, \ldots, q_n)$, the abstract model execution Engine computes the next interaction to be executed as follows.

1. It first computes the set of *enabled* interactions $\gamma_s \subseteq \gamma$ at state $s = (q, w, t)$, from given sets of transitions issued from q_i for each component M^i. According to Definition 21, an interaction $a = \{\ a_i \mid i \in I\ \} \in \gamma$ is *enabled* from state s if $(q_1, \ldots, q_n) \xrightarrow{a,g,r}_{\gamma} (q'_1, \ldots, q'_n)$ and $g = [l \leq t \leq u]^{\tau}$ statisfy $l \leq u$ and $t \leq u$, i.e. $next(a) \leq +\infty$. The timing constraint g of a is the conjunction of the guards g_i of actions a_i and r is the union of the resets r_i of actions a_i, that is, $g = \bigwedge_{i \in I} g_i$, $r = \bigcup_{i \in I} r_i$, for all $i \in I$ we have $q_i \xrightarrow{a_i, g_i, r_i} q'_i$ in M^i and for all $i \notin I$ we have $q'_i = q_i$.
2. It chooses an interaction $a = \{\ a_i \mid i \in I\ \} \in \gamma_s$ enabled from state $s = (q, w, t)$, that is, such that there exists a time instant $t' \geq t$ at which the guard a holds (i.e. $\mathsf{next}_s(a) < +\infty$), and no timing constraint is violated (i.e. $\mathsf{next}_s(a) \leq D = \mathbf{min}_{a \in \gamma_s}\ \mathsf{deadline}_s(a)$). The choice of a depends on the considered real-time scheduling policy. For instance, EDF (Earliest Deadline First) scheduling policy can be used, that is, the chosen interaction a satisfies $\mathsf{deadline}_s(a) = D$.
3. It executes a with minimal waiting time, that is, at time instant $\mathsf{next}_s(a)$. The execution of a corresponds to the execution of all actions a_i, $i \in I$, followed by the computation of a new valuation w and the update of control locations.

Algorithm 4 gives an implementation of the Execution Engine for the composition of abstract models. It basically consists of an infinite loop that first computes enabled interactions at current state s of the composition (line 3). It stops if no interaction is possible from s (i.e. deadlock) at line 5. Otherwise, it chooses an interaction a (line 7) and executes a (line 12) with minimal waiting time (line 9) by updating the logical time t. Finally, the state s is updated in order to take into account the execution of a (lines 13 and 14).

Algorithm 4 Abstract Model Execution Engine

Require: abstract models $M^i = (\mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i)$, $1 \leq i \leq n$, initial control location $(q_0^1, \ldots, q_0^n)$, set of interactions γ

1: $s = (q_1, \ldots, q_n, w, t) \leftarrow (q_0^1, \ldots, q_0^n, 0, 0)$ *// init.*
2: **loop**
3: $\gamma_s = EnabledInteractions(s)$
4:
5: **if** $\exists a \in \gamma_s$. $\mathsf{next}_s(a) < +\infty$ **then**
6: $D \leftarrow \mathbf{min}_{a \in \gamma_s}\ \mathsf{deadline}_s(a)$ *// next deadline*
7: $a = \{\ a_i \mid i \in I\ \} \leftarrow RealTimeScheduler(\gamma_s)$
8:
9: $t \leftarrow \mathsf{next}_s(a)$ *// consider minimal waiting time*
10:
11: **for all** $i \in I$ **do**
12: Execute(a_i) *// execute involved component*
13: $w \leftarrow w[r_i \mapsto t]$ *// reset clocks*
14: $q_i \leftarrow q_i'$ *// update control location*
15: **end for**
16: **else**
17: **exit**(DEADLOCK)
18: **end if**
19: **end loop**

2 Physical Models Execution Engine

A physical model is a representation of an application software running on a given platform. It is obtained by assigning execution times to the transitions of an abstract model. We consider that the application software is a set of interacting components where each component is represented by an abstract model and φ is the execution time function that gives for each action labelling the transitions of the abstract model its execution time $\varphi(a)$. We will first explain how we build the composition of physical models corresponding to the application. We then present the real-time execution engine that ensures the execution of the application on a target platform. With a correct traking of time, it stops execution when time-safety violations occur.

2.1 Composition of Physical Models

We consider abstract models M^i, $1 \leq i \leq n$, and corresponding physical models $M^i_{\varphi_i} = (\mathsf{A}_i, \mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i, \varphi_i)$, with disjoint sets of actions and clocks.

Definition 22 (composition of physical models) *Given a set of interactions γ, and an associative and commutative operator $\oplus : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$, the* composition *of physical models $M^i_{\varphi_i}$ is the physical model M_φ corresponding to the abstract model M which is the composition of M^i, $1 \leq i \leq n$, with the execution time function $\varphi : \gamma \to \mathbb{T}$ such that $\varphi(a) = \bigoplus_{i \in I} \varphi_i(a_i)$ for interactions $a = \{\ a_i \mid i \in I\ \} \in \gamma$, $a_i \in \mathsf{A}_i$.*

The definition is parameterized by an operator $\oplus$ used to compute the execution time $\varphi(a)$ of an interaction a from execution times $\varphi(a_i)$ of the actions a_i involved in a. The

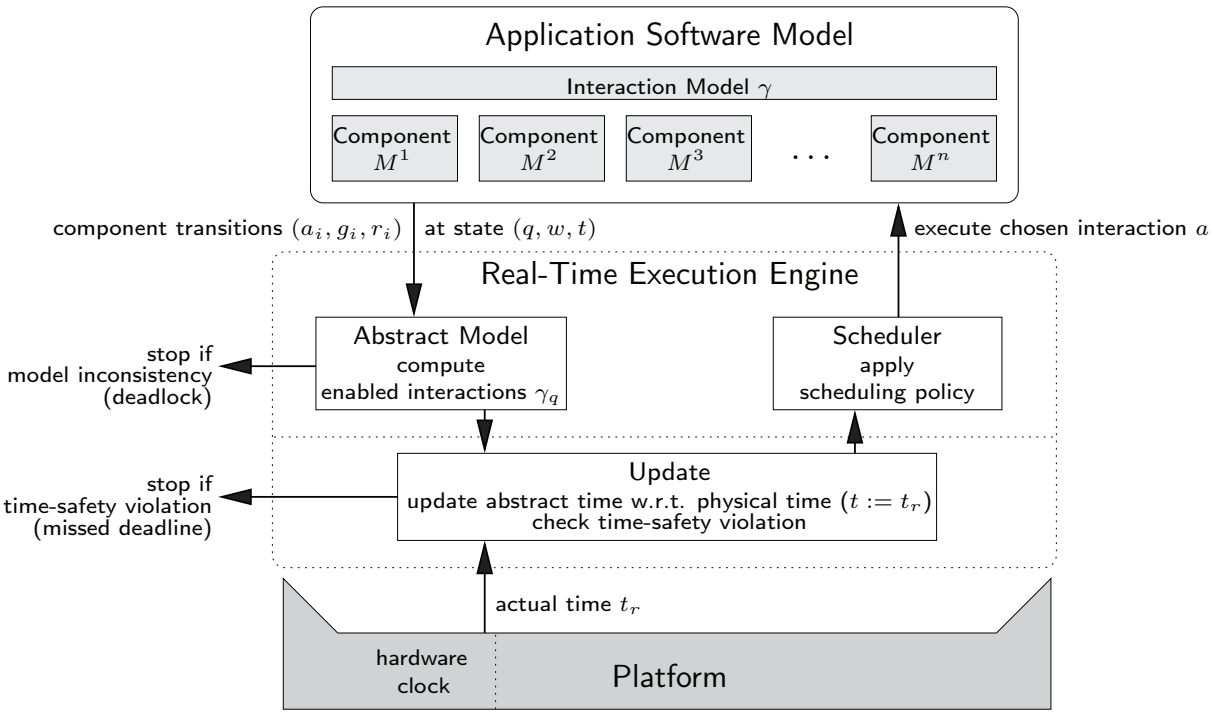


Figure 5.5: Real-time Execution Engine.

choice of this operator depends on the considered execution platform and in particular how components (abstract models) are parallelized. For instance, for a single processor platform (i.e. sequential execution of actions), $\oplus$ is addition. If all components can be executed in parallel, $\oplus$ is **max**.

As a rule, it is difficult to obtain execution times for the actions (i.e. block of code) of an application software. Execution times vary a lot from an execution to another, depending on the contents of the input data, the dynamic state of the hardware platform (pipeline, caches, etc.). There exists techniques for computing upper bounds of the execution time of a block of code, that is, estimates of the worst-case execution times [88]. Given abstract models M^i, and functions φ_i specifying WCET for the actions of M^i, the abstract composition M can be safely implemented if the physical composition M_φ (defined above) is time-robust.

2.2 Execution Algorithm of Physical Models

We define a real-time execution Engine that does not need an a priori knowledge of execution time functions φ_i. It ensures the real-time execution of a component-based application on the target platform, and stops if the implementation is not time-safe, that is, a deadline is missed during the execution (see Figure 5.5). The real-time execution Engine differs from the abstract model execution engine (see Figure 5.4) on the update section that updates the model time t upon the physical time t_r of the platfom's clock. It actually adds to the abstract time t the execution time of the previous interaction, which is the composition of execution times of the actions involved in the interaction and is computed using the operator $\oplus$ (see Definition 30). The implementation method preserves the abstract model semantics and ensures a correct correspondance between the abstract time and the physical time.

Given a state $s = (q, w, t)$, $q = (q_1, \ldots, q_n)$, the real-time execution Engine computes the next interaction to be executed as follows.

1. It first computes the set of *enabled* interactions $\gamma_s \subseteq \gamma$ at state $s = (q, w, t)$, from given sets of transitions issued from q_i for each component M^i (similary to the abstract model execution Engine).
2. It updates the abstract time of the Engine with the physical time of the platform. Before the update, the abstract time t represents the instant of execution of the previous interaction a'. We update the abstract time by adding the execution time of the interaction a'. If timing constraints are not violated, that is no deadline is missed (i.e. $\mathsf{next}_s(a) \leq D = \mathbf{min}_{a \in \gamma_s}\ \mathsf{deadline}_s(a)$), it exists an execution sequence in the abstract model that corresponds to one of the physical model.
3. It chooses an interaction $a = \{\ a_i \mid i \in I\ \} \in \gamma_s$ enabled at state $s = (q, w, t)$, such that there exists a time instant $t' \geq t$ at which the guard a holds (i.e. $\mathsf{next}_s(a) < +\infty$) and such that no other enabled interaction $a' \in \gamma_s$ is more urgent (i.e. $\mathsf{next}_s(a) < +\infty$), and no timing constraint is violated (i.e. $\mathsf{next}_s(a) \leq \mathbf{min}_{a' \in \gamma_s}\ \mathsf{deadline}_s(a')$). When more than one enabled interaction is possible, the choice of a depends on the considered real-time scheduling policy.
4. It executes a as soon as a is enabled, that is, at time instant $\mathsf{next}_s(a)$. The execution of $a = \{a_i, i \in I\}$ corresponds to the execution of all actions a_i, $i \in I$, followed by the computation of a new valuation w that happens at exactly abstract time t even if the reset is at the actual time $t_r = t + \epsilon$, wich avoids the accumulation of drifts. Finally, control locations are also updated.

Algorithm 6 gives an implementation of the Real-Time Execution Engine for a single processor platform. It differs from Algorithm 4 at lines 7, 8 and 13. It updates the current value of abstract time t with respect to the current value of physical time t_r (line 7) in order to take into account execution time of interactions for the considered execution platform. It stops if time-safety is violated, that is, if t is greater than the next deadline D (line 8). It also waits for the physical time to reach the next activation time ($\mathsf{next}_s(a)$) of the chosen interactions a (line 13).

3 Real-Time BIP Component based Framework

We implemented the proposed method for the component-based framework BIP [14]. BIP (Behavior Interaction Priority) is a framework for building systems consisting of heterogeneous components. We extended the initial BIP framework presented in Chapter 2, so as to handle real-time features in a rigorous manner. The implementation consists of two main extensions, the extension of the BIP language to allow the expression of real-time systems and the extension of the single-threaded Engine to achieve real-time execution. The real-time Engine performs the computation of schedules meeting the timing constraints of the application, depending on the physical time provided by the real-time clock of the platform, as explained in the previous section. Finally, we studied time-safety and time-robustness for a multimedia application—an adaptive MPEG video encoder modeled in BIP. We show that the application is not time-robust and we also explain how its time-robustness can be enforced using two different methods.

Algorithm 5 Real-Time Execution Engine

Require: abstract models $M^i = (\mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i)$, $1 \leq i \leq n$, initial control location $(q_0^1, \ldots, q_0^n)$, interactions γ

1: $s = (q_1, \ldots, q_n, w, t) \leftarrow (q_0^1, \ldots, q_0^n, 0, 0)$ *// init.*
2: **loop**
3: $\gamma_s = EnabledInteractions(s)$
4:
5: **if** $\exists a \in \gamma_s \,.\, \mathsf{next}_s(a) < +\infty$ **then**
6: $D \leftarrow \mathbf{min}_{a \in \gamma_s}\, \mathsf{deadline}_s(a)$ *// next deadline*
7: $t \leftarrow t_r$ *// update Engine clock w.r.t. actual time*
8: **if** $t \leq D$ **then**
9: **if** $\exists a \in \gamma_s \,.\, \mathsf{next}_s(a) < +\infty$ **then**
10: $a = \{\ a_i \mid i \in I\ \} \leftarrow RealTimeScheduler(\gamma_s)$
11:
12: $t \leftarrow \mathsf{next}_s(a)$ *// update Engine clock*
13: **wait** $t_r \geq t$ *// real-time wait*
14:
15: **for all** $i \in I$ **do**
16: Execute(a_i) *// execute involved component*
17: $w \leftarrow w[r_i \mapsto t]$ *// reset clocks*
18: $q_i \leftarrow q_i'$ *// update control location*
19: **end for**
20: **else**
21: **exit**(DEADLOCK)
22: **end if**
23: **else**
24: **exit**(DEADLINE_MISS)
25: **end if**
26: **else**
27: **exit**(DEADLOCK)
28: **end if**
29: **end loop**

3.1 Real-Time Extensions in the BIP framework

In BIP, a component describes a behavior that is given by an automaton whose transitions are labelled by ports and can execute C++ code (i.e. local data transformations). Connectors between communication ports of components define a set of enabled interactions which are synchronizations between components. The execution of interactions may involve transfer of data between the synchronizing components, Priorities is a mechanism for conflict resolution that allows direct expression of scheduling policies between interactions. Components, connectors and priorities are used for building hierarchically new components, namely *compound* components. BIP models can be compiled to C++ code and the generated code is intended to be executed by the dedicated Engine implementing the semantics of BIP. We extended the BIP language and compiler to provide the new Real-Time Engine the timing features.

Atomic Components

An atomic component has only local data, and its interface is given by a set of communication ports. Without using the proposed real-time extensions of BIP, time can be handled by synchronizing all components in order to update local integer variables representing clocks with a global Tick connector. Thus, we introduce in atomic components the notion of real-time clock. A real-time clock is used to measure the actual advance of the physical time and can be reset when executing a transition. We can express timing constraints over the values of the clocks. They are used for expressing and enforcing real-time properties in the model.

We declare real-time clocks in atomic components, like local data variables. In the previous version of BIP, a transition in a component behavior can be guarded by a boolean condition depending on the local variables. We extend the property by considering that transitions can also be guarded by timing constraints. An atomic component is now defined as follows:

Definition 23 (Atomic component) *An* atomic component *represents behavior* B *as a transition system, extended with variables and functions, represented by* $(\mathsf{V}, \mathsf{P}, \mathsf{X}, \mathsf{Q}, \longrightarrow)$*, where:*

- Q *is a set of* control states $\mathsf{Q} = \{Q_1 ... Q_n\}$*, Control States denote places at wich the components await for synchronization.*
- P *is a set of* communication ports $P = \{p_1 ... p_n\}$*,*
- V *is a set of* variables *used to store (local) data. Variables may be associated to ports.*
- X *is a finite set of* clocks*,*
- $\longrightarrow$ *is a set of transitions modeling computation steps of components. Each transition is a tuple of the form* $(q_1, p, g_p, gt_p, f_p, q_2)$*, representing a step from control state* q_1 *to control state* q_2*, denoted as* $q_1 \overset{p,g_p,g_tp,f_p}{\longrightarrow} q_2$*, where* g_p *is a boolean condition on* V*,* f_p *is a computation step consisting of data transformations and* gt_p *is a timing constraint over* X*.*

A timing constraint gt is a real-time constraint defined by $gt = (\tau, c, it)$, where

- τ is an urgency type, that is $\tau \in \{lazy, delayable, eager\}$,
- c is a name of a real time clock which is declared in the considered component, and

- *it* is defined as an interval $I = [l, u]$, representing a subset of integer values of c for which the transition is enabled by the real-time constraint.

An abstract syntax of clocks declaration is given in Figure 5.6. We introduce the keyword **clock** to create instances of clocks. Clocks are then referenced by a name and a time unit given by the user. The time unit corresponds to seconds or milliseconds or microseconds, otherwise the unit is by default the millisecond.

clock-definition ::=
clock *clock-name* { **,** *clock-name* }* [*time-unit*]

time-unit ::= **second** | **millisecond** | **microsecond**

Figure 5.6: *Clock declaration syntax in BIP*

In the following (See Figure 5.7), we consider that a timing constraint declared in a component are of the form $(\tau, c, [l, u])$, where l is an integer, and u is an integer or ∞ such that $l \leq u$. Finally, an abstract syntax of an atomic component including the proposed real-time extensions is given in Figure 5.8.

timed-guard ::=
timed-constraint { **,** *timed-constraint* }*

timed-constraint ::=
urgency clock-name **in** **(** *integer* **,** *x-integer* **)**
urgency ::= **lazy** / **delayable** / **eager**
x-integer ::= *interger* / **infinity**

Figure 5.7: *Timed guard declaration syntax in BIP*

Figure 5.9 gives the abstract model representation (left) and declaration (right) of the encoder example using the new BIP language.

Connectors

Connectors allow the composition of atomic components that interact by meeting constraints of an interaction model. Connectors are used to specify interactions between ports of components. Since we introduced timing constraints in atomic components, connectors should take them into account. Clock synchronizations between components are not made explicitly by the user, but are left to the real-time BIP execution Engine. Indeed, in order to compute the interactions at a given state, the real-time execution Engine synthesizes dynamically the timing constraints associated to interactions These are computed by combining real-time constraints associated to components ports involved in an interaction. As explained in Section 1, timing constraints associated to ports are expressed using the same

```
transition-definition ::=
on port-name
from place-name to place-name
provided untimed-guard when timed-guard do action

atomic-type-definition ::=
atomic type atomic-type-name
[ ( c-type-name fpar-name { , c-type-name fpar-name }* ) ]
{ data-definition }*
{ port-definition }*
{ clock-definition }*
{ place-definition }*
{ transition-definition }*
{ export port-type-name port-name = port-reference }*
end
```

Figure 5.8: *An atomic component syntax in BIP*

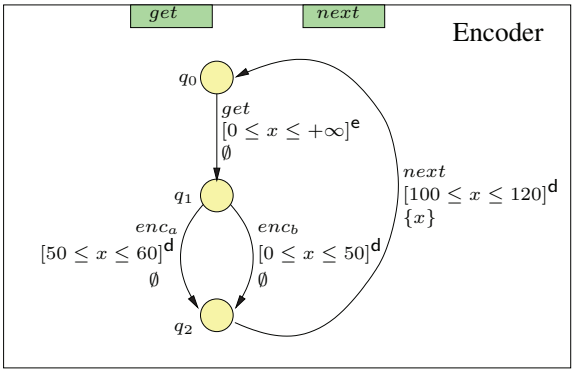
global clock t. The result of the conjunction of these timing constraints from synchronizing components is a simple timing constraint belonging to the interaction. We can now assume that interactions are guarded by both boolean functions and timing constraints. Timing constraints are computed at run-time by the real-time BIP execution Engine. Thus, they are not expressed by the BIP language.

Let $\{B_i\}_{1\leq i\leq n}$ be a set of components and $a = \{p_j\}_{j\in J}$, $J \subseteq \{1,..,n\}$ be an interaction defined in a connector γ, such that for all $j \in J$, p_j is a port of B_j. Given a state of the compound system, we denote by TG_a the timed guard corresponding to an interaction a defined in the connector γ and $g_t(p_i)$ the timed guard labelling each transition issued from B_i and involving port p_i in the interaction a. We note that we consider translated timed guards $g_t(p_i)$ in terms of the single global clock t.

We give a formal definition of a connector as follows.

Definition 24 (Connector) *A* connector γ *defines sets of ports of atomic components* B_i *which can be involved in an interaction. It is formalized by* $\gamma = (P_\gamma, A_\gamma, p[x])$ *where:*

- P_γ *is the support set of* γ*, that is the set of ports that* γ *may synchronize.*
- $A_\gamma \subseteq 2^{P_\gamma}$ *is a set of interactions each labelled by the triple* (P_a, G_a, TG_a, F_a) *where:*
  - P_a *is the set of ports* p_i*,* $i \in I$ *and* $I \subseteq [1, n]$*, that take part of an interaction* a*,*
  - G_a *is the guard of* a*, a predicate defined on variables* $\bigcup_{p_i\in a} V_{p_i}$*,*
  - TG_a *is the timing constraint of* a *over* t*, which corresponds to the conjuction of the timing constraints of ports* p_i*,* $i \in I$ *involved in* a*:* $TG_a = \bigwedge_{p_i\in a} gt(p_i)$*,*
  - F_a *is the data transfer function of* a*, defined defined on variables* $\bigcup_{p_i\in a} V_{p_i}$*.*
- p *is the exported port of the connector* γ*. We also associate to this port a timed guard* g_t *corresponding to the union of the timed guards of the interactions of connector* γ*, that is* $g_t = \bigcup_{a\in A_\gamma} TG_a$




```
atomic type Encoder
    export port intPort get
    export port intPort intPort next
    port intPort enc_a compute
    port intPort enc_b

    clock x unit millisecond

    place q0
    place q1
    place q2

    initial to q0

    on get from q0 to q1
    when x in [0,-] eager
    on enc_a from q1 to q2
    when x in [50,60] delayable
    on enc_b from q1 to q2
    when x in [0,50] delayable
    on next from q2 to q0
    when x in [100,120] delayable
    reset x
end
```

Figure 5.9: The encoder component declaration in BIP.

Priorities

Priorities are used for inhibiting an interaction, called the low interaction, whenever another interaction, called the high interaction, is enabled. Priorities can be guarded by boolean conditions, which depend on the value of variables. We can extend priorities with the notion of time, by adding delays for the application of priority rules. A priority with a delay of d means that its lower interaction is inhibited by its high interaction whenever the latter is possible in d units of time.

Definition 25 (Priority Rule) *A priority is a tuple $(C, \prec_d)$, where C is a state predicate (boolean condition) characterizing the states where the priority applies and $\prec_d$ is a partial order that gives the priority order on a set of interactions $A = \bigcup A_\gamma$ and d is the delay of application of the priority.*

For $a_1 \in A$ and $a_2 \in A$, a priority rule is textually expressed as $C \rightarrow a_1 \prec_d a_2$. When the state predicate C is true and both interactions a_1 and a_2 specified in the priority rule are enabled, the higher priority interaction, i.e., a_2 is selected for execution with a delay of d time units.

3.2 Experimental Results: Adaptive Video Encoder

We consider an adaptive MPEG video encoder componentized in BIP [35] (15000 lines of code) and running on a STm8010 board from STMicroelectronics. It takes streams of frames of 320×144 pixels as an input, and computes the corresponding encoded frames (see Figure 5.10). Since input frames are produced by a camera at a rate of 10 frames/s (i.e. every 100 ms), encoding each frame must be done within $D = 100$ ms.

Description of the application

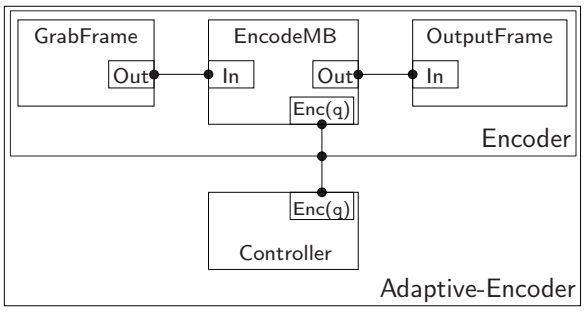


Figure 5.10: Adaptive video encoder architecture.

The adaptive MPEG video encoder consists of two main components.

Encoder corresponds to the functional part of the video encoder, that is, it involves no time constraint. Input frames are treated by GrabFrame. Each frame is split into $N = 180$ macroblocks of 16×16 pixels which are individually encoded by EncodeMB for given quality levels $q_i \in Q = \{ 0, 1, \ldots, 8 \}$. The higher the quality levels are, the better the video quality is. A bitstream corresponding to the encoded frames is produced by OutputFrame.

Controller is a controller for Encoder. It chooses quality levels q_i for encoding macroblocks so as not to exceed the time budget of $D = 100$ ms for encoding a frame. To keep low the overhead due to the computation of Controller, quality levels are only computed every 20 macroblocks, that is, there are 9 control points in a frame.

Components Encoder and Controller interact as follows. At each control point $i \in \{ 0, \ldots, 8 \}$ Controller triggers Encoder for encoding the next 20 macroblocks at a quality level q_i. The computation of q_i is based on the time t elapsed since the beginning of the encoding of the current frame, and estimates of the average execution times C_q for encoding 20 macroblocks at quality level q. Execution times have been obtained by profiling techniques using different input streams of frames (see Table 5.1). C_q is increasing with quality level q. A quality level q is enabled at control point i only if $t+(9-i)C_q \leq D$, where $(9-i)C_q$ is an estimate of the average execution time for encoding the remaining macroblocks of the current frame. This condition is equivalent to the guard $g_q(i) \equiv [t \leq D - (9-i)C_q]^{\mathsf{d}}$. In order to maximize video quality, we give higher priority for higher quality levels, that is, for all $q \in \{0, \ldots, 7\}$ we have Enc$(q+1)$ > Enc(q) (see Figure 5.11). The chosen quality level q_i is transmitted by Controller to Encoder through the port Enc. After encoding the last 20 macroblocks (i.e. $i = 9$), Controller waits for the next frame, that is, for $t = D$.

