Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Mahieddine Dellabani

Thèse dirigée par **Saddek Bensalem**, et
coencadrée par **jacques Combaz**

préparée au sein du **Laboratoire Verimag**
dans **l'École Doctorale Mathématique, Sciences et Technologies de l'information (MSTII)**

Formal Methods For Distributed Real-Time Systems

Thèse soutenue publiquement le **01/10/2018**,
devant le jury composé de :

Mme, Ahmed Lbath
Professeur, Université Grenoble Alpes, Président
M, Stavros Tripakis
Professeur, Université de Aalto, Rapporteur
M, Oleg Sokolsky
Professeur, Université de Pennsylvanie, Rapporteur
M, Eugene Asarin
Professeur, Université Patis 7, Membre
M, Yamine Ait-Ameur
Professeur, INP Toulouse – ENSEEIHT, Membre
M, Saddek Bensalem
Professeur, Université Grenoble Alpes, Directeur de thèse
M, Jacques Combaz
Ingénieur de recherche, CNRS, Coencadrant de thèse

Formal Methods for Distributed Real-Time Systems

Dellabani Mahieddine

Laboratoire Verimag
Université Grenoble Alpes

This dissertation is submitted for the degree of
Doctor of Philosophy

Ecole Doctorale Mathématique,
Sciences et Technologies de
l'Information, Informatique (MSTII)

October 2018

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Saddek Bensalem for giving me the opportunity to carry out my PhD at the Verimag laboratory, for providing ideal research conditions and for all his trust and support during this period. I would like to deepely thank my co-advisor Dr. Jacques Combaz for his continuous support, patience, motivation, and immense knowledge. His guidance as well as the tremendous amount of time spent with me all along my PhD are pricesless. I want to thank Dr. Marius Bozga for his precious suggestions, advices and for sharing with me all his knowledge in formal methods.

I would like to thank Prof. Stavros Tripakis and Prof. Oleg Sokolsky for accepting to review this thesis. Many thanks to Prof. Florence Maraninchi, Prof. Patricia Bouyer-Decitre for accepting to be part of the jury.

This journey would not have been possible without my family. To my parents, who always favored their children without asking or expecting anything in return. Thank you for all your encouragement, assistance and for the emotional and financial supports. Special thanks to my sisters for all the time and efforts spent while trying to cheer me up during difficult moments. Special thoughts to my maternal grandmother who passed away during my first year of PhD.

My sincere thanks to Lotfi Mediouni for all the time spent together during coffee breaks, for his help and especially for his optimism (I think I have never seen a person that optimistic in my life). Special thanks to all my colleagues Hosein Nazarpour, Ayoub Nouri, Rany Kahil, Rim El-Belouli and all my Verimag collegues.

Last but not least, thanks you my beloved Anne for her continuous support, and to her familly for all their kindness and the endless invitations to family dinners, christmass and every other occasion. They made me feel I am home.

Abstract

Nowadays, real-time systems are ubiquitous in several application domains. Such an emergence led to an increasing need of performance (resources, availability, concurrency, etc.) and initiated a shift from the use of single processor based hardware platforms, to large sets of interconnected and distributed computing nodes. This trend introduced the birth of a new family of systems that are intrinsically distributed, namely *Networked Embedded Systems*. Such an evolution stems from the growing complexity of real-time software embedded on such platforms (e.g. electronic control in avionics and automotive domains), and the need to integrate formerly isolated systems so that they can cooperate, as well as, share resources improving thus functionalities and reducing costs. Undoubtedly, the design, implementation and verification of such systems are acknowledged to be very hard tasks since they are prone to different kinds of factors, such as communication delays, CPU(s) speed or even hardware imprecisions, which increases considerably the complexity of coordinating parallel activities.

In this thesis, we propose a rigorous design flow intended for building distributed real-time applications. We investigate timed automata based models, with well defined semantics, in order to study the behavior of a given system with some imposed timing constraints when deployed in a distributed environment. Particularly, we study *(i)* the impact of the communication delays by introducing a minimum latency between actions executions and the effective date at which actions executions have been decided, and *(ii)* the effect of hardware imperfections, more precisely clocks imprecisions, on systems execution by breaking the perfect clocks hypothesis, often adopted during the modeling phase. Nevertheless, timed automata formalism is intended to describe a high level abstraction of the behavior of a given application. Therefore, we use an intermediate representation of the initial application, that besides having "equivalent" behavior, explicitly expresses implementation mechanisms, and thus reduces the gap between the modeling and the concrete implementation. Additionally, we contribute in building such systems by *(iii)* proposing a knowledge based optimization method that aims to eliminate unnecessary computation time or exchange of messages during the execution.

We compare the behavior of each proposed model to the initial high level model and study the relationships between both. Then, we identify and formally characterize the potential problems resulting from these additional constraints. Furthermore, we propose execution strategies that allow to preserve some desired properties and reach a "similar" execution scenario, faithful to the original specifications.

Résumé

Aujourd'hui, les systèmes temps réel sont omniprésents dans plusieurs domaines. Une telle expansion donne lieu à un besoin croissant en terme de performance (ressources, disponibilité, parallélisme, etc.) et a initié par la même occasion une transition de l'utilisation de plateformes matérielles à processeur unique, à de grands ensembles de nœuds de calcul inter-connectés et distribués. Cette tendance a introduit la naissance d'une nouvelle famille de systèmes connue sous le nom de *Networked Embedded Systems*, qui sont intrinsèquement distribués. Une telle évolution provient de la complexité croissante des logiciels temps réel embarqués sur de telles plateformes (par exemple les système de contrôle en avionique et dans domaines de l'automobile), ainsi que la nécessité d'intégrer des systèmes autrefois isolés afin d'accomplir les fonctionnalités requises, améliorant ainsi les performances et réduisant les coûts. Sans surprise, la conception, l'implémentation et la vérification de ces systèmes sont des tâches très difficiles car elles sont sujettes à différents types de facteurs, tels que les délais de communication, la fréquence du CPU ou même les imprécisions matérielles, ce qui augmente considérablement la complexité de coordonner les activités parallèles.

Dans cette thèse, nous proposons un workflow rigoureux destiné à la construction d'applications distribuées temps réel. Pour ce faire, nous étudions des modèles basés sur les automates temporisés, avec une sémantique bien définie, afin d'étudier le comportement d'un système donné avec des contraintes de temps imposées lorsqu'il est déployé dans un environnement distribué. En particulier, nous étudions *(i)* l'impact des délais de communication en introduisant une latence minimale entre les exécutions d'actions et la date à laquelle elles ont été décidées, et *(ii)* l'effet des imperfections matérielles, plus précisément les imprécisions d'horloges, sur l'exécution des systèmes. Le paradigme des automates temporisés reste néanmoins destiné à décrire une abstraction du comportement d'une application donnée. Par conséquent, nous utilisons une représentation intermédiaire de l'application initiale, qui en plus d'avoir un comportement "équivalent", exprime explicitement les mécanismes mis en œuvre durant l'implémentation, et donc réduit ainsi l'écart entre la modélisation et l'implémentation réelle. De plus, nous contribuons à la construction de tels systèmes en *(iii)* proposant une optimisation basée sur la *connaissance*, qui a pour but d'éliminer les temps de calcul inutiles et de réduire les échanges de messages pendant l'exécution.

Nous comparons le comportement de chaque modèle proposé au modèle initial et étudions les relations entre les deux. Ensuite, nous identifions et caractérisons formellement les problèmes

potentiels résultants de ces contraintes supplémentaires. Aussi, nous proposons des stratégies d'exécution qui permettent de préserver certaines propriétés souhaitées et d'obtenir des scénarios d'exécution "similaires", et fidèles aux spécifications de départs.

Contents

1 Introduction **9**
- 1.1 Real-Time Systems . . . 9
- 1.2 Control Aspects in Real-Time Systems . . . 11
  - 1.2.1 Temporal Control Versus Logical Control . . . 11
  - 1.2.2 Event-Triggered Control Versus Timed-Triggered Control . . . 11
- 1.3 System Design . . . 12
  - 1.3.1 Design Flows . . . 12
  - 1.3.2 Design Styles . . . 12
- 1.4 Existing Approaches For Building Real-Time Systems . . . 14
  - 1.4.1 Giotto . . . 14
  - 1.4.2 Oasis . . . 15
  - 1.4.3 PTIDES . . . 16
  - 1.4.4 Lustre . . . 17
- 1.5 Contribution . . . 17
- 1.6 Outline . . . 19

I Preliminaries **21**

2 Timed Systems and Semantics **23**
- 2.1 Timed Transition Systems . . . 23
  - 2.1.1 Comparing Timed Transition Systems . . . 24
  - 2.1.2 Reactive Timed Systems . . . 25
- 2.2 Timed Systems Syntax and Semantics . . . 25
  - 2.2.1 Timed Components Syntax and Semantics . . . 26
  - 2.2.2 Timed Systems . . . 30
  - 2.2.3 Example . . . 32
- 2.3 Timed Systems with Data . . . 33
- 2.4 Verification of Timed Systems . . . 34
  - 2.4.1 Symbolic Reachability . . . 34
  - 2.4.2 Compositional Verification . . . 35

3 Modeling Distributed Real-Time Systems **37**
- 3.1 Target Architecture . . . 38
  - 3.1.1 Interface . . . 38
  - 3.1.2 From Local Time to Global Time . . . 39
  - 3.1.3 Conflicting Interactions and Interaction Partitioning . . . 39
- 3.2 3-Layer Send/Receive Model . . . 40
  - 3.2.1 Send/Receive Components . . . 41
  - 3.2.2 Scheduling Layer . . . 44

3.2.3 Conflict Reservation Protocol . . . 48
3.3 Modeling Distributed Real-Time Constraints . . . 51
3.3.1 Communication Delays . . . 51
3.3.2 Clock Drift . . . 52

II Contribution **55**

4 Knowledge Based Optimization of Distributed Real-Time Systems **57**
4.1 Conflicting Interaction Calculation . . . 57
4.2 Knowledge Based Reduction of Potentially Conflicting Interaction . . . 58
4.2.1 Linear Invariants . . . 59
4.2.2 History Clocks Inequalities . . . 60
4.3 Impact of Conflict Reduction on Send/Receive Models . . . 62

5 The Local Planning Semantics **65**
5.1 Local Planning of Interactions . . . 66
5.1.1 Definition of the LPS . . . 66
5.1.2 Properties of the LPS . . . 68
5.2 Enforcing Deadlock-Free Planning . . . 71
5.3 Planning Semantics as Real-Time Controller Synthesis . . . 74
5.3.1 Planning Zones . . . 74
5.3.2 Infinite Planning Transitions . . . 75
5.3.3 Discussion . . . 79

6 Clock Drift **81**
6.1 Distributed Timed Automata . . . 81
6.1.1 Expressing Clock Constraints Using Local Clock . . . 81
6.1.2 Distributed Component . . . 82
6.1.3 Correctness . . . 83
6.2 Robust Distributed Semantics . . . 85
6.3 Related Work . . . 86

7 Implementation and Experimentation **89**
7.1 The BIP Framework . . . 89
7.2 The BIP Toolbox . . . 93
7.3 Tools Developed in this Thesis . . . 95
7.4 Experimentation . . . 97

III Conclusion **99**

8 Conclusion and Perspectives **101**

List of Figures **103**

List of Tables **105**

Bibliography **107**

Chapter 1

Introduction

1.1 Real-Time Systems

Computer systems consist of a combination of hardware, software, user and data. Such systems are constantly evolving and will become undoubtedly a constituent part of our daily lives. Nowadays, computer systems are widespread in several application domains. From simple systems like espresso machines to complex larger systems such as an airplane, these systems are continuously developing to facilitate and enhance our lifestyle by always providing improved solutions (more comfort, energy efficient, security, etc.) that answer directly to real life problems. *Real-time* systems are defined as those systems in which the correctness of the system depends not only on the logical results of computations, but also on the physical time at which the results are produced [Sta88]. In other terms, real-time systems are computing systems that have temporal constraints to meet and thus, are required to guarantee a response within specified timing constraints. Such systems need to achieve different requirements that can be structured into three main categories [Kop11]: *(i)* functional requirements, *(ii)* temporal requirements, and *(iii)* dependability requirements. Functional requirements refer to the functions that the real-time system must perform. In order to accomplish such requirements, the system needs to be able to observe its current state, that is, the values of variables describing the latter. For instance the position, speed and level of oil in a car are possible state variables. Additionally, a given system needs to control its state variables (actuation) for regulation purposes, and provide an interface to the system operator that allows its monitoring. On the other hand, temporal requirements are born from the requirements of control loops. A typical example is the time bounds for which a gear change needs to be achieved in the control system operating in modern vehicles. Temporal requirements of a given system might be of different order of magnitude. For instance the man-machine interface, in comparison with fast control loops, is less strict because of the human perception of time. Finally, dependability requirements are additional requirements regarding the quality of service that a given system produces, that is, the criteria for deciding whether the services provided by the system can justifiably be trusted. The concept of dependability encompasses the following attributes [ALRL04]: availability (readiness for correct services), reliability (continuity of correct services), safety (absence of catastrophic consequences), integrity (absence of improper system alterations), and maintainability (ability to undergo modifications and repairs).

Real-time systems can be classified from different perspectives [Kop11] based on either the characteristics of the application (perspective A), i.e., on factors outside the computer system,

or on the design and implementation (perspective B), i.e., on factors inside the computer system. Tables 1.1 presents a classification of real-time systems based on these two perspectives:

Perspective	Classification	Description
A	hard real-time vs soft real-time	A hard real-time system is a system that must meet *hard deadlines*, that is, deadlines that if missed could result in severe consequences. On the other hand, for soft real-time systems the goal becomes to meet a certain subset of deadlines in order to optimize some specific criteria. Particularly, deadline violations result in degraded quality. However, the system keeps operating and may recover in the future.
	fail-safe vs fail-operational	Fail-safe systems refer to those systems where a safe state can be identified and quickly reached upon the occurrence of a failure. On the contrary, fail-operational systems are systems that must remain operational and provide a minimal level of service even in case of failure (no safe state).
B	guaranteed-response vs best effort	Real-time systems where the probability of failure is reduced to the probability that some assumptions, on peak load or fault number and types for instance, do not hold in reality are called guaranteed-response. On the other hand, best-effort systems are systems where no analytical arguments for correctness can be made. The latter is rather established during the test and integration phases.
	resource-adequate vs resource-inadequate	While Resource-adequate systems are systems that provide sufficient computational resources to handle a specified peak load and fault scenarios, resource-inadequate systems rely on dynamic resource allocation strategy. It is based on resource sharing and probabilistic arguments about expected load and fault scenarios.
	event-triggered vs time-triggered	Event-triggered systems are systems where all communication and processing activities are initiated whenever a significant event (rather than the regular clock tick) occurs. Oppositely, in time-triggered systems all activities are initiated by the progress of real-time.

Table 1.1 – Classification Of Real-Time Systems

1.2 Control Aspects in Real-Time Systems

1.2.1 Temporal Control Versus Logical Control

When building real-time systems, a clear distinction between the aspects related to the time domain and those related to the value domain needs to be made. The latter are respectively referred to as the concepts of *temporal control* and *logical control*. Temporal control is related to the progression of *real-time*. It is of concern when it comes to determining the instants in the domain of real time for the accomplishment of computations, that is, when computations must be activated. Such instants are derived from the dynamic of the application.

Logical control is related to the control flow *within* a given task, meaning that it is based on the given task structure and the particular input data. For instance, the execution of a branch condition and the selection of one of the alternatives is an example of logical control. The *execution time* represents the time interval needed to execute a task performing the logical control. If the temporal control and the logical control aspects are mixed in a program segment, then it is impossible to determine the worst-case execution time of this program segment without analyzing the behavior of its environment.

Example 1.2.1. A semaphore wait statement is a temporal control statement. If a semaphore wait statement is contained in a program segment that also includes logical control (algorithmic) statements, then the temporal behavior of this program segment depends on both, the progress of execution time and the progress of real-time.

Thus, a good design decouples the reasoning about temporal constraints, governed by the application, from the reasoning about logical inner aspects of the algorithmic part of the application. Synchronous real-time languages, such as LUSTRE [HLR92] and ESTEREL [BG92] distinguish clearly between logical control and temporal control. In these languages, the progression of real-time is partitioned into an (infinite) sequence of intervals of specified real-time duration, which we call steps. Each step begins with a tick of a real-time clock that starts a computational task (logical control). The computational model assumes that a task, once activated by the tick of a real-time clock (temporal control), finishes its computation quasi immediately. Practically, this means that a task must terminate its execution before the next triggering signal (the next tick of the real-time clock) initiating the next execution of the task.

1.2.2 Event-Triggered Control Versus Timed-Triggered Control

In real-time systems, the activation (start-up) of computation tasks is usually achieved through triggering signals, that is, control signals that specify the instant when an activity (computation) should start in the temporal domain. Two main paradigms are commonly used for triggering activities of a real-time system, namely, the event-triggered control and the time-triggered control. We say that the control is event-triggered when the triggering signal is associated to the occurrence of a significant event, such as the arrival of a particular message, the fulfillment of an activity within a component, or the occurrence of an external interrupt. Time-triggered control relies on the progression of time. The activation of computation is pre-determined and usually depends on the periodic overflow of a timer. Time-triggered systems are based on the notion of *logical execution time (LET)* introduced by the programming language Giotto [HHK03]. LET specifies the amount of time between the activation time of a computation and its due time. It abstracts the execution time of programs in the sense that even if they start before their

activation date and finish before their due time, they behave as if they exactly consumed their LET.

1.3 System Design

System design is the process of defining the elements of a system such as the requirements, the architecture, the system components and their interfaces as well as the data flowing through the system. System design implies a systematic approach to the design of a system. It may adopt a bottom-up or a top-down approach, but either way the process is systematic wherein it takes into account all related variables of the system that need to be created. From the architecture, to the required hardware and software, right down to the data and how it travel and transform throughout its travel through the system. System design, then, overlaps with system analysis, system engineering and system architecture. The system design approach first appeared right before World War II, when engineers were trying to solve complex control and communication problems. They needed to be able to standardize their work into a formal discipline with proper methods, especially for new fields like information theory, operational research and computer science in general.

System design [BBB+11a] differs radically from pure software design in the sense that it does not focus only on functional requirements. It also accounts for nonfunctional properties such as timing, energy consumption, or even fault tolerance. Meeting such extra functionalities is essential and requires evaluation of how design choices affect the overall system behavior.

1.3.1 Design Flows

Design of real-time systems is a rather complex task [Mar11] which has to be broken down into a number of subtasks. In theory, the design process should be structured into distinct phases starting from the analysis phase until the final validation of the system and its effective deployment. In practice, such a strict sequential decomposition is hardly possible. In fact the design flow is rather decomposed into several iterations as depicted in Figure 1.1. Each iteration needs to include test generation and an evaluation of testability. A given design is then evaluated with respect to the different objectives (performance, dependability, energy consumption, etc.).

1.3.2 Design Styles

In what follows, we present some important design styles used in the design of real-time systems. Notice that the presented design styles are not exclusive in the sense that a given approach may combine several design styles.

Model-Based Design

Model-based design [JCL11, KSLB03] is one promising approach in building real-time systems. It relies on mathematical modeling for the design, analysis and validation of systems. This approach captures not only what the dynamic and the expected properties of the system are, but also what is assumed about the system's environment. Thus, It enables developers to verify the logic of their application, assumptions about its environment and end-to-end behavior at early stages of the design cycles. First, an application model expressing the behavior and dynamic

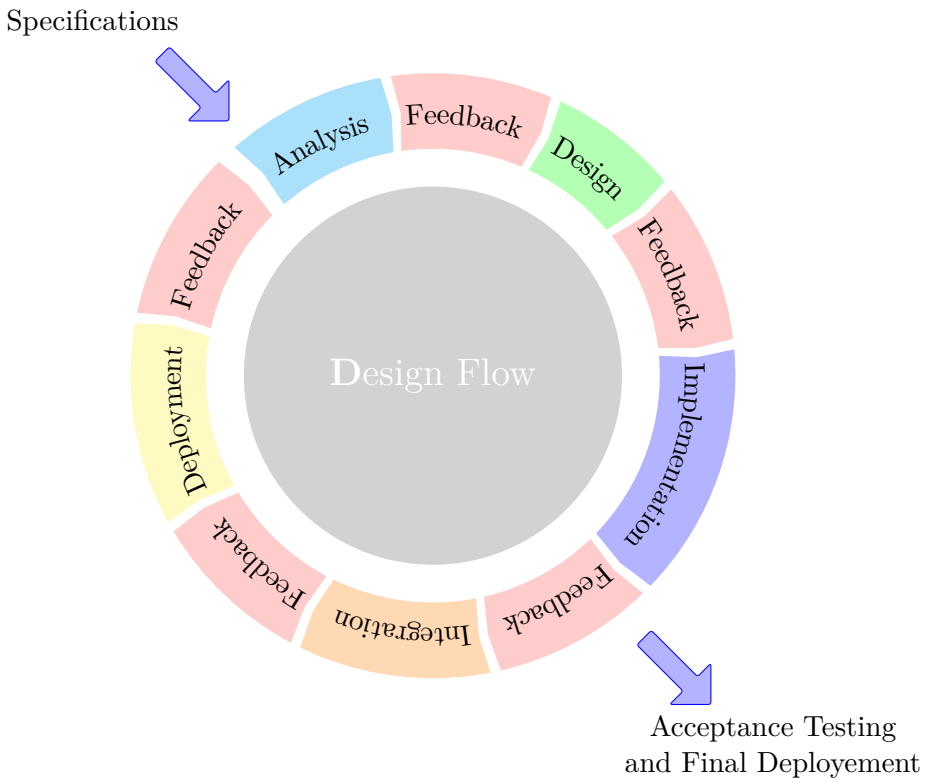


Figure 1.1 – Overal Design Flow

of the considered system is built. It represents an abstraction that is platform independent, meaning that it does not consider any hardware specification such as communication delays or CPU(s) speed, which allows to model the system at early stages without any knowledge of the target platform, verify the obtained model against some safety properties (functional requirements), or even to synthesize a control strategy for tuning the system. Thereafter, the application source code, which represents the actual implementation of the system on a given platform, is automatically generated from the high level model, and integrated in a simulated environment.

Component-Based Design

Large scale systems are complex systems that usually require to assemble several components with wide-range functionalities. Many issues may rise when designing such systems [LS11] ranging from the design process and the relationships with suppliers to incomplete specifications and testing. A typical example is the Toyota sticky accelerator problem that was caused in part by components provided by two contractors whose interaction was not verified appropriately. Component-based design tackles these issues by proposing an approach where such systems are build by assembling strongly encapsulated entities called *component* with stable, well-defined, and rigorous interface specifications [Kop10]. This approach is based on the principle of re-usability of heterogeneous components and focuses on the idea that internal knowledge about the design or the implementation of the latter is not needed. In some cases, this knowledge is not even available.

Architecture-Based Design

A system architecture is a conceptual model that describes the structure, behavior, and constraints of a system along with the interaction with its environment. Designing the architecture of a system is acknowledged to be a very hard task since it must address practical concerns of the engineering efforts [DOL03] involved in system development (short term) and evolution (long term). Moreover, it needs to support a set of requirements whose details may be unknown until an advanced late step of the development process. Thus, a methodical design approach that provides means for coping with requirements uncertainty, and proposes guidance that helps in the decision making during the design process, is needed. Architecture-based design [BBC+00] helps not only to detect design errors early in the development process, but also supports engineering efforts, which allows to produce high-quality code, by addressing the aforementioned problems.

1.4 Existing Approaches For Building Real-Time Systems

1.4.1 Giotto

Giotto [HHK03] is a programming approach for representing embedded systems at the architecture level using a timed-triggered programming language. It separates the platform-independent functionalities and timing aspects from platform-dependent scheduling and communication issues. It allows thus, an intermediate abstraction that enables the design engineers to annotate the functional programming modules with temporal attributes that are derived from the high-level stability analysis of control loops. Figure 1.2 depicts the workflow of Giotto. A platform-abstract Giotto program is written. This program captures all the functionalities and the timing aspects necessary for ensuring consistency with the mathematical model of the control design. Thereafter, the program is implemented on a given platform. Notice that this step is completely decoupled from the first step in the sense that it does not require any interaction with the control engineer, and can in large parts be automated using powerful compilers. Giotto compilation facilitates the evaluation and the optimization phases by guarantying the preservation of functionalities and timings. In Giotto, all data are communicated through ports

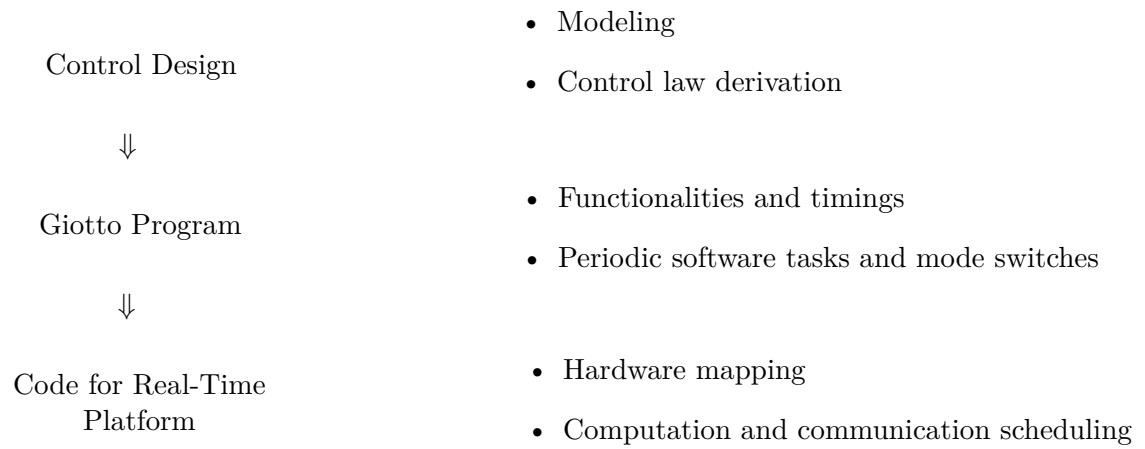


Figure 1.2 – Giotto Workflow

that can be classified into three categories: sensor ports, actuator ports and task ports. Sensor ports are updated by the environment, whereas all other ports are updated by the Giotto program. Task ports are used to communicate data between concurrent tasks or to transfer data from one mode (execution mode) to another. A Giotto task consists of a set of `In` (input) ports, a set of `out` (output) ports, and a function `f` from its input ports and current state, which can be viewed as a set of private ports inaccessible from outside the task, to its output ports and its next state. The invocation of Giotto tasks are based on two essential concepts: instantaneous communication and time determinism. In fact, tasks are activated on periodic cycles P (Figure 1.3). The Giotto logical abstraction does not specify when and how the actual computation of a task `f` is performed, it only determines the instant at which input are read t_{start} and output are produced t_{end}.

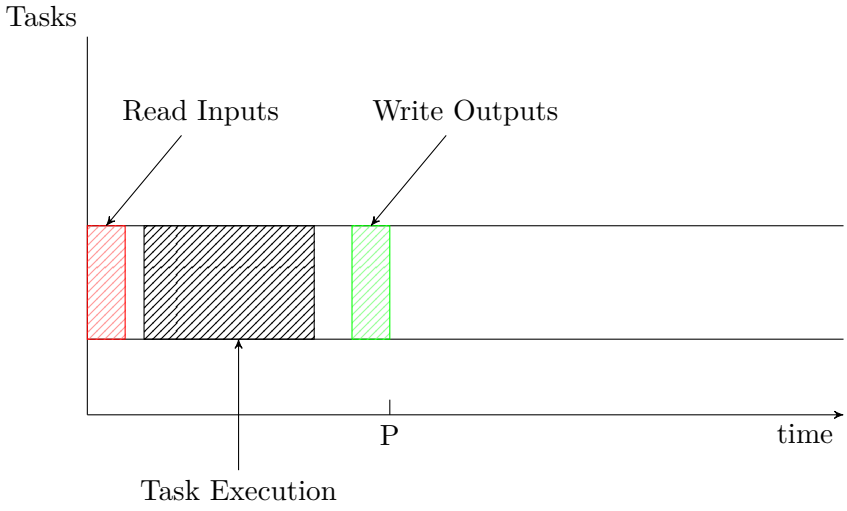


Figure 1.3 – Example of a Giotto Task Execution

1.4.2 Oasis

Oasis [CDA+05, CCDA07] is a framework that provides methods for the design and implementation of safety-critical real-time systems. It includes a complete set of development tools (code generation, validation, simulation and execution) that eases the design and verification stages while complying with general standards. The Oasis approach relies on the time-triggered concept to build systems that are fully deterministic, predictable, and reproducible in both the logical and temporal domains, even in case of failure. An Oasis application consists of a static set of communicating agents, i.e., autonomous execution entities including processing operations. The execution of a processing operation is time-triggered on a time window that is automatically deduced form the agent's timing. Every Oasis agent has an associated real-time window during which input and output data are visible.

Figure 1.4 illustrates the execution of a processing operation within its time interval: each processing operation performed by an agent takes place between two temporal instants (points) of the real-time. These points are the beginning date (t_{start}), which is the earliest instant when the processing can be started, and the ending date (t_{end}) which is the latest date when it

must be terminated. These two dates allow to explain a formal duration that is constant and represents a constraint for the actual duration of execution.

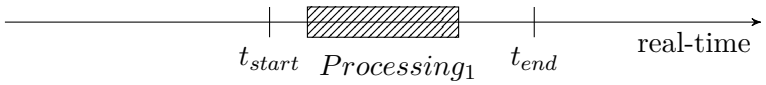


Figure 1.4 – Execution of a Processing Operation in Oasis

1.4.3 PTIDES

PTIDES [ZLL07] is an event-triggered programming model that serves as a coordination language for model-based design of distributed real-time embedded systems. It leverages network time synchronization to provide a coherent global temporal semantics. PTIDES builds on top of a strong timed semantic foundation, based on discrete-event (DE) model [Lee99], and provides a mathematical framework for presenting strategies that explore concurrency of implementations. It also allows deterministic schedulability analysis. PTIDES structures real-time software as an interconnection of components communicating using timestamped events.

Figure 1.5 (taken from [ELM+12]) illustrates the PTIDES workflow.