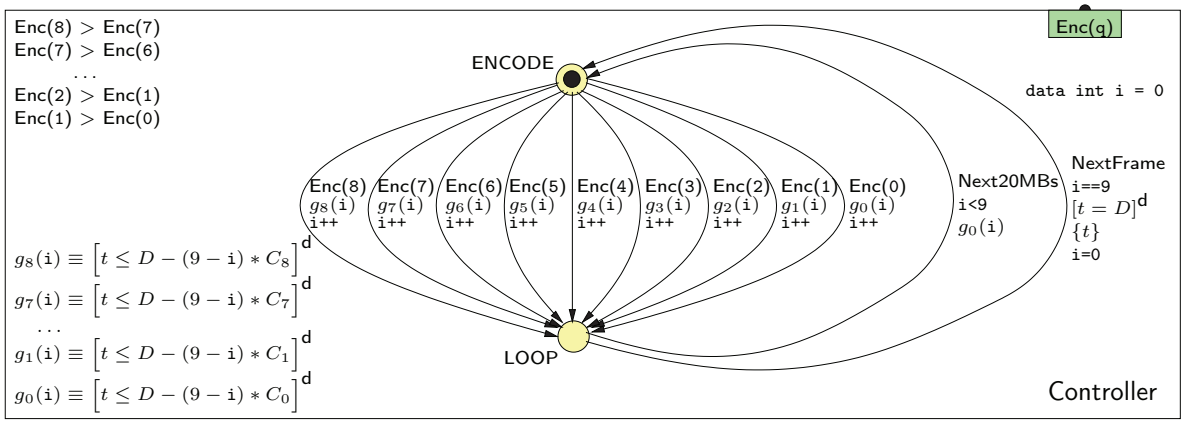


Figure 5.11: Controller component.

q	0	1	2	3	4	5	6	7	8
C_q	4	4.6	5.4	6	8.2	10	12	14.4	16

Table 5.1: Estimates of average execution times (ms).

Time-Safety

As execution times of the video encoder may vary a lot from a frame to another [60], we studied time-safety for a family of execution time functions $K\varphi$, where the parameter K ranges in $[0.001, 2]$, and where φ denotes an execution time function corresponding to the actual execution of the video encoder on the target platform for a particular frame.

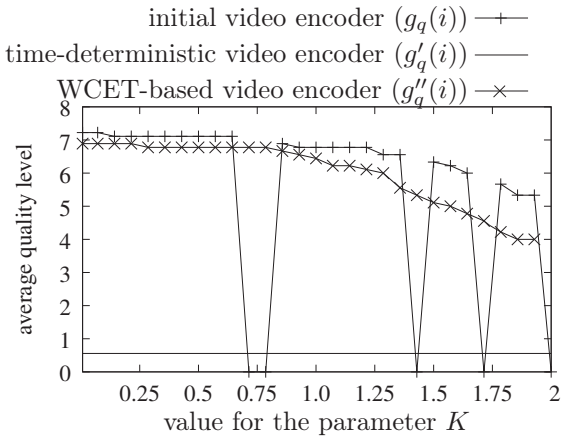


Figure 5.12: Video encoder execution for execution time functions $K\varphi$.

Figure 5.12 shows average quality levels chosen for different values of the parameter K. They are increasing as K is decreasing. Time-safety is violated for $K = 1.7$ and $K = 1.4$, even if time-safety is guaranteed for $K \in [0.9, 1.3]$ (i.e. lower execution times). That is, the application is not time-robust. This is due to the fact that the controller uses estimates of execution times which can be different from the actual execution times. This difference depends on the chosen quality levels, that is, on the value of K. Therefore, increasing the platform speed (i.e. reducing K) does not guarantee time-safety: time-safety violations

occur for $K = 0.7$ and $K = 0.8$ (see Figure 5.12).

When time-safety is violated by the video encoder, the current frame is skipped which is equivalent to encoding all its macroblocks at quality level 0. This leads to a drastic degradation of the video quality.

Time-robustness is a desirable property of an application since it allows better predictability of its behavior, that is, a time-robust application is time-safe for any execution times provided that it is time-safe for worst-case execution times. We consider two methods for enforcing time-robustness of the adaptive video encoder.

Enforcing Time-Robustness by Time-Determinism

As explained by Proposition 1 of Section 2.2, time-robustness can be guaranteed by enforcing time-determinism. This can be achieved by modifying all inequalities involved in guards $g_q(i)$ of Controller into delayable equalities $g'_q(i) \equiv [t = D - (9-i)C_q]^{\mathsf{d}}$. Using equalities $g'_q(i)$ instead of inequalities $g_q(i)$ for Controller leads to the following execution. At intial state (ENCODE, 0), Controller waits for the enabledness of a transition issued from ENCODE. As $D - (9-i)C_q$ is minimal for $q = 8$, Controller executes action Enc(8) after waiting for $D - 9C_8$. Actions Enc(0), Enc(1), ..., Enc(7) cannot be chosen from the initial state since Enc(8) is urgent at $D - 9C_8$, that is, the chosen quality level at the first iteration is $q_0 = 8$. This leads to the control location LOOP at which only Next20MBs is enabled when t reaches $D - (9-1)C_0$, leading back to control location ENCODE with $t = D - (9-1)C_0$. That is, only the quality level 0 can be chosen (i.e. $q_1 = 0$). Similarly, chosen quality levels q_i, $i > 0$, for the remaining iterations are also 0.

The time-deterministic video encoder chooses the same quality levels (i.e. $q_0 = 8$, $q_i = 0$ for $i > 0$) for all considered values of K, that is, there is no adaptation of the quality levels with respect to actual execution times $K\varphi$. Time-robustness incurs a severe reduction of the quality of the video as shown in Figure 5.12).

Enforcing Time-Robustness using WCET

Time-robustness can also be achieved by enforcing time-safety for the component Controller when execution times of actions are equals to worst-case execution times (WCET) C_q^{wc}, as explained in [42]. Notice that these values satisfy $C_q \leq C_q^{wc}$. The principle is to strengthen guards $g_q(i)$ for transitions, based on a WCET analysis of the controlled system. Given a quality level q chosen for an iteration i, an estimate of the worst-case execution time of the controlled video encoder for encoding the remaining macroblocks of the current frame is given $C_q^{wc} + (8-1)C_0^{wc}$, that is, we consider the worst-case estimates at quality level q for the next iteration, and the worst-case estimates at minimal quality level q_0 for the remaining iterations. Following [42] we consider a controller based on guards $g''_q(i) \equiv t \leq D - (\mathbf{max}\ (9-i)C_q\ ,\ (C_q^{wc} + (8-1)C_0^{wc}))$ that combined both estimates of the worst-case execution times and the average execution times. They ensure that there always exist a strategy for completing before the deadline—at worst-case the minimal quality level is chosen, even if actual execution times are equal to estimates of the worst-case execution times.

As shown in Figure 5.12, this conservative approach guarantees time-robustness by a slight reduction of the chosen quality levels with respect to the ones chosen by the initial video encoder. This is due to the use of the strengthen guards $g''_q(i)$ that are more

conservative. It appears to be a better approach for enforcing time-robustness than using time-determinism.

4 Conclusion

In this chapter, we provided a rigourous implementation method using a real-time execution Engine that faithfully implements physical models, which are abstract models with given execution times on the target platfom. We consider that the application software is a set of interacting components such as each component is represented by an abstract model. The real-time execution Engine coordinates the execution of the application software by applying a sequential composition of three micro-steps:

1. Computes the timing constraints of enabled interactions intervals by applying the semantics of the abstract model. Timing constraints are specifed by using a global abstract time clock t.

2. Updates the abstract time t by the physical time t_r provided by the execution platform. If t_r exceeds the earliest deadline of the enabled interactions, time-safety is violated and the execution stops.

3. Schedules amongst the enabled interactions the next interaction to be executed.

The real-time execution Engine detects time-safety violations when the implementation does not correspond to the abstract model specification. It is correct-by-construction because it executes an algorithm that directly implements the operational semantics of the physical model. The method leads, under some robustness assumptions for WCET times, to a correct implementation. If robustness cannot be guaranted for a model, the real-time execution Engine checks online if the execution is correct, that is, deadlock-freedom and time-safety violation.

We implemented the method for the component-based framework BIP presented in chapter 2. We extended the BIP language for expressing real-time features in the BIP models, that is, clocks and timing constraints for transitions in atomic components. The resulting real-time BIP models can be compiled to C++ code then executed by the real-time BIP execution Engine. We then extended the initial single-threaded BIP Engine to correctly execute the models by producing correct schedules of interactions, meeting to timing constraints and by detecting time-safety violations. In the initial BIP framework, variables are used to model local clocks. In the BIP model, executing transitions takes zero time, as a result, time (i.e. clocks) can only advance on states. Time advance is achieved synchronously by an implicit tick connector. At each tick, every variable representing a clock is increased by one unit of time and in order to model urgency, time advance can be disabled under some conditions. We have seen that, although this model can be interesting for modeling and simulating systems, it cannot be used in practice for implementation purposes. Using global tick synchronizations often leads to inefficient implementations, especially when the considered system is asynchronous. For thoses systems, unnecessary clock synchronizations are introduced which consumes a lot of CPU time. Moreover, a global tick requires the same level of granularity for every clock (timed data) in the model. This means that the execution time of each block of C code has to be lower than one unit of time, which is very restrictive and requires rewriting some parts of the application code.

We also studied time-safety and time-robustness for an adaptative MPEG video encoder modeled and executed using the real-time BIP framework. We show that time-safety violation leads to a drastic degradation of the video quality, due to incorrect estimates of the average execution times for encoding the macroblocks. Indeed, the choice of quality for encoding a macroblock dependends on the time elapsed since the beginning of the encoding of the frame and the time budget left before the deadline. Enforcing time-robustness, that is a desirable property of an application allowing better predictability of its behavior, entails some performance penalty. We have tested two methods to enforce time-robustness. The first method consists of enforcing time-determinism but looses adaptation of the quality levels. The second method is based on WCET analysis and leads only to a slight reduction of the chosen quality levels with respect to the ones chosen by the initial video encoder, thus, it seems to be a better solution.

In chapter 5, we will study the real-time properties for open real-time systems, that is, systems that interact with their execution environment while meeting timing constraints. Indeed, the behavior of an open system depends not only on its current state (like for a closed system), but also on the behavior of its environment. We extended the real-time BIP framework so as to take into account the communication with the environment.

Chapter 6

Open Real-Time Systems

In the previous chapters, we presented a rigorous model-based design and implementation method for building real-time systems. We gave formal definitions on which we rely for the implementation of real-time systems correct-by-construction. We also presented an implementation method based on a component-based approach using the BIP framework. Nevertheless, we only considered closed systems without taking into account the interactions of the application with its external environment. In this chapter, we extend the previous formal definitions and implementation method for open real-time systems. An open real-time system interacts with its external environment while meeting timing constraints. The behavior of an open system depends not only on its current state (like for closed systems), but also on the behavior of its environment.

Providing methodologies encompassing open systems design is too often a neglected issue. Nowadays, component-based approaches are privileged instead of monolithic approaches since they favor flexible development and reusability. They rely on the principle of encapsulation allowing building applications by composition of existing components. Many of the implementations involve ad-hoc mechanisms for the communication with the environment, as it is not explicitely modeled (e.g. shared memory), which violates this encapsulation principle and makes the analysis of the system intractable. We present a model-based method for building open real-time systems based on the use of two models:

- The **open abstract model** represents the timing behavior of the application without considering any execution platform. Interactions with the environment are modeled using Input/Output automata [70], in which actions correspond either to internal computations, or to communications with the environment, i.e. inputs and outputs. Internal actions and outputs are triggered by the application, whereas inputs are triggered by the environment. Timing constraints correspond only to user-requirements (deadlines, periodicity, occurence of inputs ..), based on an abstract notion of time (i.e. timeless execution of actions).

- The **open physical model** represents the behavior of the abstract model running on a given platform, that is, it takes into account execution times. It is obtained from the abstract model by assigning execution times to actions. It is necessary for checking the adequacy of the open abstract model to an execution platform and a given environment behavior.

The provided implementation method for component-based applications defines an open real-time Engine that performs the online computation of schedules meeting the semantics of the physical model depending on the actual execution times and the occurrence of inputs provided by the environment. In contrast to standard even-driven programming techniques, our method allows static analysis and online checking of essential properties such as time-safety and time-robustness.

The chapter is organized as follows. Section 1 introduces the notion of open real-time systems. Section 2 presents the definitions of open abstract and open physical models, and the properties of time-safety and time-robustness under the environment constraints. This section is an extension of the abstract and physical models definitions and properties presented in Chapter 4. In Section 3, we present our implementation method which is also an extension of the implementation method presented in Chapter 5. In Section 4, we present the experiments that has been conducted using the real-time BIP framework.

1 Introduction

Embedded Systems (ES) are in general open real-time systems since they have to communicate with a physical environment under real-time constraints. Such applications include communication systems, aircraft control systems, automotive systems, games and toys, etc. In general, the associated software is encapsulated in a box with no outside connectivity that can alter the behavior. So most such embedded systems are closed "boxes" that do not expose the computing capability to the outside. In a networked environment, it becomes impossible to test the software under all possible conditions. Moreover, when it comes to network a set of communicating ES devices, it becomes difficult to achieve predictable timing in the face of such openness.

Event-driven [49,50,74] design of real-time systems usually considers components (tasks) that can be triggered by events captured by interruptions. Scheduling theory guarantees only estimates of system response time for periodic execution of the components when periods and worst-case execution times (WCETs) are known. The resulting system behavior is strongly related to the chosen execution platform. In contrast, synchronous programs consider synchronized components whose execution is a sequence of non-interruptible steps that define a logical notion of time. In a step each component performs a quantum of computation. Time-triggered approaches generalize the notion of logical time. They consider different computation steps for the components. Each component defines a sequence of actions with specified logical execution times (LET) defining the difference between their release time and their due time. The system behaves as if actions consume exactly their LET: even if they start after their release time and complete before their due time, their effect is visible exactly at these times. This is achieved by reading for each action its input exactly at its release time and its output exactly at its due time. LET guarantee the system behavior predictability since it is independent from the platform, as long as actions complete before their due time. Components may describe arbitrary sequences of LET [8], or may be restricted to periodic execution (i.e. uniform LET) for each component with possible global mode switches as in [58].

Mixing event-driven programming and time-triggered behavior is a promising approach introduced in [51]. They consider time-triggered actions whose instantiation may depend on external (asynchronous) events. In the previous chapters, we extended this principle by considering more general timing constraints than (fixed) LET, such as lower bounds, upper bounds, time non-determinism. The contribution of this thesis is the improvement of the approach by considering not only internal actions but also input and output communications with the environment.

Open real-time systems must also react to multiple real-time streams of sensor stimuli and control multiple actuators concurrently. Regrettably, the mechanisms of interaction with sensor and actuator hardware, built for example on the concept of interrupts, are not well represented in programming languages. They have been left to be the domain of operating systems, not of software design and the interactions with hardware are exposed to programmers through the abstraction of threads. The purpose of our method is to ascend the possibility to deal with mechanisms of interaction between the application and its environment in the design level.

We focus on a general schema of communication between the application software and its environment via sensors and actuators. Sensors and actuators offer to the application some interfaces in order to observe and modify the state of the environment (see Figure 6.1). Those

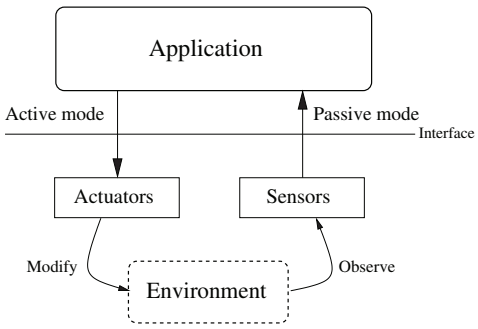


Figure 6.1: Communication modes between an application and its environment.

known interfaces implement different modes of communications between the application and the environment. We consider two modes:

- An active mode in which the environment is always ready for interacting, that is, the application can communicate at any time. It is referred in the paper as *output* following [70], and is often used for implementing the interactions between the application and the actuators of the platform.
- A passive mode in which the application is waiting for the environment to be ready for interacting. It is referred as *input* following [70], and is often used for implementing the interactions between the application and the sensors of the platform, that is, the application waits for events and associated data produced by the environment.

2 Open Abstract and Physical Models

2.1 Open Abstract Models

In order to represent the behavior of an open real-time application, we use Input/Output (I/O) timed automata [43, 70]. An I/O timed automaton is a timed automaton such that actions labeling transitions are partitioned into *inputs*, *outputs* and *internal actions*. The distinction between inputs and other actions is a fundamental property, based on who determines when the action is performed. The actions whose performance is under the control of the environment are inputs and actions whose performance is under the control of the application are outputs and internal actions.

The open abstract model formal definition is now given as follows.

Definition 26 (open abstract model) *An* open abstract model *is an I/O timed automaton* $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ *such that:*

- $\mathsf{A} = \mathsf{A}^{\mathsf{in}} \bigcup \mathsf{A}^{\mathsf{out}} \bigcup \mathsf{A}^{\mathsf{int}}$ *is a finite set of* actions *partitioned into inputs* A^{in}*, outputs* $\mathsf{A}^{\mathsf{out}}$ *and internal actions* $\mathsf{A}^{\mathsf{int}}$*,*
- Q *is a finite set of* control locations*,*
- X *is a finite set of* clocks*,*

- *and* $\longrightarrow \subseteq \mathsf{Q} \times (\mathsf{A} \times \mathsf{G}(\mathsf{X}) \times 2^{\mathsf{X}}) \times \mathsf{Q}$ *is a finite set of labeled transitions. A transition is a tuple* (q, a, g, r, q') *where* a *is an action,* g *is a timing constraint, that is a*guard *over* X *and* r *is a subset of clocks that are reset by the transition. We write* $q \xrightarrow{a,g,r} q'$ *for* $(q, a, g, r, q') \in \longrightarrow$.

An open abstract model is different from an abstract model (Definition 27) in the fact that actions are partionned into internal, input and output actions. In open abstract models, transitions labeled by an input are triggered by the environment, and correspond to reception of this input by the application. We make the assumption that timing constraints associated to transitions labelled by inputs correspond to the expected timing behavior of the environment. That is, inputs arrive within the time interval of the timing constraint. Timing constraints associated to transitions labelled by internal and output actions take into account only requirements (e.g. deadlines, periodicity, etc.). The semantics of open abstract models, like for abstract models, assume timeless execution of actions. This corresponds to the ideal and platform independent behavior of a real-time application.

The semantics of an open abstract model is the same as an absract model. Given an open abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ a finite (resp. an infinite) *execution sequence* of M from an *initial state* (q_0, v_0) is a sequence of actions and time-steps $(q_i, v_i) \xrightarrow{\sigma_i} (q_{i+1}, v_{i+1})$ of M, $\sigma_i \in \mathsf{A} \cup \mathbb{T}$ and $i \in \{\ 0, 1, 2, \ldots, n\ \}$ (resp. $i \in \mathbb{N}$). From any state of an open abstract model it is always possible to execute either an action or a time-step, or both. A state from which only time can progress, i.e. from which all execution sequences are only composed of time-steps, is a *deadlock*. We also consider abstract models that are structurally non-zeno [30]. This class of abstract models does not have time-locks, that is, time can always eventually progress.

Example 14 *Consider an open abstract model* $M = (\{\mathit{in}, \mathit{out}, \mathit{compute}\}, \mathsf{Q}, \{x\}, \longrightarrow)$ *with an input* *in*, *an output* *out*, *and an internal action* *compute*. *It has also a set of control locations* $\mathsf{Q} = \{\mathit{init}, \mathit{get}, \mathit{exec}\}$, *and the following set of transitions (Figure 6.2):*

$$\begin{aligned} \longrightarrow \ = \{\ & (\mathit{init}, \mathit{in}, [x \geq D]^{\mathsf{I}}, \{x\}, \mathit{get}), \\ & (\mathit{get}, \mathit{compute}, [x \leq D]^{\mathsf{d}}, \emptyset, \mathit{exec}), \\ & (\mathit{exec}, \mathit{out}, [x \leq D]^{\mathsf{d}}, \emptyset, \mathit{init}).\ \ \}. \end{aligned}$$

The model represents a cyclic execution of a system that receives an input *in* from the environment (transition from *init* to *get*), performs an internal computation (transition from *get* to *exec*), and sends an output *out* to the environment (transition from *exec* to *init*). A clock x is used to measure the time elapsed since the last occurrence of *in* (i.e. x is reset by *in*). Both *compute* and *out* must be done before x reaches D. Notice that *in* is not enabled at control locations *exec* and *get*, and at control location *init* when $x < D$.

It can easily be shown that M admits execution sequences from the initial state (init, D) of the following form: $(\mathit{init}, D) \xrightarrow{\delta_1} (\mathit{init}, D{+}\delta_1) \xrightarrow{\mathit{in}} (get, 0) \xrightarrow{\delta_2} (get, \delta_2) \xrightarrow{\mathit{compute}} (\mathit{exec}, \delta_2) \xrightarrow{\delta_3}$ $(\mathit{exec}, \delta_2 + \delta_3) \xrightarrow{\mathit{out}} (\mathit{init}, \delta_2 + \delta_3) \xrightarrow{\delta_4} (\mathit{init}, \delta_2 + \delta_3 + \delta_4) \xrightarrow{\mathit{in}} (get, 0)$.

Due to the guards of *compute*, *out*, and *in*, waiting times δ_i must satisfy $\delta_2 + \delta_3 \leq D$ and $\delta_2 + \delta_3 + \delta_4 \geq D$, meaning that there is a delay of at least D time units between two consecutive occurrences of *in*.

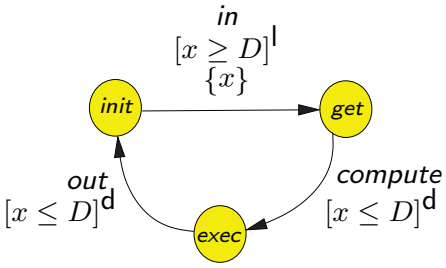


Figure 6.2: Open abstract model example

2.2 Open Physical Models

Open physical models are open abstract models modified so as to take into account non-null execution times. They represent the behavior of an application software running on an execution platform and interacting with its environment. We consider that an open physical model is time-safe if its execution sequences are also execution sequences of the corresponding open abstract model, that is, the execution times are compatible with the timing constraints and the expected behavior of the environment is compatible with the actual arrival of inputs from the environment.

Definition 27 (open physical model) *Let* $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ *be an open abstract model and* $\varphi : \mathsf{A} \rightarrow \mathbb{T}$ *be an* execution time function *that gives for each action* a *its execution time* $\varphi(a)$.

The open physical model $M_\varphi = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow, \varphi)$ *corresponds to the abstract model* M *modified so that each transition* (q, a, g, r, q') *of* M *is decomposed into two consecutive transitions:*

1. *The first transition* $(q, a, g, r \cup \{x_a\}, wait_a)$ *corresponds to the beginning of the execution of the action* a. *It is triggered by guard* g *and it resets the set of clocks* r, *exactly as* (q, a, g, r, q') *in* M. *It also resets an additional clock* x_a *used for measuring the execution time of* a.

2. *The second transition* $(wait_a, end_a, g_{\varphi(a)}, \emptyset, q')$ *is labeled by internal action* end_a *and corresponds to the completion of* a. *It is constrained by* $g_{\varphi(a)} \equiv [x_a = \varphi(a)]^{\mathsf{d}}$ *that enforces waiting time* $\varphi(a)$ *at control location* $wait_a$, *which is the time elapsed during the execution of the action* a.

The above definition of physical models corresponds to a purely sequential execution of actions. In particular, inputs cannot occur when an action is executing. In practice most of the platforms offer hardware mechanisms such as interruptions that can be used to react to inputs while the application is running. Exploiting these mechanisms for safely taking into account occurrences of inputs during actions execution is discussed in Section 3.

Input-Enablness

In open abstract models, timing constraints, that is, guards of transitions, take into account only requirements (e.g. deadlines, periodicity, etc.). Transitions labeled by an input are triggered by the environment, and correspond to reception of this input by the application.

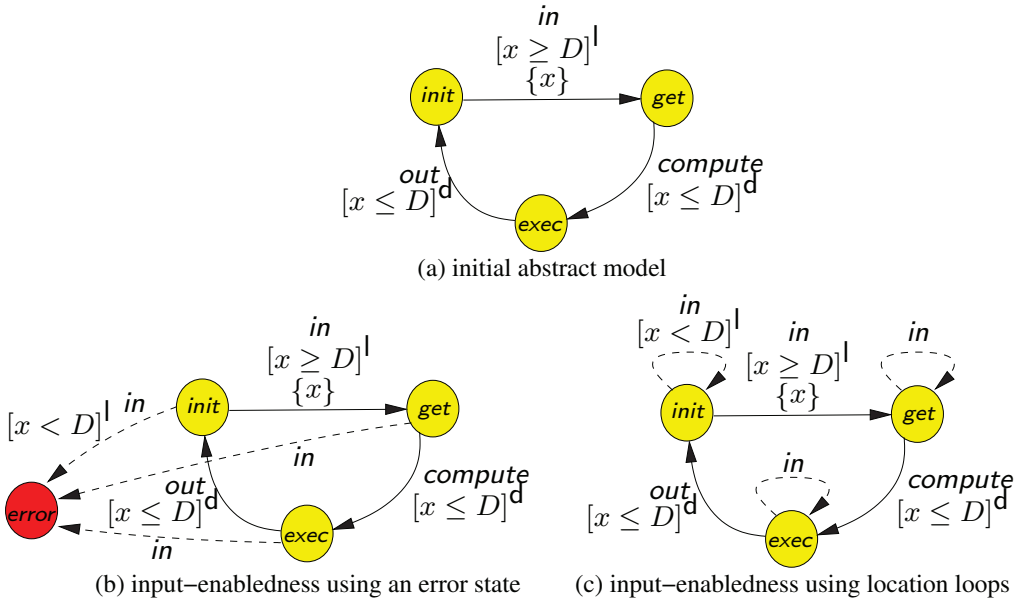


Figure 6.3: Enforcing input-enabledness.

Since an application cannot block its environment, inputs may occur even if they are not enabled by the model, as explained in [43]. An open abstract model that enables all its inputs from all its states is said to be *input-enabled*. When this property does not hold, that is, the environment does not behave as expected, there are two ways for interpreting the occurrence of an input at a state for which it is disabled.

Error. It can be interpreted as an error. In this case, the timing constraints of the open abstract model provide assumptions on the behavior of the environment. If the actual behavior of the environment violates one of these assumptions, an error is reported.

Ignore. It can be ignored. In this case the timing constraints of the abstract model are used for masking or filtering the behavior of the environment.

These two interpretations can be modeled by introducing a default behavior in case of absence of an input transition from a state of an open abstract model. Figure 6.3 shows an example of abstract model (a) and its corresponding abstract models that are obtained for the first interpretation (error) (b), and for the second interpretation (ignore) (c).

In the first interpretation, the error policy is modeled using transitions, labelled by the input *in*, leading to an error state. The transition issued from state *init* leads to the *error* state whenever an input occurs such as $x < D$. The transitions issued from state *get* and state *exec* lead to the *error* state whenever an input occurs because only the execution of *compute* (resp. out) action is possible at state *get* (resp. exec).

In the second interpretation, the ignore policy is modeled using transitions corresponding to location loops in every state. Indeed, they correspond to the same added transitions as the error policy, except that they don't lead to an error state. Instead, they allow to stay in the same state until an enabled transition in the initial open abstract model is possible.

2.3 Time-Safety and Time-Robustness

A crucial question is the divergence of the implementation from its abstract specification, that is, the difference between the behavior of an open physical model and the behavior of its corresponding open abstract model. In an open physical model M_φ, every execution of an action a is immediately followed by waiting for $\varphi(a)$ time units enforced at an intermediate state. This waiting in M_φ may not correspond to a behavior specified by the open abstract model M if there exists in M either an urgent transition or an enabled input, which corresponds to a deadline miss or input miss.

Let's consider the execution in an open physical model M_φ of an action a at state (q, v). We have the following execution sequence of M_φ:

$$(q,v) \xrightarrow{a} (wait_a, v') \xrightarrow{\varphi(a)} (wait_a, v'')) \xrightarrow{end_a} (q', v''),\ where\ v'' = v' + \varphi(a).$$

Deadline Miss

We consider the case where no input occurs during the execution of a in M_φ, i.e. no input occurs at states $(wait_a, v' + \delta)$, $0 \leq \delta \leq \varphi(a)$. This is equivalent, if it exists, to the following execution of the corresponding open abstract model M:

$$(q, v_{|\mathsf{X}}) \xrightarrow{a} (q', v'_{|\mathsf{X}}) \xrightarrow{\varphi(a)} (q', v'_{|\mathsf{X}} + \varphi(a)), \tag{6.1}$$

where $v'_{|\mathsf{X}}$ denotes the restriction of v' to clocks X, that is, $v'_{|\mathsf{X}}$ is a valuation of clocks X such that $v'_{|\mathsf{X}}(x) = v(x)$ for all $x \in \mathsf{X}$. In the following, we write $(q, v) \xrightarrow{a,\varphi(a)} (q', v' + \varphi(a))$ for execution sequence (6.1) in M and its corresponding execution sequence in M_φ. Notice that (6.1) may not be an execution sequence of M if there exists a transition $q' \xrightarrow{a',g',r'} q''$ such that $\mathsf{urg}[g'](v'_{|\mathsf{X}} + \delta)$ and $\delta \in [0, \varphi(a)[$, meaning that the physical model violates timing constraints defined in the corresponding abstract model. In this case we say that the action a' *misses its deadline* (see Figure 6.4).

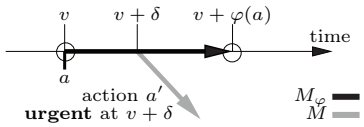


Figure 6.4: Action a' miss its deadline in M_φ.

Input Miss

We consider the case where an input occurs during the execution of a in M_φ, i.e. an input occurs at states $(wait_a, v' + \delta)$, $0 \leq \delta \leq \varphi(a)$. The behavior of M is not equivalent to the one of M_φ if it exists a transition labeled by i enabled at state $(q', v'_{\mathsf{X}} + \delta)$ in M, i.e. $q' \xrightarrow{i,g',r'} q''$ such that $g'(v'_{|\mathsf{X}} + \delta)$. In this case we say that the input i *is missed* in M_φ (see Figure 6.5).

If an input $i \in \mathsf{A^{in}}$ occurs during the execution of a, i.e. at a state $(wait_a, v' + \delta)$, $0 \leq \delta \leq \varphi(a)$, it will be either interpreted as an error or ignored as explained in Section 2.2.

Since outputs and internal actions are triggered by the application, their execution can be controlled by scheduling policies. We consider *minimally waiting* schedulers, that is,

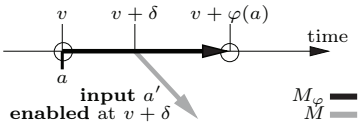


Figure 6.5: Input a' missed in M_φ.

$Case:$	1	2	3	4
$\varphi(in)$	10	20	10	10
$\varphi(compute)$	40	50	40	30
$\varphi(out)$	10	10	30	10

Table 6.1: Execution times for the open abstract model example (ms) for different cases.

execution sequences of physical models such that waiting times for outputs and internal actions are minimal. More formally, if $(q,v) \xrightarrow{\delta} (q, v+\delta) \xrightarrow{a} (q', v')$ is an execution sequence of M_φ such that $a \in \mathsf{A}^{\mathsf{out}} \cup \mathsf{A}^{\mathsf{int}}$, then $\delta = \mathbf{min}\ \{\ \delta' \geq 0 \mid g(v+\delta')\ \}$ where g is the guard of the action a at control location q. This assumption cannot apply to inputs as they are controlled by the environment, that is, we consider they can occur at any time instant meeting the timing constraints.

Example 15 *We consider the open abstract model presented in Example 14 (see Figure 6.2). We study the safety execution of its corresponding open physical model for different cases corresponding to different execution times of actions (see Table 6.1). We fix the deadline $D = 60ms$ and the initial state is (init, D).*

Case 1: Safe Execution.

This case study corresponds to a safe execution of the open physical model. The execution time function φ is such that $\varphi(\mathsf{in}) = 10$, $\varphi(\mathsf{compute}) = 40$ and $\varphi(\mathsf{out}) = 20$ and its corresponding execution sequence is (see Figure 6.6) :
$(init, D) \xrightarrow{in} (get, 10) \xrightarrow{compute} (exec, 50) \xrightarrow{out} (init, 60)$
This execution sequence is correct since the execution times of actions do not violate the timing constraints of the open abstract model. After the execution of input in, the value of the clock is $x = 10ms$. The execution of action $compute$ is possible since the deadline $D = 60ms$ is not reached. After the execution action $compute$, the value of the clock is $x = 10 + \varphi(\mathsf{compute}) = 50ms$. The execution of the output out is also possible since clock x did not reach the deadline $D = 60ms$. After the execution of out, the value of the clock is $x = 50 + \varphi(\mathsf{compute}) = 60ms$, the application is ready to receive another input.

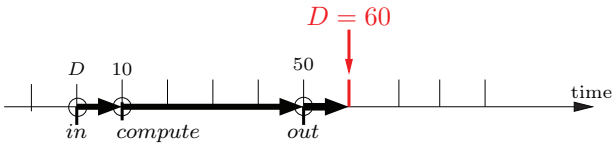


Figure 6.6: Safe Execution of M_φ.

Case 2: Deadline Miss.

In this case study, the execution of the physical model is not safe is not compatible with the timing constraints of actions, that is , a dealine miss is detected. The execution time function φ is such that $\varphi(\mathit{in}) = 20$, $\varphi(\mathit{compute}) = 50$ and $\varphi(\mathit{out}) = 10$ and the corresponding execution sequence is as follows (see Figure 6.7):

$(init, D) \xrightarrow{in} (get, 20) \xrightarrow{compute} (exec, 70) \xrightarrow{out} (init, 80)$

This execution sequence is not correct because the deadline $D = 60ms$ of the transition labelled by the output *out* is missed. After the execution of action *compute*, the value of the clock is $x = 70m$. The transition labelled by *out* should have been executed before the value of clock x becomes greater than $D = 60ms$. Thus, the system detectes a timing constraint violation.

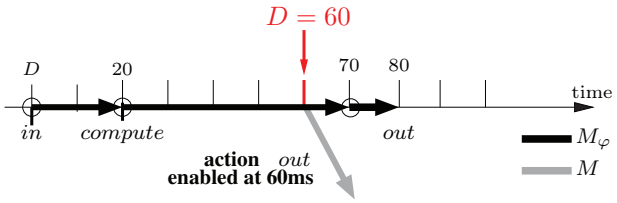


Figure 6.7: Deadline Miss in M_φ.

Case 3: Input Miss.

In this case study, the execution of the physical model is not compatible with the arrival of inputs from the environment, that is, an input miss is detected. The execution time function φ is such that $\varphi(\mathit{in}) = 10$, $\varphi(\mathit{compute}) = 40$ and $\varphi(\mathit{out}) = 30$ and the corresponding execution sequence should be as follows (see Figure 6.8) :

$(init, D) \xrightarrow{in} (get, 10) \xrightarrow{compute} (exec, 50) \xrightarrow{out} (init, 80)$

This execution sequence is correct only if an input doesn't occur during the execution of an internal or output action. If an input *in* occurs while the execution of output *out*, that is such as $60 \leq x \leq 80$, the new input *in* is missed because it is enabled in the open abstract model. Indeed, a new input *in* may occur at any time such as $x \geq D = 60ms$.

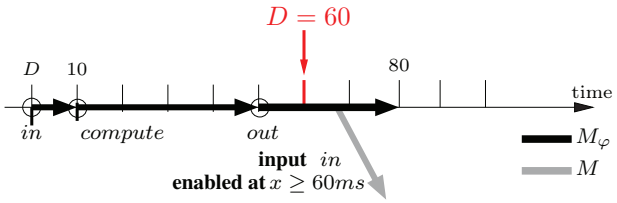


Figure 6.8: Deadline Miss in M_φ.

Case 4: Input-Enablness.

In this case study, the arrival of inputs is not compatible with the abstract model specifications. The execution time function φ is such that $\varphi(\mathit{in}) = 10$, $\varphi(\mathit{compute}) = 30$ and $\varphi(\mathit{out}) = 10$ and the corresponding execution sequence should be as follows (see Figure 6.9):

$(init, D) \xrightarrow{in} (get, 10) \xrightarrow{compute} (exec, 40) \xrightarrow{out} (init, 50)$

This execution sequence is correct since all the deadlines are met. Nevertheless, if we consider that a new input *in* occurs after the execution of output *out*, such as $50 \leq x < 60$, the new input *in* is not enabled in the open abstract model. Only inputs that occur such as $x \leq D = 60ms$ are enabeled. We can apply the input-enablness policies for the corresponding open abstract model. We can either choose to ignore the new input *in* or to rise an error.

This case study may occur in the other cases whenever an input occurs while the execution of an action.

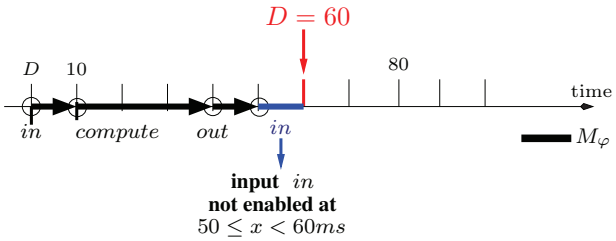


Figure 6.9: Deadline Miss in M_φ.

Definition 28 (time-safety and time-robustness) *A physical model* $M_\varphi = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow, \varphi)$ *is* time-safe *for an initial state* (q_0, v_0) *if the set of execution sequences of* M_φ *from* (q_0, v_0) *is contained in the set of execution sequences of* M*.*

A physical model M_φ *is* time-robust *if* $M_{\varphi'}$ *is time-safe for all execution times* $\varphi' \leq \varphi$*. An abstract model is* time-robust *if all its time-safe physical models are time-robust.*

Time-safety for an open physical model expresses that its execution times are compatible with the timing constraints and the possible occurrences of inputs defined in the corresponding open abstract model. Another important property for an open physical model is time-robustness, that is, time-safety is preserved when reducing the execution times. Worst-case analysis of the behavior of a system usually requires time-robustness since it is often based on upper bounds of the execution times (WCETs).

Consider the abstract model M given in Example 14, the execution time function φ such that $\varphi(\mathsf{in}) = \alpha$, $\varphi(\mathsf{compute}) = \beta$ and $\varphi(\mathsf{out}) = \gamma$, and the initial state (init, D). Time-safe physical models satisfy $\alpha + \beta + \gamma \leq D$. They admit execution sequences of the form:

$$(\mathsf{init}, D) \xrightarrow{\delta_1} (\mathsf{init}, D{+}\delta_1) \xrightarrow{\mathsf{in},\alpha} (\mathsf{get}, \alpha) \xrightarrow{\mathsf{compute},\beta} (\mathsf{exec}, \alpha{+}\beta) \xrightarrow{\mathsf{out},\gamma} (\mathsf{init}, \alpha{+}\beta{+}\gamma) \xrightarrow{\delta_2} (\mathsf{init}, D),$$

$$where\ \delta_2 = D - (\alpha + \beta + \gamma).$$

Notice that for Example 14 time-safety implies time-robustness, but this is not the case in general. An example of non time-robust abstract model is provided in Chapter 4.

We extend results for time-deterministic abstract models given in Chapter 4 . A time-deterministic model is such that each action has a logical execution time (LET) which is a fixed time budget for its execution. It guarantees that for a given sequence of inputs, it always executes outputs at the same time instants.

Proposition 3 *A* time-deterministic *open abstract model, that is, such that its outputs and internal actions are guarded by delayable or eager equalities, is time-robust*

Proof 4 *of proposition Let M be an open abstract model such that its internal actions and outputs are guarded by delayable or eager equalities, and let $\varphi' \leq \varphi$ be execution time functions such that M_φ is time-safe. We have to show that $M_{\varphi'}$ is also time-safe, i.e. all its execution sequences are also execution sequence of M.*

We prove by induction that execution sequences of $M_{\varphi'}$ are also execution sequences of M_φ. Consider the following execution sequence of $M_{\varphi'}$:

$$(q, v) \xrightarrow{a,\varphi'(a)} (q', v' + \varphi'(a)) \xrightarrow{\delta} (q', v' + \varphi'(a) + \delta) \xrightarrow{b,\varphi'(b)},$$

and assume that (q, v) is a reachable state of both M_φ and $M_{\varphi'}$. We will show that there exists a corresponding execution sequence in M_φ. Notice that by definition of open physical models, $(q', v' + \varphi'(a)) \xrightarrow{\delta} (q', v' + \varphi'(a) + \delta)$ is a transition of M. Moreover, as M_φ is time-safe, $(q, v) \xrightarrow{a,\varphi(a)} (q', v' + \varphi(a))$ is also a transition of M. By definition of open abstract models and as $\varphi' \leq \varphi$, transition $(q, v) \xrightarrow{a,\varphi'(a)} (q', v' + \varphi'(a))$ is also a transition of M. As a result, no transition is urgent at states $(q', v + \varepsilon)$, $\varepsilon \in [0, \varphi'(a) + \delta[$, and no input is enabled at states $(q', v + \varepsilon)$, $\varepsilon \in [0, \varphi(a)[$.

Case #1:** b **is an internal action or an output.

As the guard of b is a delayable or eager equality, b is urgent at state $(q', v' + \varphi'(a) + \delta)$, and is only enabled at this state. Time-safety of M_φ implies that $\varphi(a) \leq \varphi'(a) + \delta$. Since no other transition is urgent before, we conclude that b can be executed in M_φ at this state without violating the minimally waiting principle, according to the following execution sequence of M_φ:

$$(q, v) \xrightarrow{a,\varphi(a)} (q', v' + \varphi(a)) \xrightarrow{\gamma} (q', v' + \varphi'(a) + \delta) \xrightarrow{b,\varphi(b)},$$
$$\text{where } \gamma = \varphi'(a) + \delta - \varphi(a).$$

Case #2:** b **is an input.

Since b is an input enabled at $(q', v' + \varphi'(a) + \delta)$ and no transition is urgent at states $(q', v + \varepsilon)$, $\varepsilon \in [0, \varphi'(a) + \delta[$, we have the following execution is M_φ:

$$(q, v) \xrightarrow{a,\varphi(a)} (q', v' + \varphi(a)) \xrightarrow{\gamma} (q', v' + \varphi'(a) + \delta) \xrightarrow{b,\varphi(b)},$$
$$\text{where } \gamma = \varphi'(a) + \delta - \varphi(a).$$

3 Open Real-time Execution Engine

Based on the results of the previous section , we propose an implementation method for a given open physical model. We extended the capabilities of the real-time Engine presented in Chapter 5 which already guarantees a correct implementation of closed real-time systems, that is, it checks that the timing constraints requirements are met by the execution on a target platform. We focus in this section on the correct implementation of real-time systems interacting with their environment. If the corresponding open abstract model is time-robust, then time-safety for the worst-case execution times ensures time-safety of the implementation. Otherwise, time-safety is checked online and the execution is stopped if it is violated.

3.1 Composition of Models

We consider that the application software is a set of interacting components. Each component is represented by an open abstract model. Thus the open abstract model M corresponding to the application is the parallel composition of the I/O timed automata representing the components. This composition is parameterized by a set of interactions defining synchronizations between internal actions of the components and, inputs and outputs defining the interactions with the environment.

Let $M^i = (\mathsf{A}_i, \mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i)$, $1 \leq i \leq n$, be a set of open abstract models with disjoint sets of actions $\mathsf{A}_i = \mathsf{A}_i^{\mathsf{in}} \cup \mathsf{A}^{\mathsf{out}} \cup \mathsf{A}_i^{\mathsf{int}}$ and clocks, that is, for all $i \neq j$ we have $\mathsf{A}_i \cap \mathsf{A}_j = \emptyset$ and $\mathsf{X}_i \cap \mathsf{X}_j = \emptyset$. We compose the open abstract models M^i using interactions. A set of *interactions* γ is a subset of ports such that any interaction $a \in \gamma$ contains at most one (internal) action of each component M^i, that is, $a = \{\ a_i \mid i \in I\ \}$ where $a_i \in \mathsf{A}_i^{\mathsf{int}}$ and $I \subseteq \{\ 1, 2, \ldots, n\ \}$.

Definition 29 (composition of open abstract models) *For a set of open abstract models $M^i = (\mathsf{A}_i, \mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i)$, $1 \leq i \leq n$ and its set of interactions γ, we define the* composition *of the open abstract models M^i as the open abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow_\gamma)$ over the set of actions $\mathsf{A} = \mathsf{A}^{\mathsf{in}} \cup \mathsf{A}^{\mathsf{out}} \cup \gamma$ as follows:*

- *A is partitioned into inputs A^{in}, outputs $\mathsf{A}^{\mathsf{out}}$, and internal actions which are interactions γ,*
- *$\mathsf{Q} = \mathsf{Q}_1 \times \mathsf{Q}_2 \times \ldots \times \mathsf{Q}_n$,*
- *$\mathsf{X} = \mathsf{X}_1 \cup \mathsf{X}_2 \cup \ldots \cup \mathsf{X}_n$,*
- *for an interaction $a = \{\ a_i \mid i \in I\ \} \in \gamma$ we have $q \xrightarrow{a,g,r}_\gamma q'$ in M with $q = (q_1, q_2, \ldots, q_n)$ and $q' = (q'_1, q'_2, \ldots, q'_n)$ if and only if $g = \bigwedge_{i \in I} g_i$, $r = \bigcup_{i \in I} r_i$, $q_i \xrightarrow{a_i, g_i, r_i} q'_i$ in M^i for all $i \in I$, and $q'_i = q_i$ for all $i \notin I$,*
- *for an input or an output $a \in \mathsf{A}^{\mathsf{in}} \cup \mathsf{A}^{\mathsf{out}}$ we have $q \xrightarrow{a,g,r} q'$ in M with $q = (q_1, q_2, \ldots, q_n)$ and $q' = (q'_1, q'_2, \ldots, q'_n)$ if and only if $q_i \xrightarrow{a,g,r} q'_i$ in M^i and $q'_j = q_j$ for all $j \neq i$.*

The composition $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow_\gamma)$ of open abstract models M^i, $1 \leq i \leq n$, corresponds to a general notion of product for the timed automata M^i. It can execute two types of actions: interactions $a \in \gamma$ which are user-defined synchronizations (i.e. rendez-vous) between internal actions, and inputs/outputs $a \in \mathsf{A}_i^{\mathsf{in}} \cup \mathsf{A}_i^{\mathsf{out}}$ of M^i.

Example 16 *Figure 6.10 describes the interaction model of three open abstract models encapsulated in components **Sensor**, **Actuator** and **Computing**. The **Sensor** waits for the occurrence of the input in from the environment such as $x \geq D$. The **Actuator** executes an output out such as $y \leq D$. The two models interact with the computing unit **Computing** following this interaction model:*

$\gamma = \{\ inform1.compute\ ;\ inform2.send\ ;\ end.finish.ready\ \}$.

First, the sensor informs the computing unit about the arrival of the input (interaction $\{inform1.compute\}$), then the computing unit performs an internal computation such as $z \leq D$. When it finishes the computation, it informs the actuator (interaction $\{inform2.send\}$). finally, after the execution of the output, the actuator informs the computing unit and the sensor that it has finished (interaction $\{end.finish.ready\}$).

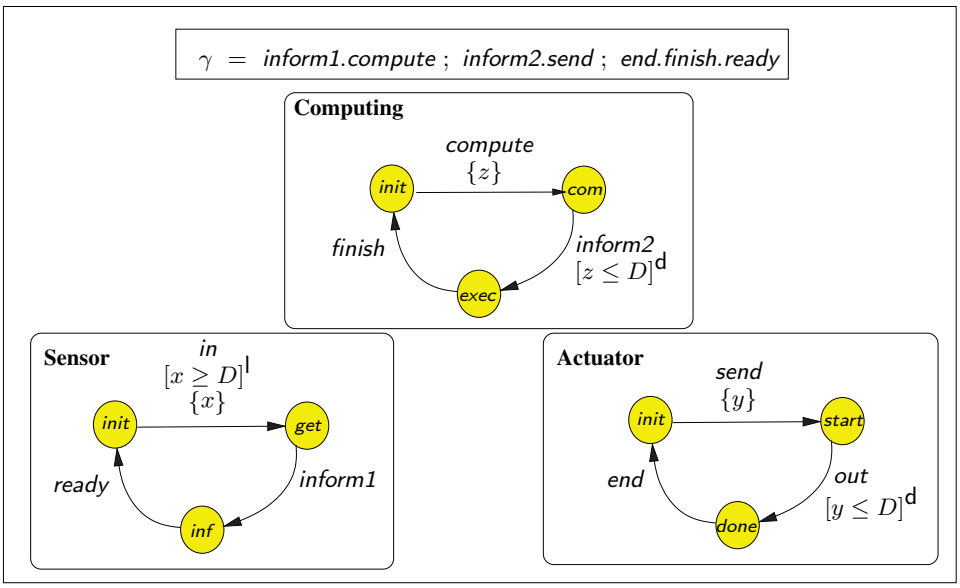


Figure 6.10: Interacting sensor and actuator open abstract models.

The resulting composition of the three open abstract models is given as follows (see Figure 6.11) :

$SA = (\mathsf{A}_{sa}, \{init', get', com', start', end'\}, \{x, y, z\}, \longrightarrow)$, composed of the set of actions $\mathsf{A}_{sa} = \mathsf{A}^{\mathsf{in}} \cup \mathsf{A}^{\mathsf{out}} \cup \gamma$, where :

- A^{in} is the set of input actions $\mathsf{A}^{\mathsf{in}} = \{\ in\ \}$.
- A^{in} is the set of output actions $\mathsf{A}^{\mathsf{in}} = \{\ out\ \}$.
- γ is the set of interactions $\gamma = \{\ \gamma_1\ ,\ \gamma_2,\ \gamma_3\ \}$, where $\gamma_1 = inform1.compute$, $\gamma_2 = inform2.send$ and $\gamma_3 = end.finish.ready$.

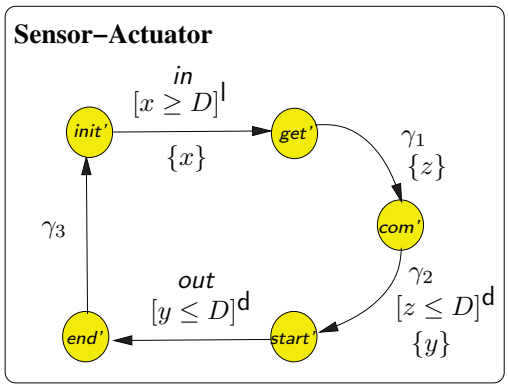


Figure 6.11: Composition of models from the sensor and actuator example.

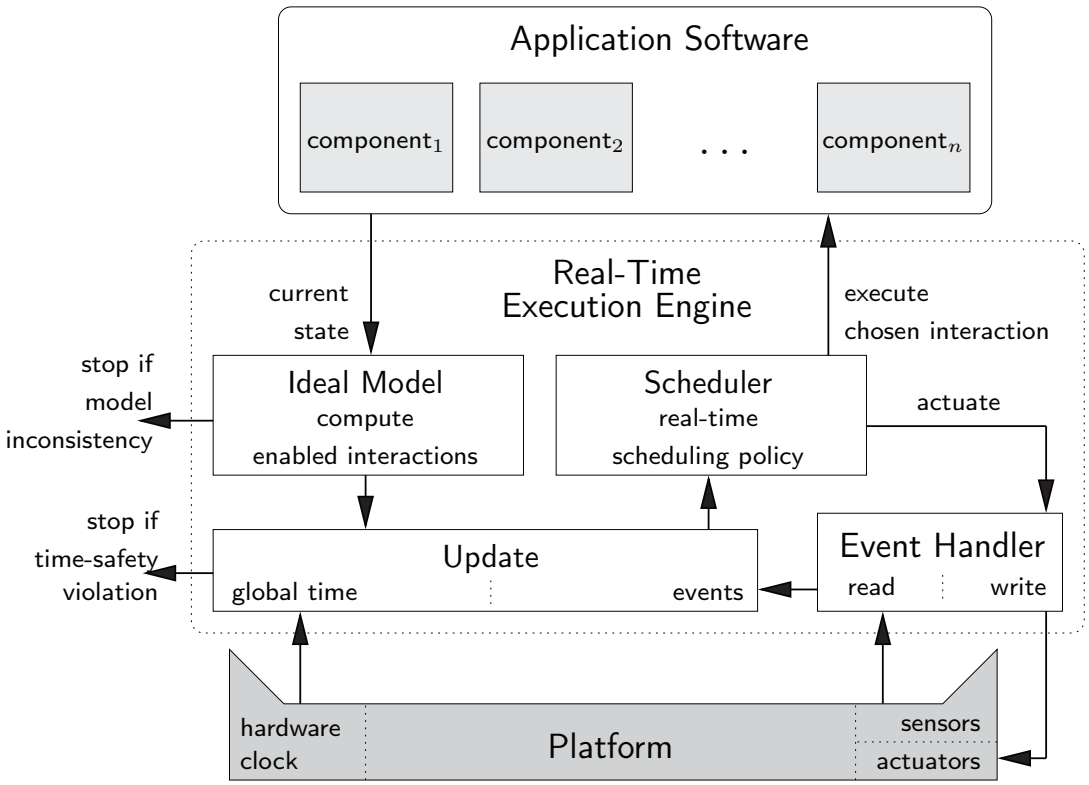


Figure 6.12: Architecture of the open real-time Engine.