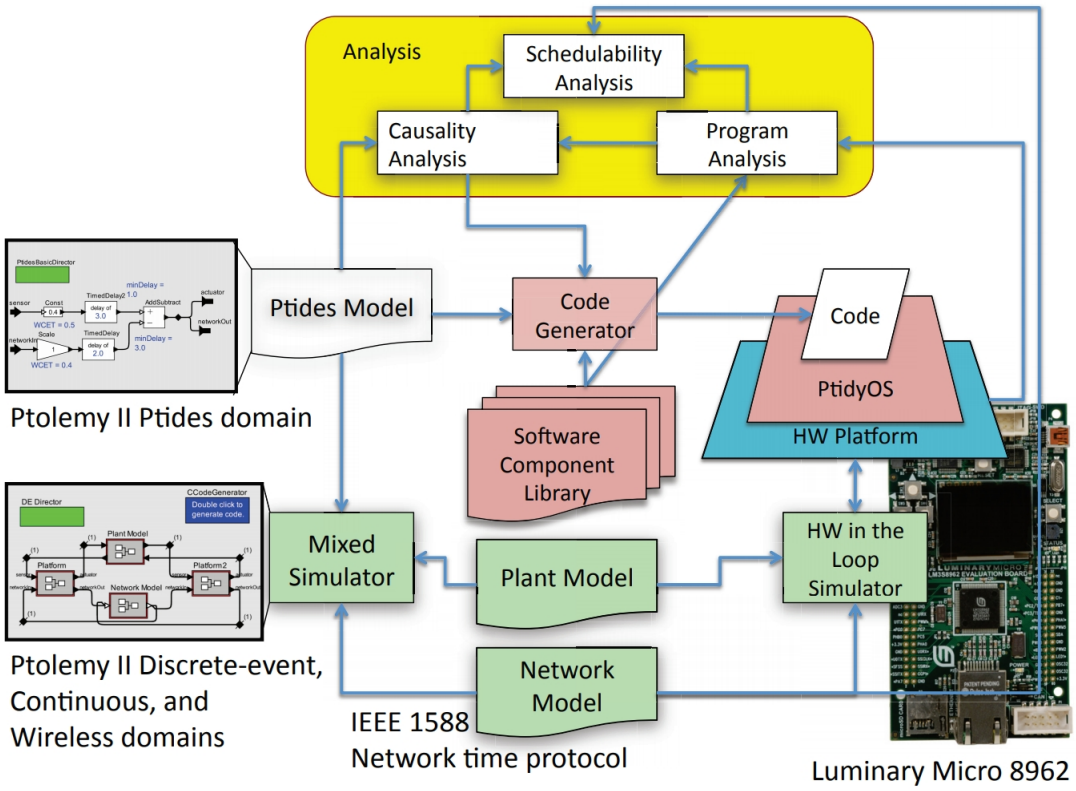


Figure 1.5 – PTIDES Code Generation and Analysis Workflow

The PTIDES design environment is an extension of the Ptolemy II framework [LXL01], which supports modeling, simulation, and design of systems using mixed models of computation. The physical part of the system can be modeled in the continuous domain. The simulation can be instantiated with different ODE solvers that suit diverse time scales present in the model. PTIDES models define the functional and temporal interaction of distributed software components, the networks that bind them together, sensors, actuators, and physical dynamics. Simulation can be done on such models, such that functionality and timing can be tested. In particular, to get a picture of the temporal behavior of a particular implementation of a model, each actor can be endowed with a platform-dependent execution time, and simulation can be performed to determine whether real-time deadlines can be met for a given set of inputs.

1.4.4 Lustre

Lustre [HLR92] is a synchronous dataflow language used for the development of reactive systems, that is, systems that continuously interact with their environment. The behavior of a Lustre program is a sequence of reactions consisting in reading the current inputs, computing the current outputs, and updating the internal state of the program. Lustre is based on the synchrony hypothesis [Hal98]: A reaction is often said to *take no time*. In other words, this means that a given system is faster than its environment.

A Lustre program consists in a set of nodes operating on flows of values (inputs and outputs are described by their flows of values along with time). The synchronous dataflow approach consists in adding a time dimension to the dataflow model. This is done by associating time with the rate of dataflow, that is, the manipulated entities can be interpreted as function of time. For instance, given a variable or an expression x, it represents an infinite sequence of values $(x_0, \ldots, x_n, \ldots)$, x_n being the value of x at the logical time instant n, that is, at the n-th reaction of the program. Figure 1.6 shows a small example of a Lustre program.

```
1  node Nand(X,Y: bool) returns (Z: bool);
2     var U: bool;
3  let
4     U = X and Y;
5     Z = not U;
6  tel
```

Figure 1.6 – A Lustre Program

The above program defines a node that takes two Boolean inputs X and Y, and returns the negation of “X and Y”. It expresses the following relation:

For any instant t, Z_t= `not` (X_t `and` Y_t)

1.5 Contribution

Nowadays, real-time systems are ubiquitous in several application domains, and such an emergence led to an increasing need of performance: resources, availability, concurrency, etc. This expansion initiates a shift from the use of single processor based hardware platforms, to

large sets of interconnected and distributed computing nodes. Moreover, it prompts the birth of a new family of systems that are intrinsically distributed, namely, *Networked Embedded Systems*. Such an evolution stems from an increase in complexity of real-time software embedded on such platforms (e.g. electronic control in avionics and automotive domains [Cha09]), and the need to integrate formerly isolated systems [Kop04] so that they can cooperate as well as share resources improving thus functionality and reducing costs. To deal with such complexity, the community of safety critical systems often restricts its scope to predictable systems, which are represented with domain specific models (e.g. periodic tasks, synchronous systems, time-deterministic systems) for which the range of possible executions is small enough to be easily analyzed, allowing the pre-computation of optimal control strategies. *Networked Embedded Systems* usually describe a set of real-time systems, distributed across several platforms, and interacting through a network. Because of their adaptive behavior, the standard practice when implementing such systems is not to rely on models for pre-computation of execution strategies but rather to design systems dynamically adapting at runtime to the actual context of execution. Such approaches, however, do not offer any formal guarantee of timeliness. Also, the lack of a priori knowledge on system behavior leaves also little room for static optimization.

In this thesis, we propose a rigorous design flow intended for building distributed real-time applications using the BIP Framework. BIP [BBB+11b] (Behavior, Interaction, Priority) is a model-based and component-based framework where systems consist of components represented as timed automata that may synchronize on particular actions to coordinate their activities. Particularly, we investigate timed automata based models, with well defined semantics, in order to study the behavior of a given system with some imposed timing constraints when deployed in a distributed environment.

Knowledge Based Optimization of

In Chapter 3, we propose an intermediate model more appropriated for the representation of distributed real-time systems. It provides details on how the implementation of a system with multiparty interactions can be derived by explicitly expressing the ongoing communication mechanism. This model is obtained using model transformations that are in partly based of the notion of *conflicting interactions*. In other words, in a distributed context components may compete on the same resources at the same time. Thus, any distributed implementation must preserve the overall system consistency by ensuring mutual exclusion in such cases. In fact, *conflicts* are hard to characterize for real life case studies and their computation is based on over-approximation. Chapter 4 proposes a knowledge based optimization in order to optimize the computation of the conflicting interactions set.

Communication Delays

Chapter 5 tackles the communication delays problem inherent to distributed real-time systems. This is achieved by considering additional delays between the decision to execute an interaction and its actual execution. These delays result from the transmission delays between the component responsible for such a decision and the components involved in the interaction, and may have a huge impact in the satisfaction of timing constraints in real-time. This will particularly help to anticipate the execution of interactions at least some delay beforehand, corresponding to the actual worst estimation of communication delays of a given platform, which will alleviate the effect of those delays on the system behavior. Indeed, such delays may

introduce behavioral flaws (e.g. deadlocks) when dealing with arbitrary timing constraints (i.e. no restriction to the non-decreasing deadlines case as in [Tri15]). The proposed approach introduces a semantics based on partial states of the system components and formalizes in a precise way the effect of the delays in this context. It also provides practical means for enforcing system correctness in their presence.

Clock Drift

The verification of real-time systems against some considered properties relies on the fact that clocks are perfectly synchronous, that is, clocks advance at the same rate, which is not the case in practice. Clocks are in fact implemented using oscillators and counter registers, thus, their precision depend highly on the quality of the oscillators and parameters of their underlying environment (temperature, humidity, etc.). Chapter 6 studies the effect of this hardware imperfections by breaking the perfect clocks assumption. More precisely, this problem is formulated as a robustness problem, i.e., does the system still satisfies the specification when subject to different kind of perturbations (in our case clock imperfection). First, a timed automata model for distributed real-time system with drifting clocks is introduced. Thereafter, we revisit the robustness by proposing a strategy that allows to have "similar" or close execution scenarios in the latter and the system with perfect clocks.

1.6 Outline

The rest of this thesis is structured as follows:

- Part I introduces all the preliminaries and includes the following chapters:
  - Chapter 2 gives formal definitions of timed transition systems, timed automata, their semantics and properties.
  - Chapter 3 presents an intermediate timed automata based model for the representation of distributed real-time systems. It also tackles two important constraints related the distributed real-time context.
- Part II includes our contributions. It consists of the following chapters:
  - Chapter 4 proposes a knowledge based optimization for systems modeled using the approach of Chapter 3.
  - Chapter 5 tackles the problem of communication delays inherent to distributed real-time systems by proposing a complete formalization of the latter through a new semantics more suited for distributed real-time executions. It also provides different methods for the verification of systems behavior against deadlock freedom.
  - Chapter 6 investigates the clock drift problem and revisits the robustness approach when studying systems with clock imperfections.
  - Chapter 7 gives an overview of the BIP toolbox and the accomplished implementation. It also presents the experimental results on different case studies
- Part III concludes the dissertation and includes the following chapter:

  - Chapter 8 concludes with an overview of the accomplished work as well as some interesting perspectives.

A complete list of the publications carried out during this thesis is presented below.

- [DCBB16] Mahieddine Dellabani, Jacques Combaz, Marius Bozga, and Saddek Bensalem. Local planning of multiparty interactions with bounded horizons. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, pages 199–216, 2016

- [DCBB17] Mahieddine Dellabani, Jacques Combaz, Saddek Bensalem, and Marius Bozga. Knowledge based optimization for distributed real-time systems. In *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, pages 751–756, 2017

- [DCBB18a] Mahieddine Dellabani, Jacques Combaz, Marius Bozga, and Saddek Bensalem. Local planning semantics : a semantics for distributed real-time systems. *Leibniz Transactions on Embedded Systems*, 2018 (in review)

- [MNB+18] Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, Mahieddine Dellabani, Jacques Combaz, Axel Legay, and Saddek Bensalem. Sbip 2.0: Statistical model checking stochastic real-time systems. Technical Report TR-2018-5, Verimag Research Report, 2018 (in review)

Part I

Preliminaries

In this part, we give formal definitions and present results that will be used in subsequent chapters. Chapter 2 provides formal definitions of timed transition systems, timed automata, their semantics and properties as well as a variant of the latter. It also discusses the verification technique used in this thesis. Chapter 3 explains how an intermediate representation, based on the timed automata formalism, can be used to represent a realistic view of a distributed real-time systems. It also tackles two important constraints that an application may incur when being deployed in a distributed environment under real-time restrictions.

Chapter 2

Timed Systems and Semantics

Contents

2.1 Timed Transition Systems 23
2.1.1 Comparing Timed Transition Systems 24
2.1.2 Reactive Timed Systems 25
2.2 Timed Systems Syntax and Semantics 25
2.2.1 Timed Components Syntax and Semantics 26
2.2.2 Timed Systems 30
2.2.3 Example 32
2.3 Timed Systems with Data 33
2.4 Verification of Timed Systems 34
2.4.1 Symbolic Reachability 34
2.4.2 Compositional Verification 35

2.1 Timed Transition Systems

Transition systems provide a general and convenient method for modeling systems and have been used frequently to model the behavior of software and hardware systems. They define graphs where nodes represent the possible *states* of the system, and edges model *transitions*, that is, state changes. A state encodes all the relevant information at a certain instant, whereas a transition describes how the system evolves between two states.

Nowadays, several variants of transition system formalisms have been proposed. For instance, a labeled transition system is a transition system where the set of transitions is labeled by *actions*. In this thesis, we use *Timed Transition Systems* (TTS) to explicitly model the effect of time passage (besides the actions) on the states of the system. Formally, it is defined as follows:

Definition 2.1.1 (Timed Transition System)**.** A timed transition system is a tuple $\mathtt{T} = (\mathcal{Q}, q_0, \Sigma \cup \mathbb{K}, \rightarrow)$ such that $\mathcal{Q}$ is a set of states, $q_0 \in \mathcal{Q}$ is the initial state, Σ is a set of actions, $\mathbb{K}$ is a time domain and $\rightarrow \subseteq \mathcal{Q} \times (\Sigma \cup \mathbb{K}) \times \mathcal{Q}$ is the transition relation.

Consequently, we distinguish two types of transition:

- *action step* $\xrightarrow{\sigma}$ for $\sigma \in \Sigma$ and we write $q \xrightarrow{\sigma} q'$.

- *time step* $\xrightarrow{d}$ for $d \in \mathbb{K}$ and we write $q \xrightarrow{d} q'$.

We write $q \xrightarrow{d,\sigma} q''$ if there exists $q' \in \mathcal{Q}$ such that $q \xrightarrow{d} q' \xrightarrow{\sigma} q''$. We say that q' is a time successor of q.

Given a TTS $\mathtt{T} = (\mathcal{Q}, q_0, \Sigma \cup \mathbb{K}, \rightarrow)$, a *run* ϱ of $\mathtt{T}$ (also called *execution sequence*), is a path that alternates action steps and time steps, that is:

$$\varrho = s_0\sigma_0 s_1\sigma_1 s_2 \cdots \text{ such that } s_i \in \mathcal{Q},\ s_i \xrightarrow{\sigma_i} s_{i+1},\ \text{and } i \in \mathbb{Z}_{\geq 0}, \sigma_i \in \Sigma \cup \mathbb{K}$$

We denote by $\mathbf{time}(\varrho, \mathrm{i})$ the total elapsed time until point i, that is, $\sum_{j<i} \sigma_j$ such as $\sigma_j \in \mathbb{K}$. In the same way, $\mathbf{time}(\varrho)$ represents the total elapsed time during ϱ, and is defined to be the limit of $\mathbf{time}(\varrho, \mathrm{i})$ if the sequence converges and ∞ otherwise. A run ϱ is said to be an *initial run* if $s_0 = q_0$.

A state $q \in \mathcal{Q}$ is called reachable if there exists an initial run that leads to state q. We put $Reach(\mathtt{T})$ to denote the set of all reachable states of $\mathtt{T}$.

2.1.1 Comparing Timed Transition Systems

In this thesis, we use the concept of (bi)simulation [HLY91] in order to attest the similarity of timed transition systems.

Definition 2.1.2 (Simulation)**.** Given two TTS $\mathtt{T}_1 = (Q_1, q_{0_1}, \Sigma \cup \mathbb{K}, \rightarrow_1)$ and $\mathtt{T}_2 = (Q_2, q_{0_2}, \Sigma \cup \mathbb{K}, \rightarrow_2)$, a simulation relation from $\mathtt{T}_1$ to $\mathtt{T}_2$ is a binary relation $\mathcal{R} \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$ such that:

- $\forall (q_1, q_2) \in \mathcal{R}, \forall \sigma \in \Sigma,\ q_1 \xrightarrow{\sigma}_1 q_1' \Rightarrow \exists q_2' \in \mathcal{Q}_2$ such that $q_2 \xrightarrow{\sigma}_2 q_2' \wedge (q_1', q_2') \in \mathcal{R}$
- $\forall (q_1, q_2) \in \mathcal{R}, \forall d \in \mathbb{K},\ q_1 \xrightarrow{d}_1 q_1' \Rightarrow \exists q_2' \in \mathcal{Q}_2$ such that $q_2 \xrightarrow{d}_2 q_2' \wedge (q_1', q_2') \in \mathcal{R}$

$\mathtt{T}_2$ simulates $\mathtt{T}_1$, denoted by $\mathtt{T}_1 \sqsubseteq_{\mathcal{R}} \mathtt{T}_2$ means that $\mathtt{T}_2$ can do everything $\mathtt{T}_1$ does. Notice that if $\mathtt{T}_1 \sqsubseteq_{\mathcal{R}} \mathtt{T}_2$ and $\mathtt{T}_2 \sqsubseteq_{\mathcal{R}} \mathtt{T}_1$, we say that $\mathtt{T}_1$ and $\mathtt{T}_2$ are bisimilar, denoted by $\mathtt{T}_1 \sim_{\mathcal{R}} \mathtt{T}_2$.

In some cases, this notion of simulation is refined in order to consider only a part of a system behavior. This is usually the case when a system performs *internal* (or *silent*) actions not visible by external observers. This variant of simulation is called *weak simulation*.

Definition 2.1.3 (Weak Simulation)**.** Given two TTS $\mathtt{T}_1 = (\mathcal{Q}_1, q_{0_1}, \Sigma \cup \{\tau\} \cup \mathbb{K}, \rightarrow_1)$ and $\mathtt{T}_2 = (\mathcal{Q}_2, q_{0_2}, \Sigma \cup \{\tau\} \cup \mathbb{K}, \rightarrow_2)$, where τ actions represent silent (unobservable) actions, a weak simulation relation from $\mathtt{T}_1$ to $\mathtt{T}_2$, denoted $\mathtt{T}_1 \dot{\sqsubseteq}_{\mathcal{R}} \mathtt{T}_2$, is a binary relation $\mathcal{R} \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$ such that:

- $\forall (q_1, q_2) \in \mathcal{R},\ q_1 \xrightarrow{\tau}_1 q_1' \Rightarrow \exists q_2' \in \mathcal{Q}_2$ such that $q_2 \xrightarrow{\tau^*}_2 q_2' \wedge (q_1', q_2') \in \mathcal{R}$
- $\forall (q_1, q_2) \in \mathcal{R}, \forall \sigma \in \Sigma,\ q_1 \xrightarrow{\sigma}_1 q_1' \Rightarrow \exists q_2' \in \mathcal{Q}_2$ such that $q_2 \xrightarrow{\tau^*\sigma\tau^*}_2 q_2' \wedge (q_1', q_2') \in \mathcal{R}$
- $\forall (q_1, q_2) \in \mathcal{R}, \forall d \in \mathbb{K},\ q_1 \xrightarrow{d}_1 q_1' \Rightarrow \exists q_2' \in \mathcal{Q}_2$ such that $q_2 \xrightarrow{\tau^* d \tau^*}_2 q_2' \wedge (q_1', q_2') \in \mathcal{R}$

We say that $\mathtt{T}_1$ and $\mathtt{T}_2$ are observationally equivalent, denoted $\mathtt{T}_1 \dot{\sim}_{\mathcal{R}} \mathtt{T}_2$, if it exists a weak simulation from $\mathtt{T}_1$ to $\mathtt{T}_2$ and vice versa.

In chapter 6, we will introduce an even weaker notion of simulation that characterizes the degree of closeness (in term of delays) between two timed systems.

2.1.2 Reactive Timed Systems

Reactive systems are supposed to execute forever, that is, they are supposed to act infinitely often. We refer to this characteristic as the *requirement of progress* [Tri99]. Particularly, a timed system evolves either through an action step or by letting the time pass (time step). This evolution imposes thus two requirements of progress: discrete progress (resp. time progress) meaning that a timed system should be able to perform action steps (resp. time steps) infinitely often. In the physical world however, no matter how fast a system can evolve, it cannot be infinitely fast. This induces the following constraints on the progress of time:

1. Only a finite number of actions can happen in a certain amount of time
2. Only a bounded number of actions can happen in zero time

Time Progress (Zeno runs & Timelocks)

We distinguish two types of anomalies that infringe the time progress in a timed system namely *zeno* runs and *timelocks*. A run ϱ is called *zeno* if it is an infinite run and if $\mathbf{time}(\varrho) \neq \infty$. Such a run transgresses the first point of the time progress presented above. Timelocks are states from which all infinite runs starting from these states are zeno.

Discrete Progress (Deadlocks)

States violating the discrete progress are called *deadlock* states. Formally, a state is said to be deadlock if no action can be executed from that state nor from any of its time successors.

Any model of a reactive timed system should properly capture the behavior of the latter. Particularly, the corresponding model must react infinitely often, that is, it must not block time or execute an unbounded number of actions in zero time. In that sense, we can say that deadlocks and timelocks are modeling errors that needs to be cleared, either during the modeling process (which is tedious for large scale systems) or by providing verification methods that guarantee their absence.

2.2 Timed Systems Syntax and Semantics

Clocks

In order to represent and measure the dense time domain, we rely on positive real valued variables, the *clocks*. Clocks are positive real variables increasing synchronously (with the same rate) in a given system. They are used to express the progress of time and impose a certain dynamic on the execution of a timed system.

Given a finite set of clocks $\mathcal{X}$, we define the valuation function $\upsilon : \mathcal{X} \to \mathbb{R}_{\geq 0}$ assigning to each clock x a positive real value $\upsilon(x)$. We put $\mathbb{R}^{\mathcal{X}}_{\geq 0}$ to denote the set of all valuations. For a valuation $\upsilon \in \mathbb{R}^{\mathcal{X}}_{\geq 0}$ and $d \in \mathbb{R}_{\geq 0}$, $\upsilon + d$ is the valuation satisfying $(\upsilon + d)(x) = \upsilon(x) + d$, while for a subset of clocks $r \subseteq \mathcal{X}$, $\upsilon[r]$ is the valuation obtained from υ by resetting clocks of r, that is, $\upsilon[r](x) = 0$ for $x \in r$ and $\upsilon[r](x) = \upsilon(x)$ otherwise. We write $\mathbf{0}$ for the valuation that assigns 0 to every clock.

An *atomic clock constraint* is an expression of the form:

$$c := \mathit{true} \mid x \# k \mid x - y \# k$$

where x and y are clocks in $\mathcal{X}$, $\# \in \{<, \leq, =, \geq, >\}$, and $k \in \mathbb{Z}$. A clock constraint is a conjunction of atomic clock constraint, that is:

$$c := true \mid x \# k \mid x - y \# k \mid c_1 \wedge c_2 \tag{2.1}$$

with c_1 and c_2 being atomic clock constraints. We write $\mathcal{C}(\mathcal{X})$ for the set of clock constraints over $\mathcal{X}$. Given a clock constraint c and a valuation υ, we say that υ satisfies c, denoted $\upsilon \models c$, if all constraints are satisfied when each $x \in \mathcal{X}$ is replaced by $\upsilon(x)$. We also consider for a clock constraint c the classical *backward* and *forward* operators on clock constraints:

Backward	$:\upsilon \models \swarrow c \Leftrightarrow \exists d \in \mathbb{R}_{\geq 0}.\ \upsilon + d \models c$
Forward	$:\upsilon \models \nearrow c \Leftrightarrow \exists d \in \mathbb{R}_{\geq 0}.\ \upsilon - d \models c$

Additionally, we also use another variant of the backward operator considering lower bounds $l \in \mathbb{Z}_{\geq 0}$ and upper bounds $u \in \mathbb{Z}_{\geq 0} \cup \{+\infty\}$:

$$\upsilon \models \swarrow_l^u c \Leftrightarrow \exists d \in \mathbb{R}_{\geq 0}, l \leq d \leq u.\ \upsilon + d \models c$$

2.2.1 Timed Components Syntax and Semantics

In this thesis, components are timed automata and systems are compositions of timed automata with respect to multiparty interactions. The timed automata we use are essentially the ones from [AD94], however, slightly adapted to embrace a uniform notation the dissertation.

Definition 2.2.1 (Timed Component)**.** A component is a tuple $B = (\mathcal{L}, \ell_0, \mathcal{X}, \mathcal{A}, \mathcal{E}, \mathcal{I})$, where:

- $\mathcal{L}$ is a finite set of locations, with $\ell_0 \in \mathcal{L}$ is the initial location,
- $\mathcal{X}$ is a finite set of clocks,
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{E} \subseteq \mathcal{L} \times (\mathcal{A} \times \mathcal{C}(\mathcal{X}) \times 2^{\mathcal{X}}) \times \mathcal{L}$ is a finite set of transitions labeled with an action, a clock constraint (guard), and a set of clocks to be reset,
- $\mathcal{I} : \mathcal{L} \to \mathcal{C}(\mathcal{X})$ is the function assigning an invariant to each location. Notice that invariants are restricted to conjunction of atomic clock constraints of the form $x \leq k$.

Throughout this thesis, we consider *deterministic* timed components, that is, at a given location ℓ and for a given action a, there is up to one outgoing transition from ℓ labeled by a. A transition $e = (\ell, (a, g, r), \ell') \in \mathcal{E}$ is also denoted by $\ell \xrightarrow{a,g,r} \ell'$. We write $\mathsf{source}(e)$, $\mathsf{target}(e)$, $\mathsf{action}(e)$, $\mathsf{guard}(e)$ and $\mathsf{reset}(e)$ for ℓ, ℓ', a, g and r, respectively. We also denote by $\Phi(a, \ell)$ the guard of the transition labeled by a and outgoing from ℓ if it exists, and *false* otherwise. It is formalized as follows:

$$\Phi(a, \ell) = \begin{cases} g, & \text{if } \exists e = (\ell, a, g, r, \ell') \in \mathcal{E} \\ false, & otherwise \end{cases}$$

Example 2.2.1. Figure 2.1 depicts a timed component B where locations are represented by circles and transitions are the directed arrows from a location to another. The initial location (ℓ_0 here) is represented by a double circle. The component $B = (\mathcal{L}, \ell_0, \mathcal{X}, \mathcal{A}, \mathcal{E}, \mathcal{I})$ is defined such that:

- $\mathcal{L} = \{\ell_0, \ell_1\}$,
- $\mathcal{X} = \{x\}$,
- $\mathcal{A} = \{a, b\}$,
- $\mathcal{E} = \{e_1, e_2\}$ where
  - $e_1 = (\ell_0, a, 2 \leq x \leq 4, \emptyset, \ell_1)$
  - $e_2 = (\ell_1, b, \mathit{true}, \{x\}, \ell_0)$
- $\mathcal{I}$ assigns the following invariants for locations: $\mathcal{I}(\ell_0) = x \leq 4$ and $\mathcal{I}(\ell_1) = \mathit{true}$

Notice that when a clock constraint (respectively an invariant is not shown on a transition (respectively location) it is interpreted as *true*. This is the case for transition e_2 and location ℓ_2.

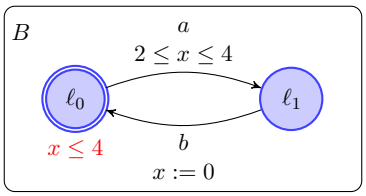


Figure 2.1 – Example of a Timed Component

Definition 2.2.2 (Standard Semantics)**.** The semantics of a timed component $B = (\mathcal{L}, \ell_0, \mathcal{X}, \mathcal{A}, \mathcal{E}, \mathcal{I})$ is given by the timed transition system $\mathtt{T} = (\mathcal{Q}, q_0, \mathcal{A} \cup \mathbb{R}_{>0}, \rightarrow)$ where:

- $\mathcal{Q} = \mathcal{L} \times \mathbb{R}^{\mathcal{X}}_{\geq 0}$ denotes the states of B with $q_0 = (\ell_0, 0)$ being the initial state,
- $\rightarrow \subseteq \mathcal{Q} \times (\mathcal{A} \cup \mathbb{R}_{>0}) \times \mathcal{Q}$ denotes the set of transitions between states according to the rules:
  - $(\ell, v) \xrightarrow{a} (\ell', v[r])$ if $\ell \xrightarrow{a,g,r} \ell'$, $v \models g$, and $v[r] \models \mathcal{I}(\ell')$ (action step)
  - $(\ell, v) \xrightarrow{d} (\ell, v + d)$ if $\forall d' \in [0, d]$, $v + d' \models \mathcal{I}(\ell)$ (time step)

Notice that since the invariants are restricted to conjunction of upper bound atomic constraints, the time step can be simplified to:

$$(\ell, v) \xrightarrow{d} (\ell, v + d) \text{ if } v + d \models \mathcal{I}(\ell)$$

In this thesis, we always assume components with *well formed* guards, that is, for a transition $\ell \xrightarrow{a,g,r} \ell'$, $(v \models g) \Rightarrow (v \models \mathcal{I}(\ell) \wedge v[r] \models \mathcal{I}(\ell'))$ for any $v \in \mathbb{R}^{\mathcal{X}}_{\geq 0}$. This ensures that the reachable states always satisfy the location invariants. The rule on action step becomes then:

$$(\ell, v) \xrightarrow{a} (\ell', v[r]) \text{ if } \ell \xrightarrow{a,g,r} \ell' \text{ and } v \models g$$

Remark 2.2.1. When used in predicate definition, clock constraints are straightforwardly applied to clock valuations of states and thus interpreted as *true* or *false*.

We define the predicate $Enabled(a)$ characterizing states (ℓ, υ) from which an action a is enabled, that is, such that $(\ell, \upsilon) \xrightarrow{a} (\ell', \upsilon')$. It is formalized as follows:

$$Enabled(a) = \bigvee_{\ell \in \mathcal{L}} \mathsf{at}(\ell) \wedge \Phi(a, \ell)$$

where $\mathsf{at}(\ell)$ is *true* on states whose location is ℓ. In the same way, we define the predicates $Enabled\swarrow(a)$, $Enabled\swarrow_l^u(a)$ and $Enabled\nearrow(a)$ describing, respectively, states from which an action a can be executed after some time step, some bounded time step or states that are time successors of states satisfying $Enabled(\alpha)$ (see Figure 2.2). These predicates can be formally written as follows:

$$\begin{aligned} Enabled\swarrow(a) &= \bigvee_{\ell \in \mathcal{L}} \mathsf{at}(\ell) \wedge \swarrow \Phi(a, \ell) \\ Enabled\swarrow_l^u(a) &= \bigvee_{\ell \in \mathcal{L}} \mathsf{at}(\ell) \wedge \swarrow_l^u \Phi(a, \ell) \\ Enabled\nearrow(a) &= \bigvee_{\ell \in \mathcal{L}} \mathsf{at}(\ell) \wedge \nearrow \Phi(a, \ell) \end{aligned}$$

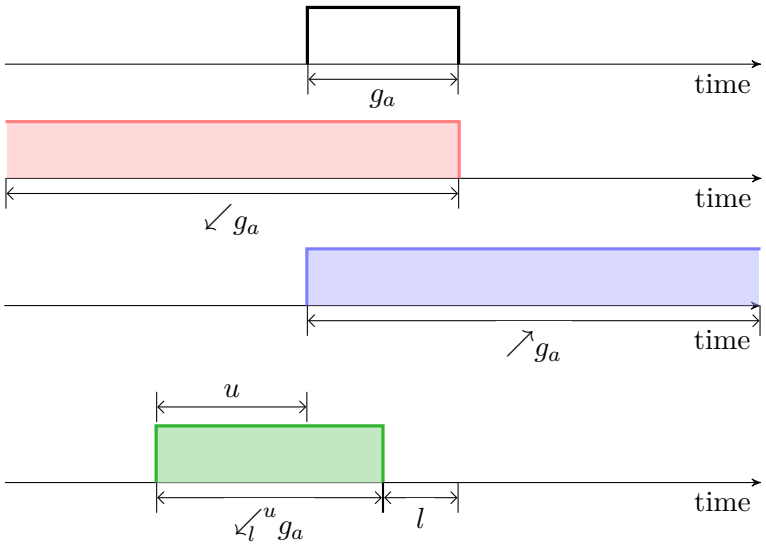


Figure 2.2 – Backward and Forward Operators on a Guard

A state (ℓ, υ) is said *urgent* if time cannot progress from this state, that is, $\nexists d \in \mathbb{R}_{>0}$ such that $(\ell, \upsilon) \xrightarrow{d} (\ell, \upsilon + d)$. Urgent states of a component B are characterized by the predicate:

$$Urgent(B) = \bigvee_{\ell \in \mathcal{L}} \mathsf{at}(\ell) \wedge urg(\ell) \tag{2.2}$$

where $urg(\ell)$ is a clock constraint characterizing the valuations from which time cannot progress with respect to the location invariant $\mathcal{I}(\ell)$, that is, assuming that $\mathcal{I}(\ell) = \bigwedge_{i=1}^{m} x_i \leq k_i$ then

$urg(\ell) = \bigwedge_{i=1}^{m} x_i \geq k_i$. Notice that due to well formed guards, an urgent reachable state satisfies also Expression 2.2 if inequalities $x_i \geq k_i$ on clocks are replaced by equalities $x_i = k_i$ in the expression of $urg(\ell)$.