Definition 30 (composition of open physical models) *Consider open abstract models M^i, $1 \leq i \leq n$, and corresponding open physical models $M^i_{\varphi_i} = (\mathsf{A}_i, \mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i, \varphi_i)$, with disjoint sets of actions and clocks.*

Given a set of interactions γ, and an associative and commutative operator $\oplus : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$, the composition *of open physical models $M^i_{\varphi_i}$ is the open physical model M_φ corresponding to the abstract model M which is the composition of M^i, $1 \leq i \leq n$, with the execution time function $\varphi : \mathsf{A} \to \mathbb{T}$ such that $\varphi(a) = \bigoplus_{i \in I} \varphi_i(a_i)$ for interactions $a = \{\ a_i \mid i \in I\ \} \in \gamma$, $a_i \in \mathsf{A}_i^{\mathsf{int}}$, and $\varphi(a) = \varphi_i(a)$ for $a \in \mathsf{A}_i^{\mathsf{in}} \cup \mathsf{A}_i^{\mathsf{out}}$.*

The definition is parameterized by an operator $\oplus$ used to compute the execution time $\varphi(a)$ of an interaction a from execution times $\varphi(a_i)$ of the actions a_i involved in a. The choice of this operator depends on the considered execution platform and in particular how executions of components (open abstract models) are parallelized. For instance, for a single processor platform (i.e. sequential execution of actions), $\oplus$ is addition. If all components can be executed in parallel, $\oplus$ is **max**.

3.2 Execution Algorithm

Our method relies on an open real-time Engine implementing the semantics of open physical models. It executes actions by taking into account their timing constraints. The proposed open real-time Engine does not need an a priori knowledge of execution times and the occurrence time of the inputs (see Figure 6.12). It ensures the real-time execution of an application by taking into account actual execution times on the target platform and the

actual occurrence of the inputs. It stops the application when time-safety is violated, that is, when a deadline or an input is missed as explained in Section 2.2. It also implements one of the policies (error or ignore) for enforcing input-enabledness, also presented in Section 2.2.

Interactions with the environment are not directly performed by the application but are managed by the Event Handler which is the part of the open real-time Engine that realizes the interface between inputs/outputs of the application and the environment. We assume that the Event Handler can detect and store all inputs and their actual occurrence time in a FIFO. There are different ways for implementing this mechanism: interruptions, signals, threads executing in parallel with the application, etc.

Timing Features

To check enabledness of actions (i.e. internal actions, inputs or outputs), the open real-time Engine expresses the timing constraints in terms of a single clock t measuring the absolute time elapsed (See Chapter 5 Section 1). Thus, the open real-time Engine considers states of the form (q, w, t) where q is a control location of M, $w : \mathsf{X} \rightarrow \mathbb{T}$ is the reset time of the clocks with the respect to clock t, and $t \in \mathbb{T}$ is the (absolute) current time. We have explained that we rewrite each atomic expression $l \leq x \leq u$ involved in a timing constraint by using the global clock t and reset times w, that is, $l \leq x \leq u \ \equiv \ l+w(x) \leq t \leq u+w(x)$. For a given state $(s, t) = (q, w, t)$ of M, we also associate to the action a its next activation time $\mathsf{next}_t(a)$ and its next deadline $\mathsf{deadline}_t(a)$. We now also associate to the interaction a, its last activation time $\mathsf{last}_t(a)$. Values $\mathsf{next}_t(a)$, $\mathsf{last}_t(a)$ and $\mathsf{deadline}_t(a)$ are computed from the timing constraint g of a, $g = [l \leq t \leq u]^\tau$, as follows:

$$\mathsf{next}_t(a) = \begin{cases} \mathbf{max} \{ t, l \} \text{ if } l \leq u \text{ and } t \leq u \\ +\infty \text{ otherwise,} \end{cases}$$

$$\mathsf{last}_t(a) = \begin{cases} \mathbf{max} \{ u \} \text{ if } l \leq u \text{ and } t \leq u \\ -\infty \text{ otherwise,} \end{cases}$$

$$\mathsf{deadline}_t(a) = \begin{cases} u \text{ if } l \leq u \ \wedge \ t \leq u \ \wedge \ \tau = \mathsf{d} \\ l \text{ if } l \leq u \ \wedge \ t < l \ \wedge \ \tau = \mathsf{e} \\ t \text{ if } l \leq u \ \wedge \ t \in [l, u] \ \wedge \ \tau = \mathsf{e} \\ +\infty \text{ otherwise.} \end{cases}$$

We extend these definitions to non-empty subsets of actions $\mathcal{A} \subseteq \mathsf{A}$: $\mathsf{next}_t(\mathcal{A}) = \mathbf{min}_{a \in \mathcal{A}} \mathsf{next}_t(a)$, $\mathsf{last}_t(\mathcal{A}) = \mathbf{max}_{a \in \mathcal{A}} \mathsf{last}_t(a)$ and $\mathsf{deadline}_t(\mathcal{A}) = \mathbf{min}_{a \in \mathcal{A}} \mathsf{deadline}_t(a)$. We also define defaults values $\mathsf{next}_t(\emptyset) = \mathsf{deadline}_t(\emptyset) = +\infty$ and $\mathsf{last}_t(\emptyset) = -\infty$.

We assume that the actual time t_r is provided by the real-time clock of the platform. It is used to update the absolute time t used by the Real-Time Engine.

Algorithm

The proposed Engine (Algorithm 6) corresponds to the ignore policy presented in Section 2.1, we have also an implementation for the error policy. It also checks for time-safety violations which correspond to inputs or deadlines missed. Notice that the proposed algorithm directly implements the semantics of physical models, that is, it corresponds to a sequential execution in which inputs can only occur during waiting periods. It can be slightly modified to safely handle inputs occurring during actions execution as explained below. Moreover, the Real-Time Engine can be modified to allow parallel execution of the components and the interactions as explained in [13].

Algorithm 6 Real-Time Engine

Require:

1: $(s,t) = (q,w,t) \leftarrow (q_0, 0, 0)$ *// initialize system state*
2: **loop**
3: $(\mathcal{I}_s, \mathcal{O}_s, \gamma_s) \leftarrow \text{Enabled}(s)$ *// enabled actions*
4: **if** $\mathsf{next}_t(\mathcal{I}_s \cup \mathcal{O}_s \cup \gamma_s) = +\infty$ **then**
5: **exit**(`DEADLOCK`) *// nothing enabled from* (s,t)
6: **end if**
7: $D \leftarrow \mathsf{deadline}_t(\mathcal{I}_s \cup \mathcal{O}_s \cup \gamma_s)$ *// next deadline*
8: $t \leftarrow t_r$ *// update model time w.r.t. actual time*
9: **if** $t > D$ **then**
10: **exit**(`DEADLINE_MISSED`) *// missed deadline*
11: **end if**
12: **if** $\text{Inputs}(\mathcal{I}_s) \neq \emptyset$ **then**
13: **exit**(`INPUT_MISSED`) *// missed input*
14: **else**
15: **if** $\mathsf{next}_t(\mathcal{O}_s \cup \gamma_s) < +\infty$ **then**
16: $a \leftarrow \text{RealTimeScheduler}(\mathcal{O}_s \cup \gamma_s)$ *// plan a*
17: $t \leftarrow \mathsf{next}_t(a)$ *// at absolute time t*
18: **else**
19: $a \leftarrow \emptyset$ *// no enabled output or internal action*
20: $t \leftarrow \textbf{min}\ \{\ \mathsf{last}_t(\mathcal{I}_s), D\ \}$ *// after t, deadlock*
21: **end if**
22: **wait** $\text{Inputs}(\mathcal{I}_s) \neq \emptyset$ **or** $t_r \geq t$
23: **if** $\text{Inputs}(\mathcal{I}_s) \neq \emptyset$ **then**
24: $(a,t) \leftarrow \text{PurgeFirst}(\mathcal{I}_s)$ *// retrieved input*
25: **end if**
26: **end if**
27: **if** $a \neq \emptyset$ **then**
28: $\text{Execute}(a)$ *// execute a at global time t*
29: **else**
30: **exit**(`DEADLOCK`) *// nothing enabled from* (s,t)
31: **end if**
32: **end loop**

Given a state (s, t), the Real-Time Engine computes the next action to execute using the followings steps.

Compute enabled actions. It first computes enabled actions from $s = (q, w)$ (line 2 of Algorithm 6). It correspond to the state of the system after components execution. We distinguish between enabled inputs $\mathcal{I}_s$, enabled outputs $\mathcal{O}_s$ and enabled internal actions γ_s using the primitive Enabled(s): $(\mathcal{I}_s, \mathcal{O}_s, \gamma_s) \leftarrow \text{Enabled}(s)$.

Update of Time. It computes the next deadline D with respect to the current state (s, t) (line 7). The absolute time t is then updated with respect to the actual time t_r in order to take into account the previous action execution time (line 8).

Check for time-safety. It stops the execution if the deadline D is missed (line 10). It also stops the execution if an enabled input occurred during the previous action execution (line 13). We use the primitive Inputs() of the Event Handler which returns the list of inputs (and their corresponding timing constraints given as parameter), resulting from the filtering of the current content of its FIFO by the enabled inputs $\mathcal{I}_s$.

An other policy would be to safely handle these inputs a posteriori by replacing line 13 by:

$$(a, t) \leftarrow \text{PurgeFirst}(\mathcal{I}_s).$$

The primitive PurgeFirst() of the Event Handler returns the oldest occurrence of an enabled input and removes it from the FIFO (first enabled input in the FIFO) . This input is treated even after its occurrence time but its behavior could be treated exactly at this time if there are enough spare time.

Schedule an output or an internal action. It chooses, if it exists, an output or an internal action a according to a real-time scheduling policy and plan to execute a at its next activation time $\mathsf{next}_t(a)$ (lines 16 and 17). Otherwise $a = \emptyset$ (line 19). The system can only progress if an input is received before the next deadline D and before the last instant of activation of the inputs $\mathsf{next}_t(\mathcal{I}_s)$ (line 20).

Wait. If $a \neq \emptyset$, it waits for the instant of execution of a, otherwise it waits for an input until there is a deadlock (line 22). If an enabled input occurs while waiting, the Event Handler notifies the Engine to purchase the execution. The input is then immediately selected and executed instead of the planed action (line 24).

Execute. It executes the chosen action a which corresponds to either an output or an internal action planed by the real-time scheduler, or an enabled input (line 28). The execution of an output also includes a notification to the Event Handler which executes the corresponding actuate actions as shown in Figure 6.12. Notice that inputs occurring during the execution of a are treated at the next iteration according to the semantics of M_φ.

4 Implementation Method for the Real-Time BIP Framework

We implemented the proposed method for the component-based framework BIP used for building systems consisting of heterogeneous components. We extended the real-time BIP framework, presented in Chapter 5, in order to handle both real-time features and the communications between the application model and its environment. We make extensions in the BIP language to allow the expression of inputs and outputs in the model which corresponds to the interface of communication with the environment. Thus, we changed the compilation process in order to make a mapping between the inputs and outputs of the application model and the physical events from sensors and actuators. An Event Handler is added in the real-time BIP Engine to manage the arrival of inputs from the environment and the execution of actuate actions assigned to outputs.

4.1 The BIP Language Extensions

We introduce in the real-time BIP language special tags on ports declaration in atomic components. These tags enable the openness of components to their physical environment. An atomic component has only local data, and its interface is given by a set of communication ports. In addition to ports which correspond to internal actions, we introduce input and output actions using tags. An "input" tag on a port turns it as an input port and an "output" tag on a port turns it as an output port. Input and Output ports are used to specify direct interaction with the environment, they cannot be involved in interactions. Output ports are denoted by empty triangles are freely triggered by the system. In contrast, input ports are denoted by empty circles and are triggered by the environment.

Figure 6.13 gives the input and output ports representation and its equivalent BIP model without the utilization of input and output tags. In case (a) the application model defines an input port *in* for which the application is waiting for an input from the environment. This is equivalent to the passive mode of communication where the sensor triggers the execution of the application when an input is enabled. In case (b) the application model defines an output port *out* for which the application sends an output to the environment. This is equivalent to the active mode of communication where the application triggers the execution of the actuator when an output is enabled.

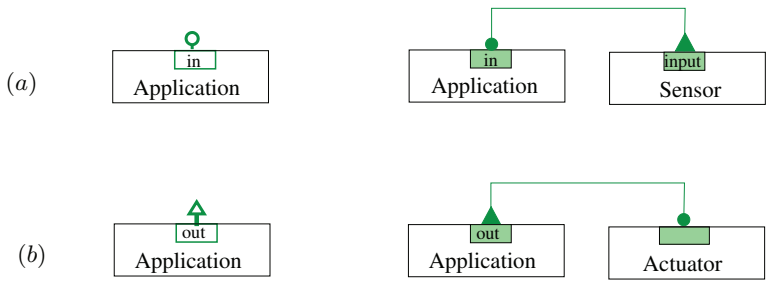


Figure 6.13: Input and Output ports representation.

An atomic component definition is given as follows:

Definition 31 (Atomic component) *An* atomic component *represents behavior* B *as a transition system, extended with variables and functions, represented by* $(\mathsf{V}, \mathsf{P}, \mathsf{X}, \mathsf{Q}, \longrightarrow)$*, where:*

- Q *is a set of* control states $\mathsf{Q} = \{Q_1 ... Q_n\}$*, which are places at which the components await for synchronization.*
- P *is a set of* communication ports $P = \{p_1 ... p_n\}$*, partitioned into normal ports* $\mathsf{P}^{\mathsf{int}}$ *(i.e. communicating with other components) and environment ports of type input* P^{in} *and output* $\mathsf{P}^{\mathsf{out}}$ *(i.e. communicating with the environment).*
- V *is a set of* variables *used to store (local) data. Variables may be associated to ports.*
- X *is a finite set of* clocks*,*
- $\longrightarrow$ *is a set of transitions modeling computation steps of components. Each transition is a tuple of the form* $(q_1, p, g_p, gt_p, f_p, q_2)$*, representing a step from control state* q_1 *to control state* q_2*, denoted as* $q_1 \overset{p,g_p,gtp,f_p}{\longrightarrow} q_2$*, where* g_p *is a boolean condition on* V*,* f_p *is a computation step consisting of data transformations and* gt_p *is a timing constraint over* X*.*

We notice that variables may be associated to input and output ports to enable the exchange of data between the application and its environment.

An abstract syntax of port declarations is given in Figure 6.14. We introduce the keywords **input** and **output** to create instances of input and output ports. They are also referenced by their port type and a port name. The port name must be unique and specific to the input or output port. Indeed, input and output ports are mapped to their associated physical events according to their names, or (and) to the components they belong. Figure 6.13 gives the open abstract model representation (left) and declaration (right) of Example 6.2) using the new BIP language (see Figure 6.15).

port-definition ::= *normal-port* | *environment-port*

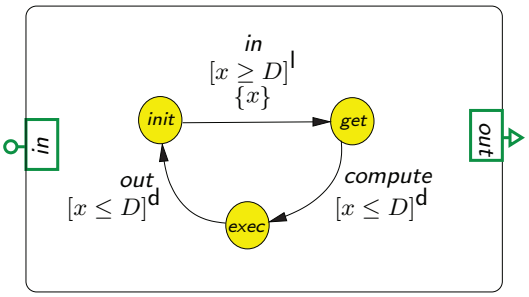
normal-port ::= [**export**] **port** *port-type-reference port-name* [*variables*]
port-type-reference ::= [*library-name* **.**]*port-type-name*

environment-port ::= [**export**] *envport-type* **port**
envport-type-reference port-name [*variables*]

envport-type ::= **input** | **output**

envport-type-reference ::= *port-type-name*

Figure 6.14: *Ports declaration syntax*




```
atomic type Open-Atom
    input port intPort in
    output port intPort out
    port intPort intPort compute

    clock x unit millisecond

    place init
    place get
    place com
    place exec

    initial to init

    on in from init to get
    when x in [D,-] lazy
    reset x
    on compute from get to exec
    when x in [-,D] delayable
    on out from exec to init
    when x in [-,D] delayable
end
```

Figure 6.15: An example of an open atomic component declaration.

4.2 Mapping Inputs and Outputs with Physical Events

The BIP language is completed by a compiler that generates C++ code from BIP models. The generated code is intended to be combined with a dedicated Engine implementing the semantics of BIP. We introduced in the language environment ports (inputs and outputs). The Event Handler is the part of the open real-time Engine that is responsible for the communication with the environment. It receives events from the environment and notifies the model of the occurrence of the corresponding inputs. It also receives outputs from the model and executes their corresponding actuate actions. A mapping between input and output ports specified in the model and their corresponding events managed by the Event handler is necessary. In this purpose, we introduce the notion of Event representing an input or an output. It is the concrete representation of the port for which we can associate an action. We introduce an event handler that manages a list of Drivers. Each Driver handles the communication with the environment (actuators, sensors, software code). A driver manages a list of events. An *event* is a structure associated to an input or output. It is used to store data produced by the environment. the event handler makes the mapping between the events handled by Drivers and ports involved in the computation of interactions.

4.3 Use Case

We conducted experiments on the marXbot [28], a miniature mobile robot embedding a multitude of sensors and actuators. The hardware architecture of the robot is composed

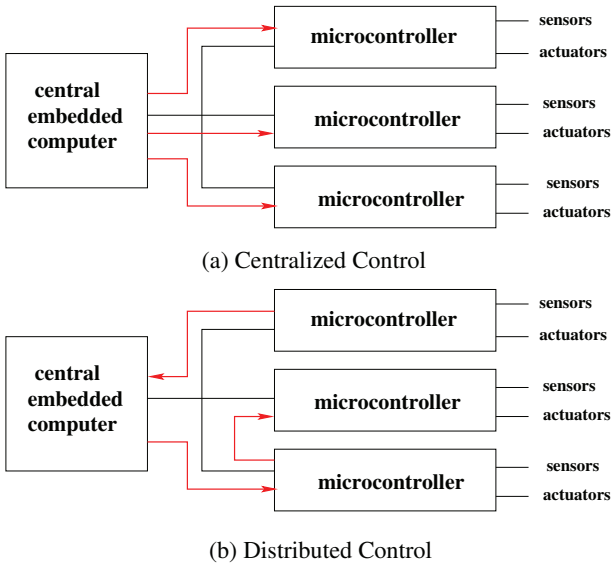


Figure 6.16: Two hardware and control architecture paradigms.

of a centralized CPU running a Linux platform and several distributed microcontrollers managing the sensors and actuators (See Figure 6.16). Compared to a centralized control approach (a), the management of sensors and actuators via the network of distributed microcontrollers (b) limits the overloading and the latency concerning the response time to basic external stimuli. It also reduces the robot's limitations in terms of speed of operation, compared to centralized approach, since each microcontroller manages its nearest device. The control architecture would be still partially centralized for complex applications that need more CPU utilization and memory allocations. This is possible thanks to the event-based communication at the microcontroller level with the centrilized CPU(b) instead of polling hardware devices in(a).

Description of the robot architecture

The marXbot robot is composed of the following modules (See Figure 6.17):

- A base module provides rough-terrain mobility thanks to wheels . It also contains the basic bricks for obstacle avoidance and odometry, as it embeds proximity sensors, a 3-axis gyroscope, and a 3D accelerometer.
- An attachment module provides self-assembling capabilities with peer marXbots. This module allows the docking of the other robots and can feel the force they apply.
- A range and bearing module allows the robot to compute a rough estimate of the direction and the distance of the neighboring robots.
- A distance scanner module allows the robot to build a 2D map of its environment.
- Finally a computer module provides a complete Linux-based operating system to the robot. This module thus enables advanced cognitive capabilities. The main computer also drives two cameras, one looking front and one oriented towards an omnidirectional hyperbolic mirror.

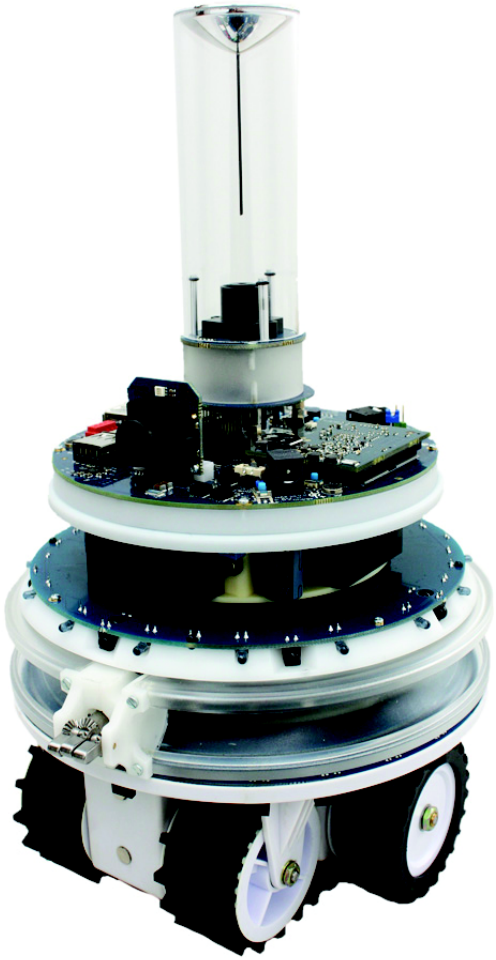


Figure 6.17: The marXbot robot.

A controller running in the central computer reads sensors, processes the data, and sets actuators at regular intervals, and these must transit through a communication bus. A intermediate solution would be to make the centralized computer communicate with the network of microcontrollers that manage the complexity of communication with the sensors and devices. A modular event-based distributed architecture, called *ASEBA* [71], has been developped in order to program the network of microcontrollers. All the microcontrollers send events and react to incoming events through a *CAN* bus in which communication is asynchronous. Indeed, the information in the sensors is dispatched by their microcontrollers to the rest of the robot only if and when the information is relevant to the application. In *ASEBA*, the description of the robot behavior and the events emission and reception policy is described in a scripting language. The language is a simple imperative programming language with arrays of 16-bits signed integers as the only data type. *ASEBA* provides also an IDE that provides each microcontroller a tab with the list of variables for the real-time edition of the values of the sensors, the actuators and the user-defined variables, a script editor and debug controls. The IDE compiles scripts into bytecodes and loads them to the microcontrollers, through the communication bus, to be executed in a lightweight Virtual

Machine.

Nevertheless, *ASEBA* does not enable the writing of complex applications and memory allocations are limited in the microcontrollers. The *ASEBA* scripting language cannot provide the use of 32-bits integers for example or the definition of complex computations. The central computer that has a Linux-based operating system, communicates with the microcontrollers via a software switch, that extends the communication bus to local TCP/IP connexions. For those reasons, we use the open Real-time BIP framework in order to build applications running in the centralized computer. In one hand, the BIP language provides the high-level language for the description of inputs and outputs and, in an other hand, the Event-Handler uses event-driven mechanisms compatible with the *ASEBA* approach. The Mapping between the model's inputs and outputs to the events provided by *ASEBA* becomes staightforward.

Modelling applications using BIP

We consider a simple experiment setup for an obstacle avoidance scenario. The robot is initially moving straight with an initial speed and stops when the front proximity sensor detects an obstacle at a distance less or equal to some value. Stopping the robot is equivalent to setting the speed of the robot's left and right wheels to zero. In our experiments, we use only two modules. In the first module, we use the wheels and obstacle avoidance microcontrollers. The second module contains the computer module running on unix that executes the application scenario. We notice that the scenario is very simple and could have been implemented only in the microcontrollers. This wouldn't be the case for more complex applications, such as building a map of the robot's trajectory or computing the best path to arrive in a target place and avoiding obstacles. The considered obstacle avoidance application is composed of three main components (see Figure 6.18).

- **LeftWheel** sets the speed of the robot's left wheel. It is initially in state *move* until an obstacle is detected, port *obstacle* is then triggered by the Sensor component. It computes the speed to stop the left wheel and sends an output *Lmove* to the microcontroller responsible for the corresponding actuator.
- **RightWeel** sets the speed of the robot's right wheel. It behaves exactly as the LeftWheel and sends an output *Rmove* to the microcontroller responsible for setting the speed of the robot's right wheel.
- **Sensor** detects obstacles. It is initially at state *clear* and waits for the enabledness of port input *detectObst* by the microcontroller that detects the obstacles. It is the component responsible for triggering the other components when an obstacle is detected.

The second step consists of mapping the input and output ports to the events produced by *ASEBA*. We have three ports to be mapped, the input port *detectObst* and the two output ports *Lmove* and *Rmove*.

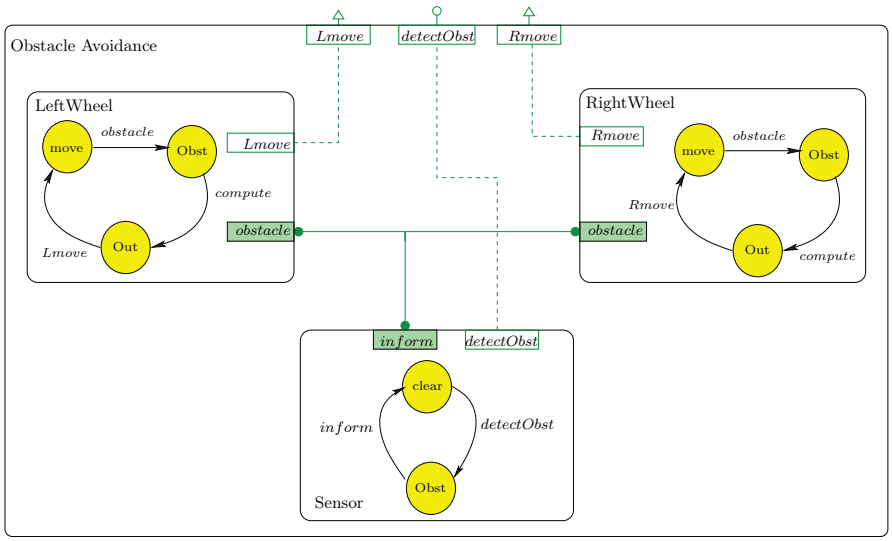


Figure 6.18: The obstacle avoidance model.

5 Conclusion

This chapter provides a rigorous design and implementation method for open real-time systems. The method extends the concepts presented in the previous chapters by not only considering closed systems but also taking into account communications with an external environment. An open real-time system interacts with a physical environment while meeting timing constraints. Its behavior depends not only on its current state, like for closed systems, but also on the behavior of its environment.