Definition 2.2.3 (Strongly Non-Zeno Timed Component)**.** Given a timed component B, a *structural loop* of B is a sequence of distinct transition $e_1 \cdots e_m$ such that $\forall i \in \{1, \cdots, m\}, \mathsf{target}(e_i) = \mathsf{source}(e_{i+1})$. B is called *strongly non-zeno* if for every structural loop there exists a clock x and some $0 \leq i, j \leq m$ such that:

- x is reset in step i, that is, $x \in \mathsf{reset}(e_i)$
- x is bounded from below in step j, that is, $(x < 1) \wedge \mathsf{guard}(e_j) = \mathit{false}$

Intuitively, this definition implies that at least 1 time unit elapses at each loop of B.

Lemma 2.2.1 ([Tri99])**.** *If B is strongly non-zeno then every infinite run of B is non-zeno*

The following corollary is an immediate consequence of lemma 2.2.1.

Corollary 2.2.1. *Given a timed component B, if B is strongly non-zeno then it is timelock free.*

Corollary 2.2.1 highlights an interesting fact of strong non-zenoness. It discharges from the trouble of checking time progress. In particular, checking the progress of a timed system is reduced to checking its deadlock freedom.

Definition 2.2.4 (Deadlocks)**.** Given a timed component B. We say that a state (ℓ, υ) of B is a *deadlock* if and only if no action can be executed from this state or any of its time successors, that is:

$$\neg\Big(\exists a \in \mathcal{A}.\ (\ell, \upsilon) \xrightarrow{a} (\ell', \upsilon') \vee \exists d > 0.\ (\ell, \upsilon) \xrightarrow{d} (\ell, \upsilon + d) \xrightarrow{a} (\ell', \upsilon')\Big)$$

Deadlock states are characterized by the following predicate:

$$\bigvee_{\ell \in \mathcal{L}} \mathsf{at}(\ell) \wedge \neg\Big(\bigvee_{a \in \mathcal{A}} \not\swarrow\big(\mathit{Enabled}(a) \wedge \mathcal{I}(\ell)\big)\Big)$$

Because of well formed guards, the above can be simplified into:

$$\bigwedge_{a \in \mathcal{A}} \neg \mathit{Enabled}\not\swarrow(a) \tag{2.3}$$

A deadlock (ℓ, υ) is called an *action-time-lock* when no action can execute nor time can progress from (ℓ, υ), that is:

$$\neg\Big(\exists a \in \mathcal{A}.\ (\ell, \upsilon) \xrightarrow{a} (\ell', \upsilon') \vee \exists d > 0.\ (\ell, \upsilon) \xrightarrow{d} (\ell, \upsilon + d)\Big)$$

Action-time-locks verifies the following predicate:

$$\bigwedge_{a \in \mathcal{A}} \neg \mathit{Enabled}(a) \wedge \bigvee_{\ell \in \mathcal{L}} \big(\mathsf{at}(\ell) \wedge urg(\ell)\big) \tag{2.4}$$

2.2.2 Timed Systems

The common practice in component-based timed systems is to have several components executing in parallel while their clocks increase synchronously. Moreover, it is often mandatory to restrict the components behaviors so as to achieve a given global property. This is usually achieved by coordinating the execution of actions in components using synchronization mechanisms. In what follows, components communicate by means of *multiparty interactions*. A multiparty interaction is a rendez-vous synchronization between actions of a fixed subset of components. It takes place only if all the participants agree to execute the corresponding actions. Given n components B_i, with disjoint sets of actions $\mathcal{A}_i$, an interaction is a subset of actions $\alpha \subseteq \cup_{1\leq i\leq n} A_i$ containing at most one action per component, that is, $\alpha \cap \mathcal{A}_i$ is either empty or a singleton $\{a_i\}$. Thus, an interaction α can be put in the form $\alpha = \{a_i\}_i \in I$ with $I \subseteq \{1, \cdots, n\}$ and $a_i \in \mathcal{A}_i$ for all $i \in I$. We denote by $part(\alpha)$, the set of components *participating* in α, that is, $part(\alpha) = \{B_i\}_{i\in I}$.

Definition 2.2.5 (Timed System)**.** Given n components $B_i = (\mathcal{L}_i, \ell_{0_i}, \mathcal{X}_i, \mathcal{A}_i, \mathcal{E}_i, \mathcal{I}_i)$ with $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset$, $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$, and $\mathcal{X}_i \cap \mathcal{X}_j = \emptyset$ for any $i \neq j$, the composition with respect to the interaction set γ, denoted by $S = \gamma(B_1, \cdots, B_n)$, is defined by the timed component $(\mathcal{L}, \ell_0, \mathcal{X}, \gamma, \mathcal{E}_\gamma, \mathcal{I})$ where:

- $\mathcal{L} = \mathcal{L}_1 \times \cdots \times \mathcal{L}_n$
- $\ell_0 = (\ell_{0_1}, \cdots, \ell_{0_n})$,
- $\mathcal{X} = \mathcal{X}_1 \cup \cdots \cup \mathcal{X}_n$,
- $\mathcal{I}(\ell) = \mathcal{I}_1(\ell_1) \wedge \cdots \wedge \mathcal{I}_n(\ell_n)$, for $\ell = (\ell_1, \cdots, \ell_n)$,
- $\mathcal{E}_\gamma$ is defined by:

$$\mathcal{E}_\gamma = \left\{ \begin{array}{c|c} \ell \xrightarrow{\alpha,g,r} \ell' & \ell{=}(\ell_1,\cdots,\ell_n){\in}\mathcal{L},\ \ell'{=}(\ell'_1,\cdots,\ell'_n){\in}\mathcal{L} \\ \text{for } \alpha{=}\{a_i\}_{i\in I}{\in}\gamma & \text{if } i{\notin}I,\ \ell'_i{=}\ell_i, \text{ and for } i{\in}I,\ \ell_i \xrightarrow{a_i,g_i,r_i} \ell'_i \text{ and} \\ & g{=}\bigwedge_{i\in I} g_i,\ r{=}\cup_{i\in I} r_i \end{array} \right\}$$

In a composition S of n components B_i, an action a_i can execute only as part of an interaction α such that $a_i \in \alpha$, that is, along with the execution of all other actions $a_j \in \alpha$. This corresponds to the usual notion of multiparty interactions. In practice, we do not explicitly build compositions of timed components as presented in Definition 2.2.5. We rather interpret their semantics by evaluating enabled interactions based on current states of components.

Property 2.2.1 (Semantics of a Composition)**.** *Given a set of components $\{B_1, \cdots, B_n\}$ and an interaction set γ, the semantics of the composite component $S = \gamma(B_1, \cdots, B_n)$ with respect to the set of interaction γ, is defined by the timed transition system $T_g = (\mathcal{Q}_g, q_{0_g}, \gamma \cup \mathbb{R}_{>0}, \rightarrow_\gamma)$ where:*

- *$\mathcal{Q}_g = \mathcal{L} \times \mathbb{R}^{\mathcal{X}}_{\geq 0}$ is the set of global states, where $\mathcal{L} = \mathcal{L}_1 \times \cdots \times \mathcal{L}_n$ and $\mathcal{X} = \bigcup_{i=1}^n \mathcal{X}_\rangle$. We write a state $q = (\ell, \upsilon)$ where $\ell = (\ell_1, \cdots, \ell_n) \in \mathcal{L}$ is a global location and $\upsilon = (\upsilon_1, \cdots, \upsilon_n) \in \mathbb{R}^{\mathcal{X}}_{\geq 0}$ is a global clock valuation. The initial state is $q_0 = (\ell_0, 0)$,*
- *γ is the set of interactions,*
- *$\rightarrow_\gamma$ is the set of transitions defined by the rules:*

– *Interaction step:*

$$\frac{\alpha = \{a_i\}_{i\in I} \in \gamma, \quad \forall i \in I.(\ell_i, \upsilon_i) \xrightarrow{a_i}_{\gamma} (\ell'_i, \upsilon'_i), \quad \forall i \notin I.(\ell_i, \upsilon_i) = (\ell'_i, \upsilon'_i)}{(\ell, \upsilon) \xrightarrow{\alpha}_{\gamma} (\ell', \upsilon')}$$

– *Time step:*

$$\frac{d \in \mathbb{R}_{>0}, \quad \forall i \in \{1, \cdots, n\}.\upsilon_i + d \models \mathcal{I}_i(\ell_i)}{(\ell, \upsilon) \xrightarrow{d}_{\gamma} (\ell, \upsilon + d)}$$

To simplify notations, predicates defined on individual components B_i are straightforwardly interpreted on global states (ℓ, υ) of the composition by considering the projection (ℓ_i, υ_i) of (ℓ, υ) on B_i. For instance, $\mathsf{at}(\ell_i)$ evaluates to true on (ℓ, υ) iff $\ell \in \mathcal{L}_1, \times \cdots \times \mathcal{L}_{i-1} \times \{\ell_i\} \times \mathcal{L}_{i+1} \times \cdots \times \mathcal{L}_n$. Similarly, clock constraints of component B_i are applied to clock valuation functions of the composition by restricting υ to clocks in $\mathcal{X}_i$ of B_i. This allows to write the predicate $Enabled(\alpha)$, characterizing states (ℓ, υ) from which an interaction $\alpha = \{a_i\}_{i\in I} \in \gamma$ can be executed, as:

$$Enabled(\alpha) = \bigwedge_{i\in I} Enabled(a_i), \tag{2.5}$$

$$= \bigwedge_{i\in I} \bigvee_{\ell_i \in \mathcal{L}_i} \mathsf{at}(\ell_i) \wedge \Phi(a_i, \ell_i), \tag{2.6}$$

$$= \bigwedge_{\substack{\ell\in\mathcal{L} \\ \ell=(\ell_1,\cdots,\ell_n)}} \mathsf{at}(\ell) \wedge \bigwedge_{\substack{i\in I \\ a_i\in\alpha}} \Phi(a_i, \ell_i) \tag{2.7}$$

Expression 2.6 expresses the predicate $Enabled(\alpha)$ using location of individual components whereas Expression 2.7 formalizes it on global location configurations. Notice that the above formulation of $Enabled(\alpha)$ corresponds to locations enumeration of all components participating in interaction α. In practice, we rather consider only a subset of locations $\mathcal{L}_\alpha \subseteq \mathcal{L}$, from which the execution of α is possible. This corresponds to $\Pi_{i\in I}|\mathcal{L}_{a_i}|$ possible configuration, where $\mathcal{L}_{a_i} \subseteq \mathcal{L}_i$ is a subset of locations from which there exists a transition labeled by action $a_i \in \alpha$, and $|\mathcal{L}_{a_i}|$ denotes the cardinality of $\mathcal{L}_{a_i}$, which is reasonably small in practical examples but can be (at the worst case) equal to $\Pi_{i\in I}|\mathcal{L}|$. The predicates $Enabled\swarrow(\alpha)$, $Enabled\swarrow_l^u(\alpha)$ and $Enabled\nearrow(\alpha)$ becomes:

$$Enabled\swarrow(\alpha) = \bigvee_{\substack{\ell\in\mathcal{L} \\ \ell=(\ell_1,\cdots,\ell_n)}} \mathsf{at}(\ell) \wedge \swarrow\Big(\bigwedge_{\substack{i\in I \\ a_i\in\alpha}} \Phi(a_i, \ell_i)\Big)$$

$$Enabled\swarrow_l^u(\alpha) = \bigvee_{\substack{\ell\in\mathcal{L} \\ \ell=(\ell_1,\cdots,\ell_n)}} \mathsf{at}(\ell) \wedge \swarrow_l^u\Big(\bigwedge_{\substack{i\in I \\ a_i\in\alpha}} \Phi(a_i, \ell_i)\Big)$$

$$Enabled\nearrow(\alpha) = \bigvee_{\substack{\ell\in\mathcal{L} \\ \ell=(\ell_1,\cdots,\ell_n)}} \mathsf{at}(\ell) \wedge \nearrow\Big(\bigwedge_{\substack{i\in I \\ a_i\in\alpha}} \Phi(a_i, \ell_i)\Big)$$

Notice that for a clock constraint $c = c_1 \wedge c_2$, we have:

$$\diamond c = \diamond(c_1 \wedge c_2) \neq \diamond c_1 \wedge \diamond c_2$$

with $\diamond \in \{\swarrow, \nearrow\}$.

The definitions of deadlocks and action-time-locks are also trivially extended to composition of timed components. Deadlocks can be characterized as follows:

$$\bigvee_{\ell=(\ell_1,\cdots,\ell_i)\in\mathcal{L}} \mathsf{at}(\ell) \wedge [\bigwedge_{\alpha\in\gamma} \neg\swarrow(Enabled(\alpha) \wedge \bigwedge_{1\leq i\leq n} \mathcal{I}_i(\ell_i))]$$

and action-time-locks by:

$$(\bigwedge_{\alpha\in\gamma} \neg Enabled(\alpha)) \wedge (\bigvee_{1\leq i\leq n} \bigvee_{\ell_i\in\mathcal{L}_i} \mathsf{at}(\ell_i) \wedge urg(\ell_i))$$

2.2.3 Example

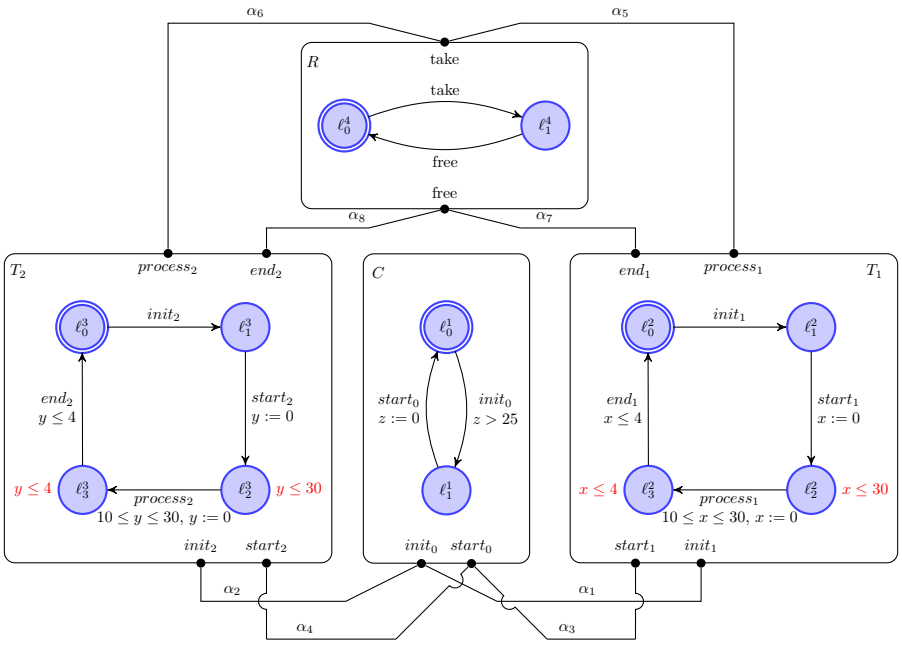


Figure 2.3 – Task Manager

Figure 2.3 depicts a timed system composed of four components C, T_1, T_2, and R. Component C represents a controller that initializes then releases tasks T_1 and T_2. Tasks use the shared resource R during their executions. To implement such behavior, we consider the following interactions between C, R, and T_1: $\alpha_1 = \{init_0, init_1\}$, $\alpha_3 = \{start_0, start_1\}$, $\alpha_5 = \{take, process_1\}$, $\alpha_7 = \{free, end_1\}$, and similar interactions α_2, α_4, α_6, α_8 for task T_2, as shown by connections on Figure 2.3. The controller is responsible for firing the execution of each task. First, it non-deterministically initializes one of the two tasks, i.e., executes α_1 or α_2, and then releases it through interaction α_3 or α_4. Tasks perform their processing independently of the controller, after being granted an access to the shared resource (α_5 or α_6). When finished,

a task releases the resource (interactions α_7 or α_8) and goes back to its initial location. An example of execution sequence of this system is given below. Valuation v of clocks x, y, and z are represented as tuples $(v(x), v(y), v(z))$:

$$((\ell_0^1, \ell_0^2, \ell_0^3, \ell_0^4), (0,0,0)) \xrightarrow{26}_\gamma ((\ell_0^1, \ell_0^2, \ell_0^3, \ell_0^4), (26,26,26)) \xrightarrow{\alpha_1}_\gamma ((\ell_1^1, \ell_1^2, \ell_0^3, \ell_0^4), (26,26,26)) \xrightarrow{\alpha_3}_\gamma$$
$$((\ell_0^1, \ell_2^2, \ell_0^3, \ell_0^4), (0,26,0)) \xrightarrow{10}_\gamma ((\ell_0^1, \ell_2^2, \ell_0^3, \ell_0^4), (10,36,10)) \xrightarrow{\alpha_5}_\gamma ((\ell_0^1, \ell_3^2, \ell_0^3, \ell_1^4), (0,36,10)) \xrightarrow{2}_\gamma$$
$$((\ell_0^1, \ell_3^2, \ell_0^3, \ell_1^4), (2,38,12)) \xrightarrow{\alpha_2}_\gamma ((\ell_1^1, \ell_3^2, \ell_1^3, \ell_1^4), (2,38,12))$$

2.3 Timed Systems with Data

Timed models introduced in the previous section focuses on the timing behavior of a given system. In order to achieve a higher degree of expressiveness, we extend these models with data variables. Data allows additional representations of complex behavior. Similarly to clock variables, they may appear in the guards of transitions as additional conditions and may be updated when transitions fire.

Definition 2.3.1 (Guards on clocks and Data)**.** Let $\mathcal{X}$ be a set of clock variables and $\mathcal{D}$ be a set of data variables. We denote by $\mathcal{G}(\mathcal{X}, \mathcal{D})$ the set of guards induced by the following grammar:

$$g := g_x \mid g_d \mid g_1 \wedge g_2$$

where $g_x \in \mathcal{C}(\mathcal{X})$, g_d is a predicate on a subset of data variables of $\mathcal{D}$, and g_1, g_2 are guards over clocks and/or data variables.

We extend the notion of valuation to data variables in the following manner:

- Valuations assign values to data variable (in addition to clocks),
- The satisfaction of a valuation to a constraint is straightforwardly extended to data variables,
- Data variables are insensitive to the progress of time, that is, for $k \in \mathcal{D}$ and $d \in \mathbb{R}_{>0}$, $(v + d)(k) = (v)(k)$,
- Update operations are also defined for data variables using transfer functions

We use transfer functions to express update operations on data variables of $\mathcal{D}$. A transfer function $f : \mathcal{V}(\mathcal{D}) \to \mathcal{V}(\mathcal{D})$ assigns to each variable $d \in \mathcal{D}$ a new value $f(v)$ based on the current values of variables of $\mathcal{D}$. Additionally, given a set $\mathcal{D}' \supseteq \mathcal{D}$, applying f on $\mathcal{D}'$ does not change the values of variables in $\mathcal{D}' \setminus \mathcal{D}$.

Definition 2.3.2 (Timed Component with Data)**.** A timed component with data is a tuple $B_d = (\mathcal{L}, \ell_0, \mathcal{X}, \mathcal{D}, \mathcal{A}, \mathcal{E}, \{f_e\}_{e \in \mathcal{E}}, \mathcal{I})$ such that:

- $\mathcal{L}$, ℓ_0, $\mathcal{X}$, $\mathcal{A}$ and $\mathcal{I}$ are defined as in Definition 2.2.1,
- $\mathcal{D}$ is a finite set of data variables,
- $\mathcal{E} \subseteq \mathcal{L} \times (\mathcal{A} \times \mathcal{G}(\mathcal{X}, \mathcal{D}) \times 2^{\mathcal{X}}) \times \mathcal{L}$ is a finite set of labeled transitions with an action, an extended guard, and a set of clocks to be reset. For each transition $e \in \mathcal{E}$, we include the transfer function f_e that updates elements of $\mathcal{D}$.

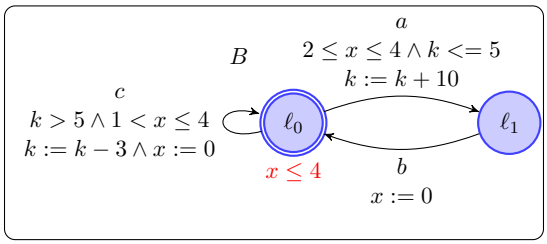


Figure 2.4 – Example of an Extended Timed Component

Example 2.3.1. Component of Figure 2.4 is an extended timed component with actions a, b and c. The set of data is $\mathcal{D} = \{k\}$. It is used on the guards of the transitions labeled by a and c ($k \leq 5$ and $k > 5$ respectively). The update operations for both transitions are respectively $\{k := k + 10\}$ and $\{k := k - 3 \wedge x := 0\}$.

Remark 2.3.1. Notice that extending a timed component with data will result in a restriction of the behavior of the initial timed components since extending guards to data variables will only constrain the execution of transitions. Consequently, for a timed component with data $B_d = (\mathcal{L}, \ell_0, \mathcal{X}, \mathcal{D}, \mathcal{A}, \mathcal{E}, \{f_e\}_{e\in\mathcal{E}}, \mathcal{I})$, the timed component $B = (\mathcal{L}, \ell_0, \mathcal{X}, \mathcal{A}, \mathcal{E}, \mathcal{I})$ represents an abstraction of the latter.

Definition 2.3.3 (Timed System with Data)**.** Given n components $B_{d_i} = (\mathcal{L}_i, \ell_{0_i}, \mathcal{X}_i, \mathcal{D}_i, \mathcal{A}_i, \mathcal{E}_i, \{f_e\}_{e\in\mathcal{E}_i}, \mathcal{I}_i)$ and an interaction set γ, the composition $S_d = \gamma(B_{d_1}, \cdots, B_{d_n})$ with respect to the interaction set γ is defined by the timed component $(\mathcal{L}, \ell_0, \mathcal{X}, \mathcal{D}, \gamma, \mathcal{E}_\gamma, \{f_e\}_{e\in\gamma}, \mathcal{I})$ where:

- $\mathcal{L}$, ℓ_0, $\mathcal{X}$, γ and $\mathcal{I}$ are defined as in Definition 2.2.5.
- $\mathcal{D} = \mathcal{D}_1 \cup \cdots \cup \mathcal{D}_n$
- $\mathcal{E}_\gamma$ is straightforwardly extended by considering guards on data variables and the application of transfer functions on executions of interactions

The semantics of timed system with data extends the semantics of Property 2.2.1 in the sense that an interaction takes place if the guards on data variables also evaluates to *true*. Also, interactions executions apply the underlying transfer functions of the corresponding components transitions.

In what follows, when referring to timed component or timed system we imply by that extended timed component or extend timed systems, unless explicitly stated.

2.4 Verification of Timed Systems

2.4.1 Symbolic Reachability

The semantics of timed systems as presented in Property 2.2.1 defines states as pairs of locations configurations and clock valuations. By considering a continuous time domain ($\mathbb{R}_{\geq 0}$ here), the resulting timed transition system yields an infinity of states. Consequently, the usual practice is to rely on a symbolic representation of states to make this state space finite. A symbolic state is defined by a pair (ℓ, ξ) where ℓ is a location and ξ is a zone, a set of clock valuations

defined by clock constraints (as defined in 2.1). Consequently, the set of reachable states of a timed component B (system) can be put on the form:

$$Reach(B) = \bigvee_{j \in J} \mathsf{at}(\ell_j) \wedge \xi_j$$

Given a timed component with data B_d and its abstraction of data B, we have $Reach(B_d) \subseteq Reach(B)$. This means that $Reach(B)$ can be used as an over-approximation of the reachable state of the timed component with data B_d.

2.4.2 Compositional Verification

Standard verification techniques such as model checking are based on explicit exploration and exhaustive enumeration of all the reachable symbolic states of a given system. The main issue with this method is the combinatorial explosion when considering large scale systems. Compositional verification have been introduced to cope with state explosion problem, and thus achieves scalability when verifying large scale systems. This approach is based on the concept of divide-and-conquer in order to break up the verification problems into smaller subsequent problems. Compositional verification have been extensively studied under different manners e.g., assume-guarantee reasoning [Lam77, OG76], contract-based verification [BCP07, BFM$^+$08], deductive verification [MP95], etc. In this thesis, we choose to use a deductive compositional verification method that exploits compositionality for analysis of timed systems using *invariants*. Invariants are symbolic approximations of the set of reachable states of the system as opposed to the exact reachability analysis in model-checking. The key principle of this approach is the computation of a global invariant as the conjunction of other invariants (components invariants, interaction invariants, etc.).

Let $S = \gamma(B_1, \cdots, B_n)$ be a system composed of n timed components B_i synchronizing through the interaction set γ, and let ψ be a property of interest. Assuming that $GI(S)$ is the global invariant for this system, the verification rule of ψ can be intuitively written as follows:

$$\frac{GI(S) \Rightarrow \psi}{\gamma(B_1, \cdots, B_n) \models \Box\psi}$$

where the notation "$\gamma(B_1, \cdots, B_n) \models \Box\psi$" is to be read as "$\psi$ holds in every reachable state of the composition $\gamma(B_1, \cdots, B_n)$".

Usually when verifying timed systems, the common practice is to verify the system against some *given property* (safety property, liveness, deadlock, etc.). These properties allows to assert that a given model satisfies the specifications. We use satisfiability checking to verify that the global invariant of a system implies properties of interest. Particularly, for a system S characterized by the global invariant $GI(S)$, and given a property ψ, we check the unsatisfiability of $GI(S) \wedge \neg\psi$.

Chapter 3

Modeling Distributed Real-Time Systems

Contents

3.1 Target Architecture 38
3.1.1 Interface 38
3.1.2 From Local Time to Global Time 39
3.1.3 Conflicting Interactions and Interaction Partitioning 39
3.2 3-Layer Send/Receive Model 40
3.2.1 Send/Receive Components 41
3.2.2 Scheduling Layer 44
3.2.3 Conflict Reservation Protocol 48
3.3 Modeling Distributed Real-Time Constraints 51
3.3.1 Communication Delays 51
3.3.2 Clock Drift 52

In the previous chapter, we presented a timed automata based model for representing timed systems with multiparty interactions. The semantics of such model is based on the notion of *global states*, that is, interactions executions are based not only on the state of participating components but on the states of all components of the system. Moreover, this type of model does not provide any details on how an implementation of multiparty interactions can be derived. Conversely, a distributed system can be seen as a collection of loosely coupled independent components communicating by explicit messages passing (components state may be known only through communication). In order to reduce the gap between the high level abstraction of a system and its concrete implementation, we propose an intermediate model more suited for the distributed real-time context, and that is obtained by applying transformation rules on the initial model. The main idea is to explicitly express the ongoing communication mechanism as well as allowing interactions executions based only on their participating components. The key concept of this approach is to structure a given system in two main layers: *(1)* an application layer that consists of a set of distributed components and *(2)* a scheduling layer that is responsible for scheduling the execution of the latter. Additionally, a third layer may be needed by the scheduling layer in order to achieve global consistency.

3.1 Target Architecture

In a distributed context, we consider that components communicate through asynchronous messages passing. Consequently, each component is able either to send a message, to wait for a notification or to execute an internal computation. Our approach proposes an architecture for executing multiparty interactions as a two way handshake protocol [Tri15, TBCB15, BBJ+12] involving asynchronous exchange of messages between *distributed* components and a second layer responsible for triggering interactions, the scheduling layer (see Figure 3.1). In order to evaluate the enabled interactions at a given state, distributed components are required to send their current local information (e.g. enabled actions, invariants, clock constraints, etc.) to the scheduling layer using an *offer* messages. As offers are sent asynchronously, the scheduling layer may not have a global knowledge of the system. It may decide to execute an interaction based only on a partial knowledge, that is, once it accumulates enough offers. We require that the exchange of messages is sender-triggered and not blocking, that is, each time a sender is ready to transmit the corresponding receiver is ready to receive. The class of models satisfying this restriction are called Send/Receive models.

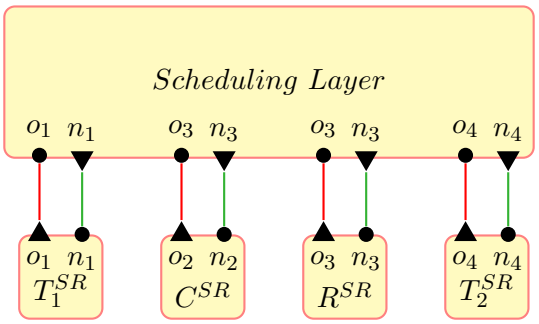


Figure 3.1 – High Level Representation of the Target Architecture

Example 3.1.1. Figure 3.1 depicts a high level representation of the timed system 2.3 in a distributed setting. Components C, T_1, T_2 and R are transformed into the distributed components C^{SR}, T_1^{SR}, T_2^{SR} and R^{SR} respectively. Each component sends information about its current state to the scheduling layer through offer messages ($\{o_1, \cdots, o_4\}$), and is notified through notification messages ($\{n_1, \cdots, n_4\}$). Triangles (respectively dots) indicates the sender (respectively the receiver).

3.1.1 Interface

In order to express the message passing mechanism, we introduce the notion of *communication ports*. A communication port defines the interface of a distributed components, that is how it interacts with the rest of the system. For our purpose, we distinguish three types of ports: send ports, receive ports, and unary ports. A send port is used to export data outside of the sender when sending offers, whereas a receive port imports data inside the receiver when being notified. Unary ports correspond to independent execution of a distributed component, which is formally expressed using a unary interaction (singleton). Effectively, each action of a timed component as presented in Definition 2.2.1 will correspond to a receive port in a distributed component, which is responsible for triggering the execution of the underlying action.

3.1.2 From Local Time to Global Time

In the timed systems model of Chapter 2, every timed component can define a set of local clocks to be used for expressing clock constraints on transitions or the allowed time progress on locations, locally. In our intermediate model, we choose to make use of *global clocks*. In fact, a global clock measures the absolute time elapsed since the system startup and is never reset. This approach allows to have a common timescale between the distributed components and the scheduling layer, which reduces considerably the effort when keeping track of the actual time progress since one needs to maintain only the global clock(s). Notice that any component clock x can be derived from the global clock simply by shifting its value by an amount of time that is constant between successive resets of x. Consequently, achieving the global time mechanism is done by removing local clocks from individual components and adding global clock(s) to the scheduling layer. Moreover, given a global clock g and for each local clock x, we include a variable ρ_x^g that stores the absolute time of its last reset with respect to g. This variable is updated each time a transition resetting x is executed. Then, the value of x can be found by the equality $x = g - \rho_x^g$. As a result, any clock constraint c involved in a component can be expressed using clock g as follows:

$$c = \bigwedge_{x \in \mathcal{X}} l_x \leq x \leq u_x = \bigwedge_{x \in \mathcal{X}} l_x + \rho_x^g \leq g \leq u_x + \rho_x^g \tag{3.1}$$

3.1.3 Conflicting Interactions and Interaction Partitioning

In a distributed context, interaction executions may occur in parallel. However, when two interactions share at least a component it is impossible to execute both interaction concurrently. Particularly, if these interactions are enabled from the same state then they are *conflicting* since they will compete on the same resources (shared component(s)) at the same time. In general, conflicts can be very hard to characterize for real life case studies since they depend on the reachability of particular states. In [Tri15, TBCB15], the computation of the conflicting interactions set relies on over-approximations. It is based on a notion of *potential conflicts* that can be detected by simple syntactic pre-checks, as depicted in Figure 3.2, and are used to quickly exclude conflicts since two interactions that are not potentially conflicting are also not conflicting.