We present a model-based method that, based on an initial open abstract model, builds its corresponding physical open abstract model and verifies its properties. Open abstract models represent the timing behavior of the application without considering any execution platform. They model the interactions with the environment using input and output actions, that is, open abstract models are modeled using Input/Output automata. Internal actions correspond to internal computations, outputs are triggered by the application, whereas inputs are triggered by the environment. The open physical model represents the implementation of the open abstract model on a execution environment. It represents the behavior of the open abstract model running on a given platform, that is, it takes into account execution times. It is obtained from the open abstract model by assigning execution times to actions and by taking into account the actual arrival times of inputs. An open physical model is correct if all its execution traces are execution traces of the open abstract model. Otherwise, it is either a deadline-miss or an input-miss problem and the open physical model is thus not time-safe. We also define an execution Engine that faithfully implements physical models on a target platform. We consider that the application software is a set of interacting components such that each component is represented by an open abstract model. The execution Engine detects time-safety violations when the implementation does not correspond to the open abstract model specification.

The method is correct-by-construction since the proposed engine directly implements the operational semantics of the open physical model. It leads, under some robustness as-

sumptions for WCET times, to a correct implementation. We implemented the method for the component-based BIP framework. We first extended the BIP language by adding input or output tags on ports and we added an Event handler to take into account the communications with the physical environment. We provided an example of an application model for the marXbot, a miniature mobile robot embedding a multitude of sensors and actuators, using the framework BIP. We show a possible connexion between the robot's centralized computer and the modular event-based distributed IDE, called $ASEBA$, that has been developed in order to program the network of sensors and actuators microcontrollers. This work is still in construction.

In Chapter 7, we present results on the design and implementation of more complex autonomous systems. The current BIP framework has been successfully used for the design and implementation of the rover Dala for both the functional and decisional levels. We show the benefits obtained in terms of CPU utilization and amelioration in the latency of reaction.

Part III

Use Cases : Autonomous Systems Design and Implementation

Chapter 7

Building Robot Software Modules

For the large scale deployment of robots in places such as homes, shopping centers and hospitals, where there is close and regular interaction with humans, robot software integrators and developers may soon need to provide guarantees and formal proofs to certification bodies that their robots are safe, dependable, and behave correctly. This also applies to robots such as extraterrestrial rovers, used in expensive and distant missions, which need to avoid equipment damage and mission failure. Such guarantees may involve proofs that a rover will not move while it is communicating or even worse, while it is drilling, that the navigation software has no fatal deadlock, or that a service robot will not extend its arm dangerously while navigating or will not open its gripper while holding a breakable object.

In this purpose, a first work has been done to combine a state of the art tool (GenoM) [19], developped in the LAAS laboratory, to build functional modules of robotic systems with the component based framework BIP for implementing embedded real-time systems. The functional modules are in the fuctional level, which is the robot lowest level that includes all the basic built-in action and perception capabilities of the robot. Little attention has been drawn in the past, to the development of these modules whose robustness is paramount to the robustness of the overall platform. Using formal methods for developing modules of the functional level of robots is then a fundamental issue. To this end, a successfull tool has been developed based on the BIP/GenoM component based design approach [17,18] and has been applied on the functional level of a complex exploration rover, the Dala rover. The gains were *(i)* the production of a very fine grained formal computational model of the robot functional level; *(ii)* running the BIP engine on the real robot, which executes and enforces the model at runtime; and *(iii)* checking the model offline for deadlock freedom, as well as other safety properties.

Nevertheless, the timing properties haven't been taken into account in the formalisms. We extend and improve the approach in various directions: *(i)* Modeling timing features with the real-time BIP model and *(ii)* Execution of the functional level with the real-time BIP engine.

This chapter is structured as follows. In Section 1, we discuss about the existing tools to build robots functional level using formal methods. In Section 2, we present the Dala rover architecture and the design method of its modules, using the previous BIP/GenoM component based design approach. In Section 3, we give the experimental results conducted

on the Antenna module of the robot. We present the method to handle more efficiently the timing features of the module and its communications with the environment. Finally, in Section 4, we conclude the chapter.

1 Building Robot Software using Formal Methods

Despite a growing concern to develop safe, robust, and verifiable robotic systems, robot software development remains quite disconnected from the use of formal methods. The most common method to ensure the correctness of a system is testing (see [36] for a survey). Testing techniques have been effective for finding bugs in many industrial applications. Unfortunately, there is, in general, no way for a finite set of test cases to cover all possible scenarios, and therefore, bugs may remain undetected. Hence, in general, testing does not give any guarantees on the correctness of the entire system. Consequently, these approaches are impractical with complex autonomous and embedded systems for even a small fraction of the total operating space.

On the functional side of robotic systems, there are many popular software tools available (e.g., OROCOS [38], CARMEN [72], Player Stage [87], Microsoft Robotics Studio [63], and ROS [77]). In [65, 81], those methods are even compared. Yet, none of these architectural tools and frameworks propose any extension or link with formal methods, and validation or verification tools.

Recently, the LAAS laboratory has proposed the *R2C* [59], a tool used between the functional and decisional levels. The main component of *RIIC* is the *state checker*. This component encodes the constraints of the system, specified in a language named *EXOGEN*. At run-time it continuously checks if new requests are consistent with the current execution state and the model of properties to enforce. Another interesting early approach to prove various formal properties of the functional level of robotic systems is the ORCCAD system [47]. This development environment, based on the Esterel [33] language, provides extensions to specify robot "tasks" and "procedures." However, this approach remains constrained by the synchronous systems paradigm.

More generally, as advocated in [21], an important trend in modern systems engineering is model-based design, which relies on the use of explicit models to describe development activities and their products. It aims at bridging the gap between application software and its implementation by allowing predictability and guidance through analysis of global models of the system under development. The first model-based approaches, such as those based on ADA, synchronous languages [55] and Matlab/Simulink, support very specific notions of components and composition. More recently, modeling languages, such as UML [64] and AADL [48], attempt to be more generic. They support notions of components that are independent from a particular programming language, and put emphasis on system architecture as a means to organize computation, communication, and implementation constraints. Software and system component-based techniques have not yet achieved a satisfactory level of maturity. Systems built by assembling together independently developed and delivered components, often exhibit pathological behavior. Part of the problem is that developers of these systems do not have a precise way of expressing the behavior of components at their interfaces, where inconsistencies may occur. Components may be developed at different times and by different developers with, possibly, different uses in mind. Their different internal assumptions, when exposed to concurrent execution, can give rise to unexpected behavior, e.g., race conditions, and deadlocks.

All these difficulties and weaknesses are amplified in embedded robotic systems design in general. They cannot be overcome, unless we solve the hard fundamental problems concerning the definition of rigorous frameworks for component-based design.

2 Presentation of the DALA Rover

Most complex robotic systems are build following the same architecture, composed of two main levels, a functional level and a decisional level. The functional level is the robot lowest level, which includes all the basic built-in action and perception capabilities of the robot. These processing functions and control loops (e.g. image processing, obstacle avoidance, motion control) are encapsulated into controllable communicating modules. In this section, we first present the overall architecture of the Dala rover, that is, the different modules it contains. Then, we present the initial BIP/GenoM modules functional organization.

2.1 The Robot Architecture

At LAAS, the GenoM tool (part of the LAAS architecture toolbox) is used to develop the different modules consituing the robot. Each module in the functional level of the LAAS architecture is responsible for a particular functionality of the robot. For example, the functional level of our Dala rover is shown in Figure 7.1. This functional level includes two navigation modes.

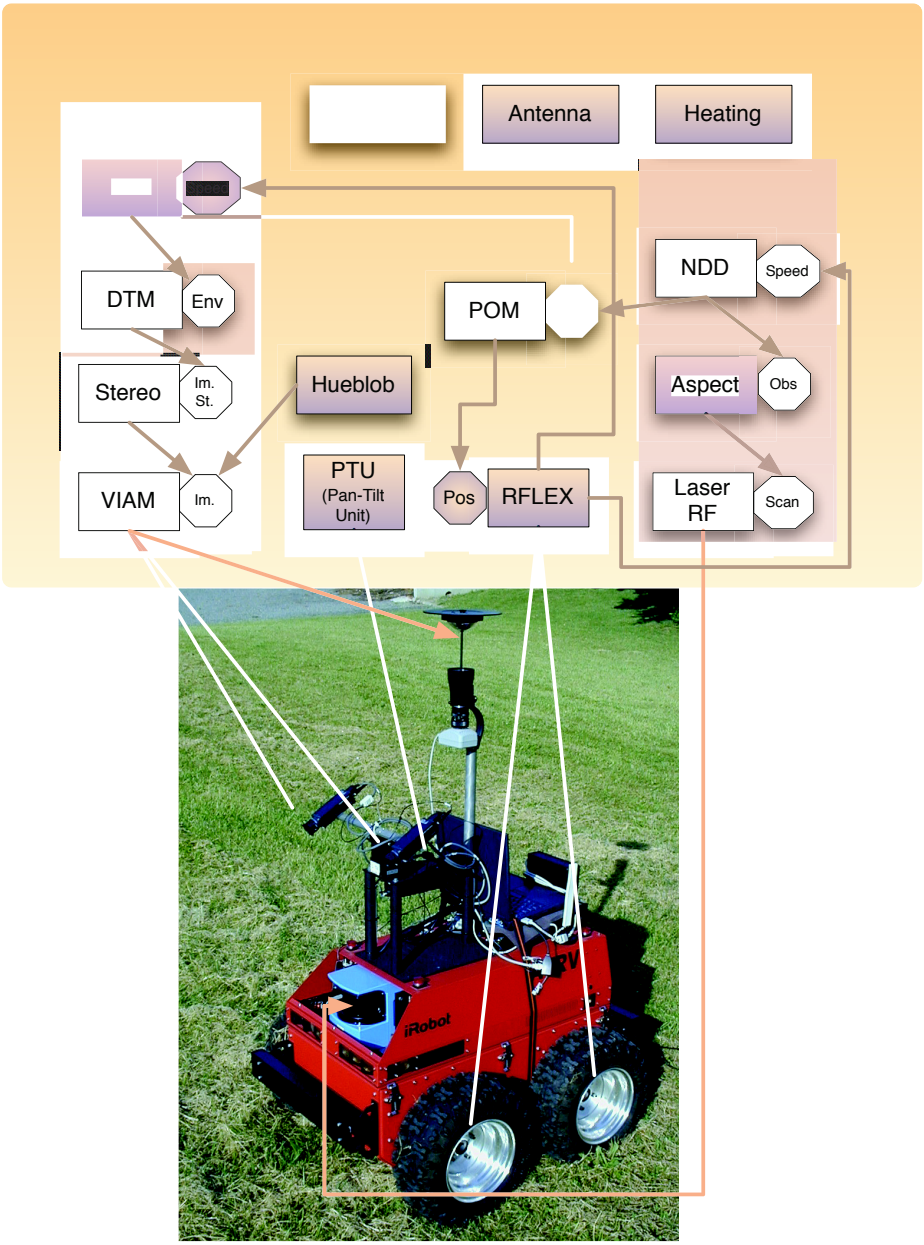
The first navigation mode, for mostly flat terrain, is laser based and contains the following modules:

- the *LaserRF* module acquires the laser scan;
- the *Aspect* module builds a 2D obstacles map,
- the *NDD* module navigates by producing a speed reference, and
- the *RFLEX* module is the robot wheel controller that uses the speed reference.

The second navigation mode, for rough terrain, is vision based.

- the *VIAM* module takes stereo images,
- the *Stereo* module correlates them and
- passes them onto the *DTM* module to build 3D map,
- which used by the trajectory planner of the *P3D* module.

Other modules are deployed to implement opportunistic science (Hueblob), and to emulate communication (Antenna) and power and energy management (Battery). Each module provides a set of services which can be invoked by the decisional level according to tasks that need to be achieved. The open real-time BIP framework has been successfully used for implementing the Antenna module of the functional level of Dala, The method can be generalized with an automatic translation of the GenoM modules into equivalent real-time BIP components.


Antenna
Heating
DTM
Env
Stereo
Im. St.
VIAM
Im.
Hueblob
POM
PTU (Pan-Tilt Unit)
Pos
RFLEX
NDD
Speed
Aspect
Obs
Laser RF
Scan
iRobot

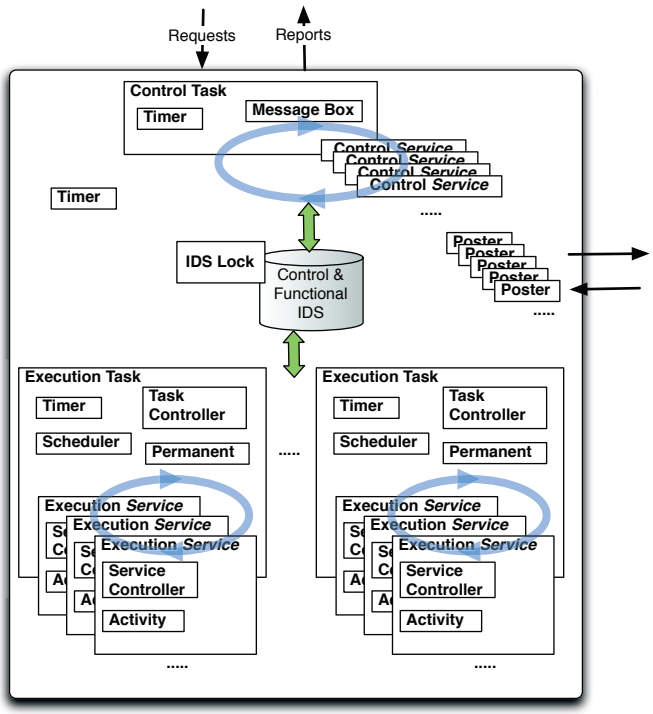


Figure 7.2: A GenoM module functional organization.

- An *execution task* has different scheduling periods and priorities in charge of executing particular user defined services. It triggers periodically services for launching and executing associated activities and upon completion the *services* return a *report* to the caller. In most implementations, an *execution task* is a POSIX thread and each *execution task* is executed one after another (they are in the same thread).

- A *control task*, which among other things, handles *requests* and *reports*, is responsible for setting and returning variable values. These *services* are implemented through the transitions of an automaton that are linked to particular elementary (C/C++) code, called *codels*, which are executed during the transitions.

- A module may also export *posters* containing "shared" data for others (modules or the decisional level) to use.

In the previous version, each active *service* is given a slice of the CPU in sequence and executes one *codel*. Time is taken into account with logical ticks provided by the BIP engine. Thus, real-time properties are modeled using *timers* and BIP automata with transitions executing C "sleep" actions. In general, a global *timer* component is responsible for executing the C "sleep" actions and synchronizes all the timer components of the modules that are used to trigger the execution of periodic tasks. In the following subsection, we give as an example, the Antenna module general structure.

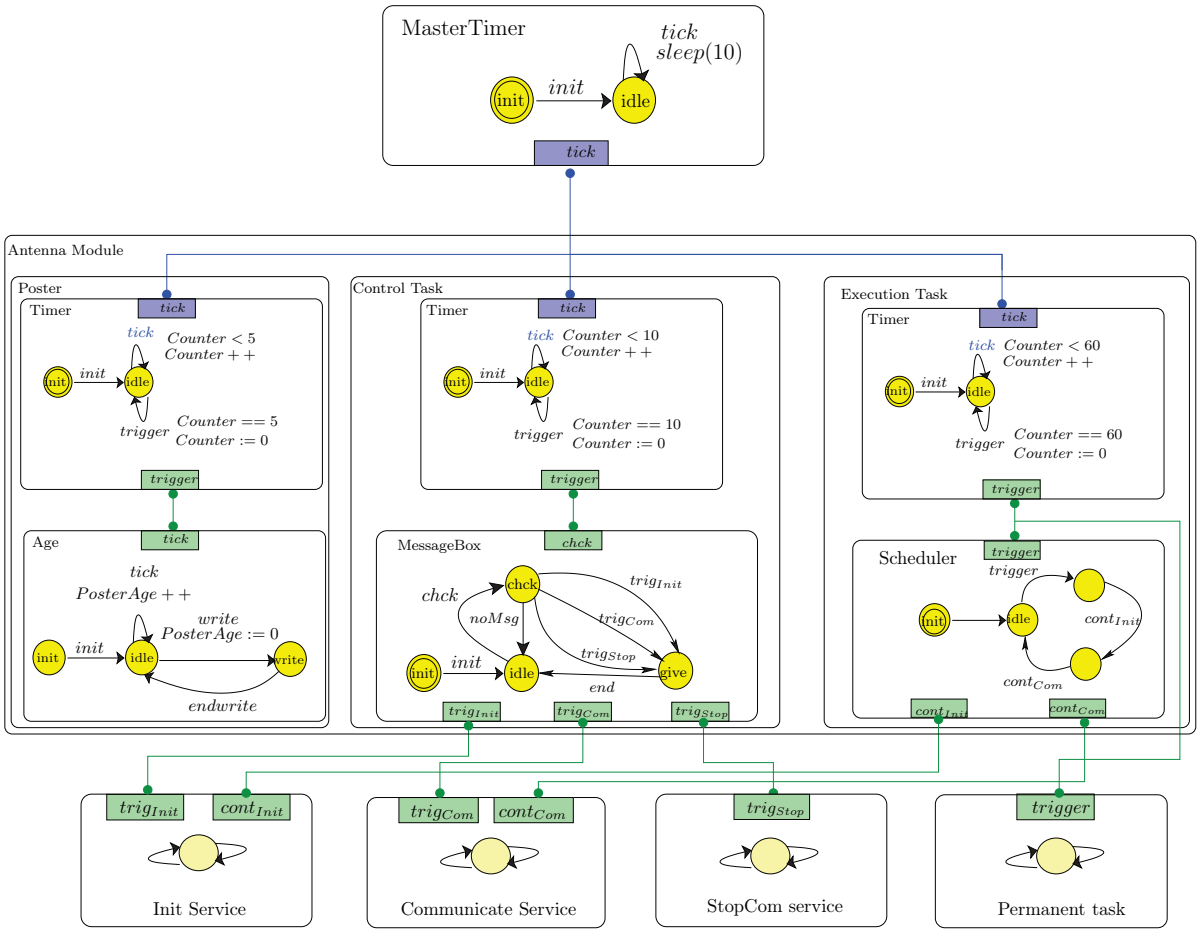


Figure 7.3: Antenna module implementing timing constraints using a Timer.

2.3 The Antenna Module Example

The Antenna module is responsible for the communication with an orbiter, and provides the following services (see Figure 7.3):

Init service initializes the communication with the orbiter. It fixes the time window for the communication between the application and the orbiter, given as parameter.

Communication service starts the communication with the orbiter. It has a parameter defining the duration of the communication.

StopCom service terminates the on-going communication between the application and the orbiter.

In the initial BIP/GenoM approach, time is measured in terms of *ticks*. *Timer* components implementing periodic activations are strongly synchronized with a *MasterTimer* component through tick ports. *MasterTimer* ensures that there are at least 10 ms between two consecutive synchronizations of the components *Timer*. This is achieved by calling sleep primitives of the platform in *MasterTimer* when executing action *tick*. *Timer* components

trigger other components at fixed periods given as parameters in terms of ticks. Periodic execution of *Timer* is enforced by a guard involving an integer variable *Counter* incremented at each *tick* execution.

- Component *Age* measures the freshness of the poster at a period of 5 ticks (50 ms). When the period *counter* == 5 is reached in the timer component, the Age component is triggered in order to increment variable *PosterAge* corresponding to the poster age.
- Component *MessageBox* checks the presence of requests using a period of 10 ticks (100 ms). When the period *counter* == 10 is reached in the timer component, the MessageBox component is triggered in order to read in the memory whether a request is present.
- Component *Scheduler* executes activities based on a period of 60 ticks (600 ms). When the period *counter* == 60 is reached in the timer component, the Scheduler component is triggered in order to launch the execution of the permanent task.

Implementing timing features with tick connectors is clearly not enough when it comes to providing and controlling a real-time model of a complex system. It can eather introduce a deadlock or block the time advance in a part of the system as explained below.

Deadlocks have been found when verifying the consistency of the model with D-Finder[?]. The first deadlock was due to the strong synchronization of *timer* components in the *NDD* module, wich uses the same structure as the antenna module. Indeed, *timer* components are also synchronized with other components to trigger the execution of a task, thus, the strong synchronization of all *timer* components can be blocked if one of the execution tasks cannot perform a tick transition. This is due to the fact that its triggering port never becoming available after the Timer's period is reached. A solution has been adopted, which is not to strongly synchronize all the timer components. The one that is responsible of the blocking will not be involved.

This solution is not very rigorous because time is not progressing in one of the timer components. Even after correcting individual modules with respect to deadlocks, it's not possible to check whether the synchronization between all related modules components are deadlock-free because of the large state space.

3 The Antenna Module Experimental Results

In this section, we explain how we improve the system by using the real-time BIP framework through the Antenna module. We present the two transformation steps. In the first step, we use the real-time BIP framework in order to handle timing constraints more efficiently. We remove all the timer components and extend the triggered components with timed automata, that is we introduce clocks and timing constraints. In the second step, we use the open real-time BIP framework in order to remove the polling mechanism for checking the presence of requests. We directly introduce an input and output ports in order to communicate with the decisional level.

3.1 Introducing Clocks and Timing Constraints

The first step is based on the real-time Engine proposed in Chapter 5. It directly expresses timing constraints in components using clocks, which avoids the use of *MasterTimer* and

	real(s)	user(s)	sys(s)	CPU utilization(%)
1^{st} implementation (ticks)	22.6	0.2	0.1	1.32
2^{nd} implementation (real-time Engine)	22.6	0.05	0.08	0.45

Table 7.1: CPU utilization for Antenna.

Timer components. Figure 7.4 is the resulting representation of the Antenna module. In this purpose, we introduced:

- Clock *Ageclk* in *Age* component measuring the freshness of the poster. That is, whenever we write a new data, clock *Ageclk* is reset to zero. A timing constraint over clock *Ageclk* can prevent the reading of a poster if the data is older than or equal to a giving value. In our case, the poster is read for $Ageclk < 100ms$.
- Clock *Mclk* in *MessageBox* component is used to enforce a period 100 ms for checking the presence of a request. The corresponding timing constraint over clock *Mclk* is $Mclk = 100$.
- Clock *Pclk* in *Scheduler* component is used to enforce a period of 600 ms to launch a task. The corresponding timing constraint over clock *Pclk* is $Pclk = 600$.

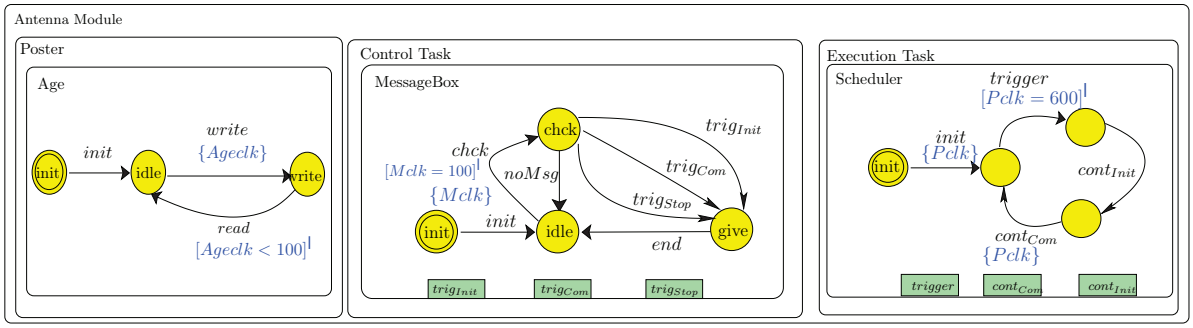


Figure 7.4: Antenna module using the Real-Time Execution Engine.

We compared the execution of the first implementation of Antenna (i.e. using the multi-thread Engine), and the second implementation (i.e. using the real-time Engine). CPU utilization is almost 3 times higher for the multi-thread Engine using ticks compared to the real-time Engine using clocks (see Table 7.1). The main reason is that the multi-thread Engine executes *tick* every 10 ms, even at states for which the application is waiting for enabledness of a guard or the arrival of a message (see Figure 7.9 (a)), whereas the real-time Engine is actually sleeping (processor is idle) for the same states (see Figure 7.9 (b)). The real-time Engine directly schedules the interactions at time instants meeting the timing constraints, avoiding the need for strong synchronization between the components when they execute a *tick*.

Moreover, executing *tick* at a given period $P = 10ms$ requires the execution times of interactions to be bounded by P, which is a strong and restrictive assumption that imposes to decompose the interactions when it does not hold. Second, *tick* involves strong synchronization of all components. The obtained model may easily deadlock: a local deadlock of a single component leads to global deadlock of the system as explained earlier. Finally, mixing

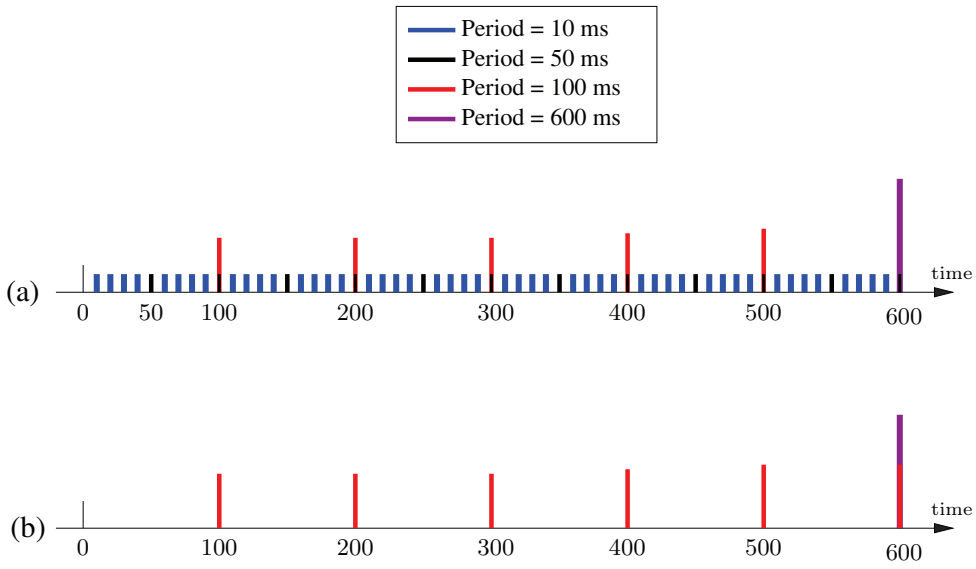


Figure 7.5: Execution trace of the antenna module using the first implementation (a) and the second implementation (b).