Definition 3.1.1 (Potential Conflict)**.** Two interactions α_1 and α_2 are potentially conflicting if $part(\alpha_1) \cap part(\alpha_2) \neq \emptyset$ and for each component $B_i \in part(\alpha_1) \cap part(\alpha_2)$ there exists two transitions of B_i $e_1, e_2 \in \mathcal{E}_i$ such that $\mathsf{source}(e_1) = \mathsf{source}(e_2)$ and $\mathsf{action}(e_1) \in \alpha_1$, $\mathsf{action}(e_2) \in \alpha_2$.

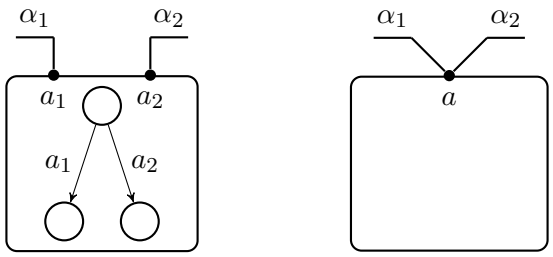


Figure 3.2 – Potential Conflict Between Interactions α_1 and α_2

In order to avoid a centralized scheduling and to introduce concurrency between interaction executions, we propose to decentralize the scheduling layer into several schedulers each one responsible of scheduling a subset of interactions. The purpose behind this practice is to *(i)* spread the workload across concurrent schedulers and as much as possible independently, and *(ii)* map schedulers as close as possible to the components that they concretely handle (with respect to the corresponding subset of interactions), which brings back the communication overhead between components to the same magnitude. Our work does not address interaction partitioning, nonetheless it is a crucial concern for load-balancing and for tuning the system to achieve a desired level of performance.

Definition 3.1.2 (Interaction Partition)**.** Given an interaction set γ, a partition of γ is a set of subset $\{\gamma_k\}_{k=1}^{m}$ such that $\gamma = \gamma_1 \cup \cdots \cup \gamma_m$ and $\forall i, j \in \{1, \cdots, m\}$ such that $i \neq j$, $\gamma_i \cap \gamma_j = \emptyset$.

Decentralizing the schedulers generates situational conflict between interactions, that is, if two interactions handled in separate schedulers (from two class of the interaction partition) are potentially conflicting, they cannot execute in parallel. We call such interactions, *externally conflicting* interactions. A simple solution to resolve such conflicts is to enforce a *conflict-free* partitioning of interactions. In spite of that, this solution will restrict the choice for distributing interactions across schedulers. Thus, another method [BBJ+12, Tri15] is to incorporate a third layer that will arbiter the execution of potentially conflicting interactions. The latter can be represented using a tier component realizing a *conflict resolution protocol* (CRP). This component implements an algorithm based on the idea of messages counting technique [PCT04]. This technique is based on counting the number of times that a component participates in an interaction. Conflicts are then resolved by ensuring that each participation number is used only once, which is achieved by counting the number of the interaction offer for each component. Then, conflicts are simply resolved by comparing the offer numbers of participating components with the numbers of their last execution. On the other hand, conflicts raised from interactions of the same class, that is, handled by the same scheduler, are resolved locally by the latter.

Example 3.1.2. Let us consider example of Figure 2.3. For the interaction set $\gamma = \{\alpha_1, \cdots, \alpha_8\}$, let $\gamma_1 = \{\alpha_{2\times i-1}\}_{i=1}^{4} \cup \gamma_2 = \{\alpha_{2\times i}\}_{i=1}^{4}$ be an interaction partition. Then from Definition 3.1.1 the set of potentially conflicting interactions two-by-two is: $\{(\alpha_1, \alpha_2); (\alpha_3, \alpha_4); (\alpha_5, \alpha_6); (\alpha_7, \alpha_8)\}$. This mean that the set of conflicting interactions of the whole system is γ.

3.2 3-Layer Send/Receive Model

Let $S = \gamma(B_1, \cdots, B_n)$ be a timed system. Given a partition of interactions $\{\gamma_k\}_{k=1}^{m}$, the Send/Receive model corresponding to S is based on the three following layers:

- The *Distributed Component Layer* consists of a transformation of timed components B_i into Send/Receive components B_i^{SR} that send, asynchronously, offer messages enclosing their current state to the scheduling layer

- The *Scheduling Layer* is responsible of interactions executions. Based on offers received form the Send/Receive components, it may decide or not to execute an interaction. In case of conflicts, the scheduling layer relies on the conflict resolution layer to grant or deny the execution of an interaction

- The *Conflict Resolution Layer* resolves conflicts between externally conflicting interactions based on messages counting technique

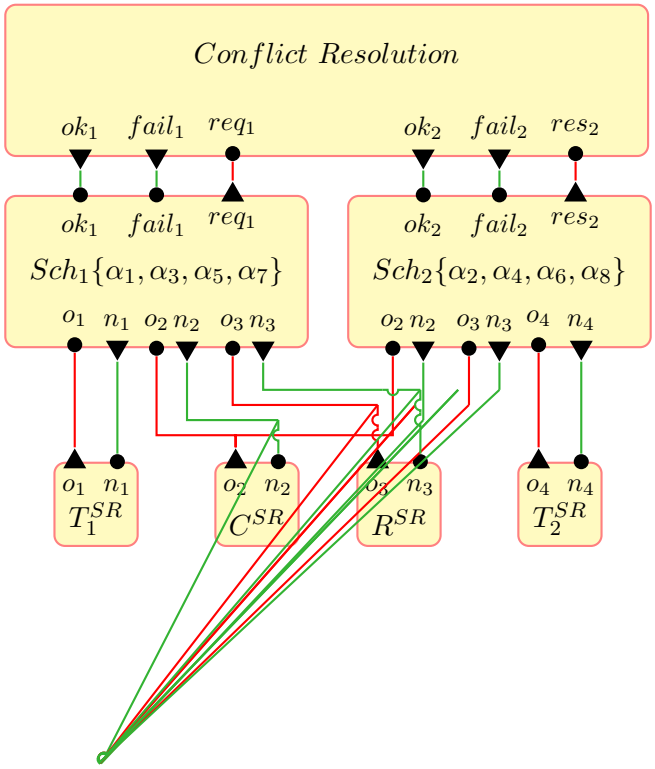


Figure 3.3 – High Level Representation of a Decentralized Send-Receive Model of the Task Manager Example

Example 3.2.1. Figure 3.3 describes a Send/Receive model with a decentralized scheduling. The set of interactions is partitioned into two classes $\gamma_1 = \{\alpha_1, \alpha_3, \alpha_5, \alpha_7\}$ and $\gamma_2 = \{\alpha_2, \alpha_4, \alpha_6, \alpha_8\}$, each one handled by a scheduler (Sch_1 and Sch_2 respectively). Since γ_1 and γ_2 are conflicting, for instance α_1 and α_2 are potentially conflicting, schedulers rely on the conflict resolution layer to resolve the conflicts. In this case, they emit a request (req_k with $k \in \{1, 2\}$) and wait for a notification granting (respectively denying) them the execution of an interaction (ok_k respectively $fail_k$). Notice that components C^{SR} and R^{SR} send offers to both schedulers since they are participating in interactions handled in both schedulers.

3.2.1 Send/Receive Components

The transformation of a timed component B into a Send/Receive component B^{SR} relies on decomposing each transition of B into two transitions: *(1)* an offer (send) transition and *(2)* a notification (receive) transition. This is done by splitting each location ℓ into two locations, ℓ itself and $\ell_\perp$ as shown in Figure 3.4.

When at $\ell_\perp$ location, the Send/Receive component is not in a stable state and is able only to send an offer to its respective scheduler(s). We require that offers are sent as soon as possible meaning that there is no delay when at location $\ell_\perp$. We call such location *urgent* location,

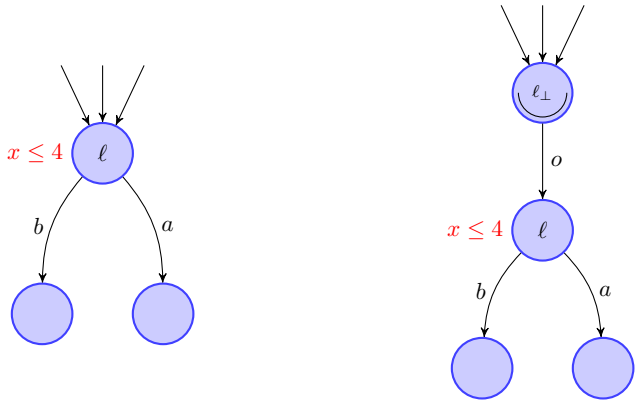


Figure 3.4 – Offer Construction

graphically represented by a $\smile$ inside the location. From a semantics point of view, an urgent location is equivalent to adding an extra clock x that is reset on all incoming edges, and having an invariant $x \leq 0$ on the location as illustrated in Figure 3.5. Thus, time is not allowed to pass when the system is in such locations. An offer contains the exact variables encoding the current state of a component. It includes the following variables:

- An invariant variable of the current location invariant
- A guard variable for each action (port) that is set to the guard (over clocks and data) of each transition outgoing from the current location and labeled by this port if it exists, otherwise to *false*
- A *Boolean* variable indicating whether the last transition reset clocks or not
- A *participation number* variable used for conflict resolution

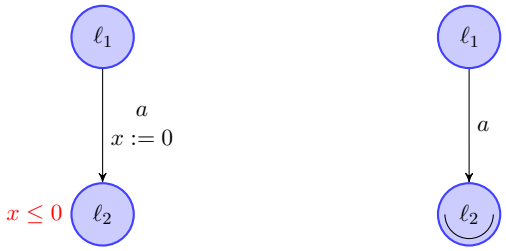


Figure 3.5 – Representation of an Urgent Locatoion

Definition 3.2.1 (Send/Receive Component)**.** Let $B = (\mathcal{L}, \ell_0, \mathcal{X}, \mathcal{D}, \mathcal{A}, \mathcal{E}, \{f_e\}_{e \in \mathcal{E}}, \mathcal{I})$ be a timed component. The corresponding Send/Receive component is defined by the timed component $B^{SR} = (\mathcal{L}^{SR}, \ell_0^{SR}, \emptyset, \mathcal{D}^{SR}, \mathcal{P}^{SR}, \mathcal{E}^{SR}, \{f_e\}_{e \in \mathcal{E}^{SR}}, \emptyset)$, such that:

- $\mathcal{L}^{SR} = \mathcal{L} \cup \mathcal{L}^\perp$, where $\mathcal{L}^\perp = \{\ell_\perp | \ell \in \mathcal{L}\}$. Locations of $\mathcal{L}^\perp$ are urgent locations.
- $\ell_0^{SR} = \ell_{0_\perp} \in \mathcal{L}^\perp$ is the initial location.

- $\mathcal{P}^{SR} = P \cup \{o\}$, where $P = \{p_a | a \in \mathcal{A}\}$ is the set of ports for each action of B and o is the offer port.
- $\mathcal{D}^{SR} = \mathcal{D} \cup \{g_{p_a}\}_{a \in \mathcal{A}} \cup \{inv_B\} \cup \{r_x\}_{x \in \mathcal{X}} \cup \{n_B\}$, where g_{p_a} are guard variables, inv_B is an invariant variable, r_x are Boolean reset variables and n_B is a participation number variable. Variables $\mathcal{D}_o^{SR} \subset \mathcal{D}^{SR} = \{g_{p_a}\}_{a \in \mathcal{A}} \cup \{i\}_B \cup \{n_B\} \cup \{r_x\}_{x \in \mathcal{X}}$ are exported by the offer port.
- For each place $\ell \in \mathcal{L}$, we include an offer transition $e_\ell = (\ell_\perp, o, true, \emptyset, \ell)$ in $\mathcal{E}^{SR}$
- For each transition $e = (\ell, a, g, r, \ell') \in \mathcal{E}$, we include a notification transition $e_{p_a} = (\ell, p_a, true, \emptyset, \ell'_\perp)$. The transfer function $f_{e_{p_a}}$ applies the original transfer function f_e of e, then update guard variables, invariant variable, reset variables and the participation number as follows:
  - $\forall a' \in \mathcal{A}, g_{p_{a'}} := \begin{cases} g_{a'} & \text{if } e' = (\ell', a', g_{a'}, r', \ell'') \in \mathcal{E} \\ false & otherwise \end{cases}$
  - $inv_B := \mathcal{I}(\ell')$
  - $\forall x \in \mathcal{X}, r_x := \begin{cases} true & \text{if } x \in r \\ false & otherwise \end{cases}$
  - $n_B := n_B + 1$

This definition of a Send/Receive component relates the execution of a transition $e = (\ell, a, g, r, \ell') \in \mathcal{E}$ from the initial component B to the following two execution steps in B^{SR}. First, an offer transition $e_\ell = (\ell_\perp, o, true, \emptyset, \ell)$ sends for each port $p \in \mathcal{P}$ the guard over clocks and data corresponding to the enabledness of p at ℓ, the location invariant inv_B, the participation number n_B for component B, as well as the reset variable r_x for each clock $x \in \mathcal{X}$, such that, $r_x = true$, if x has been reset by the previous transition execution. Reset variables r_x are used to reset clocks in the Scheduling layer before computing guard of interactions. In the second place, a notification transition $e_{p_a} = (\ell, p_a, true, \emptyset, \ell'_\perp)$ is executed upon the execution of an interaction involving p_a in the scheduling layer. In the same manner to e in B, e_{p_a} updates values of variables $\mathcal{D}$ according to to the transfer function f_e, as well as variables needed for the next offer. Figure 3.6 depicts the Send/Receive transformation of the component C of Example 2.3.

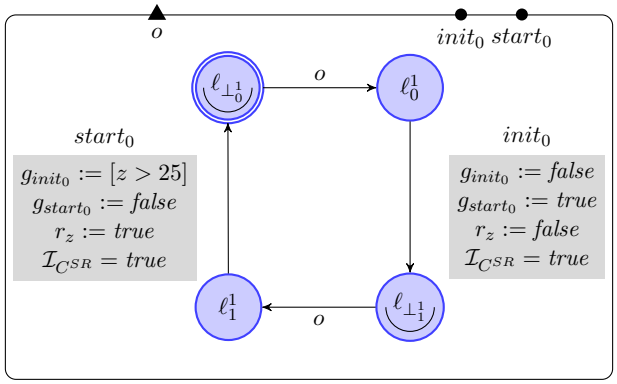


Figure 3.6 – Send/Receive Transformation of the Controller Component From Figure 2.3

3.2.2 Scheduling Layer

As explained earlier, the scheduling layer works with a partial view of the global state of the system. Initially, every scheduler is waiting for offers form the corresponding components (with respect to the interaction partition). Each received offer specifies to the schedulers the state of the sender component. In what follows, we consider work-conservative scheduler, that is, schedulers preserving the execution sequences of the initial model under the standard semantics. For the sake of distributed implementation, it is worth taking a decision as soon as possible. Thus, once a scheduler gathers enough information for scheduling interactions, it arbitrarily choose one and executes the corresponding transitions that will trigger the notification responses to the components involved in that interaction. In what follows, we use Petri nets [Mur89] to describe the scheduling layer. Petri nets are a well suited formalism for encoding parallel and concurrent executions. Particularly, we focus on a class of Petri nets (*1-Safe*) to encode the structure of the schedulers introduced by our method since it is proven that any 1-Safe Petri net can be transformed in an equivalent automaton [DS02]. Consequently, they provide a clear and compact representation of the scheduling layer.

Petri Nets

Definition 3.2.2 (Petri Net)**.** A Petri net is a 3-tuple $\mathcal{P} = (\mathcal{L}, \mathcal{A}, \mathcal{T})$ where $\mathcal{L}$ is a set of finite *places*, $\mathcal{A}$ is a finite set of actions, and $\mathcal{T} \subseteq 2^{\mathcal{L}} \times \mathcal{A} \times 2^{\mathcal{L}}$ is a set of transitions. A transition τ is a triple $({}^\bullet\tau, a, \tau^\bullet)$, where ${}^\bullet\tau$ is the set of input places of τ and $\tau^\bullet$ is the set of output places of τ.

A Petri net can be represented as a directed bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ denotes the set of vertices and $\mathcal{E}$ denotes the set of directed edges. The set of vertices is structured into two classes, namely places and transitions, that is, $\mathcal{V} = \mathcal{L} \cup \mathcal{T}$. Places are represented by circular vertices and transitions are represented by rectangular vertices as shown in Figure 3.7. The set of directed edges $\mathcal{E}$ is the union of the sets $\{(\ell, \tau) \in \mathcal{L} \times \mathcal{T} | \ell \in {}^\bullet\tau\}$ and $\{(\tau, \ell) \in \mathcal{T} \times \mathcal{L} | \ell \in \tau^\bullet\}$. A *marking* of a Petri net is a mapping $m : \mathcal{L} \to \mathbb{N}$ that describes the current *state* of a Petri net by assigning a non-negative integer to each of its places. We use tokens [Mur89] to represent the marking (number of tokens). We say that a place is *marked* if it contains at least one token. For a transition τ, we say that τ is *enabled* at a given state if all of its input places ${}^\bullet\tau$ are marked, that is, $\forall \ell \in {}^\bullet\tau,\ m(\ell) > 0$. A *firing* (execution) of an enabled transition removes one token from each input place and adds one token to each output place. Formally, the firing of a transition from a marking m results in a marking m' such that:

$$\forall \ell \in \mathcal{L},\ m'(\ell) = m(\ell) - \tau^-(\ell) + \tau^+(\ell)$$

where

$$\tau^-(\ell) = \begin{cases} 1 & \text{if } \ell \in {}^\bullet\tau \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \tau^+(\ell) = \begin{cases} 1 & \text{if } \ell \in \tau^\bullet \\ 0 & \text{otherwise} \end{cases}$$

We put $m \xrightarrow{a}_{\mathcal{P}} m'$ to denote that a transition $\tau = ({}^\bullet\tau, a, \tau^\bullet)$ can be executed at marking m and reaches marking m'. We also denote by $\to_{\mathcal{P}}$ the set of triples (m, a, m') such that $m \xrightarrow{a}_{\mathcal{P}} m'$.

Example 3.2.2. Figure 3.7 depicts an example of a Petri net with two successive markings. It includes three places $\{\ell_1, \cdots, \ell_3\}$ and four transitions $\{t_1, \cdots, t_4\}$. For clarity, places with

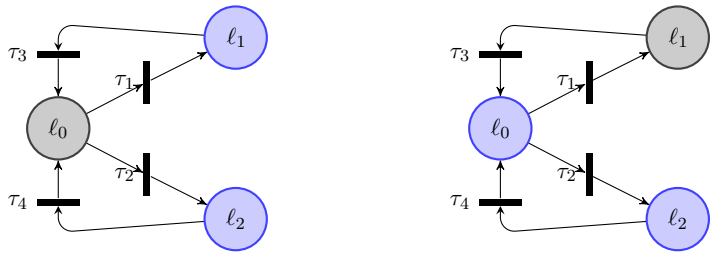


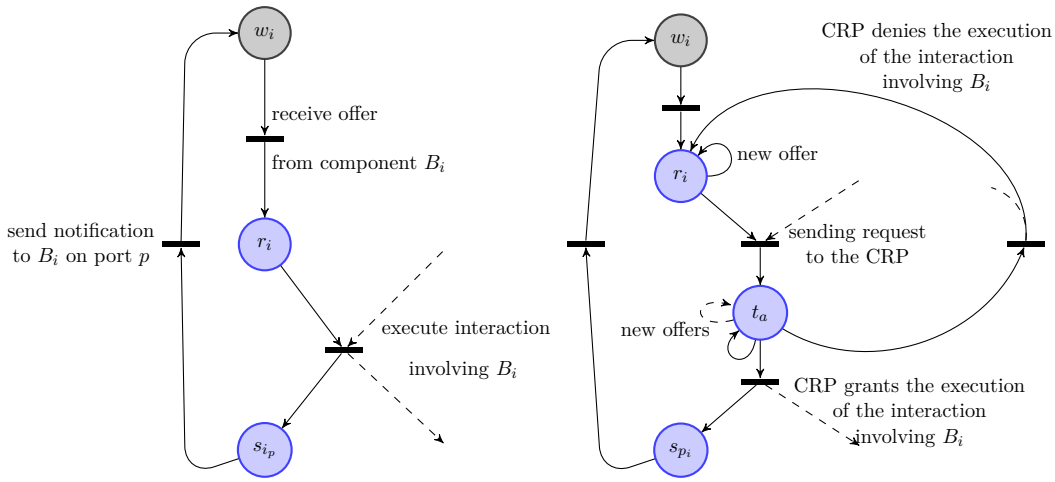
Figure 3.7 – A Simple Petri Net with Two Succesive Markins

tokens are represented with filled gray circles. The left side marking shows the initial marking of the Petri net whereas the right side marking results from the execution of transition t_1.

Given a Petri net $\mathcal{P} = (\mathcal{L}, \mathcal{A}, \mathcal{T})$ and an initial marking m_0, the marking m is *reachable* if there exists a sequence of transitions $m_0 \xrightarrow{a_1}_{\mathcal{P}} m_1 \xrightarrow{a_1}_{\mathcal{P}} \cdots \xrightarrow{a_1}_{\mathcal{P}} m$. We say that $\mathcal{P}$ is *1-Safe* if there is at most one token per place in each reachable marking. This implies at most $2^{|\mathcal{L}|}$ markings. In this thesis, we consider only this class of Petri nets. The behavior of a 1-Safe Petri net $\mathcal{P} = (\mathcal{L}, \mathcal{A}, \mathcal{T})$ is defined by the finite labeled transition system $(2^{\mathcal{L}}, \mathcal{A}, \rightarrow_{\mathcal{P}})$, where $2^{\mathcal{L}}$ is the set of states, $\mathcal{A}$ is the set of actions, and $\rightarrow_{\mathcal{P}} \subseteq 2^{\mathcal{L}} \times \mathcal{A} \times 2^{\mathcal{L}}$ is the set of transitions defined as follows. We have $(m, a, m') \in \rightarrow_{\mathcal{P}}$, denoted by $m \xrightarrow{a}_{\mathcal{P}} m'$, if there exists $\tau = ({}^\bullet\tau, a, \tau^\bullet) \in \mathcal{T}$ such that ${}^\bullet\tau \subseteq m$ and $m' = (m \backslash {}^\bullet\tau) \cup \tau^\bullet$. In this case, we say that a is enabled at m.

Building Schedulers

Given a timed system $\gamma(B_1, \cdots, B_n)$ and a partition of interactions $\{\gamma_j\}_{j=1}^m$, each class of the interaction partition is handled by a single scheduler component, namely Sch_j. The behavior of each scheduler is described as a 1-Safe Petri net in which there is a token for each component flowing between three or four different types of places as shown in Figure 4.3:



(a) Scheduling Mechanism for Internally or not Conflicting Interactions

(b) Scheduling Mechanism for Externally Conflicting Interactions

Figure 3.8 – Scheduling Mechanism

- *Waiting place*: For each component participating in an interaction handled by a scheduler, the corresponding scheduler include a waiting place signifying that it is waiting for the component offer. Waiting places are labeled by w.
- *Receive place*: When receiving an offer from a component, the corresponding token is moved from the corresponding waiting place to the receive place (one received place per component) and stays there until an interaction including this component is scheduled or requested for scheduling (through the conflict resolution layer). Receive places are labeled by r.
- *Try place*: Try places (labeled by t) concern only components that are participating in an interaction that is externally conflicting with another interaction (of another scheduler). As explained in Subsection 3.1.3, schedulers rely on the conflict resolution layer to resolve conflicts. For each externally conflicting interaction, a try place is inserted. When scheduling such interactions, tokens are moved from receive places of components to try place of that interaction, meaning that a request has been sent to the CRP. Following this request, the CRP either grants the execution of the interaction and the tokens are moved to the sending places, or denies the execution which results in moving the tokens back to the receive places. Moreover, loops for offer transitions are added on try places and receive places in order to take into account successive offers from components.
- *Sending place*: Once an interaction has been scheduled for execution, the corresponding components token are moved from receive places (or try place) to send places corresponding to ports of components participating in that interaction. There is one send place for each port (excluding offer port) for every Send/Receive component. Sending places are labeled by s.

As illustrated in Figure 4.3, tokens are initially in waiting places. Once an offer is received by a scheduler, the corresponding token moves to the receive place. The scheduler copies then the values of the offer variables to its local variables. Once offers of all components involved in an interaction have been gathered, schedulers computes its guard. If the guard evaluates to *true*, with respect to data variables and the global scheduler clock, the scheduler can either execute the interaction if it is not externally conflicting with another interaction (Figure 3.8a). Tokens are then moved to send places of components ports participating in that interaction. Otherwise (Figure 3.8b), a request is sent to the CRP and the token is moved to the corresponding *try* place. Thereafter, either the CRP grants the execution of the interaction and tokens are moved to send places, or the execution is denied and tokens are moved back to receive places. Eventually, the scheduler may receive new offers when being in *try* places. This corresponds to the execution of a conflicting interaction in another scheduler.

Definition 3.2.3 (Scheduler)**.** Let $\gamma(B_1, \cdots, B_n)$ be a timed system and $\gamma_j \subset \gamma$ be a subset of interactions. The corresponding scheduler Sch_j responsible for executing interactions of γ_j is defined by the tuple $Sch_j = (\mathcal{L}_j, \mathcal{P}_j, \mathcal{T}_j, \mathcal{X}_j, \mathcal{D}_j, \{g_\tau\}_{\tau\in\mathcal{T}_j}, \{r_\tau\}_{\tau\in\mathcal{T}_j}, \{f_\tau\}_{\tau\in\mathcal{T}_j}, \{\mathcal{I}_\ell\}_{\ell\in\mathcal{L}_j})$, where $(\mathcal{L}_j, \mathcal{P}_j, \mathcal{T}_j)$ is a 1-Safe Petri net defining the structure of the scheduler such that:

- $\mathcal{X}_j = \{t_j\} \cup \{z_j\}$ is the set of clocks of Sch_j, where t_j is the global clock used for scheduling interactions of γ_j (it is never reset) and z_j is a clock used for internal constraints.
- $\mathcal{D}_j$ is the set of variables containing:

  - Variables updated whenever an offer from a component B_i participating in interactions of γ_j is received. These variables consist of: an invariant variable inv_{B_i} and a participation number n_{B_i} for each B_i, a guard variable g_{p_a} for each action of B_i involved in an interaction of γ_j, and a Boolean reset variable for each clock of B_i.
  - Reset time variables that stores the absolute time of the last reset of each component clocks. For each clock x of B_i we include a reset time variable ρ_x.

- For each transition $\tau \in \mathcal{T}_j$, g_τ is a guard over $\mathcal{X}_j$ and $\mathcal{D}_j$.
- For each transition $\tau \in \mathcal{T}_j$, r_τ is a reset function over $\mathcal{X}_j$.
- For each transition $\tau \in \mathcal{T}_j$, f_τ is a transfer function over$\mathcal{D}_j$.
- For each place $\ell \in \mathcal{L}_j$, $\mathcal{I}_\ell$ is an invariant over $\mathcal{X}_j$.
- The 1-Safe Petri net $(\mathcal{L}_j, \mathcal{P}_j, \mathcal{T}_j)$ is structured as follows:
  - $\mathcal{L}_j$ is the set of places. It includes four types of places:
    * For each component B_i involved in interactions of γ_j, we include a waiting place w_i^j, a receive place r_i^j, where $\mathcal{I}_{w_i^j} = true$ and $\mathcal{I}_{r_i^j}$ is the invariant inv_{B_i} expressed on t_j.
    * For each action a involved in interactions of γ_j, we include a sending place s_{p_a}, where $\mathcal{I}_{s_{p_a}} = z_j \leq 0$.
    * For each interaction $\alpha \in \gamma_j$ that is externally conflicting with another interaction, we include a try place t_α with $\mathcal{I}_{t_\alpha}$ is the invariant inv_{B_i} expressed on t_j.
  - $\mathcal{P}_j$ is the set of ports. It includes the following ports:
    * For each component B_i involved in interactions of γ_j, we include a receive port o_i^j. Each port o_i^j is associated with the variables g_{p_a} and r_x for each action, respectively clock, of B_i, as well as the variables inv_{B_i} and n_i.
    * For each action a involved in interactions of γ_j, we include a send port p_a.
    * For each interaction $\alpha \in \gamma_j$ that is externally conflicting with another interaction, we include a send port rsv_α (reservation port), and receive ports ok_α (granted execution) and $fail_\alpha$ (denied execution). The port rsv_α exports the variables $\{n_i\}_{B_i \in part(\alpha)}$.
    * For each interaction $\alpha \in \gamma_j$ that is internally or not conflicting with other interactions of γ_j, we include the unary port α.
  - $\mathcal{T}_j$ is the set of transitions. It consists of the following:
    * For each component B_i, $\mathcal{T}_j$ includes the offer transitions (w_i^j, o_i, r_i^j), (r_i^j, o_i, r_i^j) and $\{(t_\alpha, o_i, t_\alpha) | B_i \in part(\alpha)\}$ where α is an externally conflicting interaction. The execution of offer transitions copy the variables attached to components offer to its local variables. Offer transitions have no guards and no reset functions. Their transfer functions update reset time variables ρ_x whenever $r_x = true$, that is, $\rho_x := t_j$.
    * For each action a involved in interactions of γ_j, $\mathcal{T}_j$ includes a transition (s_{p_a}, p_a, w_i^j) where i is the index of the component containing a. This transition notifies the corresponding Send/Receive component to execute the transition labeled by p_a. It has no guard, no reset function, and no transfer function.

- For each interaction $\alpha = \{a_i\}_{i \in I} \in \gamma_j$, that is internally conflicting or not conflicting with interactions of γ_j, $\mathcal{T}_j$ includes the transition $\tau_\alpha = (\{r_i^j\}_{B_i \in part(\alpha)}, \alpha, \{s_{p_{a_i}}\}_{a_i \in \alpha})$. The guard of this transition is $g_{\tau_\alpha} = \bigwedge_{a_i \in \alpha} g_{p_{a_i}}$. Notice that guards over data are the same, whereas guard of clocks are expressed using the global clock t_j and the reset time variables ρ_x. This transition has no transfer function and its reset function resets the clock z_j.
- For each interaction $\alpha = \{a_i\}_{i \in I} \in \gamma_j$, that is externally conflicting, $\mathcal{T}_j$ includes the following transitions:
  - $\tau_{rsv_\alpha} = (\{r_i^j\}_{B_i \in part(\alpha)}, rsv_\alpha, t_\alpha)$. The guard of this transition is $g_{\tau_{rsv_\alpha}} = \bigwedge_{a_i \in \alpha} g_{p_{a_i}}$.
  - $\tau_{ok_\alpha} = (t_\alpha, ok_\alpha, \{s_{p_{a_i}}\}_{a_i \in \alpha})$. This transition has no guard, no transfer function, and its reset function is $r_{\tau_{ok_\alpha}} = \{z_j := 0\}$.
  - $\tau_{fail_\alpha} = (t_\alpha, fail_\alpha, \{s_{p_{a_i}}\}_{a_i \in \alpha})$. This transition has no guard, no reset function, and no transfer function.

Notice that Definition 3.2.3 presents only the syntax of a scheduler. It uses the Petri net formalism only for compactness purposes and is not to be confused with any other Petri Net formalism such as Time Petri Nets or Timed Petri nets.

Example 3.2.3. Figure 4.3 depicts the internal representation of scheduler Sch_1 from Figure 3.3. The scheduler Sch_1 is responsible of the interaction class $\gamma_1 = \{\alpha_1, \alpha_3, \alpha_5, \alpha_7\}$, that is, he is responsible of notifying components C, T_1 and R whose indexes are respectively 1, 2 and 4. Notice that since every interaction of γ_1 is potentially conflicting with an interaction of γ_2 handled by scheduler Sch_2, scheduling interactions of γ_1 requires the intervention of the conflict resolution layer.

Property 3.2.1 (Scheduler Semantics)**.** *Let $Sch = (\mathcal{L}, \mathcal{P}, \mathcal{T}, \mathcal{X}, \mathcal{D}, \{g_\tau\}_{\tau \in \mathcal{T}}, \{r_\tau\}_{\tau \in \mathcal{T}}, \{f_\tau\}_{\tau \in \mathcal{E}}, \{\mathcal{I}_\ell\}_{\ell \in \mathcal{L}})$ be a tuple defining a scheduler. Let $(2^{\mathcal{L}}, \mathcal{P}, \rightarrow_{\mathcal{P}})$ be the finite labeled transition system of its underlying 1-Safe Petri net. The semantics of the scheduler Sch is equivalent to the semantics of the timed component $(2^{\mathcal{L}}, \ell_0, \mathcal{X}, \mathcal{D}, \mathcal{P}, \mathcal{E}, \{f_e\}_{e \in \mathcal{T}}, \mathcal{I})$ such that:*

- *$\ell_0 = \otimes_{\ell \in m_0 | m_0(\ell) = 1} \ell$, where m_0 is the initial marking of the Petri net $(\mathcal{L}, \mathcal{P}, \mathcal{T})$.*
- *For each $(m_1, p, m_2) \in \rightarrow_{\mathcal{P}}$ we include a transition $e \in \mathcal{E} = (\ell_1, p, g_p, r, \ell_2)$ such that:*
  - *$\ell_1 = \otimes_{\ell \in m_1 | m_1(\ell) = 1} \ell$ and $\ell_2 = \otimes_{\ell \in m_2 | m_2(\ell) = 1} \ell$. The invariant of ℓ_1 and ℓ_2 are respectively $\mathcal{I}(\ell_1) = \bigwedge_{\ell \in m_1 | m_1(\ell) = 1} I_\ell$ and $\mathcal{I}(\ell_1) = \bigwedge_{\ell \in m_2 | m_2(\ell) = 1} I_\ell$.*
  - *$g_p = g_\tau$, $f_e = f_\tau$, and $r = r_\tau$ where $\tau = ({}^\bullet\tau, p, \tau^\bullet) \in \mathcal{T}$ such that ${}^\bullet\tau \subseteq m$ and $m' = (m \backslash {}^\bullet\tau) \cup \tau^\bullet$.*

3.2.3 Conflict Reservation Protocol

In this subsection, we present the third layer of our Send/Receive architecture, namely the conflict resolution layer. The main purpose of this layer is to resolve the conflict that occur between interactions of separate schedulers at run time. Since interactions may compete on resources (here sharing components), the conflict resolution layer implements a protocol, inspired from [PCT04] and based on messages counting technique, that allows to check the freshness of offers received for the execution of an interaction from schedulers. In other words,

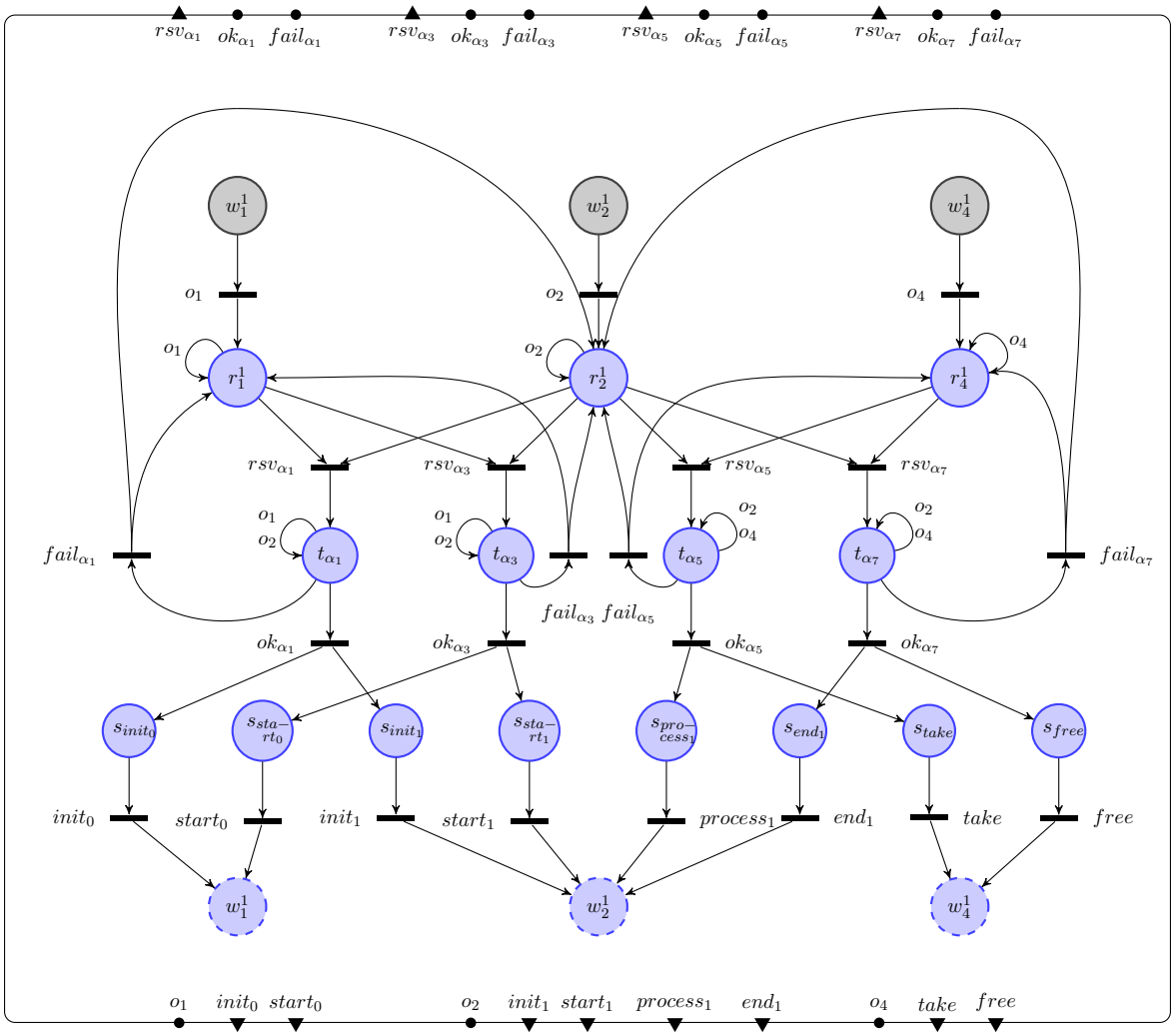


Figure 3.9 – Internal Representation of Scheduler Sch_1 from Figure 3.3

it ensures that two externally conflicting interactions cannot execute with the same offers by checking that the participation numbers of the involved components have not been yet consumed. Particularly, the protocol keeps the last participation number of each component and compares it with the participation number from the reservation request of a scheduler and thereafter, decides whether to grant a scheduler or not the execution of an interaction.

There exists several implementations of the conflict resolution protocol [Bag89]. We present here only one variant since our interest is not in studying the conflict resolution layer. It is centralized variant based on Bagrodia's protocol.

Definition 3.2.4 (Conflict Resolution Protocol)**.** Let $\gamma(B_1, \cdots, B_n)$ be a timed system and $\{\gamma_j\}_{j=1}^m$ be an interaction partition. The corresponding centralized conflict resolution protocol component is defined by the timed component $CP = (\mathcal{L}^{CP}, \ell_0^{CP}, \emptyset, \mathcal{D}^{CP}, \mathcal{P}^{CP}, \mathcal{E}^{CP}, \{f_e\}_{e \in \mathcal{E}^{CP}}, \emptyset)$ such that:

- $\mathcal{L}^{CP}$ contains for each externally conflicting interaction α a waiting location w_α and a receive location r_α. Receive locations are urgent locations.

- $\mathcal{D}^{CP}$ includes for each component B_i participating in conflicting interactions α its current participation number n_i^α as well as the last participation number N_i.
- $\mathcal{P}^{CP}$ includes for each externally conflicting interaction a reservation port rsv_α, ok_α and $fail_\alpha$. The port rsv_α imports the variables $\{n_i^\alpha | B_i \in part(\alpha)\}$.
- $\mathcal{E}^{CP}$ includes for each externally conflicting interaction α the following transitions:
  - A reservation transition $e_{rsv_\alpha} = (w_\alpha, rsv_\alpha, true, \emptyset, r_\alpha)$
  - A transition granting the execution of α, $e_{ok_\alpha} = (r_\alpha, ok_\alpha, g_{ok_\alpha}, \emptyset, w_\alpha)$ such that, $g_{ok_\alpha} = \bigwedge_{B_i \in part(\alpha)} n_i^\alpha > N_i$ and $f_{e_{ok_\alpha}} = \{\forall B_i \in part(\alpha), N_i := n_i^\alpha\}$.
  - A transition denying the execution $e_{fail_\alpha} = (r_\alpha, fail_\alpha, true, \emptyset, w_\alpha)$

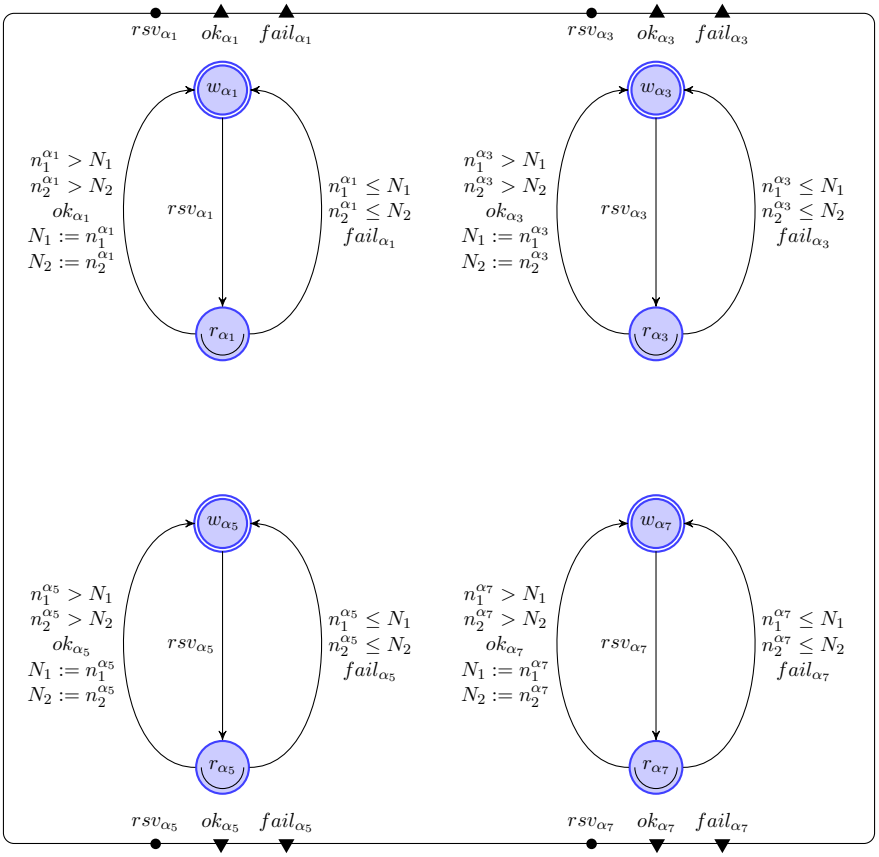


Figure 3.10 – Sub-Part of the Timed Component for the Centralized CRP of Figure 3.3 Handeling Interactions of γ_1

In what follows, we present the Send/Receive interactions that link the three layer of the presented Send/Receive model.

Definition 3.2.5 (Send/Receive Interactions)**.** Let $\gamma(B_1, \cdots, B_n)$ be a timed system and $\{\gamma_j\}_{j=1}^m$ be an interaction partition. The Send/Receive interactions γ^{SR} connecting the three layers of the Send/Receive models are:

- For each component B_i^{SR}, we include an offer interaction involving B_i^{SR} and its respective schedulers $\{B_i^{SR}.o, Sch_{j_1}.o_i, \cdots, Sch_{j_k}.o_i\}$.
- For each port p of a component B_i^{SR} and for each scheduler Sch_j handling an interaction involving p, we include a notification interaction $\{B_i^{SR}.p, Sch_j.p\}$.
- For each internally conflicting or not interaction $\alpha \in \gamma$ handled by a scheduler Sch_j, we include the unary interaction $\{Sch_j.a\}$.
- For each externally conflicting interaction $\alpha \in \gamma$, we include the following interactions:
  - $\{Sch_j.rsv_\alpha, CP.rsv_\alpha\}$
  - $\{Sch_j.ok_\alpha, CP.ok_\alpha\}$
  - $\{Sch_j.fail_\alpha, CP.fail_\alpha\}$

The correctness of the Send/Receive transformation is proved using observational equivalence, that is, weak bisimulation.

Theorem 3.2.1 (Correctness [Tri15])**.** $T \dot{\sim} \mathcal{T}^{SR}$.

The correctness of the presented approach is necessary to attest that both the initial and resulting systems have the same behavior. Nonetheless, the proof of correctness has already been established and its details are not relevant to the content of this thesis. The interested reader can find all the steps of the proof in Chapter 5 of [Tri15].

3.3 Modeling Distributed Real-Time Constraints

Distributed real-time systems are prone to different kind of problems. The immediate concern is the communication delays inherent to distributed platforms. The latter increase considerably the effort of coordinating the parallel activities of running components. Thus, scheduling such systems must cope with the induced delays by proposing execution strategies ensuring global consistency while satisfying the imposed timing constraints. Another phenomenon intrinsic to distributed platforms is clock drift. A clock is a device that consists of a counter that is incremented periodically according to the frequency of an oscillator. This implies that clocks are not perfect since the oscillator frequencies may vary during their lifetime due to several factors such as aging, temperature, humidity, etc. Consequently, clocks trend to *drift* or gradually desynchronize from a given reference time. Moreover, when having multiple clocks running in the same system, which is usually the case in distributed real-time systems, the relative clock drift between these clocks may result in an unexpected (even undesirable) behavior. The common practice is to resynchronize the clocks (internally or externally) in order to bring the difference to a certain threshold to minimize the impact of this phenomenon.

3.3.1 Communication Delays

The Send/Receive model as presented in Section 3.2 assumes implicitly that communication between the different layers is timeless. This restricts the applicability of such approach to applications where the timing constraints are far bigger than the communication delays imposed by a given target platform. To cope with those delays, a variant of the Send/Receive approach

was presented in [Tri15]. It is based on an early decision making mechanism where schedulers plan interactions executions ahead and notify components of their execution dates in advance. The main issue of this method is that when planning components to execute at a given time, all the interaction including the planned components will be ineligible for execution (and even for planning). This may locally block components, especially if all the interactions involving these components are disallowed from execution. To overcome this problem, this adaptation of the Send/Receive models suggests that each scheduler, additionally to the components he is handling, observe a subset of components that may be blocked when scheduling interactions. However, because of the nature of the location invariants (local constraints that propagate on the global level), this technique results in observing all the components of the system, instead of only a subset as presented in [Tri15].

In order to model the behavior of a system under some communication delays bounded by $h_{\min}$ ($h_{\min}$ being the worst case communication delay), we introduce the *local planning semantics*. This semantics aims to distinguish between the decision dates for executing interactions and their actual execution dates by adding a notion of *planning* on the semantics level. The delay between the planning of an interaction and its execution is thus constrained by $h_{\min}$, which is a parameter of the semantics. Although this approach is based on the same idea of anticipating the executions of interactions, it differs from the approach of [Tri15] in the following points:

- The class of system handled by the method of [Tri15] is restricted to timed components with closed guards, that is, where clock constraints are of the form:

  $$c := \mathit{true} \mid x \leq k \mid x \geq k$$

  where x is a clock and $k \in \mathbb{Z}_{\geq 0}$. Moreover, this method is restricted to timed components with non-decreasing deadlines. In other words, if time can progress by d from a given state (ℓ, υ) of a component, it can also progress by d from any state (ℓ', υ') reached by executing an action a from state (ℓ, υ). Our approach on the other hand, imposes only that the system is free of modelling errors such as deadlocks or timelocks.

- Our approach works on the semantics level whereas the method of [Tri15] is based on transformations and model constructions. The main advantage of working on the semantics level is that it allows to stay at certain level of abstraction, close enough to the original model, which reduce considerably the chance of errors during the formalization. Furthermore, one can still imagine a Send/Receive like transformation that implements this semantics.

- Unlike the standard semantics of timed system, the local planning semantics is based on a local view of the system which is more suited for the distributed context.

Chapter 5 presents a detailed description of the local planning semantics, its properties and relations with the standard semantics of timed systems. It also provides sufficient conditions that guarantee the correctness (in terms of behaviour) of a given application under some bounded communication delays.

3.3.2 Clock Drift

Reachability analysis has been used to test the behavior of timed automata models against some desired properties. However, such analysis techniques whether region-based or zone-based can

be incorrect and misleading, since they rely on several assumptions such as zero response times or infinitesimally precise clocks, which is generally not the case in reality. In practice, clocks are implemented using an oscillator and a counting register, and their precision based on the quality of the oscillator together with the operating environment. The common practice when studying the effect of clock imperfections is to define a perturbation model that approximates the behavior of a given model under clock drifts in order to study its robustness. Robust reachability has been introduced to check whether a given system still satisfies the specification when subject to different perturbations such as clocks drift. In [Pur00], Puri introduced a model of clock drift for closed timed automata by introducing a parameter $\epsilon > 0$ that bounds the clocks drift rates. This work showed that the standard reachability analysis approach is not correct when clocks drift, even by infinitesimally small amount, and subsequently suggested a region based method for calculating $Reach^*(S)$, the set of reachable states for *every* drift (the limit as $\epsilon \to 0$), that is, $Reach^*(S) = \cap_{\epsilon>0} Reach(S_\epsilon)$. Other works [DK06, Dim07] proposed a zone based algorithm for computing this reach-set more efficiently, and generalized the approach for open timed automata model [Dim07]. In [WDMR04, Pur00], another perturbation model was considered. Here, the system model is syntactically modified by *relaxing* the guards through a parametric enlargement of δ. Dewulf [WDMR04] showed that the notion of robustness defined in [Pur00], and studied in other works [DK06, Dim07], is closely related to the notion of implementability introduced in [WDMR04], that is, whether for some $\delta > 0$, the enlarged system model still satisfies the requirements expressed by the considered properties. This allowed to prove that the considered notion of implementability is decidable for timed automata. Finally, [SFK08] consider a more realistic model of drifting clocks by considering clock resynchronization, available now in most distributed real-time systems. It was proven that standard zone-based reachability analysis is exact when testing robust safety, provided a uniform strictly positive robustness margin of 1. In Chapter 6, we present a timed automata based model for distributed real-time systems where the relative drift between clocks is assumed to be bounded (clocks are assumed to be resynchronized with a certain threshold). The resulting timed transition system includes straightforwardly more states than the initial model. We then give interesting properties of the drifted model and provide a strategy that allows, for any resulting execution trace, to stay close enough to a "similar" trace of the initial model.

Part II

Contribution

This part includes our contributions to the field of modeling and validation of distributed real-time systems. First, Chapter 4 proposes a knowledge based optimization of the Send/Receive transformation. It aims at reducing the interactions between the scheduling layer and the conflict resolution layer through a reduction of the potentially conflicting interactions set. Thereafter, Chapter 5 and 6 study the behavior of a given model when subject to constraints inherent to the distributed context. Chapter 5 tackles the problem of communication delays by proposing a strategy based on anticipating the execution of components beforehand. It provides sufficient conditions that allow to check whether a given system is robust or not (in the sense not guaranteed) to communication delays. We also propose an alternative method based on real-time controller synthesis and explain how it differs from our approach. In the same way, Chapter 6 investigates the clock drift problem and proposes a strategy that ensures that executions of the drifted system stay close enough from executions of the model with perfect clocks.

Chapter 4

Knowledge Based Optimization of Distributed Real-Time Systems

Contents

4.1 Conflicting Interaction Calculation 57

4.2 Knowledge Based Reduction of Potentially Conflicting Interaction 58

4.2.1 Linear Invariants 59

4.2.2 History Clocks Inequalities 60

4.3 Impact of Conflict Reduction on Send/Receive Models 62

4.1 Conflicting Interaction Calculation

As explained in Subsection 3.1.3, two interactions α_1 and α_2 sharing a subset of components cannot execute concurrently. Particularly, if these interactions are enabled from the same state they are conflicting, meaning that they are competing on the same resources (shared components) and only one interaction will be granted the execution and not the other. Particularly, given a timed system $\gamma(B_1, \cdots, B_n)$ and an interaction partition $\{\gamma_j\}_{j=1}^{m}$, if such interactions are part of two different class of the interaction partition (externally conflicting interactions), then the resulting Send/Receive model requires the intervention of the conflict resolution layer to resolve the conflict situation. The computation of the conflicting interactions set is based on syntactic pre-checks (Definition 3.1.1), that is, it is an over-approximation that in some cases induces an unnecessary conflict resolution. The calculation of the conflicting interactions set highly impacts the structure and the performance of the underlying Send/Receive model. For every interaction that is in fact not conflicting, the corresponding scheduler includes an additional place and three transitions involved in the Send/Receive interactions involving the conflict resolution layer. In this case, executing an interaction adds not only an evaluation overhead, but also latency resulting from the communication delays between the two layers. In order to refine the conflicting interactions set, we rely on the following definition of conflicts.

Definition 4.1.1 (Conflicting Interactions)**.** Let $S = \gamma(B_1, \ldots, B_n)$ be a timed system. Two interactions α_1 and α_2 of γ are *conflicting*, and we write $\alpha_1 \# \alpha_2$, if $part(\alpha_1) \cap part(\alpha_2) \neq \emptyset$ and

there exists a reachable state from which both α_1 and α_2 are enabled, i.e., a state satisfying:

$$Conflict(S, \alpha, \beta) = Reach(S) \wedge Enabled(\alpha_1) \wedge Enabled(\alpha_2) \tag{4.1}$$

where $Rreach(S)$ is the set of reachable states of the S.

The above definition of conflicts characterizes the exact set of conflicting interactions. Especially, it considers that two interactions are not conflicting if the whole system cannot reach a state from which both can potentially execute. Clearly, such interactions do not require conflict resolution as they cannot be scheduled based on common offers. In what follows, we propose an approach that aims to reduce the set of potential conflicts. In fact instead of calculating the exact set of reachable states of a given system, we use static analysis techniques to extract a *knowledge* that represents an over approximation of the reachable states of the system on the form of invariants.

4.2 Knowledge Based Reduction of Potentially Conflicting Interaction

Knowledge as referred to it here can be interpreted as any information that gives a characterization of a given system. We distinguish two types of knowledge: Local knowledge that captures partial information of a system on components level, and a global knowledge that relates these local knowledge and link them together. Our approach includes two main steps: the first step *(i)* consists of constructing the set of potentially conflicting interactions based on Definition 3.1.1. This step aims mainly to distinguish non conflicting interactions from those that can potentially conflict in order to ease and avoid unnecessary checks during the second step. Then, the second step *(ii)* calculates then combines local and global knowledge of the system on the form of invariants. The latter will represents an over-approximation of the reachable state of the system. After that, by replacing $Reach(S)$ by its over-approximation $\overline{Reach(S)}$ in Expression 4.1, potentially conflicting interactions are reduced by checking the following precondition:

$$\overline{Conflict(S, \alpha_1, \alpha_2)} = \overline{Reach(S)} \wedge Enabled(\alpha_1) \wedge Enabled(\alpha_2) \tag{4.2}$$

Notice that since $Reach(S) \Rightarrow \overline{Reach(S)}$, we obtain that $Conflict(S, \alpha_1, \alpha_2) \Rightarrow \overline{Conflict(S, \alpha_1, \alpha_2)}$. Thus, if two interactions are established to be conflicting according to Expression 4.2, then they are conflicting according to Expression 4.1. Hereinafter, *false* conflicts refers to potential conflicts as defined in Definition 3.1.1 but that are not conflicts with respect to Definition 4.1.1.

A potential conflict between two interactions is a *false* conflict either: *(i)* because the system cannot reach a global location configuration enabling both interactions, or *(ii)* because both interactions are not enabled at the same time due to timing constraints. In the following, we show how to compute invariants for removing false conflicts of types *(i)* and *(ii)*. This invariants combined with individual reachable states of components will represent our over-approximation of $Reach(S)$.

4.2.1 Linear Invariants

Linear invariants consist of linear constraints that allow to reason on complex properties. For instance, they can be used to *count* how many processes are at a given state of a concurrent system. Such invariants have been widely used in different domain. Particularly, we are interested in the so called linear state-invariants [Mur89, KJ86, SSM03] from the Petri net community. Linear state-invariants or (S-invariants) are determined to be appropriate for proving non-coverage of subsets of individual locations, which corresponds exactly to what needed to prove that two interactions cannot be enabled from the same location configuration. Precisely, linear invariants consist of linear combinations of $\mathsf{at}(\ell)$ predicates that are equal to a constant. For instance $\mathsf{at}(\ell_0) + \mathsf{at}(\ell_1) + \mathsf{at}(\ell_2) = 1$ is a linear constraint for the Petri net of Figure 3.7.

Locations configurations reachable in a composition $S = \gamma(B_1, \ldots, B_n)$ are necessary combinations of reachable locations of individual components B_i. However, in general not all combinations are reachable in S since components are not fully independent as they synchronize through interactions. A typical example of that is a shared resource used in mutual exclusion by a set of components: any of them can potentially use it, but they should coordinate so that states in which two (or more) components use the resource are not reachable. Another illustration of this can be found in example of Figure 2.3: components T_1 (resp. T_2) may reach location ℓ_1^2 (resp. ℓ_1^3) by executing action $init_1$ (resp. $init_2$), but in the composition T_1 and T_2 cannot be simultaneously at locations ℓ_1^2 and ℓ_1^3. This is due to interactions $\alpha_1 = \{init_0, init_1\}$ and $\alpha_2 = \{init_0, init_2\}$ with component C: executing α_1 disables α_2, and vice versa. That is, the potential conflict between interactions α_3 and α_4 can be excluded if we consider reachable locations of the composed system.

Definition 4.2.1 (Linear Invariant)**.** Let $S = \gamma(B_1, \ldots, B_n)$ be a timed system and $\mathcal{L} = \bigcup_{1 \leq i \leq n} \mathcal{L}_i$ being all components locations, with $\mathcal{L}_i$ the set of locations of B_i. A *linear invariant* of S is a linear equality constraint which holds in all reachable global state of S. It is of the form:

$$\sum_{\ell \in \mathcal{L}} u_\ell \cdot \mathsf{at}(\ell) = u_0,$$

where u_ℓ, and u_0 are integers, in which predicates $\mathsf{at}(\ell)$, $\ell \in \mathcal{L}$, are interpreted as 0 for false and 1 for true.

To compute linear invariants for a timed system $S = \gamma(B_1, \ldots, B_n)$, we consider its untimed version $\tilde{S}$ abstracting all data and timing aspects of S (i.e. obtained from S by relaxing guards and location invariants of components). Note that linear invariants for $\tilde{S}$ are also linear invariants for S, since reachable locations of S are necessary included in reachable locations of $\tilde{S}$. Methods for calculating linear invariants are based on linear algebra, and more precisely on the *characteristic system* [KJ86] (also known by the place-transition matrix in the Petri nets community). It consists of a system of linear equations representing the interactions of a given system.

Definition 4.2.2. [Characteristic System] For a timed system $S = \gamma(B_1, \cdots, B_n)$ and $\mathcal{L} = \mathcal{L}_1 \times \cdots \times \mathcal{L}_n$, the set of all global location configuration. The characteristic system is defined as follows:

$$\mathcal{M}(S) = \bigwedge_{\alpha \in \gamma} \bigwedge_{\ell \in \mathcal{L}_\alpha} \Big(\sum_{\ell_i \in \alpha^\bullet} x_{\ell_i} - \sum_{\ell_j \in {}^\bullet\alpha} x_{\ell_j} \Big)$$

where $\mathcal{L}_\alpha$ is the subset of locations configurations from which α is possible, and $\alpha^\bullet$, ${}^\bullet\alpha$ denotes respectively the destinations and sources locations of components actions involved in α.

Example 4.2.1. The characteristic system for Example 2.2.3 following the enumeration of all interactions of γ is:

$$\mathcal{M}(S) = \begin{cases} x_{\ell_1^1} - x_{\ell_0^1} + x_{\ell_1^2} - x_{\ell_0^2} = 0 \\ x_{\ell_1^1} - x_{\ell_0^1} + x_{\ell_1^3} - x_{\ell_0^3} = 0 \\ x_{\ell_0^1} - x_{\ell_1^1} + x_{\ell_2^2} - x_{\ell_1^2} = 0 \\ x_{\ell_0^1} - x_{\ell_1^1} + x_{\ell_2^3} - x_{\ell_1^3} = 0 \\ x_{\ell_1^4} - x_{\ell_0^4} + x_{\ell_3^2} - x_{\ell_2^2} = 0 \\ x_{\ell_1^4} - x_{\ell_0^4} + x_{\ell_3^3} - x_{\ell_2^3} = 0 \\ x_{\ell_0^4} - x_{\ell_1^4} + x_{\ell_0^2} - x_{\ell_3^2} = 0 \\ x_{\ell_0^4} - x_{\ell_1^4} + x_{\ell_0^3} - x_{\ell_3^3} = 0 \end{cases}$$

The common techniques for solving homogeneous systems $A.x = 0$ are the Guass-Jordan elimination, Cholesky-, QR- or LU-factorization. These algorithms have low polynomial complexity and can be directly applied to solve the characteristic system $\mathcal{M}(S)$. In order to obtain the linear invariants of a given system, we use the algorithm proposed in [BBBL13]. It is a variant of Gauss-Jordan elimination that exploits the locality of unknowns as well as the particular form of the characteristic system equations. Equations are processed iteratively, one by one, while producing an equivalent left-bound system.

Let $LI(S)$ be the linear invariants characterizing a given system S. A potential conflict between interactions α_1 and α_2 is a false conflict if the following formula is not satisfiable:

$$\bigwedge_{1 \leq i \leq n} Reach(B_i) \wedge LI(S) \wedge Enabled(\alpha_1) \wedge Enabled(\alpha_2) \tag{4.3}$$

where $Reach(B_i)$ denotes the reachable states of component B_i.

Example 4.2.2. Let us reconsider the example of Figure 2.3. Among the resulting linear invariants, we focus on the following:

$$\begin{cases} 1 \cdot \mathsf{at}(\ell_1^2) + 1 \cdot \mathsf{at}(\ell_1^3) - 1 \cdot \mathsf{at}(\ell_1^1) = 0 & (4.4) \\ 1 \cdot \mathsf{at}(\ell_3^2) + 1 \cdot \mathsf{at}(\ell_3^3) - 1 \cdot \mathsf{at}(\ell_1^4) = 0. & (4.5) \end{cases}$$

We deduce from the invariant (4.4) that $\mathsf{at}(\ell_1^2)$ and $\mathsf{at}(\ell_1^3)$ cannot be true simultaneously, that is, components T_1 and T_2 cannot be simultaneously at the corresponding locations. Consequently, we can directly infer that interactions α_3 and α_4 are not conflicting, even though they are potentially conflicting. Likewise, with (4.5) we exclude the conflict between α_7 and α_8.