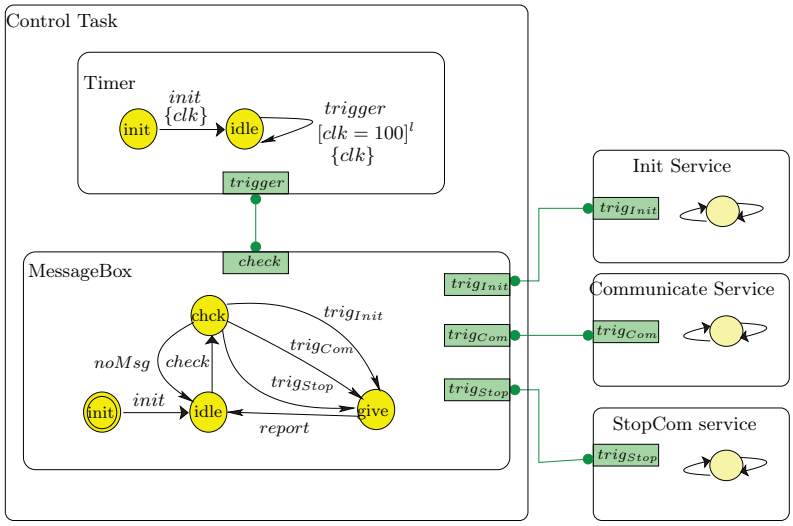


Figure 7.6: The MessageBox component representation using ticks.

timing constraints (expressed in terms of boolean guards involving integer variables) with data makes the analysis of the model more difficult. Indeed, they should be restricted to interval constraints and handled separately to exploit existing analysis techniques for timed automata, as in the proposed method.

3.2 Introducing Input and Output Ports

In the first implementation, the communication with the decisional level (the environment) and the Antenna module is achieved by using a dedicated shared buffer (see Figure 7.6). The

decisional level directly writes requests in the buffer, and Antenna periodically reads the buffer using component *MessageBox* to check their presence. The chosen period is 100 ms. Antenna also sends reports when executing the internal action *report* of *MessageBox*.

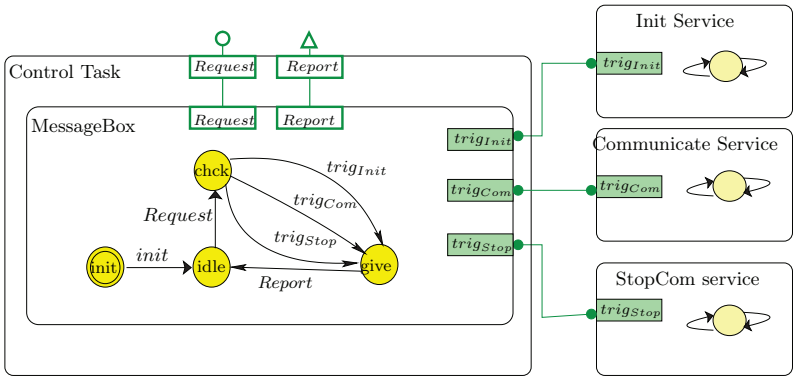


Figure 7.7: The Message Box component representation using inputs and outputs.

The extension we now make is based on the open real-time Engine, that is, we introduce the notion of Input and Output ports to communicate with the decisional level (see Figure 7.7). We introduce the input port *Request* that replaces the active wait used by the first implementation. The output port *Report* is introduced to send reports to the environment after the treament of a request. Reports were send via the transitions of the MessageBox component.

The execution of the Antenna module using the multi-thread Engine consumes almost 4 times more CPU compared to executions with the open real-Time Engine. The CPU utilization is thus 2 times higher for using active waits based on timing constraints compared to the implementation with environment ports. The reason is that the previous implementations execute actions even at states for which the application is waiting for a request arrival, whereas the second implementation is actually sleeping (processor is idle) for the same states.

The response time of Antenna (i.e. delay between sending a request to Antenna and its treatment) is also drastically improved when using input and output ports instead of active waits: it ranges from 0 to 100 ms for the first implementation whereas it is about 0.1 ms for the last one (see Figure 7.8). This is due to the fact that the open real-time Engine is instantaneously woken up in the second implementation when a request arrives during sleeping periods (see Figure 7.9). We implemented this mechanism in the decisional level by sending a Unix signal to the Event Handler each time a request is sent. It is even possible to model the minimal arrival time of requests using timing constraints, and verify if it is compatible with the other timing constraints (for example, the communication window bounds) and the execution times. Moreover, the BIP Engine is able to check that the actual arrivals of the requests meet the model constraints.

4 Conclusion

In this chapter, we presented the accomplished work for building software modules for the Dala robot. We first presented the existing work that combines a state of the art tool (GenoM), developped in the LAAS laboratory, with the component based framework BIP,

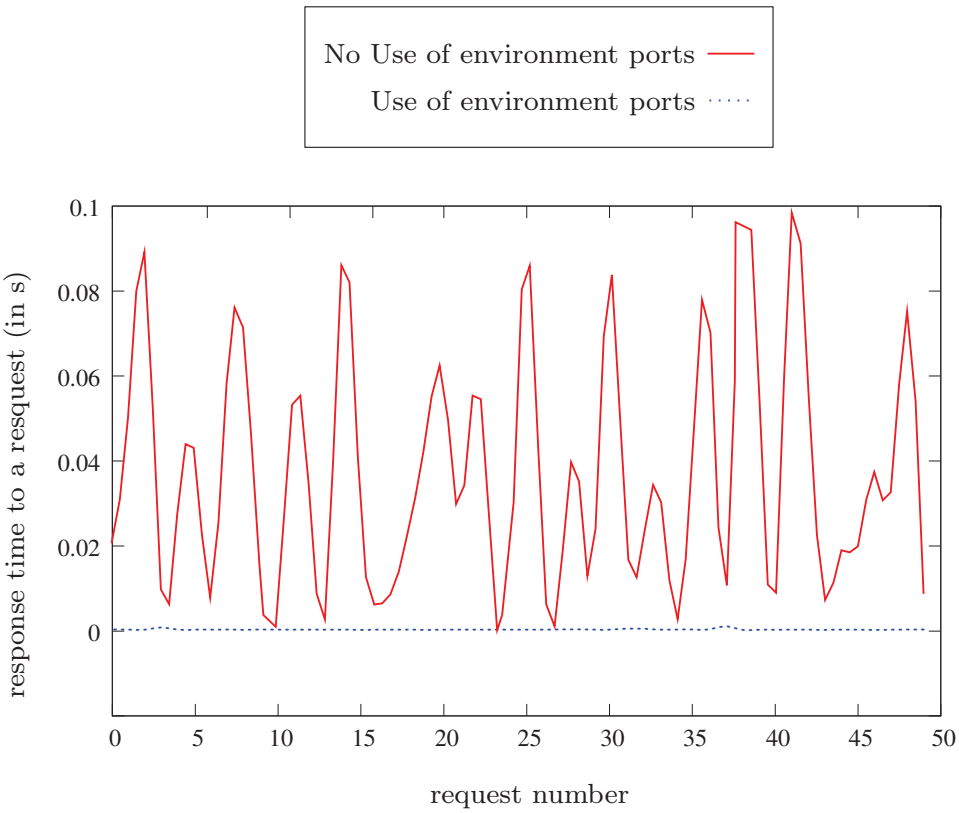


Figure 7.8: Response time of Antenna to a request.

in order to use formal methods for developing modules of the functional level of robots. To this end, a successfull tool has been developed based on the BIP/GenoM component based design approach. Nevertheless, little attention has been drawn to verify the timing properties of the considered system.

We extend and improve the approach by modeling the modules timing features with the real-time BIP models. We tested the execution of the Antenna module with the real-time BIP engine and we shown an amelioration of the CPU utilization. The main reason is that the multi-thread Engine (untimed) executes *tick* every 10 ms, even at states for which the application is waiting for enabledness of a guard, whereas the real-time Engine is actually sleeping (processor is idle) for the same states. We also extended the modelling approach by introducing input and output ports, for specifying interactions with the environment. The execution of the Antenna module with the open real-time BIP engine has also shown an amelioration of the CPU utilization. The reason is that the previous implementation executes actions even at states for which the application is waiting for a request arrival, whereas the second implementation is actually sleeping (processor is idle) for the same states. The response time of Antenna is also drastically improved. This is due to the fact that the open real-time Engine is instantaneously woken up when a request arrives during sleeping periods instead of periodically checking the presence of a request.

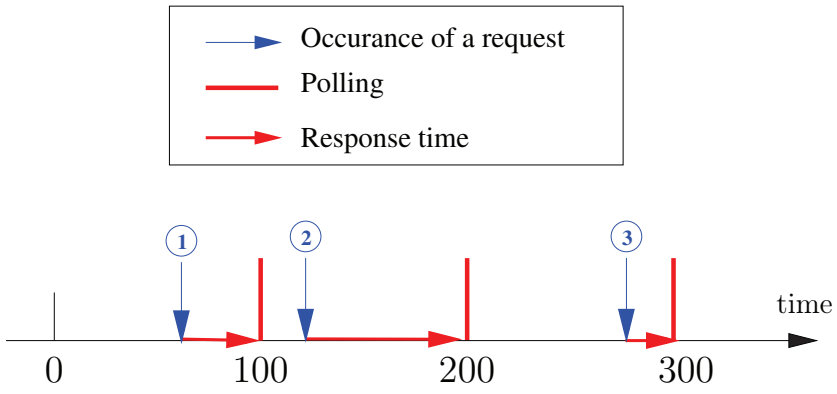


Figure 7.9: Occurrence of requests in the Antenna module example.

Chapter 8

The Allen Temporal Logic for Planning

The decisional level of an autonomous system usually involves temporal planners [59] that basically choose variable time instants for the execution of actions of a system, e.g., starting or terminating a task. The goal of the planner is to find a valid plan, that is, a list of actions (possibly with parallel execution) meeting user-defined constraints that can be expressed using various formalisms. Planners can also seek for efficiency by optimizing parameters such as latency, throughput, energy, and memory.

The robotic systems we deploy rely on a high level temporal planner and a plan execution controller using a Timeline-based planning technique. Allen's interval algebra, also called Allen's temporal logic (ATL) is one of the best established formalisms for temporal reasoning [2,3]. It is a widely used formalism in the temporal planning community for expressing constraints on plans. The constraints are expressed as boolean formulae over atomic propositions. Validation and correctness of complex plans for systems with decisional autonomy is highly desirable, if not crucial in many applications. Since ATL is the logic of planning, an automated translation from ATL to BIP models enables us to use rigorous design and implementation techniques and tools in a domain lacking them.

Moreover, the importance of controlling the execution of the plan, that is monitoring, cannot be overestimated. For example, an autonomous rover whose execution plans have been rigorously verified may still fail for reasons such as hardware or operating system failures, unexpected terrain in an unknown environment, violations of timing constraints etc. Having a controller to check the online execution of plans step by step and to trigger recovery code in case of violations is of crucial importance.

Our contribution consists of providing a rigorous and simple technique for modeling the plans. The execution and the controlling mechanisms are also handled by the open real-time BIP framework. Thus, there is no need to synthezise monitors for the execution. The BIP engine directly executes the plans actions with the respect to the constraints and monitors the execution at runtime.

This chapter is structured as follows. In Section 1, we discuss about the existing tools to build plans using formal methods and present the ATL formalism. In Section 2, we present the transformation mechanism from ATL formula to BIP models. In Section 3, we explain how to model plans using BIP models. In Section 4, we give the experimental results conducted on the Dala robot. Finally, in Section 5, we conclude the chapter.

1 Building Plans Using Formal Methods

1.1 State of the art

Formal methods have been more widely used together with "decisional components" of robotic systems. The main reason is perhaps because these decisional components already use a "model" (for planning, diagnostics, etc.). In [89], the authors propose a model-based approach where the objective is to abstract the system into a state transitions based language. The programmers specify state evolutions with invariants and a controller executes this maintaining these invariants. To do that, the controller estimates the most likely current state—using observation and a probabilistic model of physical components—and finds the most reliable sequence of commands to reach a specified goal (i.e., with a minimum probability of failure). In [52], the authors present the CIRCA SSP planner for hard real-time controllers. This planner synthesizes off-line controllers from a domain description (preconditions, postconditions and deadlines of tasks). CIRCA SSP can then deduce the corresponding timed automaton to control the system on-line, with respect to these constraints. This automaton can be formally validated with model checking techniques. Similarly, [29] discusses an approach for model checking the AgentSpeak(L) agent programming language aimed at reactive planning systems. The work describes a toolkit called CASP (Checking AgentSpeak Programs) for supporting the use of model checking techniques, in particular, for automatically translating AgentSpeak(L) programs into a language understood by a model checker. In [85], the authors present a system which allows the translation from MPL (Model-based Processing Language) and TDL (Task Description Language)—the executive language of the CLARAty architecture [73]— to SMV, a symbolic model checker language.

In [66], the authors discuss an approach for automatically generating correct-by-construction robot controllers from high-level representations of tasks given in Structured English, which are translated into a subset of Linear Temporal Logic and eventually into automata. In their work, complex and continuous missions can be specified using the basic prepositions 'between', 'near,' 'within,' 'inside,' and 'outside.' An example of such a mission is "stay near A unless the alarm is sounding," where A is a location. Likewise, [90] also deals with the synthesis of correct-by-construction controllers based on temporal logic specifications. Here, finite state automata based controllers are synthesized by a trajectory planner to satisfy a given temporal specification, which is based on an abstract model of the physical system. The authors show how the correct behavior of an autonomous vehicle can be maintained using the robot controller automatically synthesized.

1.2 Allen Temporal Logic

Allen Temporal Logic is specified as a framework to deal with incomplete relative temporal information such as "Event A is before event B". Instead of adopting time points, Allen takes intervals as the primitive temporal quantity. There are 13 basic binary relations between any two intervals, also called constraints. In planning, ATL is then used to reason about concurrency and temporal extent, where action instances and states are described in terms of temporal intervals that are linked by constraints. Attributes whose states change over time are called state variables. The history of values of a state variable over a period of time is called a "Timeline". The interval constraints among all possible values that must occur among state variables for a plan to be legal are organized in a set of compatibilities. Compatibilities are the causal and temporal relationships between attributes. They determine

the necessary correlation with other procedure invocations in a legal plan.

Example 17 *To illustrate the use of ATL, let us consider a PhD-student coffee problem, used as a running example (see Figure 8.1). A PhD-student is at his office and has a "Low" mood because he is running out of coffee. If he goes to the caffet and takes a coffee then his mood will change to "High". The available actions are "Going" from a place to another and "Preparing" the machine "(M)" to make coffee or the coffee "(C)" itself. The coffee is to be prepared in the caffet and requires the machine beeing prepared for it. To model the problem we use three Timelines as follows.*

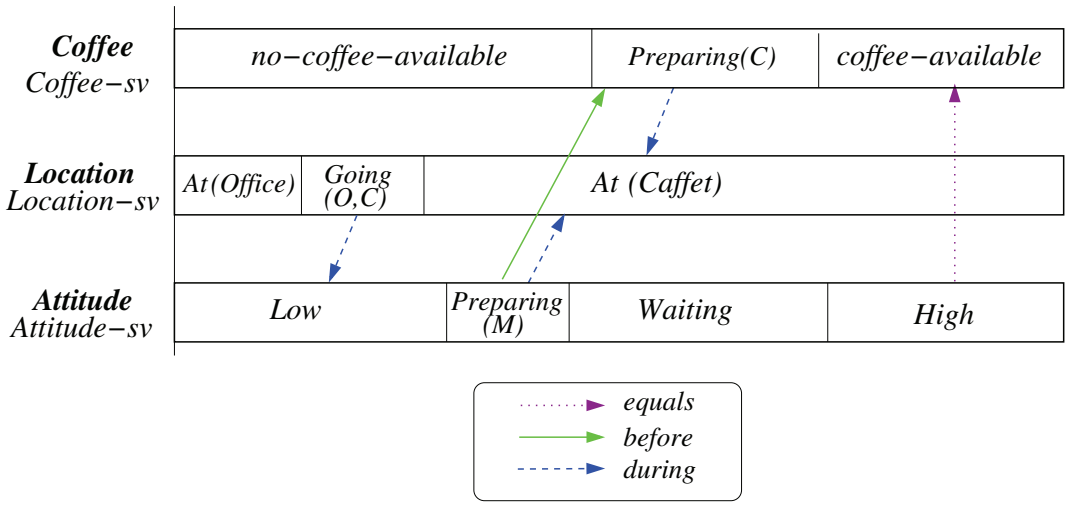


Figure 8.1: The PhD-student coffee problem.

1. **Coffee** has one state variable "Coffee-sv" saying if the coffee is available or not. The coffee goes from state "no-coffee-available" to "coffee-available" after the Coffee preparation "Preparing(C)".

2. **Location** has one variable "Location-sv" for the location of the PhD-student. The PhD-student is from state "At(Office)" to "At(Caffet)" when goes from his office to the caffet "Going(O,C)".

3. **Attitude** has one variable "Attitude-sv" for the attitude of the PhD-student. The mood of the PhD-student changes from "Low" to "High" after preparing the machine "Preparing(M)" to make coffee and "Waiting" for it to be available.

We can consider the following compatibilities :

- Coffee-available ("Ca") requires Preparing(C) ("CP") which requires no-coffee-available ("NCa"). Preparing(C) is performed while At(Caffet) ("@(Caffet)") after Preparing(M)("MP").

- At(Caffet) ("@(Caffet)") requires going from the location Office to the Caffet which requires At(Office) ("@(office)"). Going(O,C) ("G(O,C)") is performed while Low.

- High (“H”) requires Waiting (“W”) which requires Preparing(M) (“MP”) and Low (“L”). Preparing is performed while At(Caffet) and High as long as coffee-available.

To formally specify these compatibilities we use ATL formula defined as follows.

Definition 32 (Allen Temporal Logic Formula) *If P is a set of atomic propositions and T is a set of intervals, then an Allen temporal logic formula over P and T, or an ATL(P, T) is any boolean combination of basic formulae of the form:*

- *Equals(I, J),*
- *Before(I, J) and After(I, J),*
- *Overlaps(I, J) and OverlappedBy(I, J),*
- *Meets(I, J) and MetBy(I, J),*
- *Contains(I, J) and During(I, J),*
- *Starts(I, J) and StartedBy(I, J),*
- *Ends(I, J) and EndedBy(I, J),*

where I, J ∈ T.

Figure 8.2 gives a representation of some Allen relations (six of the above have a symmetrical one). We consider two time intervals $I = [s_I, t_I]$ and $J = [s_J, t_J]$, where s_I (resp. s_J) is the start time of I (resp. J) and t_I (resp. t_J) its terminating time (i.e. $s_I < t_I$ and $s_J < t_J$).

(a) Equals(I, J) means intervals I and J coincide (i.e., $s_I = s_J$ and $t_I = t_J$).

(b) Before(I, J) or After (J, I) means I terminates before J starts (i.e., $t_I < s_J$).

(c) Overlaps(I, J) or OverlappedBy(J, I) means J starts during I and J finishes after I (i.e., $s_I < s_J < t_I < t_J$).

(d) Meets(I, J) or MetBy(J, I) means J starts when I terminates (i.e., $t_I = s_J$).

(e) During(I, J) or Contains(J, I) means I is included in J (i.e., $s_J < s_i$ and $t_i < t_J$).

(f) Starts(I, J) or StartedBy(J, I) means I and J start at the same time (i.e., $s_I = s_J$).

(g) Ends(I, J) or EndedBy(J, I) means I and J finish at the same time (i.e., $t_I = t_J$).

The compatibilities extracted from example 1.2 can be formally specified in ATL as follows:

- Meets(“NCa”, “CP”) ∧ Meets(“CP”, “Ca”) ∧
  During(“CP”,“@(Caffet)”) ∧ Before(“MP”,“CP”) ∧
- Meets(“@(office)”,“G(O,C)”) ∧ Meets(“G(O,C)”,“@(Caffet)”)
  ∧ During(“G(O,C)”,“L”) ∧
- Meets(“L”,“MP”) ∧ Meets(“MP”,“W”) ∧ Meets(“W”,“H”)
  ∧ During(“MP”,“@(Caffet)”) ∧ Equals(“H”,“Ca”).

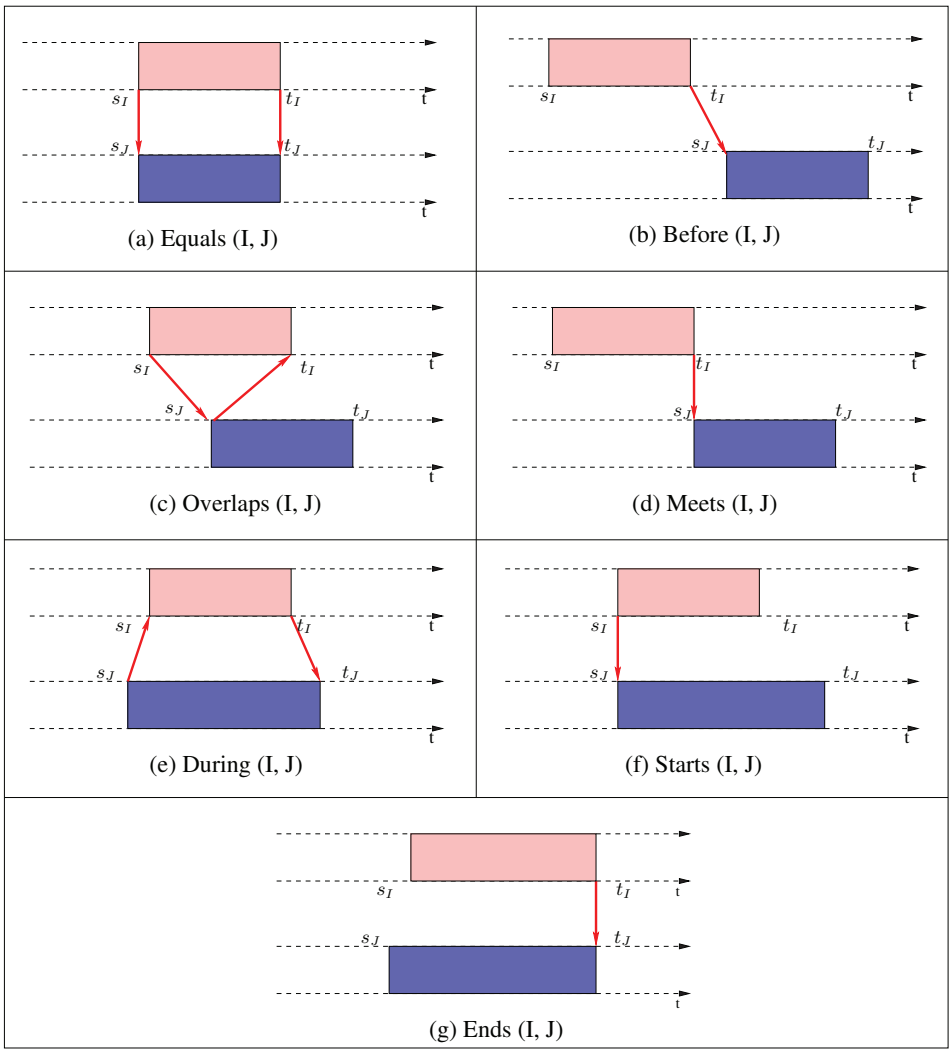


Figure 8.2: Allen Temporal Relations

2 Translating Allen Temporal Logic into BIP Models

It has been shown that Allen interval logic formulae can be translated into timed automata [79] in order to check its satisfiability. Each interval relation is translated into a little state machine that has two special states, true when the formulae holds and false otherwise. For example, if we consider the $Meets(I, J)$ relation (see Figure 8.3), one starts with the initial state $(\overline{I}, \overline{J})$ (neither I nor J), and there it stays as far as one does not enter any of the intervals. If while in this state the monitored program enters the interval J, then the relation $Meets(I, J)$ is obviously violated. Otherwise, if the interval I but not J is entered, the machine moves to state $(I, \overline{J})$ where it waits until either I is left and J is entered in which case it returns true, or otherwise, until I is left without entering J or I and J overlap, when it returns false.

We think that the translation of ATL formula into real-time BIP models can lead to smarter and smaller descriptions. The idea is to use one timed automaton (i.e., one compo-

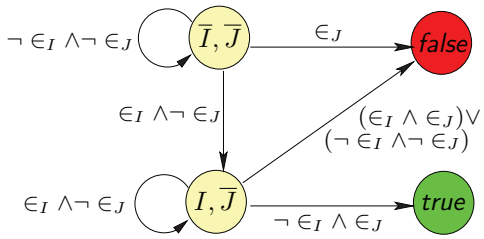


Figure 8.3: Translation of the $Meets(I, J)$ relation into timed automata.

nent) for modeling each interval. Interactions and priorities—a.k.a. the glue—in BIP offers also an elegant language for expressing constraints between intervals used to describe coincidence of actions. In this section, we explain first how to translate Allen intervals into BIP atomic components, then how to translate the different Allen constraints into connectors.

2.1 Translating Allen intervals into BIP atomic components

We consider an interval as an executing action, thus it has a starting time and a finishing time. We model an interval with an atomic component as follows (see Figure 8.4 (b)).

Component $Interval = (\emptyset, \mathsf{A}, \{init, exec, end\}, \{x, y\}, \longrightarrow)$ is composed of a set of ports $\mathsf{A} = \{\textbf{begin}, \textbf{executing}, \textbf{finish}, \textbf{no-executing}\}$, port **begin** corresponding to the beginning of the execution, port **executing** corresponding to the execution, port **finish** corresponding to the end of execution and port **no-executing** corresponding to the time after execution of the interval (see Figure 8.4 (a)). Ports **begin** and **finish** correspond to particular time instants of executions that are triggered only once in the BIP component. Ports **executing** and **no-executing** correspond to time intervals that can be triggered as long as we are in the corresponding times intervals.

The Component has also two clocks x and y involved in the following set of transitions :

$$
\begin{aligned}
\longrightarrow \quad = \{ \quad & (init, begin, [s_{lb} \leq x \leq s_{ub}]^{\mathsf{d}}, \{y\}, exec), \\
& (exec, executing, \emptyset, \emptyset, exec), \\
& (exec, finish, [t_{lb} \leq y \leq s_{ub}]^{\mathsf{d}}, \emptyset, end), \\
& (end, no-executing, \emptyset, \emptyset, end) \quad \}.
\end{aligned}
$$

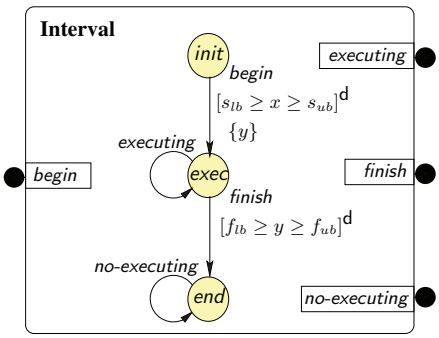


Figure 8.4: Translating an interval(a) into a BIP component(b).