4.2.2 History Clocks Inequalities

As they completely abstract time, linear invariants presented above are only partially capturing system dynamics. For example, a global location may not be reachable because components locations have disjoint clock constraints, or an interaction may not be enabled from a state because of an empty timing constraint. Such properties require extra relationships relating

clocks of different components that are not available in $Reach(B_i)$ as it is is restricted to clocks of a single component: a zone of one of its symbolic states is a conjunction of its related clocks constraints, as explained in Section 2.4.

We follow the approach of [RAB+15] for reinforcing our approach with global invariants on clocks. They are induced by simultaneity of transitions when executing an interaction and the synchrony of time progress. To compute such invariants, additional *history* clocks are first introduced in components. History clocks are associated to actions of components and to interactions, and reset upon their executions. They do not modify the behavior since they are not involved in timing constraints. They only reveal local timing of components, relevant to the interaction layer, which allows to infer further properties referred to as *history clocks inequalities* in [RAB+15], expressing the fact that history clock of an interaction is necessary equal to history clocks of its actions after its execution and until the execution of another interaction involving these actions.

Figure 4.1 – Example of a History Clock for Action a

Definition 4.2.3 (History Clocks for Actions)**.** Given a timed system $S = \gamma(B_1, \cdots, B_n)$, the history clocks for actions are defined as follows:

$$\mathcal{HA}(S) = \bigvee_{\alpha \in \gamma} [(\bigwedge_{\substack{a_i, a_j \in \alpha \\ a_k \in Act(\gamma \ominus \alpha)}} h_{a_i} = h_{a_j} \leq h_{a_k}) \wedge (\mathcal{HA}(\gamma \ominus \alpha)]$$

where $Act(\gamma \ominus \alpha)$ is the set of actions involved in the interactions $\gamma \ominus \alpha = \{\beta \setminus \alpha \mid \beta \in \gamma \wedge \beta \not\subseteq \alpha\}$ and $\mathcal{HA}(\emptyset) = true$. The predicate $\mathcal{HA}(S)$ can be interpreted as follows. Assuming that $\alpha \in \gamma$ is the last interaction executed in the system. Then, all history clocks of its involved actions are reset at the same time. Moreover, they are smaller than all the other history clocks, contained in $\gamma \ominus \alpha$. The history clocks for actions are additionally strengthened by *separation constraints* for conflicting interactions. In fact, an action involved in two conflicting interactions is exclusively executed by one of these interactions at a given time. Particularly, in some cases a minimum time lapse is required between two executive occurrences of the same action. Similarly to history clocks for actions, we introduce a history clock for each interaction that will be reset on the execution of the latter. Separation constraints are then formalized as follows:

Definition 4.2.4 (Separation Constraints)**.** Given a timed system $S = \gamma(B_1, \cdots, B_n)$, the separation constraints are defined as follows:

$$HI(S) = \bigwedge_{\substack{\alpha \neq \beta \in \gamma \\ a \in \alpha \cap \beta}} \bigwedge_{a \in \alpha} |h_\alpha - h_\beta| \geq k_a$$

where $|h|$ denotes the absolute value of h and k_a is a constant computed locally on the component containing action a. It represents the minimum time lapse between two occurrences of a.

Our method combines history clocks inequalities $\mathcal{H}(S) = \mathcal{HA}(S) \wedge \mathcal{HI}(S)$ and symbolic states of components to identify false conflicts where the following is not satisfiable:

$$\bigwedge_{1 \leq i \leq n} Reach(B_i) \ \wedge \mathcal{H}(S) \ \wedge \ Enabled(\alpha) \wedge \ Enabled(\beta) \tag{4.6}$$

Example 4.2.3. We illustrate the application of (4.6) for checking conflicts by considering again the example of Figure 2.3. It can be shown that the potential conflict between α_5 and α_6 cannot be removed using (only) linear invariants. In the following, we prove that these interactions are actually not conflicting using history clocks inequalities. Since action $start_0$ of C is synchronized with either $start_1$ of T_1 or $start_2$ of T_2, and since history clocks h_a of an action a is reset whenever a is executed, by [RAB+15] the history clock inequalities for $start_0$ are:

$$\begin{gathered}(h_{start_0} = h_{start_1} \leq h_{start_2} - 25) \\ \vee \\ (h_{start_0} = h_{start_2} \leq h_{start_1} - 25).\end{gathered} \tag{4.7}$$

Equality (4.7) states that h_{start_0} is equal to the history clock corresponding to the last synchronization, i.e., either h_{start_1} or h_{start_2}, and is lower than history clocks of previous synchronizations. Value 25 in (4.7) is obtained considering *separation constraints* computed from symbolic states of components and interactions: two occurrences of $start_0$ are separated by at least 25 time units because of timing constraints of C, and so too occurrences of $start_1$ or $start_2$ which can only execute jointly with $start_0$. To relate history clocks with components clocks, we simply include history clocks when computing symbolic states of components (i.e. *Reach* for components), which is used to establish here that $x = h_{start_1}$ and $y = h_{start_2}$ when components T_1 and T_2 are respectively at locations ℓ_2^2 and ℓ_2^3. That is, with (4.7) we obtain $x \leq y - 25$ or $y \leq x - 25$. By definition of *Enabled* we have $Enabled(\alpha_5) = \mathsf{at}(\ell_2^2) \wedge (10 \leq x \leq 30)$. Similarly, $Enabled(\alpha_6) = \mathsf{at}(\ell_2^3) \wedge (10 \leq y \leq 30)$. We obtain then: $Enabled(\alpha_5) \wedge \mathsf{at}(\ell_2^3) \Rightarrow y \leq 5 \ \wedge Enabled(\alpha_6) \wedge \mathsf{at}(\ell_2^2) \Rightarrow x \leq 5$. This proves that α_5 and α_6 are not conflicting.

4.3 Impact of Conflict Reduction on Send/Receive Models

As explained earlier, refining the conflicting interactions set enables to minimize the exchange of messages between the scheduling layer and the conflict resolution layer of the corresponding Send/Receive model. More precisely, every false conflicting interaction is scheduled using 2 exchange of messages between components participating in that interaction and the corresponding schedulers (offers and notifications), instead of 4 by considering the unnecessary exchange of messages between schedulers and the conflict resolution protocol. As a result, the Scheduler Sch_1 and the conflict resolution components from Figure 4.3 and Figure 4.2 respectively become:

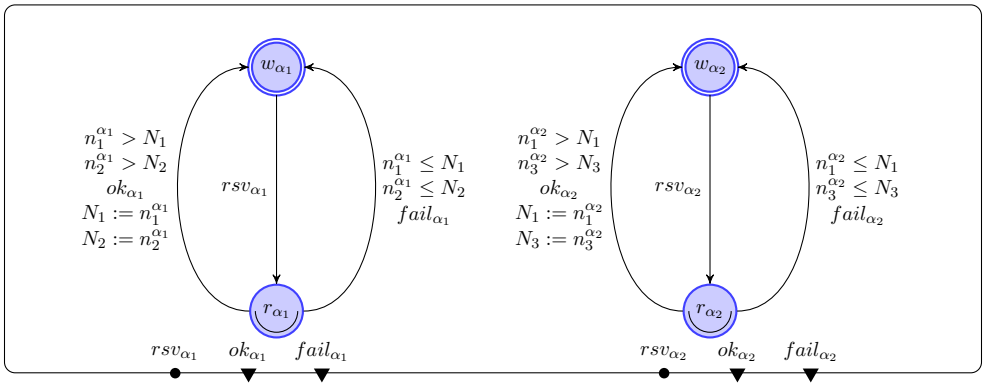


Figure 4.2 – Refined Version of the Conflict Resolution Layer from Figure 4.2

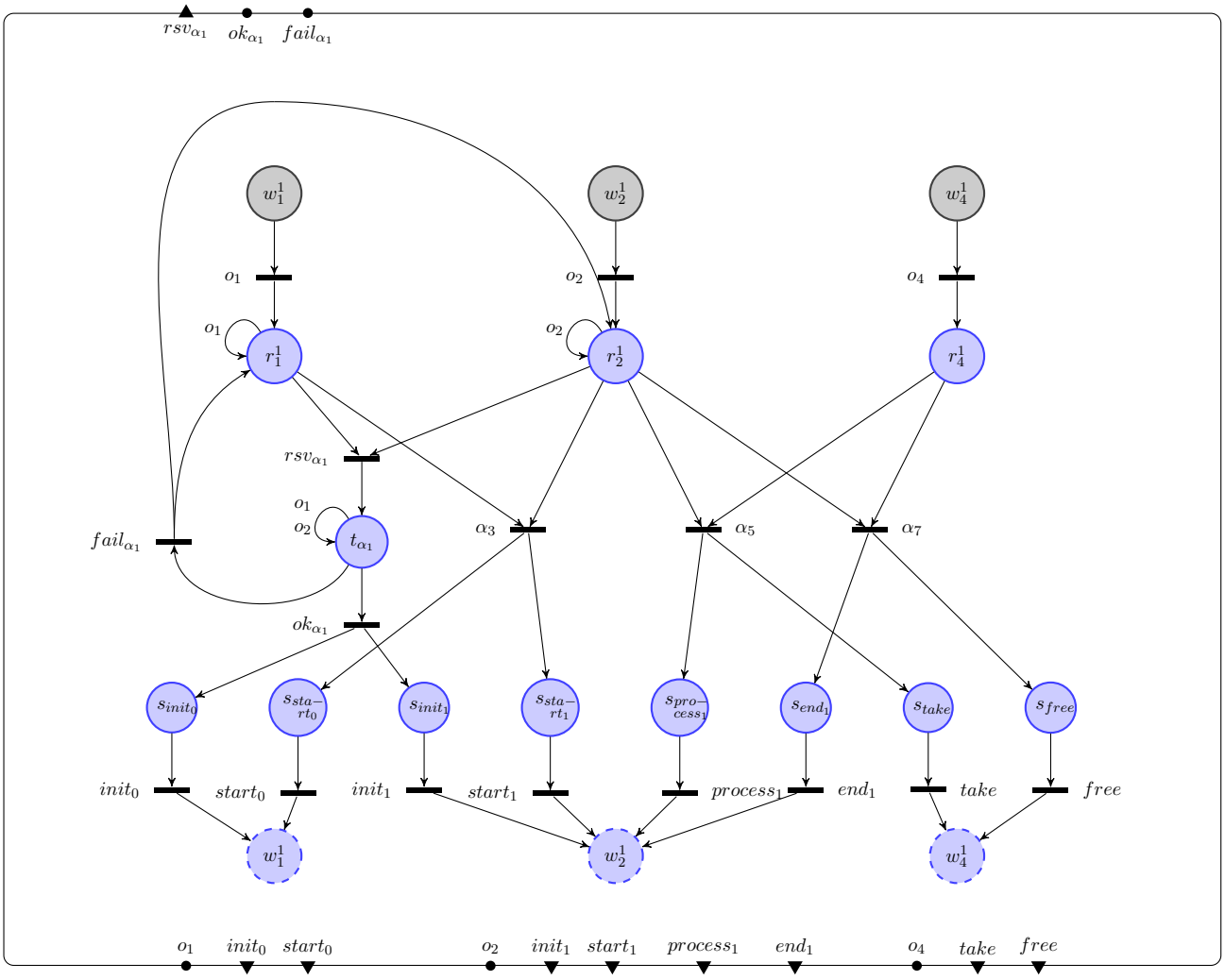


Figure 4.3 – Refined Version of Scheduler Sch_1 from Figure 4.3

Chapter 5

The Local Planning Semantics

Contents

5.1 Local Planning of Interactions 66
5.1.1 Definition of the LPS 66
5.1.2 Properties of the LPS 68
5.2 Enforcing Deadlock-Free Planning 71
5.3 Planning Semantics as Real-Time Controller Synthesis 74
5.3.1 Planning Zones 74
5.3.2 Infinite Planning Transitions 75
5.3.3 Discussion 79

In Chapter 2, we presented a timed automata model for representing timed systems with multiparty interactions. The semantics of such model is based on the notion of *global states*, that is, interactions executions are based not only on the state of participating components but on the states of all components of the system. Conversely, a distributed system can be seen as a collection of loosely coupled components that communicate with a scheduling layer which, based on a partial view of the system, is responsible of taking decisions for interactions executions and their effective execution dates. Additionally, high-level coordination primitives, such as multiparty synchronizations (interactions) are rarely built-in primitives of distributed platforms. Hence, their implementation on a distributed platform requires protocols responsible for realizing these synchronizations using simpler primitives such as point-to-point messages passing. This is classically implemented using one or more additional coordination component(s) observing the system state and deciding on interactions executions, which adds on a communication overhead not reflected by the standard semantics presented in Chapter 2.

This motivates the introduction of the *local planning semantics* (LPS). This semantics differs from the standard semantics of timed automata in two main aspect: *(i)* interactions executions are based only on partial state of the system, that is, based only on the state of components participating in the considered interaction. Thus, it allows to decide locally without monitoring the entire system. *(ii)* it distinguishes between the execution decision of an interaction (its *planning*), and the execution itself. This distinction allows us to impose a delay between the planning of an interaction and its execution. The latter is constrained by the (maximal) communication latency induced by execution platforms, which is a parameter of the semantics. Trivially, this semantics is correct in the sense that it refines (it is included in) the

standard semantics. However, being based on local states, planning decisions are too permissive and may introduce deadlocks when they are not compatible with the global state of the system.

5.1 Local Planning of Interactions

5.1.1 Definition of the LPS

Let $S = \gamma(B_1, \cdots, B_n)$ be a composition of components $B_1, \ldots, B_n$ with disjoint set of locations, actions and clocks. We define the predicate $\mathit{Plannable}(\alpha, d)$ characterizing states (ℓ, υ) from which an interaction $\alpha = \{a_i\}_{i \in I} \in \gamma$ is enabled in $d \in \mathbb{R}_{\geq 0}$ units of time (if time progresses by d units of time), that is, such that $\mathit{Enabled}(\alpha)$ evaluates to true on state $(\ell, \upsilon + d)$. It is characterized by:

$$\mathit{Plannable}(\alpha, d) = \bigvee_{\substack{\ell \in \mathcal{L} \\ \ell = (\ell_1, \cdots, \ell_n)}} \mathsf{at}(\ell) \wedge \bigwedge_{\substack{i \in I \\ a_i \in \alpha}} \left(\Phi(a_i, \ell_i) + d\right) \tag{5.1}$$

Notice that for an interaction α the predicate $\mathit{Plannable}(\alpha, d)$ depends only on states of components of $\mathit{part}(\alpha)$, which motivates the following property.

Property 5.1.1. *Let (ℓ, υ) be a state of the composition S. For any interactions $\alpha, \beta \in \gamma$ such that, $(\ell, \upsilon) \xrightarrow{\beta}_{\gamma} (\ell', \upsilon')$ and $\mathit{part}(\alpha) \cap \mathit{part}(\beta) = \emptyset$, if $\mathit{Plannable}(\alpha, d)$ holds at state (ℓ, υ) then it still holds at state (ℓ', υ').*

This property derives directly from the fact that executing an interaction β does not change the states of components participating in an interaction α, provided that α and β have disjoint sets of participating components, and thus, $\mathit{Plannable}(\alpha, d)$ is not affected by the execution of β in this case. In the following, we consider a slightly different definition of conflicts than the one presented in Chapter 3 and Chapter 4. We say that two interactions α and β *conflicts* when they have common participating components, that is, when $\mathit{part}(\alpha) \cap \mathit{part}(\beta) \neq \emptyset$, and we write $\alpha \# \beta$. We denote by $\mathit{conf}(\alpha)$ the set of interactions conflicting with α, that is, $\mathit{conf}(\alpha) = \{\beta \in \gamma \mid \alpha \# \beta\}$.

Property 5.1.2. *Let (ℓ, υ) and $(\ell, \upsilon + d')$, with $d' \in \mathbb{R}_{>0}$ be two states of the composition S. For an interaction $\alpha \in \gamma$, if $\mathit{Plannable}(\alpha, d)$ holds at state (ℓ, υ) then $\mathit{Plannable}(\alpha, d - d')$ also holds at any state $(\ell, \upsilon + d')$ such that $d' \leq d$.*

This property can be found directly by writing expression 5.1 on state $(\ell, \upsilon + d')$.

As previously explained, due to communication latencies induced by execution platforms, we assume that interactions cannot be planned in d units of time if $d < h_{\min}$, where $h_{\min} \in \mathbb{Z}_{\geq 0}$ is a parameter representing the *minimal planning horizon*, which should represent the upper bound communication latencies. Notice that for the sake of simplicity, we consider a global parameter $h_{\min}$ but we could also assume different parameters for each interaction. Additionally, we also consider upper bound planning horizons $h_{\max} : \gamma \to \mathbb{Z}_{\geq 0} \cup \{+\infty\}$ for each interaction such that for any $\alpha \in \gamma$ we have $h_{\max}(\alpha) \geq h_{\min}$. We denote by $h^{\infty}_{\max}$ the upper planning horizon assigning infinity to every $h_{\max}(\alpha)$. A direct consequence of introducing the planning horizons is that every interaction α can be planned only using a horizon d satisfying $h_{\min} \leq d \leq h_{\max}(\alpha)$, meaning that every component $B \in \mathit{part}(\alpha)$ will be blocked for a duration between $[h_{\min}, h_{\max}(\alpha)]$. Observe that while $h_{\min}$ represents the worst case estimation of the communication delays

for a given platform, the parameters $h_{\max}(\alpha)$ will be used later to find a strategy that avoids deadlocks by restricting the amount of time components can be blocked for.

For an interaction α, we define the predicate $\mathit{Plannable}(\alpha)$ characterizing states from which α can be planned in a delay respecting the planning horizons $h_{\min}$ and $h_{\max}(\alpha)$, that is:

$$\mathit{Plannable}(\alpha) \Leftrightarrow \exists d \in \mathbb{R}_{\geq 0} \,.\, h_{\min} \leq d \leq h_{\max}(\alpha) \;\wedge\; \mathit{Plannable}(\alpha, d),$$

It can be written formally as follows:

$$\mathit{Plannable}(\alpha) = \bigvee_{\substack{\ell \in \mathcal{L} \\ \ell = (\ell_1, \cdots, \ell_n)}} \mathsf{at}(\ell) \;\wedge\; \swarrow_{h_{\min}}^{h_{\max}(\alpha)} \Big(\bigwedge_{\substack{i \in I \\ a_i \in \alpha}} \Phi(a_i, \ell_i) \Big) \tag{5.2}$$

$$= \mathit{Enabled}\swarrow_{h_{\min}}^{h_{\max}(\alpha)}(\alpha) \tag{5.3}$$

Definition 5.1.1 (Plan)**.** A plan π is a function $\pi : \gamma \to \mathbb{R}_{\geq 0} \cup \{+\infty\}$ defining relative times for executing interactions, with the convention that an interaction α is planned to execute in $\pi(\alpha)$ time units only if $\pi(\alpha) < +\infty$. Plans satisfy that for any two interactions $\alpha \neq \beta$, such that $\pi(\alpha) < +\infty$ and $\pi(\beta) < +\infty$, the interactions α and β are not conflicting (i.e. $\neg(\alpha \# \beta)$).

We denote by π_0 the plan assigning $+\infty$ to every interaction of γ, that is, $\forall \alpha \in \gamma, \pi_0(\alpha) = +\infty$. For a plan π, we consider its minimum value $\min(\pi) = \min\{\pi(\alpha) | \alpha \in \gamma\}$. We also denote by $\mathit{conf}(\pi)$ the set of interactions conflicting with the plan π, i.e., $\mathit{conf}(\pi) = \{\alpha \mid \exists \beta \# \alpha. \; \pi(\beta) < +\infty\}$, and $\mathit{part}(\pi)$ the set of components participating in interactions planned by π, i.e., $\mathit{part}(\pi) = \{B_i \mid \exists \alpha \,.\, \pi(\alpha) < +\infty \wedge B_i \in \mathit{part}(\alpha)\}$. Notice that since π stores relative times, whenever time progresses by d, the value $\pi(\alpha)$ assigned by π to an interaction α should be decreased by d until it reaches 0, meaning that α has to execute. We write $\pi - d$ to describe the progress of time over the plan, that is, $(\pi - d)(\alpha) = \pi(\alpha) - d$ for interactions α such that $\pi(\alpha) < +\infty$. Similarly, $\pi[\alpha \mapsto d]$ assigns relative time d to α, $\alpha \notin \mathit{conf}(\pi)$, into existing plan π, i.e. $(\pi[\alpha \mapsto d])(\beta) = d$ for $\beta = \alpha$, $(\pi[\alpha \mapsto d])(\beta) = \pi(\beta)$ otherwise.

Definition 5.1.2 (Local Planning Semantics)**.** Given a set of components $\{B_1, \cdots, B_n\}$ and an interaction set γ, we define the local planning semantics (LPS) of the composition $\gamma(B_1, \cdots, B_n)$, as the TTS $(\mathcal{Q}_p, q_{p_0}, \gamma \cup \mathbb{R}_{>0} \cup (\gamma \times \mathbb{R}_{\geq 0}), \rightsquigarrow_{\gamma})$ where:

- $\mathcal{Q}_p = \mathcal{L} \times \mathcal{V}(\mathcal{X}) \times \Pi$, where $\mathcal{L}$ is the set of global locations, $\mathcal{V}(\mathcal{X})$ is the set of global clock valuations, and Π is the set of plans.
- $(\gamma \times \mathbb{R}_{\geq 0})$ defines the action of planning interactions of γ and their relative times.
- $\rightsquigarrow_{\gamma}$ is the set of transitions defined by the rules:
  - $(\ell, \upsilon, \pi) \overset{(\alpha,d)}{\rightsquigarrow}_{\gamma} (\ell, \upsilon, \pi[\alpha \mapsto d])$ for $\alpha \in \gamma$, $h_{\min} \leq d \leq h_{\max}(\alpha)$ and $d \neq +\infty$ if $\alpha \notin \mathit{conf}(\pi)$ and $\mathit{Plannable}(\alpha, d)$ holds on (ℓ, υ, π).
  - $(\ell, \upsilon, \pi) \overset{\alpha}{\rightsquigarrow}_{\gamma} (\ell', \upsilon', \pi[\alpha \mapsto +\infty])$ for $\alpha \in \gamma$ if $\pi(\alpha) = 0$ and $(\ell, \upsilon) \xrightarrow{\alpha}_{\gamma} (\ell', \upsilon')$.
  - $(\ell, \upsilon, \pi) \overset{d}{\rightsquigarrow}_{\gamma} (\ell, \upsilon + d, \pi - d)$ for $d \leq \min(\pi)$, $\ell = (\ell_1, \ldots, \ell_n)$, if $(\upsilon + d) \models \mathcal{I}_i(\ell_i)$ for components $B_i \in \mathit{part}(\pi)$ and $(\upsilon + d + h_{\min}) \models \mathcal{I}_i(\ell_i)$ for components $B_i \notin \mathit{part}(\pi)$.

Remark that in the above definition as well as in what follows, predicates defined on states $(\ell, \upsilon) \in \mathcal{Q}_g = \mathcal{L} \times \mathcal{V}(\mathcal{X})$ of the standard semantics are straightforwardly interpreted on states $(\ell, \upsilon, \pi) \in \mathcal{Q}_p$ considering the projection (ℓ, υ) of (ℓ, υ, π) on $\mathcal{Q}_g$.

States of the LPS do not include only locations and clock valuations, but also the relative execution times of the planned interactions stored by π. Initially, no interaction is planned, that is, initial states $(\ell_0, \upsilon_0, \pi_0)$ satisfy $\pi_0 = +\infty$. Planning an interaction α to be executed at a relative time $h_{\min} \leq d \leq h_{\max}(\alpha)$ corresponds to the operation $\pi[\alpha \mapsto d]$ on the plan, which can only be done if α is not conflicting with the latter, and becomes enabled if time progresses by d (i.e. if $\mathit{Plannable}(\alpha, d)$). On the other hand, time progress not only updates clock values but also the plan by decreasing the relative execution times of the planned interactions. To force the execution of planned interactions when their relative execution times reach 0, time cannot progress more than the relative execution times of the interactions (more than $d \leq \min(\pi)$). As for the standard semantics, time progress is limited by the location invariants of the components, but with the following significant difference: Components $B_i \in \mathit{part}(\pi)$ participating in planned interactions behave as in the standard semantics, that is, time can progress until their invariants become urgent. For components $B_i \notin \mathit{part}(\pi)$, i.e., that are not participating in planned interactions, we take into account the minimal delay $h_{\min}$ needed for planning and then executing an interaction: in components $B_i \notin \mathit{part}(\pi)$ time can progress only up to $h_{\min}$ time units before the urgency of their location invariants. By doing so, we ensure that there always remains enough time to plan interactions involving $B_i \notin \mathit{part}(\pi)$, if they exist, and execute them before their invariants expire.

Example 5.1.1. Let us consider the following execution sequence for example of Figure 2.3 under the LPS with $h_{\min} = 2$ and $h_{\max} = h^{\infty}_{\max}$.

$$
\begin{array}{ll}
((\ell_0^1,\ell_0^2,\ell_0^3,\ell_0^4),(0,0,0),+\infty) & \overset{(\alpha_1,26)}{\rightsquigarrow_\gamma}((\ell_0^1,\ell_0^2,\ell_0^3,\ell_0^4),(0,0,0),\{\alpha_1\mapsto 26\})\overset{26}{\rightsquigarrow_\gamma} \\
((\ell_0^1,\ell_0^2,\ell_0^3,\ell_0^4),(26,26,26),\{\alpha_1\mapsto 0\}) & \overset{\alpha_1}{\rightsquigarrow_\gamma}((\ell_1^1,\ell_1^2,\ell_0^3,\ell_0^4),(26,26,26),+\infty)\overset{(\alpha_3,2)}{\rightsquigarrow_\gamma} \\
((\ell_1^1,\ell_1^2,\ell_0^3,\ell_0^4),(26,26,26),\{\alpha_3\mapsto 2\}) & \overset{2}{\rightsquigarrow_\gamma}((\ell_1^1,\ell_1^2,\ell_0^3,\ell_0^4),(28,28,28),\{\alpha_3\mapsto 0\})\overset{\alpha_3}{\rightsquigarrow_\gamma} \\
((\ell_0^1,\ell_2^2,\ell_0^3,\ell_0^4),(0,28,0),+\infty) & \overset{(\alpha_2,26)}{\rightsquigarrow_\gamma}((\ell_0^1,\ell_2^2,\ell_0^3,\ell_0^4),(0,28,0),\{\alpha_2\mapsto 26\})\overset{26}{\rightsquigarrow_\gamma} \\
((\ell_0^1,\ell_2^2,\ell_0^3,\ell_0^4),(26,54,26),\{\alpha_2\mapsto 0\}) & \overset{\alpha_2}{\rightsquigarrow_\gamma}((\ell_1^1,\ell_2^2,\ell_1^3,\ell_0^4),(26,54,26),+\infty)\overset{(\alpha_4,2)}{\rightsquigarrow_\gamma} \\
((\ell_1^1,\ell_2^2,\ell_1^3,\ell_0^4),(26,54,26),\{\alpha_4\mapsto 2\}) & \overset{2}{\rightsquigarrow_\gamma}((\ell_1^1,\ell_2^2,\ell_1^3,\ell_0^4),(28,56,28),\{\alpha_4\mapsto 0\})\overset{\alpha_4}{\rightsquigarrow_\gamma} \\
((\ell_0^1,\ell_2^2,\ell_2^3,\ell_0^4),(28,0,0),+\infty) & \overset{(\alpha_6,30)}{\rightsquigarrow_\gamma}((\ell_0^1,\ell_2^2,\ell_2^3,\ell_0^4),(28,0,0),\{\alpha_6\mapsto 30\})
\end{array}
$$

This execution sequence represents a path that alternates plan actions, time progress and execution of some interactions, and leads to the action-time-lock state $((\ell_0^1,\ell_2^2,\ell_2^3,\ell_0^4),(0,0,28),\{\alpha_6 \mapsto 30\})$. In fact, the location invariant $x \leq 30$ in component T_1, imposes the planning of interaction α_7 at the latest h_{min} units of time before it becomes urgent. However, since interaction α_6 was planned in 28 units of time, α_7 cannot be planned since it is conflicting with α_6. This execution sequence shows that a given system action-time-locks under the local planning semantics , even if it is deadlock-free in the standard semantics.

5.1.2 Properties of the LPS

We use weak simulation to compare models of the standard semantics and the local planning semantics by considering the planning transitions unobservable. As shown in Example 5.1.1, the LPS does not preserve the deadlock freedom property of our system. Nevertheless, the following proves weak simulation relations between the two semantics.

Lemma 5.1.1. *Given a reachable state (ℓ, υ, π) of the LPS. If for $\alpha \in \gamma$, $\pi(\alpha) < +\infty \Rightarrow$ $Plannable(\alpha, \pi(\alpha))$.*

Proposition 5.1.1. *An interaction can execute from a state (ℓ, υ, π) in the LPS semantics only if it can execute from (ℓ, υ) in the standard semantics, that is:*

$$\forall \alpha \in \gamma.(\ell, \upsilon, \pi) \overset{\alpha}{\rightsquigarrow}_{>\gamma} (\ell', \upsilon', \pi') \Rightarrow (\ell, \upsilon) \overset{\alpha}{\rightarrow}_{\gamma} (\ell', \upsilon').$$

Proposition 5.1.1 is a consequence of Lemma 5.1.1: an interaction α can execute in the local planning semantics only if $\pi(\alpha) = 0$ (see Definition 5.1.1). That is, a state (ℓ, υ, π) of the LPS from which α can execute satisfies $Plannable(\alpha, 0)$ or equivalently $Enabled(\alpha)$, which demonstrates that α can execute from (ℓ, υ) in the standard semantics.

Proposition 5.1.2. *Time can progress by d at a state (ℓ, υ, π) in the local planning semantics only if time can progress by d at (ℓ, υ) in the standard semantics, that is:*

$$\forall d \in \mathbb{R}_{>0}.(\ell, \upsilon, \pi) \overset{d}{\rightsquigarrow}_{>\gamma} (\ell', \upsilon', \pi') \Rightarrow (\ell, \upsilon) \overset{d}{\rightarrow}_{\gamma} (\ell', \upsilon').$$

Proposition 5.1.2 is a direct consequence of the definition of time progress in the local planning semantics which is a restriction of the one in the standard semantics.

Corollary 5.1.1. *If a state (ℓ, υ, π) is reachable in the local planning semantics, then the state (ℓ, υ) is reachable in the standard semantics.*

Corollary 5.1.1 is obtained from Propositions 5.1.1 and 5.1.2 and the fact that planning transitions (labeled by (α, d)) affect only the plan π in states (ℓ, υ, π) of the LPS.

The definition of weak simulation (Definition 2.1.3) is based on the unobservability of $\beta-$transitions. In our case, $\beta-$transitions corresponds to planning transitions. Let $\mathtt{T}_g$ and $\mathtt{T}_p$ be respectively the underlying timed transition systems of the standard semantics and the local planning semantics respectively.

Corollary 5.1.2. $\mathtt{T}_p \dot{\sqsubseteq}_R \mathtt{T}_g$ *with* $R = \{((q, \pi); q) \in \mathcal{Q}_p \times \mathcal{Q}_g\}$.

Corollary 5.1.2 corresponds to a notion of correctness of the local planning semantics: any execution in the LPS corresponds to an execution in the standard semantics. In addition, if interactions are allowed to be planned with relative execution times of 0 (i.e. $h_{\min} = 0$) then timeless planning of interactions becomes possible. Thus, the planning semantics simulates the standards semantics in that case.

Corollary 5.1.3. $\mathtt{T}_g \dot{\sqsubseteq}_{R'} \mathtt{T}_p$ *with* $R' = \{(q; (q, \pi)) \in \mathcal{Q}_g \times \mathcal{Q}_p \mid h_{\min} = 0\}$.

However, this is no longer true in general if $h_{\min} > 0$ which means that not all execution sequences of the standard semantics are preserved by the local planning semantics.

Corollary 5.1.4. *If $\mathtt{T}_g$ is zeno runs free then $\mathtt{T}_p$ is too.*

Corollary 5.1.4 states that the *LPS* does not introduce any zenoness behavior if the standard semantics is free from the latter. It is a direct consequence of Corollary 5.1.2 and the fact that it is not possible to have infinite sequences of planning transitions without interaction execution (γ is finite and planning times are bounded).

Proposition 5.1.3. *If every reachable state of $\mathcal{T}_g$ is not a deadlock, then a reachable state of $\mathcal{T}_p$ is not deadlock if and only if it is not an action-time-lock.*

Proof of Propostion 5.1.3. We prove Proposition 5.1.3 by contradiction. Let us assume that the system under the standard (resp. local planning) semantics is deadlock free (resp. action-time-lock-free). Let (ℓ, υ, π) be a reachable deadlock state of the LPS. We have:

$$\nexists \sigma \in \gamma \cup (\gamma \times \mathbb{R}_{\geq 0}), \exists d.\ (\ell, \upsilon, \pi) \overset{\sigma}{\rightsquigarrow}_{\gamma} (\ell', \upsilon', \pi') \vee (\ell, \upsilon, \pi) \overset{d}{\rightsquigarrow}_{\gamma} (\ell, \upsilon + d, \pi - d) \overset{\sigma}{\rightsquigarrow}_{\gamma} (\ell', \upsilon', \pi')$$

We denote by $wait(\ell, \upsilon, \pi)$ the set of allowed waiting times at state (ℓ, υ, π), that is:

$$wait(\ell, \upsilon, \pi) = \{0\} \cup \{d \in \mathbb{R}_{>0} | (\ell, \upsilon, \pi) \overset{d}{\rightsquigarrow}_{\gamma} (\ell, \upsilon + d, \pi - d)\}$$

We also put $\max(wait(\ell, \upsilon, \pi))$ to denote the maximal waiting time at state (ℓ, υ, π). Notice that $\max(wait(\ell, \upsilon, \pi))$ may not be defined in some cases. In fact, we are not interested in its actual existence but rather in the fact that it is bounded ($< +\infty$) or not.

Lemma 5.1.2. *Let (ℓ, υ, π) be a reachable state of the local planning semantics. For $k \in \mathbb{R}_{\geq 0}$, such that $k = \max(wait(\ell, \upsilon, \pi))$, we have the following properties:*

P1 *If $k < +\infty$ then $(\ell, \upsilon, \pi) \overset{k}{\rightsquigarrow}_{\gamma} (\ell, \upsilon + k, \pi - k) \wedge wait(\ell, \upsilon + k, \pi - k) = \{0\}$*

P2 *If $\pi \neq \pi_0$ then $k \leq \min(\pi)$*

We distinguish 2 cases:

Case 1: no interaction is planned (i.e. $\pi = \pi_0$) By definition of the LPS, it is clear that for $\pi = \pi_0$, there is no interaction to execute from (ℓ, υ, π) or any of its successor $(\ell, \upsilon + d, \pi - d)$.

1. $wait(\ell, \upsilon, \pi) = \{0\}$:
   This means that time progress is not allowed at state (ℓ, υ, π). We also have $\nexists \sigma \in (\gamma \times \mathbb{R}_{\geq 0}).(\ell, \upsilon, \pi) \overset{\sigma}{\rightsquigarrow}_{\gamma} (\ell', \upsilon', \pi')$ (deadlock assumption). We can conclude that (ℓ, υ, π) is a reachable action-time-lock state, which contradicts the assumption that the system under the local planning semantics is action-time-lock-free.

2. $wait(\ell, \upsilon, \pi) \neq \{0\}$:

   (a) $\max(wait(\ell, \upsilon, \pi)) = +\infty$:

   Lemma 5.1.3. *Let (ℓ, υ, π) be a reachable state of the local planning semantics . If $\forall d \in \mathbb{R}_{>0}.\ (\ell, \upsilon, \pi) \overset{d}{\rightsquigarrow}_{\gamma} (\ell, \upsilon + d, \pi - d) \wedge \neg Plannable(\alpha)$ at (ℓ, υ, π), then we have $\neg Enabled(()\alpha)$ at $(\ell, \upsilon + d, \pi - d)$ with $d \geq h_{\min}$.*

   By P1 of Lemma 5.1.2 we can deduce that $\exists d \geq h_{\min}$ such that $(\ell, \upsilon, \pi) \overset{d}{\rightsquigarrow}_{\gamma} (\ell, \upsilon + d, \pi - d)$. We also have from the deadlock assumption and Lemma 5.1.3: $\bigwedge_{\alpha \in \gamma} \neg Enabled(()\alpha)$. Finally, since the state $(\ell, \upsilon + d, \pi - d)$ is reachable in the standard semantics, and by evaluating the deadlock characterization 2.3 on state $(\ell, \upsilon + d, \pi - d)$, we can conclude that the system under the standard semantics deadlocks, which contradicts the assumption of deadlock freedom of the system under the standard semantics.

(b) $\max(wait(\ell, \upsilon, \pi)) < +\infty$:
Considering that $k = \max(wait(\ell, \upsilon, \pi))$, then we have by P1 of Lemma 5.1.2: $(\ell, \upsilon, \pi) \overset{k}{\rightsquigarrow}_{\gamma} (\ell, \upsilon + k, \pi - k) \wedge wait(\ell, \upsilon + k, \pi - k) = \{0\}$. Using the deadlock assumption we have: $\bigwedge_{\alpha \in \gamma} \neg Plannable(\alpha)$ at state $(\ell, \upsilon + k, \pi - k)$. Since the system cannot progress beyond this state ($wait(\ell, \upsilon + k, \pi - k) = \{0\}$), we can conclude that $(\ell, \upsilon + k, \pi - k)$ is a reachable action-time-lock state, which contradicts the assumption that the system under the local planning semantics is action-time-lock-free.

Case 2: at least an interaction is planned (i.e. $\pi \neq \pi_0$) Considering that $k = \max(wait(\ell, \upsilon, \pi))$, since $\pi \neq +\infty$, we have by P2 of Lemma 5.1.2: $k < +\infty \wedge k \leq \min \pi$. Using the deadlock assumption we can infer that $k < \min \pi$, since no execution is possible from (ℓ, υ, π) or any of its successors. This means that $(\ell, \upsilon + k, \pi - k)$ is a reachable action-time-lock state, which contradicts the assumption that the system under the LPS is action-time-lock-free. □

5.2 Enforcing Deadlock-Free Planning

As explained in previous section, the local planning semantics is based on local conditions for planning interactions and may exhibit deadlocks even when the system is deadlock-free with the standard semantics. Such deadlocks are partly due to the fact that planning an interaction may block, in addition to the participating components, extra components whose timing constraints are not considered by these local conditions. In this section, we investigate simple execution strategies that only restrict the horizon used for planning interactions with upper bounds. By reducing the period of time during which components are blocked, they tend to remove deadlocks from the reachable states. In what follows, we consider a composition of components $S = \gamma(B_1, \cdots, B_n)$ such that it is deadlock-free in the standard semantics.

Corollary 5.2.1 (Sufficient Condition for Deadlock Freedom)**.** *If a reachable state of the planning semantics is not an action-time-lock then it is not a deadlock.*

Corollary 5.2.1 is a direct consequence of Proposition 5.1.3. It affirms that for systems that are initially deadlock-free under the standard semantics, it is sufficient to prove action-time-lock freedom of the LPS to prove its deadlock freedom.

Proposition 5.2.1. *A reachable state (ℓ, υ, π) of the local planning semantics is an action-time-lock if and only if:*

$$\pi > 0 \ \wedge \bigwedge_{\alpha \notin conf(\pi)} \neg Plannable(\alpha) \ \wedge \bigvee_{\substack{\ell_i \in \mathcal{L}_i \\ B_i \notin part(\pi)}} \mathsf{at}(\ell_i) \ \wedge (urg(\ell_i) + h_{\min}).$$

The above proposition derives directly from the definition of action-time-locks on a state of the local planning semantics. As shown in Example 5.1.1, the local planning semantics may introduce deadlocks. The source of deadlocks is twofold: *(i)* due to communication delays, consecutive execution in a component are separated by at least $h_{\min}$ units of time which may be incompatible with its timings constraints, and *(ii)* conditions for planning interactions are too permissive as they only take into account timing constraints of participating components whereas they may block additional components, namely the ones participating

in conflicting interactions. In what follows, we study how to generate planning strategies for preserving deadlock freedom by restricting the planning transitions of the LPS so that deadlock states become unreachable. Such a strategy may not exist when timing constraints cannot accommodate with the communication delays $h_{\min}$.

From Corollary 5.2.1, action-time-lock freedom is a sufficient condition for deadlock freedom of the LPS. By Proposition 5.2.1, a state (ℓ, υ, π) is an action-time-lock in the local planning semantics if and only if no time progress is allowed nor planning or execution of interactions from (ℓ, υ, π), that is:

$$\pi > 0 \ \wedge \bigwedge_{\alpha \in \gamma \setminus conf(\pi)} \neg Plannable(\alpha) \ \wedge \bigvee_{\substack{\ell_i \in \mathcal{L}_i \\ B_i \notin part(\pi)}} \mathsf{at}(\ell_i) \ \wedge (urg(\ell_i) + h_{\min}).$$

The above predicate characterizes the fact that no interaction can be executed or planned, nor time can progress in component $B_i \notin part(\pi)$. Consequently, we deduce that a necessary condition of action-time-lock is the existence of a component $B_i \notin part(\pi)$ such that time cannot progress in B_i and B_i cannot be planned in an interaction, that is:

$$\bigwedge_{\alpha \in \gamma(B_i) \setminus conf(\pi)} \Big(\neg Plannable(\alpha) \ \wedge \bigvee_{\ell_i \in \mathcal{L}_i} \mathsf{at}(\ell_i) \ \wedge (urg(\ell_i) + h_{\min}) \Big).$$

where $\gamma(B_i)$ denotes the subset of interactions in which B_i participates, that is, $\gamma(B_i) = \{\beta \in \gamma \mid B_i \in part(\beta)\}$. Notice that the above expression strongly depends on the plan π, which is difficult to characterize in practice. The following theorem proposes sufficient plan-independent condition characterizing action-time-lock states of the LPS .

Theorem 5.2.1. *Let ϕ be the following predicate:*

$$\bigvee_{1 \leq i \leq n} \Big[\bigvee_{\ell_i \in \mathcal{L}_i} \mathsf{at}(\ell_i) \ \wedge (urg(\ell_i) + h_{\min}) \ \wedge \bigwedge_{\alpha \in \gamma(B_i)} \Big(\neg Plannable(\alpha) \ \vee \bigvee_{\substack{\beta \in conf(\alpha) \\ B_i \notin part(\beta)}} \overline{Plannable(\beta)} \Big) \Big].$$

We prove that a reachable action-time-lock state (ℓ, υ, π) satisfies ϕ.

Proof of Theorem 5.2.1. A reachable action-time-lock state of the LPS satisfies:

$$\pi > 0 \ \wedge \bigwedge_{\alpha \in \gamma(B_i) \setminus conf(\pi)} \Big(\neg Plannable(\alpha) \ \wedge \bigvee_{\substack{\ell_i \in \mathcal{L}_i \\ B_i \notin part(\pi)}} \mathsf{at}(\ell_i) \ \wedge (urg(\ell_i) + h_{\min}) \Big).$$

In order to approximate the above formula, we distinguish two cases:

Case 1: no interaction is planned (i.e. $\pi = \pi_0$)
From $\pi = +\infty$ we deduce directly that there exists an urgent component B_i such that no interaction α involving B_i can be planned, that is:

$$\bigvee_{1 \leq i \leq n} \Big[\bigvee_{\ell_i \in \mathcal{L}_i} \mathsf{at}(\ell_i) \ \wedge (urg(\ell_i) + h_{\min}) \ \wedge \bigwedge_{\alpha \in \gamma(B_i)} \neg Plannable(\alpha) \Big]. \tag{1}$$

Case 2: at least an interaction is planned (i.e. $\pi \neq \pi_0$)
In this case, there exists an urgent component $B_i \notin part(\pi)$ such that no interaction α involving

B_i can be planned, either because it conflicts with a planned interaction β $(0 < \pi(\beta) < +\infty)$ or because $\mathit{Plannable}(\alpha)$ is not satisfied, that is $\exists \beta \in \pi, \exists B_i \notin \mathit{part}(\beta)$ satisfying:

$$(0 < \pi(\beta) < +\infty) \ \wedge \bigwedge_{\alpha \in \gamma(B_i) \setminus \mathit{conf}(\beta)} \neg \mathit{Plannable}(\alpha) \ \wedge \bigvee_{\substack{\ell_i \in \mathcal{L}_i \\ B_i \notin \mathit{part}(\beta)}} \mathsf{at}(\ell_i) \ \wedge (\mathit{urg}(\ell_i) + h_{\min}).$$

or equivalently $\exists \beta \in \pi, \exists B_i \notin \mathit{part}(\beta)$ satisfying:

$$\bigvee_{\substack{\ell_i \in \mathcal{L}_i \\ B_i \notin \mathit{part}(\beta)}} \mathsf{at}(\ell_i) \wedge (\mathit{urg}(\ell_i) + h_{\min}) \wedge \bigwedge_{\alpha \in \gamma(B_i)} \Big(\neg \mathit{Plannable}(\alpha) \vee \big(\beta \in \mathit{conf}(\alpha) \wedge (0 < \pi(\beta) < +\infty)\big)\Big).$$

By noticing that we have the following implication between quantifiers $\exists y, \forall x. Q(x, y) \implies \forall x, \exists y. Q(x, y)$, we can deduce that the above condition implies:

$$\bigvee_{1 \le i \le n} \Big[\bigvee_{\ell_i \in \mathcal{L}_i} \mathsf{at}(\ell_i) \ \wedge (\mathit{urg}(\ell_i) + h_{\min}) \ \wedge \bigwedge_{\alpha \in \gamma(B_i)} \Big(\neg \mathit{Plannable}(\alpha) \ \vee \bigvee_{\substack{\beta \in \mathit{conf}(\alpha) \\ B_i \notin \mathit{part}(\beta)}} 0 < \pi(\beta) < +\infty \Big) \Big].$$

As $\pi > 0$, and if we consider only reachable action-time-locks, we have $0 < \pi(\beta) \le h_{\max}(\beta)$, and by Lemma 5.1.1 we have $\mathit{Plannable}(\beta, \pi(\beta))$. That is, β satisfies $\mathit{Plannable}(\beta)$ in which the lower bound $h_{\min}$ is replaced by the strict lower bound 0, i.e.:

$$\overline{\mathit{Plannable}(\beta)} \Leftrightarrow \exists d > 0 \ . \ d \le h_{\max}(\beta) \ \wedge \ \mathit{Plannable}(\beta, d).$$

Then, the above expression becomes:

$$\bigvee_{1 \le i \le n} \Big[\bigvee_{\ell_i \in \mathcal{L}_i} \mathsf{at}(\ell_i) \ \wedge (\mathit{urg}(\ell_i) + h_{\min}) \ \wedge \bigwedge_{\alpha \in \gamma(B_i)} \Big(\neg \mathit{Plannable}(\alpha) \ \vee \bigvee_{\substack{\beta \in \mathit{conf}(\alpha) \\ B_i \notin \mathit{part}(\beta)}} \overline{\mathit{Plannable}(\beta)} \Big) \Big]. \quad (2)$$

By remarking that Expression 1 implies Expression 2, we can conclude that an action-time-lock of the local planning semantics satisfies:

$$\bigvee_{1 \le i \le n} \Big[\bigvee_{\ell_i \in \mathcal{L}_i} \mathsf{at}(\ell_i) \ \wedge (\mathit{urg}(\ell_i) + h_{\min}) \ \wedge \bigwedge_{\alpha \in \gamma(B_i)} \Big(\neg \mathit{Plannable}(\alpha) \ \vee \bigvee_{\substack{\beta \in \mathit{conf}(\alpha) \\ B_i \notin \mathit{part}(\beta)}} \overline{\mathit{Plannable}(\beta)} \Big) \Big].$$

□

Notice that due to the monotony of ϕ on upper bound horizons, we obtain the following lemma:

Lemma 5.2.1. *If $\mathtt{T}_p$ is action-time-lock free for the upper bound horizons function $h_{\max}$, then it is action-time-lock free for any upper bound horizon function $h'_{\max} \le h_{\max}$.*

In order to attest that planning interactions does not introduce deadlocks, we use an SMT solver to check the satisfiability of ϕ. As explained earlier, a given system is deadlock-free under the restricted LPS if $\mathit{Reach}(\mathtt{T}_p) \wedge \phi$ is unsatisfiable. Since $\mathit{Reach}(\mathtt{T}_p) \subseteq \mathit{Reach}(\mathtt{T}_g)$ (Corollary 5.1.2), we can verify the above on $\mathit{Reach}(\mathtt{T}_g)$. Effectively, we do not compute $\mathit{Reach}(\mathtt{T}_g)$ to avoid the combinatorial explosion problem, inherent to composition of timed

automata. In fact, we rather build an over-approximation, $\overline{Reach(\mathtt{T}_g)}$, of the latter, and use it during our verification. Finding a strategy granting action-time-lock-freedom is based on the idea of restricting the upper bound horizon function $h_{\max}$. In fact, since $h_{\min}$ is a parameter that is dependent of the communication latency of a given execution platform, it cannot be tuned. Instead, initially for each interaction $\alpha \in \gamma$, $h_{\max}(\alpha) = +\infty$. Thereafter, due to the monotony of ϕ (Lemma 5.2.1) on upper horizons, this parameter will be refined, that is, its maximum will be decreased until finding a function $h_{\max}$ for which $\overline{Reach(\wedge)}\phi$ is unsatisfiable or until reaching the upper horizon function $h_{\max}^{h_{\min}}$ for which $h_{\max}(\alpha) = h_{\min}$ for every $\alpha \in \gamma$ and such that $\overline{Reach(\wedge)}\phi$ is satisfiable.

5.3 Planning Semantics as Real-Time Controller Synthesis

In Section 5.3, we presented a method that provides execution strategies by restricting the upper bounds planning horizon for each interaction. This strategy aims to preserve the deadlock-freedom property of a given system under the local planning semantics without imposing further scheduling constraints. This approach relies on the verification of a given expression on over-approximations of the reachable states of the initial semantics. Thus, it may give false-positive results due to *(i)* the nature of expression to check (sufficient condition) and *(ii)* the over-approximation of the reachable states of the *LPS* using over-approximations of the reachable states of the standard semantics (Corollary 5.1.2).

In such cases, an alternative is to tackle the problem as a real-time controller synthesis problem. Real-time controller synthesis is a common method used to extract an execution strategy from formal specifications satisfying certain properties. Usually, these properties express the reachability (resp. non-reachability) of a set of winning states (resp. bad states). In case of planning interactions with bounded horizons, the idea is to restrict the transition relation so that all the remaining behaviors do not lead to states where a component is urgent and no possible execution including this component may occur. This can be formalized as a reachability game for a timed game automaton [**?**], where the main idea consists in trying to find an execution strategy guaranteeing that a given set of namely *bad states* of the system are never reached.

In order to apply this approach, it is required to encode the planning of interactions and their effects on the system, that is, *(i)* encode interactions planning as synchronizations between components, *(ii)* reserve the components of the planned interactions until their chosen execution date, i.e, keep track of the planned interactions and their execution dates, and *(iii)* characterize the set of bad states. Thereafter, tools such as UPPAAL-Tiga [**?**] can be used to find an execution strategy of the planning semantics avoiding the set of bad states, that is, deadlock states. Expressing the planning problem as a real-time controller synthesis problem is not an easy task. Hereinafter, we discuss the different issues met during the formalization process and provide suggestions for solving them.

5.3.1 Planning Zones

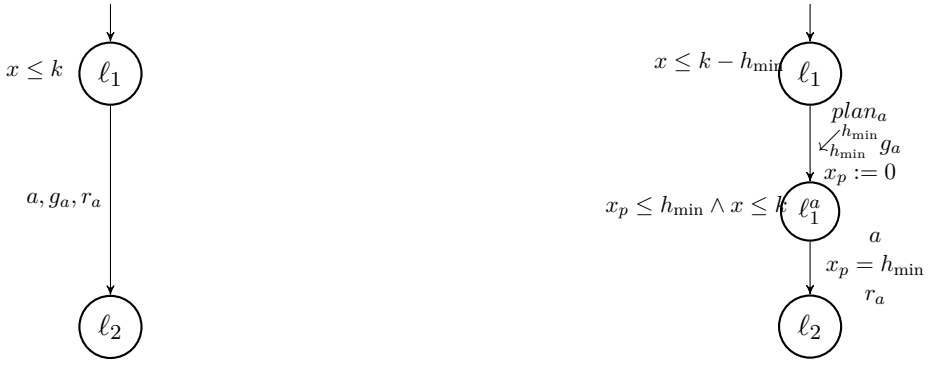
From Equation 5.2, we can see that the clocks values for planning an interaction α are calculated at a global level, that is, by applying the $\swarrow_{h_{\min}}^{h_{\max}(\alpha)}$ on the conjunction of its participating

actions timing constraints. Notice that for a timing constraint $g = g_1 \wedge g_2$, we have:

$$\swarrow_{h_{\min}}^{h_{\max}(\alpha)} g = \swarrow_{h_{\min}}^{h_{\max}(\alpha)} (g_1 \wedge g_2) \implies \swarrow_{h_{\min}}^{h_{\max}(\alpha)} g_1 \wedge \swarrow_{h_{\min}}^{h_{\max}(\alpha)} g_2 \tag{5.4}$$

The above equation bears out the fact that planning states must be encoded on the composition of the system model and not on individual components. Particularly, equation 5.4 points out the fact that encoding the planning on transitions of individual components will induce additional behavior ($\swarrow_{h_{\min}}^{h_{\max}(\alpha)} (g_1 \wedge g_2) \implies \swarrow_{h_{\min}}^{h_{\max}(\alpha)} g_1 \wedge \swarrow_{h_{\min}}^{h_{\max}(\alpha)} g_2$). This represents the first drawback of this method since building the composition may be tedious especially for big scale systems. Therefore, a simple solution to avoid computing the composition is to consider models with interactions having timing constraints on up to one of their participating actions, that is, given an interaction $\alpha = \{a_i\}_{i \in I} \in \gamma$, we have $g_\alpha = \mathit{true}$ or $g_\alpha = g_{a_i}$, with $g_{a_j} = \mathit{true}$ for $j \in I, j \neq i$. In fact, considering interactions including up to one action with timing constraints, will allow to encode the planning on individual components that, additionally to the defined synchronizations (interactions), will also synchronize their planning actions.

The idea is to split each transition of the initial model into two transitions: *(1)* a planning transition, followed by *(2)* an execution transition after the plan transition being performed. For an interaction $\alpha \in \gamma$, the choice of the planning horizon, that is, the duration for which components participating in α will be blocked for until their execution, will be encoded on the execution transition of the component whose action $a_i \in \alpha$ and $g_\alpha = g_{a_i}$. Otherwise, if $g_\alpha = \mathit{true}$ this choice is made arbitrarily. Consequently, this component will be equipped with a clock x_p that will be used to track the planning dates. Finally, location invariants must also be translated to enforce planning at the latest $h_{\min}$ units of time before their expiry. Figure 5.1 depicts an overview of such transformation for $\delta = h_{\min}$ horizon:



(a) Part of a timed automaton

(b) Planning encoding

Figure 5.1 – Planning as a Timed Automaton

5.3.2 Infinite Planning Transitions

Effectively, in order to encode the planning in timed automata, horizons values must be integer. Moreover, due to the dense time nature of the planning intervals (relative planning date for each interactionα are in $[h_{\min}, h_{\max}(\alpha)]$), we end up with an infinity of plan transitions, especially when not restricted upper bound planning horizons, i.e., $h_{\max} = h_{\max}^{\infty}$. Consequently, the

first thing to do is to restrict for each interaction $\alpha \in \gamma$ the upper bound planning horizon $h_{\max}(\alpha)$.Thereafter, we propose to discretize the planning horizons in order to obtain finite values in $\mathbb{Z}_{>0}$ (Figure 5.2). In what follows, we denote by $Disc : \gamma \longrightarrow \mathcal{D}$ the discretized horizon function defining for each interaction its respective discretized planning horizons $\mathcal{D} \subset \mathbb{Z}_{>0}$.

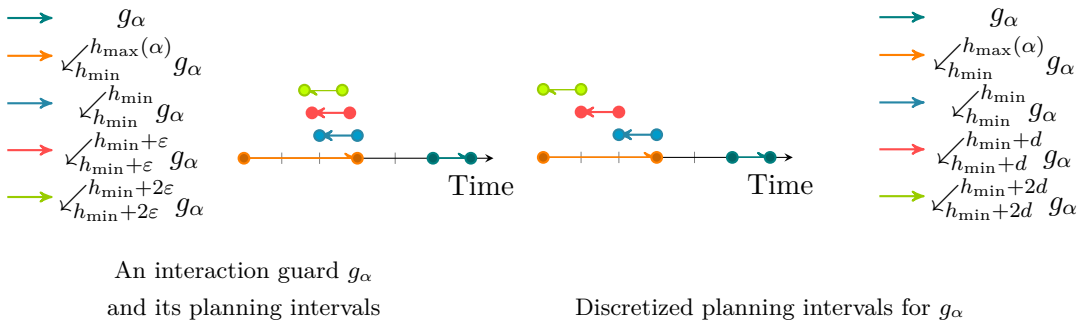


Figure 5.2 – Discretizing Planning Horizons for Interaction

Definition 5.3.1 (Planning Timed Automaton)**.** Given n timed components $\mathcal{B}_i = (\mathcal{L}_i, \ell_0^i, \mathcal{A}_i, \mathcal{T}_i, \mathcal{X}_i, \mathcal{I}_i)$ synchronizing through the interaction set γ such that, for each interaction $\alpha \in \gamma$, the guard of α is equal to the guard of one of its included actions. We define the corresponding planning model as the composition of the n timed automata $\mathcal{B}_i^p = (\mathcal{L}_i^p, \ell_0, \mathcal{A}_i \cup \mathcal{P}_i, \mathcal{T}_i^p, \mathcal{X}_i \cup \{x_i^p\}, \mathcal{I}_i^p)$, w.r.t the interaction set $\gamma \cup \mathcal{P}$, where:

- $\mathcal{P}_i = \cup_{a \in \mathcal{A}_i} p_a$ is the set of *Planning Actions*
- $\mathcal{P} = \{p_\alpha = \{p_{a_i}\}_{i \in I} | \alpha \in \gamma \wedge \alpha = \{a_i\}_{i \in I}\}$ is the set of *Planning Interactions*
- x_i^p is a *Tracking Clock* for interactions execution in each component
- $\mathcal{L}_i^p = (\mathcal{L}_i \cup \mathcal{L}_{i_p})$ is the set of control locations, where $\mathcal{L}_{i_p}$ is the set of locations following planning actions
- $\mathcal{T}_i^p$ is such that for each $(\ell_i, a_i, g_i, r_i, \ell_i') \in \mathcal{T}_i$, $a_i \in \alpha$ and for each $\delta \in Disc(\alpha)$:
  - if $g_\alpha \neq true$ we have:

    Planning transitions: $\begin{cases} \ell_i \xrightarrow{p_{a_i}, true, \emptyset} \ell_{a_i}, & \text{if } g = true \\ \ell_i \xrightarrow{p_{a_i}, \swarrow^{\delta}_{\delta}, g_i, r(x_i^p)} \ell_{a_i}^{\delta}, & \text{otherwise} \end{cases}$

    Execution transitions: $\begin{cases} \ell_{a_i} \xrightarrow{a, true, r_i} \ell_i', & \text{if } g = true \\ \ell_{a_i}^{\delta} \xrightarrow{a, g_a \wedge x_i^p = \delta, r_i} \ell_i', & \text{otherwise} \end{cases}$

    where $\ell_a, \ell_{a_i}^{\delta} \in \mathcal{L}_{i_p}$.

  - if $g_\alpha = true$, we choose one action $b \in \alpha$:

    Planning transitions: $\begin{cases} \ell_i \xrightarrow{p_{a_i}, true, \emptyset} \ell_{a_i}, & \text{if } a \neq b \\ \ell_i \xrightarrow{p_{a_i}, true, r(x_i^p)} \ell_{a_i}^{\delta}, & \text{otherwise} \end{cases}$

$$\text{Execution transitions: } \begin{cases} \ell_{a_i} \xrightarrow{a_i, true, r_i} \ell'_i, \text{ if } a \neq b \\ \ell^{\delta}_{a_i} \xrightarrow{a_i, g_i \wedge x^p_i = \delta, r_i} \ell'_i, \text{ otherwise} \end{cases}$$

- $\mathcal{I}^p_i$ is the set of *Location Invariants* , such that $\forall \ell^p_i \in \mathcal{L}^p_i$, we have:
  $\mathcal{I}^p_i(\ell^p_i) = \begin{cases} \mathcal{I}(\ell_i) - h_{\min}, \text{ if } \ell^p_i = \ell_i \in \mathcal{L}_i \\ x^p_i \leq \delta \wedge \mathcal{I}(\ell_i), \text{ if } \ell^p_i = \ell^{\delta}_{a_i} \in \mathcal{L}_{i_p} \text{ such that } \ell_i \in \mathcal{L}_i \wedge \ell_i \xrightarrow{p_{a_i}} \ell^{\delta}_{a_i}, \end{cases}$

For a composition $\gamma(B_1, \cdots, B_n)$, let $\mathtt{T}_{p'} = (\mathcal{Q}_{p'}, \gamma' \cup \mathbb{R}_{>0}, \longrightarrow_{\gamma'})$, where $\gamma' = \gamma \cup \mathcal{P}$, be the corresponding timed transition system of its planning model under the standard semantics.

Theorem 5.3.1. *$\mathcal{T}_{p'} \dot{\sqsubseteq}_{R'} \mathcal{T}_g$ where R' is the relation defined as follows: For $q^p = (\ell^p, \upsilon^p) \in \mathcal{Q}_{p'}$ and $q^g = (\ell^g, \upsilon^g) \in \mathcal{Q}_g$, such that $(q^p, q^g) \in R'$, we have:*

- *$\ell^p = (\ell^p_1, \cdots, \ell^p_n)$, $\ell_g = (\ell^g_1, \cdots, \ell^g_n)$:*

  $$\forall i \in \{1, \cdots, n\}, \ \ell^g_i = \begin{cases} \ell^p_i, \ \textit{if } \ell^p_i \in \mathcal{L}_i, \\ \ell_i, \ \textit{if } \ell^p_i \in \mathcal{L}_{i_p} \textit{ with } \ell_i \xrightarrow{a,g,r} \ell^p_i \in \mathcal{T}^p_i \wedge \ell_i \in \mathcal{L}_i, \end{cases}$$

  Notice that for the case where $\ell^p_i \in \mathcal{L}_{i_p}$, ℓ_i is unique by construction of the planning model.