Clock x is reset at intialization. It is used to ensure the beginning of the action within a starting interval $[s_{lb}, s_{ub}]$ (where s_{lb} is the lower bound and s_{ub} is the upper bound), by using constraint $s_{lb} < x < s_{ub}$ with a delayable urgency. Clock y is reset when the action execution begins. It is used to measure the execution time of the action to ensure its termination within a finishing interval $[t_{lb}, t_{ub}]$ (where t_{lb} is the lower bound and t_{ub} is the upper bound), by using constraint $t_{lb} < y < t_{ub}$ with a delayable urgency. Each transition is labeled by an exported port. Exported ports are used as an interface to enable the synchronization with other components.

Notice 1 *If there is no constraint on the beginning of the action, we introduce clock x to enforce non-null delays between communicating actions by using constraints $x > 0$.*

2.2 Translating Allen constraints into BIP connectors

Boolean conjunctions of atomic propositions can be efficiently derived using existing work that establishes the correspondence between boolean formulae and the glue of BIP [BS, 2008]. Therefore, coincidence of actions (e.g., $t_I = s_J$) can be modeled as a strong synchronization between atomic components using interactions. Each Allen constraint between two intervals can be translated into strong synchronizations between transitions of the corresponding atomic components, through their exported ports. Ordering of actions (e.g., $t_I < s_J$) can be modeled as a set of priorities.

We have a corresponding connector type for each Allen relation as follows (see Figure 8.5):

(a) Equals(I, J) synchronizes both the beginning of I and J and their finishing. It corresponds to a strong synchronization between port begin of I and port begin of J (i.e. interaction $\{I.\mathsf{begin}; J.\mathsf{begin}\}$), and a strong synchronization between port finish of I and port finish of J (i.e. interaction $\{I.\mathsf{finish}; J.\mathsf{finish}\}$) ;

(b) Before(I, J) or After (J, I) synchronizes the end of execution of I and the beginning of J. It corresponds to a strong synchronization between port no-executing of I and port begin of J (i.e. interaction $\{I.\mathsf{no\text{-}executing}; J.\mathsf{begin}\}$);

(c) Overlaps(I, J) or OverlappedBy(J, I) synchronizes the beginning of J with the execution of I, and the finishing of J with the execution of I. It corresponds to a strong synchronization between port executing of I and port begin of J (i.e. interaction $\{I.\mathsf{executing}; J.\mathsf{begin}\}$), and a strong synchronization between port no-executing of I and port finish of J (i.e. interaction $\{I.\mathsf{no\text{-}executing}; J.\mathsf{finish}\}$);

(d) Meets(I, J) or MetBy(J, I) synchronizes the finishing of I and the beginning of J. It corresponds to a strong synchronization between port finish of I and port begin of J (i.e. interaction $\{I.\mathsf{finish}; J.\mathsf{begin}\}$)

(e) During(I, J) or Contains(J, I) synchronizes the beginning of I with the execution of J, and also the finishing of I with the execution of J. It corresponds to a strong synchronization between port begin of I and port executing of J (i.e. interaction $\{I.\mathsf{begin}; J.\mathsf{executing}\}$), and a strong synchronization between port finish of I and port executing of J (i.e. interac-

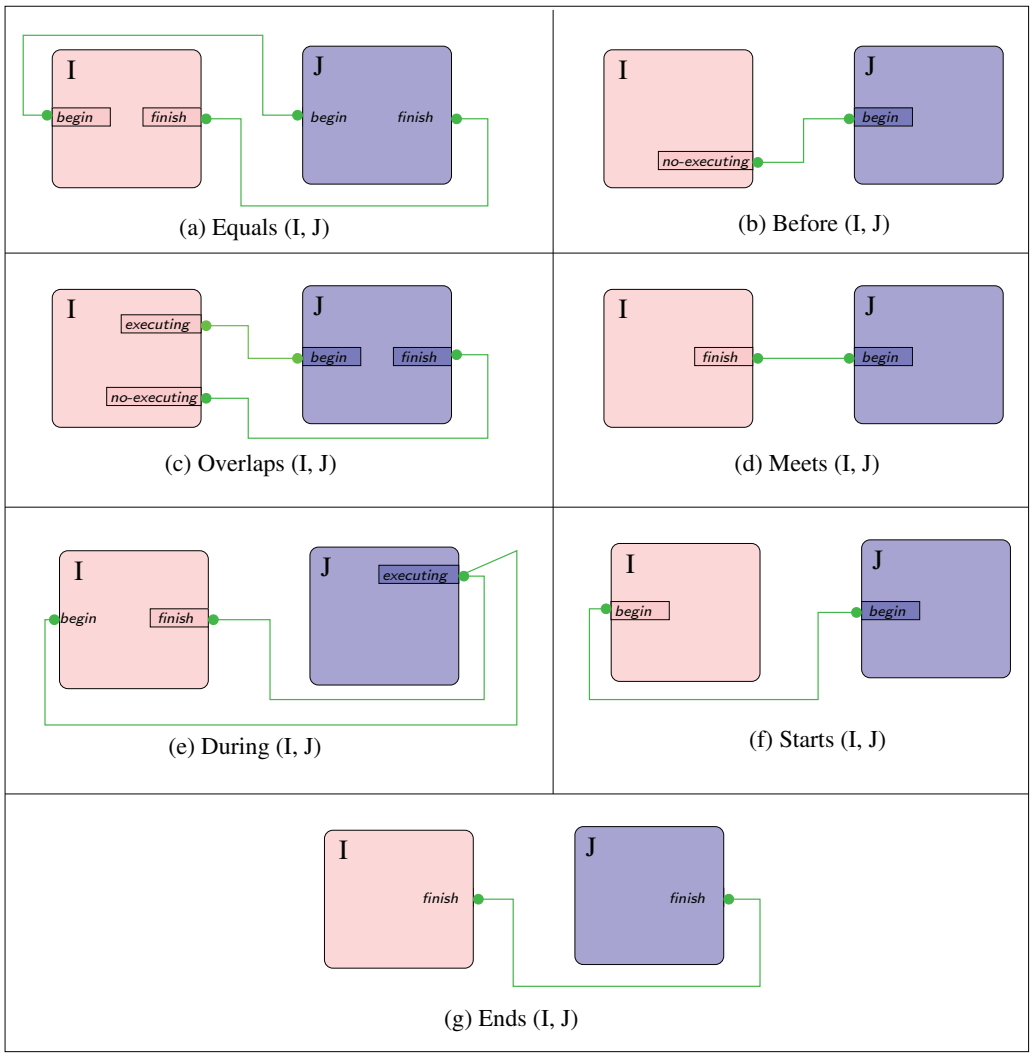


Figure 8.5: Allen Temporal Relations

tion $\{I.\mathsf{begin}; J.\mathsf{executing}\}$);

(f) Starts(I, J) or StartedBy(J, I) synchronizes the beginning of I with the beginning of J. It corresponds to a strong synchronization between port begin of I and port begin of J (i.e. interaction $\{I.\mathsf{begin}; J.\mathsf{begin}\}$),

(g) Ends(I, J) or EndedBy(J, I) synchronizes the finishing of I with the finishing of J. It corresponds to a strong synchronization between port finish of I and port finish of J (i.e. interaction $\{I.\mathsf{finish}; J.\mathsf{finish}\}$),

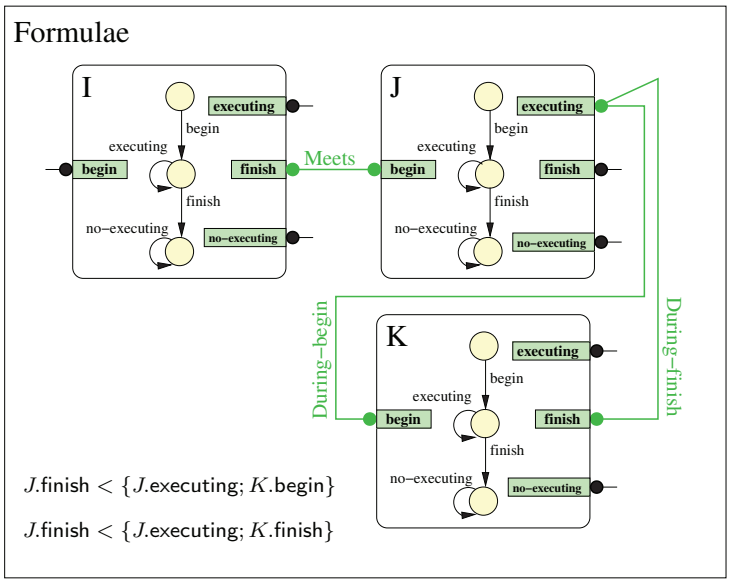


Figure 8.6: Example of translation: "I meets J and K during J" into BIP.

Example 18 *Figure 8.6 gives the translation of a formulae over 3 three intervals I, J and K into a BIP model, considering "I meets J and K during J". Actions are modeled as atomic components and constraints as interactions over those components. The beginning of the execution of J is strongly synchronized with the end of the execution of I to model "I meets J" by using a strong synchronization between port* finish *of I and port* begin *of J.*

To ensure that K is executed during the execution of J, we use strong synchronizations between port executing *of J and ports* begin *and* finish *of K. We also give a higher priority for the execution of K over the completion of J to enforce the execution of K before the completion of J. Thus, priorities enforce ordering of actions (e.g., $J.\mathsf{finish} < \{J.\mathsf{executing}; K.\mathsf{begin}\}$ and $J.\mathsf{finish} < \{J.\mathsf{executing}; K.\mathsf{finish}\}$).*

3 Plans Modeling Using BIP

We have seen how ATL formula can be translated into BIP models, we now explain how to build correct models for plans. We first explain how Timelines are modeled and how to manage the constraints between the different Timelines using BIP. We also make some extensions in the BIP framework, especially in the BIP language, in order to easily use the Allen constraints in BIP models.

3.1 Modeling Plans: First Method

Modeling Timelines

Each Timeline is modeled as a compound component composed of atomic components representing the actions. The chronological order between these actions is expressed by the

constraint "Meets" and modeled by strong synchronizations between the finish port of an action and the begin port of the next one.

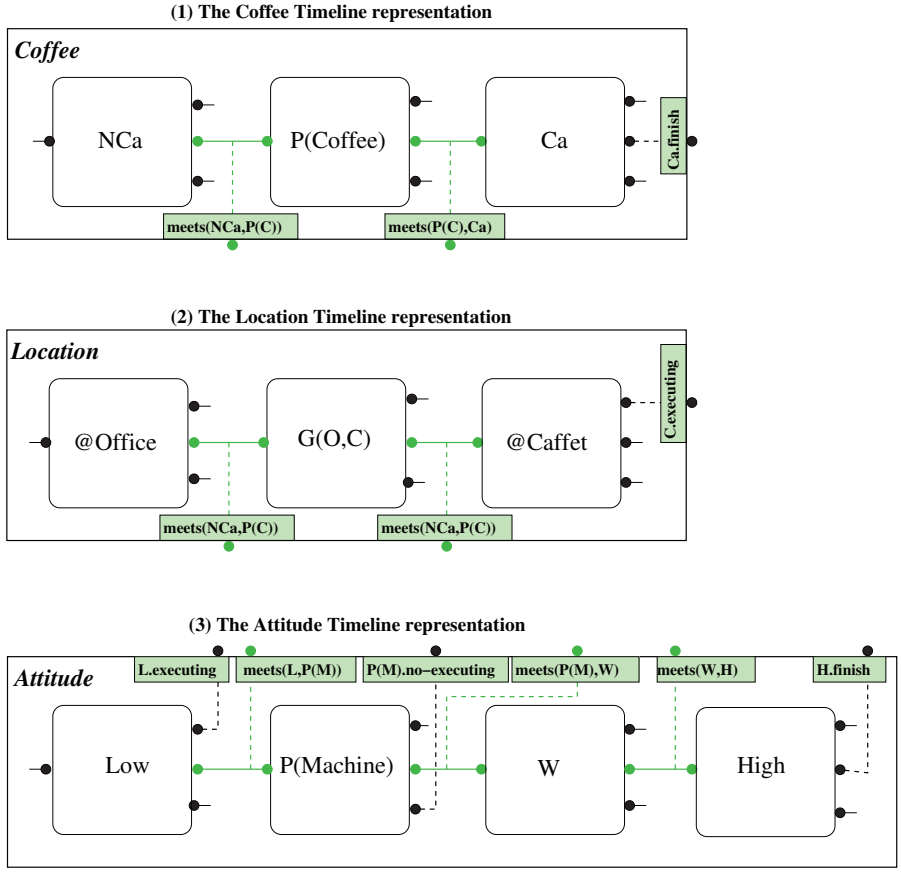


Figure 8.7: Timelines modeling for the PhD-student example.

We can see in Figure 8.7, the Timelines transformation of the PhD-student coffee problem (Plan in Figure 8.1) as three compound components:

- **Coffee** (1) is composed of three components (*NCa*, *P(C)*, *Ca*) representing each state variable of the Coffee Timeline. The formulae *Meets("NCa", "P(C)")* is modeled by the synchronization of port finish of component *NCa* with port begin of component *P(C)*, and the formulae *Meets("P(C)", "Ca")* is modeled by the synchronization of port finish of component *P(C)* with port begin of component *Ca*.
- **Location** (2) is composed of three components (@Office, G(O,C), @Caffet) representing each state variable of the Location Timeline. The formula Meets("@(office)","@(Caffet)") and Meets ("G(O,C)","@(Caffet)") are also modeled by *Meets* synchronizations between the three actions.
- **Attitude** (3) is composed of four components (Low, P(M), W, High) representing each state variable of the Attitude Timeline. The formula Meets("L","MP"), Meets ("MP","W") and Meets("W","H") are modeled by strong synchronizations between the four atomic components .

Modeling composition of Timelines

In order to model constraints over timelines, each compound component representing a Timeline has to export ports that are involved in synchronizations with other Timelines. The atomic component P(C) of the Coffee Timeline synchronizes with the atomic component @(Caffet) of the Location Timeline (for the formulae During("P(C)","@(Caffet)")) and with the atomic component P(M) of Attitude Timeline (for the formule Before("P(M)","P(C)")). Finally, the atomic component Ca synchronizes with the atomic component High of Attitude Timeline (for the formulae Equals("H","Ca")).

The Coffee Timeline has to export the ports begin and finish of P(C), the Location Timeline has to export the execution port of @(Caffet) and the Timeline Attitude has to export the port no-executing of P(M). Since ports begin and finish of P(C) and port begin of Ca are involved in *Meets* synchronizations inside the $Coffee$ Timeline, we export the ports of the connector they are involved in. Indeed, a connector has an option to define a port and export it. This allows a connector to be used as a port in other connectors or in the compound components they belong to, and create structured connectors. The port export statement is provided in the connector type definition. The resulting exported ports for the Coffee Timeline are the following.

- Exported from a synchronization: **meets(NCa,P(C))** resulting from the synchronization involving port **begin** of P(C), and **meets(P(C),NCa)** resulting from the synchronization involving port **finish** of P(C).
- Exported from components: **Ca.finish** corresponding to port **finish** of Ca, since it is not involved in any synchronization.

All the other atomic components ports that are not exported by the timeline are wrapped into singleton connectors in order to enable their execution whitout any constraint, that is synchronizations with other components.

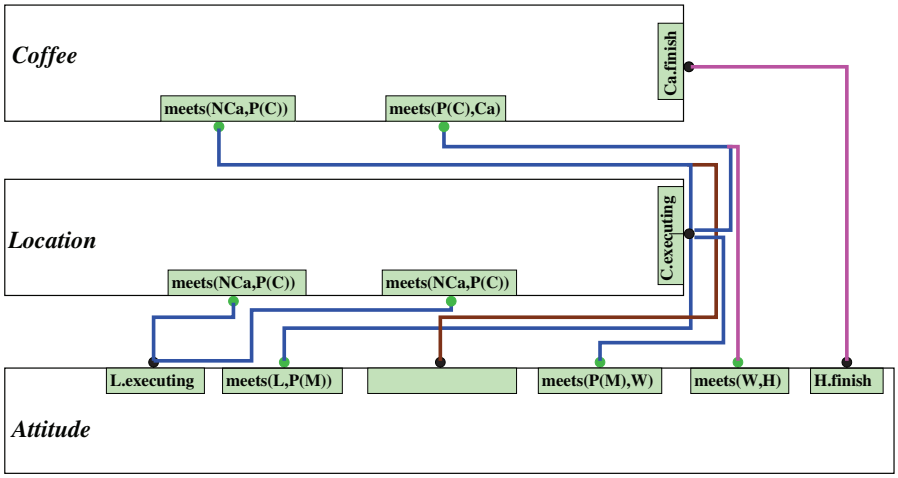


Figure 8.8: Model for the phd-student example.

Figure 8.8 is a representation of the model of the whole plan. We can see the synchronizations between the different Timelines through their exported ports. We note that

some ports are involved in several synchronizations and thus, it is then necessary to merge the interactions in which they are involved into a single connector. We merged the interaction $\{meets(P(C), Ca); meets(C.executing)\}$, part of the modeling of the formulae During("P(C)","@(Caffet)"), and interaction $\{meets(P(C), Ca); meets(W, H)\}$ of the modeling of the formulae Before("P(M)","P(C)"). It is due to the fact that the exported port meets(P(C),Ca) representing port begin of component P(C) is involved in both interactions. The merging concerns only exported ports of type begin or finish since the ports of kind executing and no-executing can be triggered several times, and as long as they are involved in During and After synchronizations types.

3.2 New language for Modeling Plan

We note that the transformation of Allen Temporal Logic into BIP models is tedious and error prone due to the merging mechanism. In this purpose, we extend the BIP language in order to express the Allen Temporal Logic between components by introducing the Allen constraints keywords. From this new language, we automatically generate the corresponding connectors by a model to model transformation tool.

The BIP Language extensions

We extend the BIP language in order to express allen constraints between components in an easy way by introducing in the BIP grammar keywords corresponding to Allen Algebra constraints. Each compound component representing a Timeline (respect. a plan), defines the components representing its actions (resp. Timelines). We add the possibility to declare allen formula between the components inside a compound component (see Figure 8.9). Figure 8.10 describes the new syntax of an Allen definition. Each declaration of an allen definition is identified by the keyword **allen**. We can optionnaly associate to it a name *allen-name*, which can be used to identify the corresponding interaction execution in the execution traces. If it is not specified, the tool allocates a special number to the allen formulae. Then, we define the allen relation between an *action1* and *action2* with the appropriate *allen-constraint* (**meets**, **before**, **overlaps**, **starts**, **ends**, **during**, **equals**).

compound-type-definition ::=
compound type *compound-type-name*
[**(** *c-type-name fpar-name* { **,** *c-type-name fpar-name* }* **)**]
{ *component-definition* }*
{ *allen-definition* }*
end

component-definition ::=
component-type component-name **(** *actual-arg* { **,** *actual-arg* }* **)**

Figure 8.9: *Allen definition in a compound component*

Notice 2 *When the compound component represents the plan, then the actions are of the*

allen-definition ::= **allen** [*allen-name*] *allen-relation*

allen-realtion ::= *action1 allen-constraint action2*

action1 ::= *component-identifier*
action2 ::= *component-identifier*

allen-constraint ::=
meets
|**before**
|**overlaps**
|**starts**
|**ends**
|**during**
|**equals**

component-identifier ::= *IDENTIFIER* (. *IDENTIFIER*) ?

Figure 8.10: *Allen Forluma declaration syntax*

form identifier1. identifier2, where identifier1 is the name of the Timeline and identifier2 is the name of the desired atomic action involved in the synchronization.

Allen to BIP Model Transformation Tool

Given a BIP file describing a plan with the new allen syntax, we build an Allen/BIP model. The Parser analyses the BIP description source and generates an intermediate model conforming to the new BIP meta-model describing the allen syntax. It performs syntactic analysis of the input program conforming to the new BIP grammar and reports the programming errors. The Allen-to-BIP transformation tool generates a correct BIP model by translating the Allen definitions decribed in the Allen/BIP model into connectors, in order to generate an executable C++ code (see Figure 8.11).

Given a plan model S, the proposed algorithm (Algorithm 7) corresponds to the following computation steps in the transformation tool.

Extracting Allen definitions from Timelines. From each Timeline declared in the plan as a compound component, we extract the Allen definitions (lines 2-5). For each allen declaration, we create a type of internal interaction using the function $CreateInternalInteraction$ (line 12). Function $CreateInternalInteraction$ takes as parameter the $Name$ of the declaration, actions Act_1 and Act_2 that are involved in the synchronization, and the type of constraint Cst. According to the type of constraint, it creates the corresponding interaction

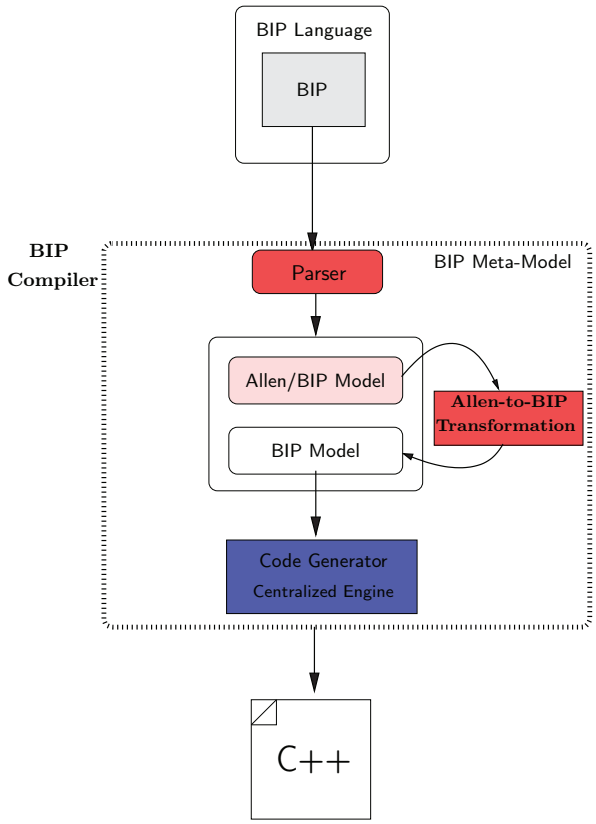


Figure 8.11: The Allen-to-BIP transformation tool.

by connecting actions Act_1 and Act_2 ports, following the rules presented in Section 2.2. Each created interaction is saved in a dictionary.

Creating Connectors. When all the interactions of a given Timeline are created, we apply the merging mechanism when it is necessary (line 15). That is, when two interactions use of the same instances of ports of kind begin or finish. After the merging, we create a connector for each interaction (line 16) and we export all the ports issued from the components or connectors.

Extracting Allen definitions from the Plan.

In the plan compound component, we extract the Allen definitions for Timelines (line 20). For each declaration, we create a type of interaction using the function *CreateInteraction* (line 22). To create the interaction corresponding to the allen definition, we use the exported ports created in the previous step. When the needed port of one of the actions is involved in a Timeline internal connector, we use the port issued from the connector, otherwise, we use the exported port of the component.

Creating Connectors. When all the interactions of the plan are created, we apply the merging mechanism (line 25). We finally create the connectors for each interaction and the singleton connectors for all the timelines exported ports that haven't been used for the

interactions.

Algorithm 7 Allen to BIP Transformation

Require: S *// The Plan model*

loop
  $T_i \leftarrow$ EnabledTimeline(S) *// For each Timeline*

  loop
    $A_j \leftarrow$ AllenConstraint(T_i) *// For each Allen declaration*
    if HasName(A_j) **then**
      $Name \leftarrow$ getAllenName(A_i)
    end if
    $Act_1 \leftarrow$ getAction1(A_j)
    $Act_2 \leftarrow$ getAction2(A_j)
    $cst \leftarrow$ getConstraint(A_j)
    CreateInternalInteraction($Name$,Act_1,Act_2,Cst) *// Create interaction*
  end loop

  $I =$ MergeInternalInteractions(C_j) *// First Merge*
  $C =$ CreateConnectors(I) *// Create Connectors*
  ExportPorts(C)

  loop
    $A_i \leftarrow$ AllenConstraint(S) *// Constraints over Timelines*
    ($Name$,Act_1,Act_2,Cst) $\leftarrow$ extraction(A_i)
    CreateInteraction($Name$,Act_1,Act_2,Cst) *// Create interaction*
  end loop

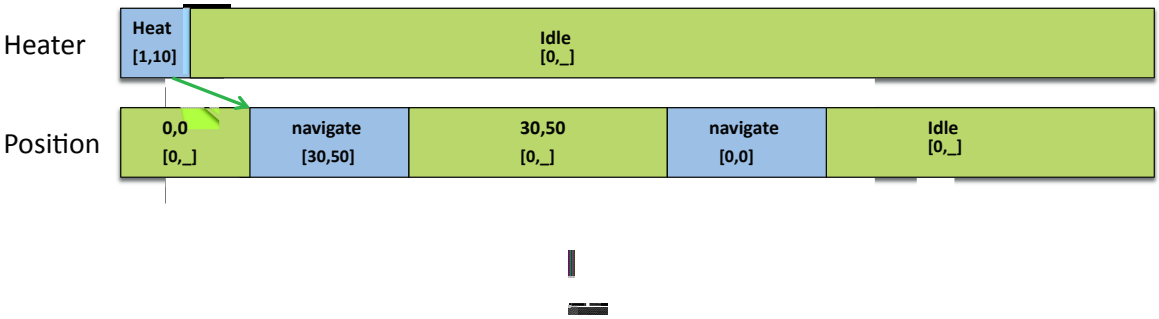
  $I =$ MergeInteractions(C_j) *// Second Merge*
  $C =$ CreateConnectors(I) *// Create final Connectors*
  CreateSingletonConnectors(C) *// Create single Connectors*

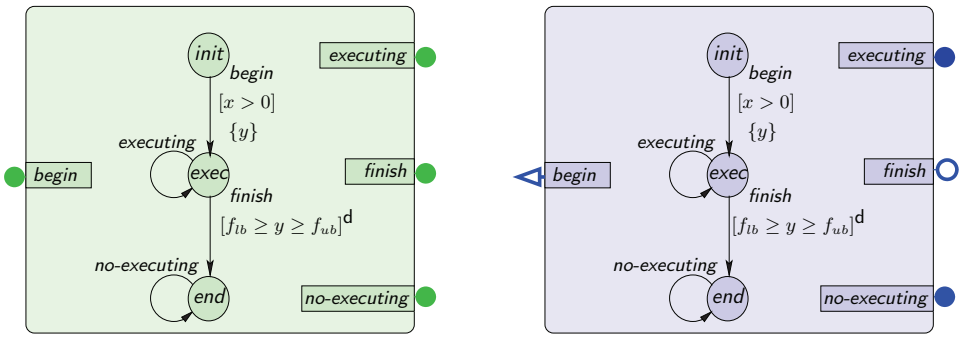
end loop

4 The Dala Rover Planning Example

We have modeled and executed a Dala plan using the open real-time BIP framework. The Dala robot mission scenario consists of navigating from an initial position to a target position, taking a picture of the place, and going back to the initial position. Each step of the mission requires a strong collaboration between the different modules of the robot. Figure 8.12 is a graphical representation of the plan. It is composed of six timelines describing the variable states for each module of the robot.

- In the Heater Timeline, the initial action has to set the heater in a given value with action *Heat*. It has a timing constraint on its termination in the interval $[1, 10]$. When it finishes, the *Heater* is at state *idle*.


Heater
Heat
[1,10]
Idle
[0,_]
Position
0,0
[0,_]
navigate
[30,50]
30,50
[0,_]
navigate
[0,0]
Idle
[0,_]



(a) State Action

(b) Uncontrollable Action

Figure 8.13: Modelling the robot basic actions using BIP.

Uncontrollable Action	Robot Command
Initialization	*InitialiseRobot*
Heat (value)	*HeatRobot* value
navigate(x,y)	*MoveRobot* x y
move(x,y)	*MovePTU* x y
shot (image)	*TakeSciencePic* image

Table 8.1: Mapping Between the uncontrollable actions and the robot commands

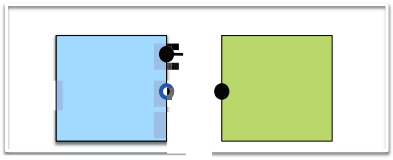
has finished. With this method, we are able to detect timing constraints violations if the actions take more time than expected.

We then use the Allen to BIP model transformation tool in order to transform Allen constraints into BIP connectors. The chronological order between actions belonging to the same timeline is expressed by the constraint meets and modeled by strong synchronizations between terminate and begin ports. We note that ports of type input or output can be involved in such synchronizations. In that case, the resulting interactions contain also synchronization with the environment. Each compound component exports ports that are involved in synchronizations with other components.

Figure 8.14 represents a BIP model for Timelines Heater and Position of the robot. Port "Heat.no-executing" is exported from port "no-executing" of component "Heat," and port "navigate.begin" is exported from the interaction in which port "begin" of component "navigate" is involved. The constraint "heat before navigate" is modeled by synchronizing those two ports.

We describe the plan by using the new BIP language. The BIP code of the Heater and Position Timelines compound components of Figure 8.14 is the following:

```
Compound type Heater
   component uncontrollable-action heater [1,10]
   component state-action idle
   allen heater meets idle
end

Compound type Position
```



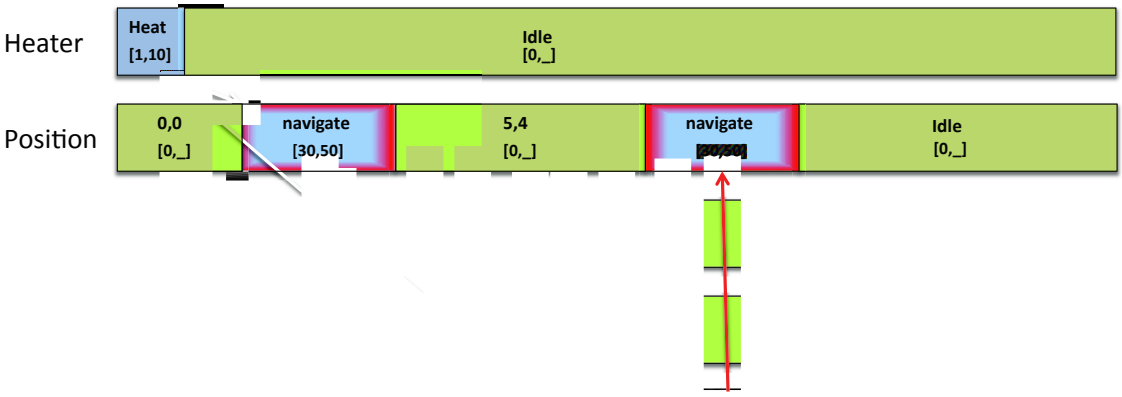
The BIP description of the whole plan is:

```
Compound type Plan
  component Heater heater
  component Position position
  component PTU ptu
  component VIAM viam
  component Antenna antenna
  component Visibility-Window visibility-window

  allen heater.heater before position.navigate1
  allen heater.heater before ptu.move1
  allen heater.heater before viam.shot
  allen heater.heater before antenna.communicate

  allen position.navigate1 during ptu.down
  allen position.navigate0-bis during ptu.down-bis

  allen ptu.move1 during position.position1
  allen ptu.move2 during position.position1
  allen ptu.move3 during position.position1

  allen viam.shot1 during ptu.left
  allen viam.shot2 during ptu.right

  allen antenna.communicate during position.idle
  allen antenna.communicate during visibility-window.orbiter

end
```

4.1 Opportunistic Science

Opportunistic science poses significant challenges for autonomous planning and execution system. In many ways, the challenges of handling opportunistic science are similar to dealing whith unexpected events and anomalies during plan execution. When an autonomous system detects an anomaly, such as a traverse taking longer than expected the system must assess the impact this event will have on its ability to complete the mission objectives. If necessary, the system will revise the plan in order to complete the remaining mission objectives as possible, or enter a safe mode and wait for assistance. Depending on the type of event, the science analysis software may request an additional image or a spectrometer measurement. We extend our planning method in order to enable the planner to take into account opportunistic science detection in the plan. We are able to model the detection of potentially interesting science events and stop the robot in order to eventually retask the rover to respond appropriately. The scenario demonstrate our current capabilities in responding to opportunisctic science events.

Figure 8.15 is a graphical representation of the plan that handles opportunistic science events. We introduce a new timeline Monitor Rock where the initial state is *idle*. Action *rock* is responsible for monitoring the presence of a rock. When an opportunistic science


Heater
Heat
[1,10]
Idle
[0,_]
Position
0,0
[0,_]
navigate
[30,50]
5,4
[0,_]
navigate
Idle
[0,_]

receiving reports from the environment. We extend its capabilities by introducing a new state*stop*. This new state is reached from state *exec* with the transition labelled by a new port *interrupt* that is triggered while the action is executing and a request for stopping the action is sent to the environment. We wait for the *finish* input to be sure that the action has been stopped.

The interruption of navigation when an opportunistic science is detected is expressed by the constraint "monitor before navigate". Each port "nav_i.interrupt" is exported from port "interrupt" of the interruptible component "nav_i" and port "monitor.finish" is exported from port "finish" of component "monitor". The constraint "monitor before navigate" is modeled by synchronizing port "nav_i.interrupt" and port "monitor.finish".

4.2 The Temporal plan execution controller

The real-time BIP engine is well-suited for executing the decisional level of robotic systems. Indeed, the real-time BIP engine becomes a temporal plan execution controller by providing a correct schedule of actions. It communicates with the functional level through the event handler that sends the request corresponding to each uncontrollable action and waits for a reply within the time interval corresponding to the termination of the action. It detects violations of timing constraints and stops the execution or reports the fault to the planner.

5 Conclusion

We have proposed a novel software engineering methodology for developing safe plans for robotic systems. With our approach, one can provide guarantees that the robot will not perform actions that may lead to situations deemed unsafe, i.e., those that may eventuate in undesired or catastrophic consequences and that timing constraints violations can be detected at execution.

We used the BIP Engine as a temporal-plan execution controller for the decisional level. The BIP framework takes into account execution time deadlines and communications with the external environment. Allen Temporal Logic is specified as a framework to deal with incomplete relative temporal information such as "Event A is before event B". Instead of adopting time points, Allen takes intervals as the primitive temporal quantity. We translated ATL formula into real-time BIP models, which merges the logic of planning with correctness. The idea was to use one timed automaton (i.e., one component) for modeling each interval. Interactions and priorities—a.k.a. the glue—in BIP offers also an elegant language for expressing constraints between intervals used to describe coincidence of actions.

We also extended the BIP language in order to express the Allen Temporal Logic between components by introducing the Allen constraints keywords. From this new language, we automatically generate the corresponding connectors by a model to model transformation tool.

We have modeled and executed a Dala plan using the open real-time BIP framework. The Dala robot mission scenario consists of navigating from an initial position to a target position, taking a picture of the place, and going back to the initial position. Each step of the mission requires a strong collaboration between the different modules of the robot.

Using a BIP model combining plans of the decisional level and the functional level is interesting for increasing the performance of the implementations since costly communications between these two levels can be expressed as BIP interactions in which are directly handled

by the engine in the same process. More importantly, this can also be used for verifying global properties, using D-Finder, involving both the decisional level and the functional level.

Part IV

Conclusions and Perspectives

Chapter 9

Conclusion

In this thesis, we provide a rigorous design and implementation methodology for building real-time systems. In this chapter we conclude about the work that has been achieved and futur work directions.

1 Achievements

Our method is based on a formally defined relation between application software written in high level languages with atomic and timeless actions and its execution on a given platform. The relation if formalized by using two models:

1. abstract models which describe the behavior of the application software as well as timing constraints on its actions;

2. physical models which are abstract models equipped with an execution time function specifying WCET for the actions of the abstract model running on a given platform.

Time-safety is the property of physical models guaranteeing that they respect timing constraints. Time-robust physical models have the property to remain time-safe for decreasing execution times of their actions. Non-robustness is a timing anomaly that appears in time non-deterministic systems.

The method is new and innovates in several aspects.

- Our method doesn't suffer limitations of existing methods regarding the behavior of the components or the type of timing constraints. Considered real-time applications include not only periodic components with deadlines but also components with non-deterministic behavior and actions subject to interval timing constraints. The method generalizes existing techniques in particular those based on LET. These techniques consider fixed LET for actions, that is, time-deterministic abstract models. In addition, their models are action-deterministic, that is, only one action is enabled at a given state. For these models time-robustness boils down to deadlock-freedom for WCET.

- It proposes a concrete implementation method using a real-time execution Engine which faithfully implements physical models. That is, if a physical model defined from an abstract model and a target platform is time-robust then the Engine coordinates

the execution of the application software so as to meet the real-time constraints. The real-time execution Engine is correct-by-construction. It executes an algorithm which directly implements the operational semantics of the physical model.

- To the best of our knowledge, the concept of time-robustness is new. It can be used to characterize timing anomalies due to time non-determinism. These timing anomalies have in principle different causes from timing anomalies observed for WCET [78]. Results on time-safety and time-robustness allow a deeper understanding of causes of anomalies. They advocate for time-determinism as a means for achieving time-robustness. An interesting question is loss in performance when in a model interval constraints are replaced by equalities on their upper bound. Time-robustness is then achieved through time-determinization entailing some performance penalty. We are currently studying performance trade-offs for transformations guaranteeing time-robustness.

We have also extended the implementation method for open real-time systems.

- This method generalizes existing techniques namely time-triggered approaches. These techniques rely on the notion of (fixed) logical execution times (LET) imposed to component actions leading to time-deterministic behaviors. We consider any type of interval timing constraints for actions, which encompasses time non-deterministic systems.

- We improve our approach by providing mechanisms that allow the system to react to external inputs produced by the environment, based on the formally defined notions of open abstract and physical models.

    1. open abstract models which describe the behavior of the application software and the expected behavior of the environment using Input/Output automata. Inputs and outputs are used to represent explicitly the communications between the application and its environment. This leads to models in which components can only access their local data which is an essential hypothesis of component-based design.
    2. open physical models which are abstract models equipped with takes into account the actual arrival of inputs from the external environment. The platform specific code used for implementing the communications with the environment is thus moved from the model to the real-time Engine that safely handles these communications, which enables for checking online for misbehavior.

    We think that our approach can be useful for implementing adaptive systems, that is, systems that adapt their behavior in accordance with the actual execution on the platform (i.e. with execution times, energy consumption, available resources, etc.) and/or the behavior of the environment.

Experimental results show the enhancements in term of performances obtained by the use of these mechanisms. We applied the method for the design and implementation of autonomous systems. We conducted successful experiments on the DALA robot from the LAAS laboratory. With our approach, one can provide guarantees that the robot will not perform actions that may lead to unsafe situations, those that may eventuate in undesired

or catastrophic consequences and that timing constraints violations can be detected at execution. We used the real-time version of the BIP framework that we have developed, which takes into account execution time deadlines. We also used BIP as a temporal-plan execution controller for the decisional level. Using a BIP model combining plans of the decisional level and the functional level should increase the performance of the implementations since costly communications between these two levels can be expressed as BIP interactions in which are directly handled by the engine in the same process. More importantly, this can also be used for verifying global properties involving both the decisional level and the functional level.

2 Perspectives

For future work, we are considering several research directions.

- We have given results on time-safety and time-robustness that allow a deeper understanding of causes of anomalies. They advocate for time-determinism as a means for achieving time-robustness. An interesting question is loss in performance when in a model interval constraints are replaced by equalities on their upper bound. Time-robustness is then achieved through time-determinization at some performance penalty. We are currently studying the loss of performance induced by this transformation.

- We also want to provide static analysis tools to verify if for a given abstract model and execution platform specifications, the corresponding physical model is time-safe and time-robust.

- Currently, powerful hardware platforms needed for executing critical systems are multi-core or many-core platforms. The application code should be optimally distributed over the platform to take advantage of its computing power. Although distributed systems are widely used nowadays, their implementation is still time-consuming and an error prone process. Coordination in BIP is achieved through multi-party interactions (i.e. those accross multiple components), and scheduling by using dynamic priorities. Transforming the semantics of the actual untimed distributed BIP version into a distributed timed implementation is clearly a non trivial task. Therefore, it constitutes one of our futur work direction.

- An other work direction, is the issue of mixed critical systems [9,10]. Mixed criticality is the concept of allowing applications at different levels of criticality to interact and co-exist on the same computational platform. Unfortunately, certification of such systems is more difficult, because it requires that even the components of less criticality be certified at the highest criticality level. It is necessary to provide strong scheduling theories for real-time and safety-critical system design and implementation.

List of Figures

Bibliography

[1] *IEEE Fourth International Symposium on Industrial Embedded Systems - SIES 2009, Ecole Polytechnique Federale de Lausanne, Switzerland, July 8 - 10, 2009*. IEEE, 2009.

[2] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[3] James F. Allen. Towards a general theory of action and time. *Artif. Intell.*, 23(2):123–154, 1984.

[4] Karine Altisen and Stavros Tripakis. Implementation of timed automata: An issue of semantics or modeling? In Pettersson and Yi [75], pages 273–288.

[5] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[6] Rajeev Alur and David L. Dill. The theory of timed automata. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 45–73. Springer, 1991.

[7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[8] Christophe Aussaguès and Vincent David. A method and a technique to model and ensure timeliness in safety critical real-time systems. In *ICECCS*, pages 2–12. IEEE Computer Society, 1998.

[9] Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. Mixed-criticality scheduling of sporadic task systems. In Camil Demetrescu and Magnús M. Halldórsson, editors, *ESA*, volume 6942 of *Lecture Notes in Computer Science*, pages 555–566. Springer, 2011.

[10] Sanjoy K. Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In Marco Caccamo, editor, *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 13–22. IEEE Computer Society, 2010.

[11] Ananda Basu. *Component-based Modeling of Heterogeneous Real-time Systems in BIP*. PhD thesis, UJF, 2008.

[12] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In Suzuki et al. [86], pages 116–133.

[13] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In Suzuki et al. [86], pages 116–133.

[14] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12. IEEE Computer Society, 2006.

[15] Ferenc Belina and Dieter Hogrefe. The ccitt-specification and description language sdl. *Computer Networks*, 16:311–341, 1989.

[16] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In Bouajjani and Maler [32], pages 614–619.

[17] Saddek Bensalem, Lavindra de Silva, Félix Ingrand, and Rongjie Yan. Towards a more dependable software architecture for autonomous robots. *IEEE Robotics and Automation Magazine*, 16(1), March 2009.

[18] Saddek Bensalem, Lavindra de Silva, Félix Ingrand, and Rongjie Yan. A verifiable and correct-by-construction controller for robot functional levels. *Journal of Software Engineering for Robotics*, 16(1):123–126, September 2009.

[19] Saddek Bensalem, Matthieu Gallien, Félix Ingrand, Imen Kahloul, and Thanh-Hung Nguyen. Designing autonomous robots. *IEEE Robotics and Automation Magazine*, 16(1):66–77, March 2009.

[20] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. D-finder 2: Towards efficient correctness of incremental design. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 453–458. Springer, 2011.

[21] Saddek Bensalem, Félix Ingrand, and Joseph Sifakis. Autonomous robot software design challenge. In *IARP/IEEE-RAS Joint Workshop on Technical Challenge for Dependable Robots in Human Environments*, Pasadena, CA, May 2008.

[22] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[23] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[24] Gordon S. Blair, Thierry Coupaye, and Jean-Bernard Stefani. Component-based architecture: the fractal initiative. *Annales des Télécommunications*, 64(1-2):1–4, 2009.

[25] Simon Bliudze and Joseph Sifakis. The algebra of connectors - structuring interaction in bip. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.

[26] Simon Bliudze and Joseph Sifakis. A notion of glue expressiveness for component-based systems. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 508–522. Springer, 2008.

[27] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. In *Proceedings of the tenth ACM international conference on Embedded software (EMSOFT'10)*, pages 209–218, 2010.

[28] Michael Bonani, Valentin Longchamp, Stéphane Magnenat, Philippe Rétornaz, Daniel Burnier, Gilles Roulet, Florian Vaussard, Hannes Bleuler, and Francesco Mondada. The marxbot, a miniature mobile robot opening new perspectives for the collective-robotic research. In *IROS*, pages 4187–4193. IEEE, 2010.

[29] Rafael H. Bordini, Michael Fisher, Carmen Pardavila, Willem Visser, and Michael Wooldridge. Model checking multi-agent programs with casp. In *CAV*, pages 110–113, 2003.

[30] Sébastien Bornot, Gregor Gößler, and Joseph Sifakis. On the construction of live timed systems. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 109–126. Springer, 2000.

[31] Sébastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Inf. Comput.*, 163(1):172–202, 2000.

[32] Ahmed Bouajjani and Oded Maler, editors. *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*. Springer, 2009.

[33] F. Boussinot and R. de Simone. The ESTEREL Language. *Proceeding of the IEEE*, pages 1293–1304, September 1991.

[34] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-source architecture transformation for performance optimization in bip. In *SIES* [1], pages 152–160.

[35] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-source architecture transformation for performance optimization in BIP. In *SIES* [1], pages 152–160.

[36] M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner. Model-based Testing of Reactive Systems. *Lecture Notes in Computer Science*, 3472, 2005.

[37] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.

[38] H Bruyninckx. Open robot control software: the orocos project. In *ICRA*, Seoul, Korea, 2001.

[39] Alan Burns and Andy J. Wellings. *Real-time systems and their programming languages*. Addison-Wesley, 3rd edition, 2001.

[40] Ana R. Cavalli and Amardeo Sarma, editors. *SDL '97 Time for Testing, SDL, MSC and Trends - 8th International SDL Forum, Evry, France, 23-29 September 1997, Proceedings*. Elsevier, 1997.

[41] Damien Chabrol, Vincent David, Christophe Aussaguès, Stéphane Louise, and Frédéric Daumas. Deterministic distributed safety-critical real-time systems within the oasis approach. In S. Q. Zheng, editor, *IASTED PDCS*, pages 260–268. IASTED/ACTA Press, 2005.

[42] Jacques Combaz, Jean-Claude Fernandez, Joseph Sifakis, and Loïc Strus. Symbolic quality control for multimedia applications. *Real-Time Systems*, 40(1):1–43, 2008.

[43] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In Karl Henrik Johansson and Wang Yi, editors, *HSCC*, pages 91–100. ACM ACM, 2010.

[44] Vincent David, Jean Delcoigne, Evelyne Leret, Alain Ourghanlian, Philippe Hilsenkopf, and Philippe Paris. Safety properties ensured by the oasis model for safety critical real-time systems. In Wolfgang D. Ehrenberger, editor, *SAFECOMP*, volume 1516 of *Lecture Notes in Computer Science*, pages 45–59. Springer, 1998.

[45] Catalin Dima. Dynamical properties of timed automata revisited. In Jean-François Raskin and P. S. Thiagarajan, editors, *FORMATS*, volume 4763 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2007.

[46] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia R. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[47] B. Espiau, K. Kapellos, and M. Jourdan. Formal verification in robotics: Why and how. In *The International Foundation for Robotics Research, editor, The Seventh International Symposium of Robotics Research*, pages 201 – 213, Munich, Germany, October 1995. Cambridge Press.

[48] Peter H. Feiler, Bruce A. Lewis, and Steve Vestal. The SAE architecture analysis & design language (AADL) a standard for engineering performance critical systems. In *IEEE International Symposium on Computer-Aided Control Systems Design*, pages 1206–1211, 2006.

[49] David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI*, pages 1–11. ACM, 2003.

[50] Thomas Genssler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter O. Müller, and Christian Stich. Components for embedded software: the pecos approach. In Shuvra S. Bhattacharyya, Trevor N. Mudge, Wayne Wolf, and Ahmed Amine Jerraya, editors, *CASES*, pages 19–26. ACM, 2002.

[51] Arkadeb Ghosal, Thomas A. Henzinger, Christoph M. Kirsch, and Marco A. A. Sanvido. Event-driven programming with logical execution times. In Rajeev Alur and George J.

Pappas, editors, *HSCC*, volume 2993 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2004.

[52] Robert P. Goldman, David J. Musliner, and Michael J. Pelican. Using model checking to plan hard real-time controllers. In *Proceedings of the AIPS Workshop on Model-Theoretic Approaches to Planning*, 2000.

[53] Gregor Gössler and Joseph Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.

[54] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305 –1320, sep 1991.

[55] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[56] Nicolas Halbwachs. About synchronous programming and abstract interpretation. *Sci. Comput. Program.*, 31(1):75–89, 1998.

[57] Nicolas Halbwachs. Synchronous programming of reactive systems. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[58] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[59] F. Ingrand, S. Lacroix, S. Lemai, and F. Py. Decisional autonomy of planetary rovers. *Journal of Field Robotics*, 24(7):559–580, 2007.

[60] Damir Isovic, Gerhard Fohler, and Liesbeth Steffens. Timing constraints of MPEG-2 decoding for high quality video: misconceptions and realistic assumptions.

[61] Mohamad Jaber. *Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP*. PhD thesis, Grenoble Universités, 2010.

[62] Mohamad Jaber. *Implémentations Centralisés et Réparties de Systèmes Corrects par construction à base des Composants par Transformations Source-à-source dans BIB*. PhD thesis, Université de Grenoble, 2010.

[63] J Jackson. Microsoft robotics studio: A technical introduction. *IEEE RAM*, 14(4):82–87, 2007.

[64] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[65] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. *Auton Robot*, Jan 2007.

[66] H. Kress-Gazit and G.J. Pappas. Automatic synthesis of robot controllers for tasks with locative prepositions. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3215 –3220, May 2010.

[67] Xiaojun Liu, Yuhong Xiong, and Edward A. Lee. The ptolemy ii framework for visual languages. In *HCC*, pages 50–. IEEE Computer Society, 2001.

[68] Stéphane Louise, Vincent David, Jean Delcoigne, and Christophe Aussaguès. Oasis project: deterministic real-time for safety critical embedded systems. In *EW 10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 223–226, New York, NY, USA, 2002. ACM.

[69] Stéphane Louise, Matthieu Lemerre, Christophe Aussaguès, and Vincent David. The oasis kernel: A framework for high dependability real-time systems. In Taghi M. Khoshgoftaar, editor, *HASE*, pages 95–103. IEEE Computer Society, 2011.

[70] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[71] Stéphane Magnenat, Basilio Noris, and Francesco Mondada. Aseba-challenge: An open-source multiplayer introduction to mobile robots programming. In Panos Markopoulos, Boris E. R. de Ruyter, Wijnand IJsselsteijn, and Duncan Rowland, editors, *Fun and Games*, volume 5294 of *Lecture Notes in Computer Science*, pages 65–74. Springer, 2008.

[72] M Montemerlo, N Roy, and S Thrun. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit. In *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pages 2436–2441, Las Vegas, NV, 2003.

[73] Issa A Nesnas, Anne Wright, Max Bajracharya, Reid Simmons, and Tara Estlin. Claraty and challenges of developing interoperable robotic software. In *IROS*, Las Vegas, NV, Oct 2003. invited paper.

[74] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew P. Black, Peter O. Müller, Christian Zeidler, Thomas Genssler, and Reinier van den Born. A component model for field devices. In Judy M. Bishop, editor, *Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2002.

[75] Paul Pettersson and Wang Yi, editors. *Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings*, volume 3829 of *Lecture Notes in Computer Science*. Springer, 2005.

[76] Juraj Polakovic, Ali Erdem Özcan, and Jean-Bernard Stefani. Building reconfigurable component-based os with think. In *EUROMICRO-SEAA*, pages 178–185. IEEE, 2006.

[77] M Quigley, B Gerkey, K Conley, J Faust, T Foote, J Leibs, E Berger, R Wheeler, and A Ng. Ros: an open-source robot operating system. In *International Conference on Robotics and Automation*, Kobe, Japan, 2009.

[78] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In Frank Mueller, editor, *WCET*, volume 06902 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[79] Grigore Rosu and Saddek Bensalem. Allen linear (interval) temporal logic - translation to ltl and monitor synthesis. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2006.

[80] Vassiliki Sfyrla. *Modélisation des Systemes Synchrones sur BIP*. PhD thesis, University of Grenoble, 2011.

[81] Azamat Shakhimardanov and Erwin Prassler. Comparative evaluation of robotic software integration systems: A case study. In *IROS*, page 7, San Diego, CA, Sep 2007.

[82] Joseph Sifakis. Component-based construction of real-time systems in bip. In Bouajjani and Maler [32], pages 33–34.

[83] Joseph Sifakis. Embedded systems design - scientific challenges and work directions. In *DATE*, page 2. IEEE, 2009.

[84] Joseph Sifakis. Embedded systems design - scientific challenges and work directions. In Roderick Bloem and Natasha Sharygina, editors, *FMCAD*, page 11. IEEE, 2010.

[85] R. Simmons, C. Pecheur, and G. Srinivasan. Towards automatic verification of autonomous systems. In *IEEE/RSJ International conference on Intelligent Robots & Systems*, 2000.

[86] Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors. *Formal Techniques for Networked and Distributed Systems - FORTE 2008, 28th IFIP WG 6.1 International Conference, Tokyo, Japan, June 10-13, 2008, Proceedings*, volume 5048 of *Lecture Notes in Computer Science*. Springer, 2008.

[87] R Vaughan and B Gerkey. Reusable robot software and the player/stage project. *Software Engineering for Experimental Robotics*, pages 267–289, 2007.

[88] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2010.

[89] B. C. Williams, M. D. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. T. Sullivan. Model-Based Programming of Fault-Aware Systems. *Artificial Intelligence*, pages 61–75, winter 2003.

[90] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, HSCC '10, pages 101–110, New York, NY, USA, 2010. ACM.

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