- *$\upsilon^g = equ(\upsilon^p)$, where $equ(\upsilon^p)$ is the projection of υ^p on clocks of υ^g*

Proof of Theorem 5.3.1. To prove that $\mathtt{T}_{p'} \dot{\sqsubseteq}_{R'} \mathtt{T}_g$, we need to prove that:

1. $\forall (q^p, q^g) \in R', \sigma \in \gamma \cup \mathbb{R}_{>0}$ such that $q^p \xrightarrow{\sigma}_{\gamma'} q'^p \Rightarrow \exists q'^g . (q'^p, q'^g) \in R' \wedge q^g \xrightarrow{\sigma}_{\gamma} q'^g$
2. $\forall (q^p, q^g) \in R', p_\alpha \in \mathcal{P}$ such that $q^p \xrightarrow{p_\alpha}_{\gamma'} q'^p \Rightarrow (q'^p, q^g) \in R'$

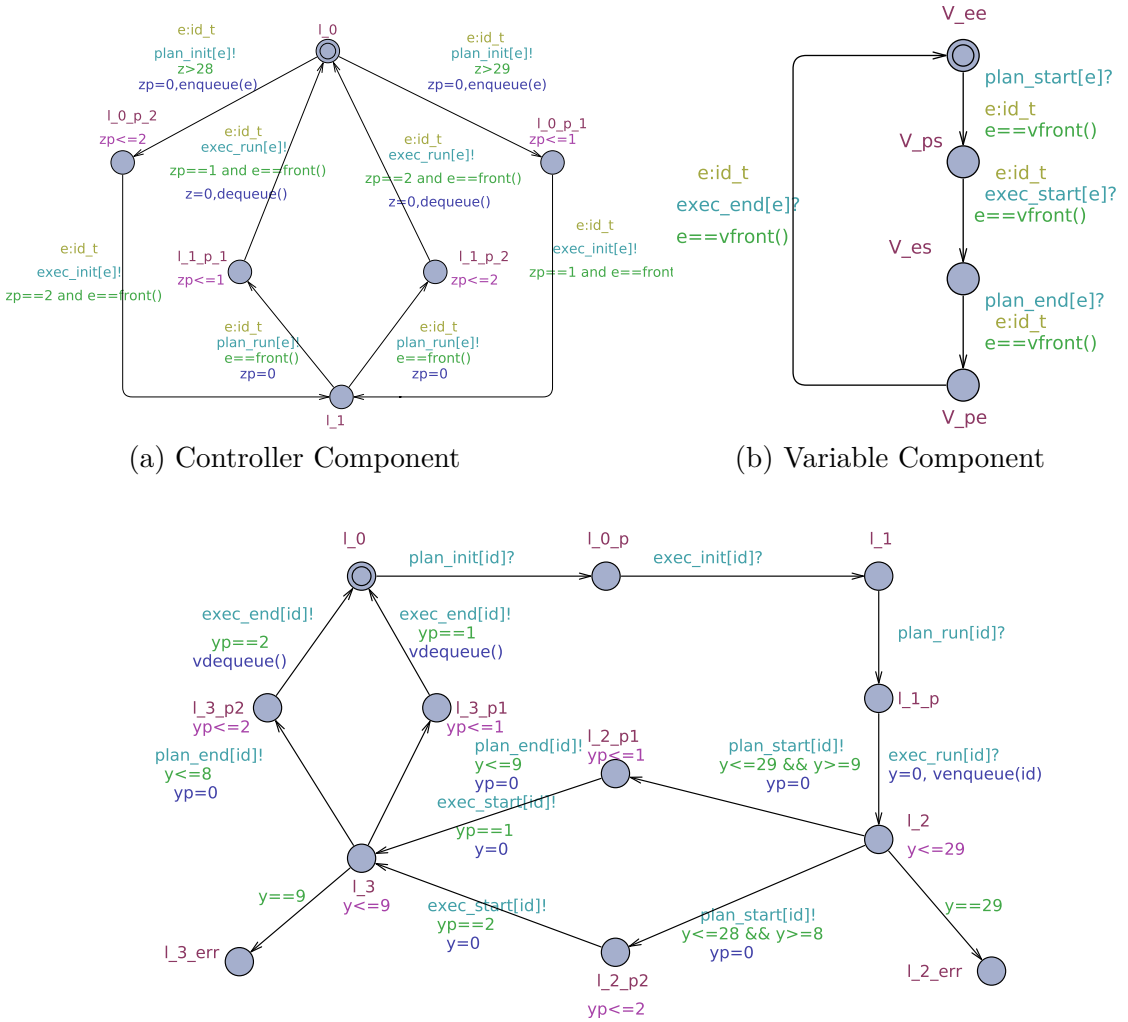
1. (a) Suppose that $(q^p, q^g) \in R', \sigma = \alpha \in \gamma$ and $q^p \xrightarrow{\alpha}_{\gamma'} q'^p$ with $q'^p = ((\ell'^p_1, \cdots, \ell'^p_n), \upsilon'^p)$. We have: $q^p \xrightarrow{\alpha}_{\gamma'} q'^p \Rightarrow g_\alpha$ is *true*, and for $\alpha = \{a_i\}_{i \in \mathcal{I}}$, by construction of the planning automaton, we have: $\ell^g_i \xrightarrow{a_i, g_i, r_i} \ell'^g_i$ such that $\ell'^g_i = \ell'^p_i$. Moreover, since the same clocks are reset by the execution of α in both models, we deduce that $\upsilon'^g = equ(\upsilon'^p)$. By remarking that the state of components not participating in α remains the same, we conclude that $\exists q'^g$ such that $q^g \xrightarrow{\alpha}_{\gamma} q'^g \wedge (q'^p, q'^g) \in R'$.

   (b) Suppose that $(q^p, q^g) \in R', \sigma \in \mathbb{R}_{>0}$ and $q^p \xrightarrow{\sigma}_{\gamma'} q'^p$. For $q^p_i = (\ell^p_i, \upsilon^p_i)$, we define $\mathcal{I}_g$ the set of indexes such that $\ell^p_i \in \mathcal{L}_i$, and $\mathcal{I}_p$ the set of indexes such that $\ell^p_i \in \mathcal{L}_{p_i}$.

   - $\forall i \in \mathcal{I}_g . \ell^p_i = \ell^g_i \wedge q^p_i \xrightarrow{\sigma} q'^p_i \Rightarrow q^g_i \xrightarrow{\sigma} q'^g_i$. This implication is a direct result of the planning model definition since: $\sigma \leq \mathcal{I}(\ell^p_i) \leq \mathcal{I}(\ell^g_i) - h_{\min}$.
   - $\forall i \in \mathcal{I}_p . \ell^g_i = \ell_i$ such that $\ell^p_i \in \mathcal{L}_{i_p}$ with $\ell_i \xrightarrow{a,g,r} \ell^p_i \in \mathcal{T}^p_i \wedge \ell_i \in \mathcal{L}_i$. Thus $q^p_i \xrightarrow{\sigma} q'^p_i \Rightarrow q^g_i \xrightarrow{\sigma} q'^g_i$, since $\mathcal{I}(\ell^p_i) \implies \mathcal{I}(\ell^g_i)$.

   We conclude that $\exists q'^g$ such that $q^g \xrightarrow{\sigma}_{\gamma} q'^g \wedge (q'^p, q'^g) \in R'$.

2. Suppose that $(q^p, q^g) \in R'$ and $q^p \xrightarrow{p_\alpha}_{\gamma'} q'^p$, with $p_\alpha \in \mathcal{P}$ and $q'^p = ((\ell'^p_1, \cdots, \ell'^p_n), \upsilon'^p)$. We have: $q^p \xrightarrow{p_\alpha}_{\gamma'} q'^p \Rightarrow$ for $\alpha = \{a_i\}_{i \in \mathcal{I}}$ $\ell^g_i = \ell^p_i \wedge \ell^g_i \xrightarrow{p_{a_i}} \ell'^p_i$. Moreover, since planning actions reset only the clocks x^p_i for tracking execution time, we can deduce that $(q'^p, q^g) \in R'$.



(a) Controller Component (b) Variable Component

(c) Task Component

Figure 5.3 – Planning Automata for the Task Manager Example

□

Once interactions planning encoded, one last thing to do is to add the set of bad states to each planning automaton (if needed) and find a strategy to avoid those states. Figure 5.3 depicts the corresponding planning automata for example of Figure 2.3 with respect to Definition 5.3.1. Locations suffixed by p, correspond to locations following planning actions, whereas locations ending with err define the bad states, that is, states with urgent location invariant(s) and no possible execution removing the urgency. In this example, for each interaction $\alpha \in \gamma$, we chose $\mathcal{D}(\alpha) = \{1, 2\}$. Notice that for this example, we consider that all actions are controllable actions since it is a closed system in the sense that there is no interaction with the environment.

We performed the verification on the Task Manager examples with 20 tasks. The winning condition being a safety condition: avoid all *"err"* locations. This was translated into the

following property:

$$\text{control: A[ ] forall (i : int[0,N-1]) not (Task(i).l\_2\_err or Task(i).l\_3\_err)} \tag{5.5}$$

The property of interest was successfully verified. Additionally, we were also able to synthesize all wining actions of all states using the command line of UPPAAL-Tiga. A sample of the resulting output is provided below Figures 5.4. Notice that the average execution time[1] for verifying Property 5.5 is 0.1141 seconds (0.6534 seconds when requesting the generation of a strategy).

```
State:  ( Controller.l_1_p_1 Task(0).l_0 Task(1).l_1_p Task(2).l_3_p2
Task(3).l_0 Task(4).l_0 Task(5).l_0 Task(6).l_0 Task(7).l_0 Task(8).l_0
Task(9).l_0 Task(10).l_0 Task(11).l_0 Task(12).l_0 Task(13).l_0 Task(14).l_0
Task(15).l_0 Task(16).l_0 Task(17).l_0 Task(18).l_0 Task(19).l_0 Var.V_pe
) vlist[0]=2 vlist[1]=0 vlist[2]=0 vlist[3]=0 vlist[4]=0 vlist[5]=0
vlist[6]=0 vlist[7]=0 vlist[8]=0 vlist[9]=0 vlist[10]=0 vlist[11]=0
vlist[12]=0 vlist[13]=0 vlist[14]=0 vlist[15]=0 vlist[16]=0 vlist[17]=0
vlist[18]=0 vlist[19]=0 vlen=1 Controller.list[0]=1 Controller.list[1]=0
Controller.list[2]=0 Controller.list[3]=0 Controller.list[4]=0
Controller.list[5]=0 Controller.list[6]=0 Controller.list[7]=0
Controller.list[8]=0 Controller.list[9]=0 Controller.list[10]=0
Controller.list[11]=0 Controller.list[12]=0 Controller.list[13]=0
Controller.list[14]=0 Controller.list[15]=0 Controller.list[16]=0
Controller.list[17]=0 Controller.list[18]=0 Controller.list[19]=0
Controller.len=1
When you are in (Controller.zp==1 && Task(2).yp<=2), take transition
Controller.l_1_p_1-> Controller.l_0 { zp == 1 && 1 == front(), exec_run[1]!,
z := 0, dequeue() } Task(1).l_1_p->Task(1).l_2 { 1, exec_run[id]?, y := 0,
venqueue(id) } When you are in (Task(2).yp==2 && Controller.zp<=1), take
transition Task(2).l_3_p2-> Task(2).l_0 { yp == 2, exec_end[id]!, vdequeue()
} Var.V_pe->Var.V_ee { 2 == vfront(), exec_end[2]?, 1 }
```

Figure 5.4 – Sample of the Output Strategy from UPPAAL-Tiga

5.3.3 Discussion

In this section, we explained how the problem of planning interactions can be formalized into a real-time controller synthesis approach. However, this approach has some drawbacks. In order to encode planning of interactions in components as timed automata, this approach restricts its scope to discretized horizon values which results in having less control over the planning dates of interactions, and leads in case of a high number of discretized values, to an explosion in the number of planning transitions. Unfortunately, we do not have an immediate solution for this problem. In fact, it is user dependent since one user may just want to have a ASAP execution for a given interaction, for instance because the components involved in

[1]The experiments have been conducted on a HP machine with Ubuntu 16.04, an Intel® Core™i5-4300U processor of frequency 1.90GHz×4, and 7.7GiB memory.

this interaction are often requested, and in that case the practice will be to always plan with $h_{\min}$. In other cases, the user may want to plan an interaction with flexible amount time. Additionally, this approach considers only a class of systems where interactions have timing constraints on up to one of their participating components action. Otherwise, the planning should be encoded on the composition, which represents a tedious work because of the state space explosion problem. Nevertheless, this approach differs form the usual scheduler synthesis approach since it is not performed on the regular semantics of timed automata. Particularly, here we are interested in avoiding bad states of the planning semantics (states that verify the expression of Theorem 5.2.1). Consequently, unless finding an automatic general method for generating such complex expression in the query language accepted by such tools, and without ignoring that finding a strategy avoiding those states may be hard in term of computational complexity, our real-time controller synthesis approach seems more straightforward and much simpler but it comes with some feasibility restriction.

Chapter 6

Clock Drift

Contents

6.1 Distributed Timed Automata 81
6.1.1 Expressing Clock Constraints Using Local Clock 81
6.1.2 Distributed Component 82
6.1.3 Correctness 83
6.2 Robust Distributed Semantics 85
6.3 Related Work 86

6.1 Distributed Timed Automata

In the previous section, we introduced a timed automata model that describes a high level representation of systems execution. However, this type of model assumes that components clocks are perfectly synchronous which is hardly the case in practice. Effectively, clocks are able to measure time up to a certain precision and will be likely to drift since they are implemented based on oscillators that are not perfect (the oscillator frequency is not constant, it changes depending on environmental conditions and aging). In this section, we present a distributed timed automata model where clocks advance at different rates and we study the effect of clock drift on the system behavior.

6.1.1 Expressing Clock Constraints Using Local Clock

When building distributed real-time systems, a common practice is to use local clocks as time references [**?** **?**]. These clocks measure the absolute time elapsed since the system startup and are never reset. This approach reduces the effort of keeping track of the actual time progress in components and enable to have a common time scale.

The idea consists in mapping each clock of a component to a (unique) local clock. Thus, the value of component clocks are obtained by simply shifting their corespondent local clock by a constant amount of time as soon as the clocks are not reset. Effectively, for each clock x of a component, we introduce a real variable ρ_x that stores the absolute time of its last reset (w.r.t to its local clock), that is, if x is mapped to a local clock g, then each time x is reset, ρ_x is update to the current value of g. Notice that the value of x can be found by the equality

$x = g - \rho_x$. As a result, any timing constraints c of a component B_i can be expressed using a local clock g as follows:

$$c = \bigwedge_{x_i \in \mathcal{X}_i} l \lhd x_i \rhd u = \bigwedge_{x_i \in \mathcal{X}_i} l + \rho_{x_i} \lhd g \rhd u + \rho_{x_i} \tag{6.1}$$

where $\lhd \in \{<, \leq\}$ and $\rhd \in \{>, \geq\}$. Notice any timing constraints of Definition **??** can be written on the form of inequalities.

6.1.2 Distributed Component

For an interaction α, we denote by $\mathsf{clock}(\alpha)$ the set of clocks appearing in its timing constraints, that is, $\mathsf{clock}(\alpha) = \{x \in \mathcal{X} | \forall (\ell, \alpha, g, r, \ell') \in \mathcal{T}, x \in g\}$. We also put for a subset of interaction γ_i, $part(\gamma_i) = \{\cup_{\alpha \in \gamma_i} part(\alpha)\}$.

Definition 6.1.1 (Interaction Partition)**.** Given a set of interaction γ, a partition of γ is a subset $\{\gamma_k\}_{k=1}^m$, such that $\gamma = \gamma_1 \cup \cdots \gamma_m$ and $\forall i, j \in \{1, \cdots, m\}$ such that $i \neq j$, $\gamma_i \cap \gamma_j = \emptyset$.

In what follows, we assign to each interaction partition a unique local clock based on which interactions constraints are evaluated. Moreover, we imposes that interaction partitions do not share clocks, that is, for a given partition $\{\gamma_k\}_{k=1}^m$:

$$\bigwedge_{\substack{i,j \in \{0,\cdots,m\} \\ i \neq j}} part(\gamma_i) \cap part(\gamma_j) = \emptyset$$

This avoid timing inconsistency and ensure that each clock of a component is evaluated using a unique local clock. We call this type of partitions timing conflict free partitions (TCF).

Definition 6.1.2 (Distributed Component)**.** Let $S = (\mathcal{L}, \ell_0, \gamma, \mathcal{X}, \mathcal{T}, \mathcal{I})$ be the composition of n timed automaton synchronized through the interaction set γ. Given a TCF partitioning of interaction $\{\gamma_k\}_{k=1}^m$, we define the corresponding distributed timed automaton as the tuple $S^{dt} = (\mathcal{L}, \ell_0, \gamma, \mathcal{X}^{dt}, \pi, \mathcal{P}^{dt}, \mathcal{T}^{dt}, \mathcal{I}^{dt})$ such that:

- $\mathcal{X}^{dt}$ is the set of local clocks (a unique clock per interaction partition)
- $\pi : \mathcal{X} \longrightarrow \mathcal{X}^{dt}$ is a many to one mapping between component clocks and local clocks
- $\mathcal{P}^{dt} = \{\cup_{x \in \mathcal{X}} \rho_x | \rho_x \in \mathbb{R}_{\geq 0}\}$ is the set of positive real variables storing the last relative reset times
- $\mathcal{T}^{dt}$ is such that for every $(\ell, \alpha, g, r, \ell') \in \mathcal{T}$, $\alpha \in \gamma_i \subset \pi$, we have the corresponding transition $(\ell, \alpha, g^{dt}, r^{dt}, \ell') \in \mathcal{T}^{dt}$ where:
  - g^{dt} is the guard g expressed using the local clock $x_i \in \mathcal{X}^{dt}$
  - $r^{dt} = \{\rho_x | x \in r\}$ is the update function for reset variables
- $\mathcal{I}^{dt}$ is the set of location invariants expressed using local clocks

We extend the notion of reset to variable of $\mathcal{P}^{dt}$ as follows:

$$\upsilon[r](\rho_x) = \begin{cases} \upsilon(\pi(x)), \text{ if } \rho_x \in r \\ \upsilon(\rho_x), \text{ otherwise} \end{cases}$$

Property 6.1.1 (Semantics)**.** *A distributed component* $S^{dt} = (\mathcal{L}, \ell_0, \gamma, \mathcal{X}^{dt}, \pi, \mathcal{P}^{\mathcal{X}}, \mathcal{T}^{dt}, \mathcal{I}^{dt})$ *w.r.t a given partitioning of interaction* π *defines the* $LTS^{dt} = (\mathcal{Q}^{dt}, q_0^{dt}, \gamma, \longrightarrow_\gamma)$ *where:*

- $\mathcal{Q}_{dt} = \mathcal{L} \times \upsilon(\mathcal{X}_{dt} \times \mathcal{P}_{dt}) \times \Delta$ *where* $\mathcal{L}$ *is the set of global locations,* $\mathbb{R}^{\mathcal{X}}_{\geq 0}(\mathcal{X}_{dt} \times \mathcal{P}_{dt})$ *is the set of clock and data valuations, and* Δ *is the the offset of clock in* $\mathcal{X}_{dt}$ *w.r.t to an implicit referene clock (perfect)*
- $q_0^{dt} = (\ell_0, 0, 0)$ *is the initial state.*
- $\longrightarrow_\gamma \subseteq \mathcal{Q} \times (\gamma \cup \mathbb{R}_{>0}) \times \mathcal{Q}$ *is the set of labeled transitions defined by the rules:*
  - $(\ell, \upsilon, \delta) \xrightarrow{\alpha}_\gamma (\ell', \upsilon[r], \delta)$ *for* $\alpha \in \gamma$*, if* $(\ell, \alpha, g_{dt}, r_{dt}, \ell') \in \mathcal{T}_{dt} \wedge g_{dt} \models \upsilon$
  - $(\ell, \upsilon, \delta) \xrightarrow{d}_\gamma (\ell, \upsilon', \delta')$ *for* $d \in \mathbb{R}_{>0}$*, such that* $\upsilon' = \upsilon + d - \delta + \delta' \wedge \upsilon' \models \mathcal{I}(\ell) \wedge \upsilon' \geq \upsilon$

In what follows, we consider a more realistic scheme where clock can drift up to a certain value ϵ with respect to a reference clock. This induces that $\delta \in [-\epsilon, \epsilon]^{|\mathcal{X}^{dt}|}$.

6.1.3 Correctness

Property 6.1.2. $\forall x, y \in \mathcal{X}^{dt}, x \neq y$ *we have* $\upsilon(x) - \upsilon(y) \in [-2\epsilon, 2\epsilon]$

Property 6.1.2 states that the relative drift between local partitions clock is bounded by 2ϵ. This results from the fact that all local clocks are kept within ϵ of a reference clock.

In order to attest the correctness of the distributed semantics, we compare its corresponding LTS and the LTS of the standard semantics.

Definition 6.1.3 (Simulation)**.** A simulation over $A = (\mathcal{Q}_A, \sum, \rightarrow_A)$ and $B = (\mathcal{Q}_B, \sum, \rightarrow_B)$ is a relation $R \subseteq \mathcal{Q}_A \times \mathcal{Q}_B$ such that we have:

- $\forall (q, r) \in R, a \in \sum,\ q \xrightarrow{a}_A q' \implies \exists r' \in \mathcal{Q}_B$ such that $r \xrightarrow{a}_B r' \wedge (q', r') \in R$
- $\forall (q, r) \in R, d \in \mathbb{R}_{\geq 0},\ q \xrightarrow{d}_A q' \implies \exists r' \in \mathcal{Q}_B$ such that $r \xrightarrow{d}_B r' \wedge (q', r') \in R$

B simulates A, denoted by $A \sqsubseteq_R B$, means that B can do everything A does. Notice that if $A \sqsubseteq_R B$ and $B \sqsubseteq_R A$ we say that A and B are bisimilar ($A \sim_R B$).

Let R be the relation:

$$R = \{(q_{dt}, q) \in \mathcal{Q}_{dt} \times \mathcal{Q} | q_{dt} = (\ell_{dt}, \upsilon_{dt}, \delta), q = (\ell, \upsilon)$$
$$\text{such that: } \exists k \in [-\epsilon, \epsilon]^{|\mathcal{X}^{dt}|} \begin{cases} \ell_{dt} = \ell, \\ \forall x \in \mathcal{X}, \upsilon(x) = \upsilon^{dt}(\pi(x)) - \rho_x - \delta \end{cases} \}$$

The relation R_{dt} relates states of the distributed semantics with states of the standard semantics having the same location configuration, and whose clocks valuations expressed on local clocks are $\epsilon-$close. We call such states $\epsilon-$similar.

Lemma 6.1.1. *For* $\Delta = \{0\}$*, we have* $LTS^{dt} \sim_R LTS$

Lemma 6.1.1 describes the fact that a given system and its corresponding distributed model are bisimilar when $\Delta = \{0\}$, that is, when clocks advance at the same rate (perfect clocks).

Property 6.1.3. *Let $(q^{dt}, q) \in R$ such that q^{dt} satisfies Enabled$(()\alpha)$, then q satisfies Enabled$^{\swarrow^{\epsilon}}(\alpha) \vee$ Enabled$(()\alpha) \vee$ Enabled$^{\nearrow^{\epsilon}}(\alpha)$.*

Property 6.1.3 expresses that for states $(q^{dt}, q) \in R$ if it exists an interaction enabled at q^{dt} then this interaction is either enabled at q, will be enabled after a time progress of ϵ or is up to ϵ after the deadline of α. This property flows directly from the $\epsilon-$similarity of states $(q^{dt}, q) \in R$, the form of interactions timing constraints (conjunction of intervals) and the fact that clocks involved in the same interaction advances at the same rate. It points out that any execution from state q^{dt} might not be always possible from state q or any of its time successor.

Lemma 6.1.2. *Let $(q^{dt}, q) \in R$ such that q^{dt} satisfies Enabled$(()\alpha) \wedge$ Enabled$^{\swarrow^{\epsilon}}(\alpha)$, then q satisfies Enabled$(()\alpha) \vee$ Enabled$^{\swarrow^{\epsilon}}(\alpha)$.*

Lemma 6.1.2 can be deduced straightforwardly from property 6.1.3. It expresses that for states $(q^{dt}, q) \in R$, if it exists an interaction $\alpha \in \gamma$ such that α is enabled and that clock valuations are up to ϵ before the deadline of this interaction then, α can be executed from q or by doing a time progress up to ϵ.

The usual notion of simulation as defined in 6.1.3 is too precises. It requires that each trace in one system can be matched *exactly* by a trace in the other system, that is, two states can be distinguished even for an infinitesimally small mismatch between timings ($\delta \in \Delta \neq \{0\}$). Thus, we rely on the following quantitative variant of simulation [TFL10].

Definition 6.1.4 ($\epsilon-$simulation)**.** Given two LTS, $LTS_1 = (\mathcal{Q}_1, \sum, \longrightarrow_1)$ and $LTS_2 = (\mathcal{Q}_2, \sum, \longrightarrow_2)$, a relation $R \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$ is a:

- Strong timed $\epsilon-$simulation, if for any $(q, r) \in R$, $\sigma \in \sum$, $d \in \mathbb{R}_{\geq 0}$
  - $q \xrightarrow{\sigma} q'$ implies $r \xrightarrow{\sigma} r'$ for some $r' \in \mathcal{Q}_2$ with $(q', r') \in R$
  - $q \xrightarrow{d} q'$ implies $r \xrightarrow{d'} r'$ for some $r' \in \mathcal{Q}_2$ and $d' \in \mathbb{R}_{\geq 0}$ with $|d' - d| \leq \epsilon$ and $(q', r') \in R$
- Timed action $\epsilon-$simulation, if for any $(q, r) \in R$, $\sigma \in \sum$, $d \in \mathbb{R}_{\geq 0}$
  - $q \xrightarrow{d,\sigma} q'$ implies $r \xrightarrow{d',\sigma} r'$ for some $r' \in \mathcal{Q}_2$ and $d' \in \mathbb{R}_{\geq 0}$ with $|d' - d| \leq \epsilon$ and $(q', r') \in R$

If there exists a strong timed (resp. timed action) $\epsilon-$simulation between LTS_1 and LTS_2 w.r.t R_ϵ, then we write $LTS_1 \sqsubseteq^{\epsilon}_{R} LTS_2$ (resp. $LTS_1 \sqsubseteq^{\epsilon*}_{R} LTS_2$

This approach characterizes the degree of closeness between timed systems: it generalizes the (boolean) notions of timed simulation (yes or no) to metrics over timed system. Formally, for a positive real number ϵ, a state q is told $\epsilon-$similar to another state r if there is a time-abstract simulation that can relates both states in the sense that the difference between the delays of time-step transitions is at most ϵ.

Although this definition of simulation is less restrictive, Property 6.1.3 gives the intuition that for some state $(q^{dt}, q) \in R$ there may be interactions that can be executed from q^{dt} but not from q, which make the $\epsilon-$simulation impossible.

6.2 Robust Distributed Semantics

As explained in previous sections, the distributed semantics may exhibit new behavior with respect to the standard semantics. In this section, we identify the problematical states of the distributed semantics and provide sufficient conditions that will guarantee action (or strong) timed $\epsilon-$simulation.

Definition 6.2.1 (Potentially Bad States)**.** We denote by PBS the set of potentially bad states of the distributed semantics characterized as follows:

$$PBS(S^{dt} = \{q^{dt} = (\ell, \upsilon, \delta) | \exists \alpha \in \gamma, \upsilon \models g_\alpha \wedge \upsilon + \epsilon \not\models g_\alpha\}$$

The intuition behind this characterization results form Property 6.1.3. Restricting the progress of time to this amount before any interaction deadline will prevent any execution not possible in the corresponding state of the standard semantics. Consequently, we restrict the progress of time in the distributed semantics as follows:

$$(\ell, \upsilon, \delta) \xrightarrow{d}_\gamma (\ell, \upsilon', \delta') \text{ for } d \in \mathbb{R}_{>0}, \text{ such that } \upsilon' = \upsilon + d - \delta + \delta' \wedge \upsilon' \models \mathcal{I}(\ell) \wedge (\ell, \upsilon, \delta) \notin PBS \tag{6.2}$$

Proposition 6.2.1. *Let $Reach(S^{dt*})$ be the set of reachable state of the restricted distributed semantics and LTS^{dt*} its respective labeled transition system. We have:*

$$PBS(S^{dt*}) \cap Reach(S^{dt*}) = \emptyset \implies LTS^{dt*} \sqsubseteq^{\epsilon*}_R LTS^g$$

Proposition 6.2.1. In order to prove proposition 6.2.1, we need to show that if for any $(q^{dt}, q) \in R$, $\sigma \in \gamma$ and $d \in \mathbb{R}_{\geq 0}$:

$$q^{dt} \xrightarrow{d,\sigma} q^{dt'} \implies \exists q' \in \mathcal{Q}^g, q \xrightarrow{d',\sigma} q', \text{ with } d' \in \mathbb{R}_{\geq 0}, \ |d' - d| \leq \epsilon \text{ and } (q^{dt'}, q') \in R$$

Let $(q^{dt}, q) \in R$ such that $PBS(S^{dt}) \cap Reach(S^{dt}) = \emptyset$ and $\exists \sigma \in \gamma, \exists d \in \mathbb{R}_{\geq 0}$, $q^{dt} \xrightarrow{d,\sigma} q^{dt'}$. We distinguish two cases:

Case 1: $d > 0$ We have $q^{dt} \xrightarrow{d,\sigma} q^{dt'}$. This means:

$$\begin{cases} \upsilon^{dt'} = (\upsilon^{dt} + d - \delta + \delta')[r^{dt}] \\ \upsilon^{dt} + d - \delta + \delta' \models \mathcal{I}^{dt}(\ell) \wedge g^{dt}_\sigma \\ \upsilon^{dt} + d - \delta + \delta' + \epsilon \models \mathcal{I}^{dt}(\ell) \wedge g^{dt}_\sigma \quad \textit{by } 6.2 \end{cases}$$

which expressed on original clocks becomes:

$$\begin{cases} \upsilon^{dt'} = (\upsilon + \rho + k + d - \delta + \delta')[r^{dt}] \\ \upsilon + d + k - \delta + \delta' \models \mathcal{I}(\ell) \wedge g_\sigma \\ \upsilon + d + k - \delta + \delta' + \epsilon \models \mathcal{I}(\ell) \wedge g_\sigma \end{cases}$$

We also have $k - \delta + \delta' \in [-3\epsilon, 3\epsilon]$. Consequently we have:

1. $k - \delta + \delta' \in [0, \epsilon]$, then:

$$\begin{cases} v^{dt'} = \upsilon + d + k - \delta + \delta' + (\rho)[r^{dt}] \\ \upsilon + d + \epsilon \models \mathcal{I}(\ell) \wedge g_\sigma \end{cases}$$

   We put $d' = d + \epsilon$, we obtain:

$$\begin{cases} v^{dt'} = \upsilon + d' + k - \delta + \delta' - \epsilon + (\rho)[r^{dt}] \text{ since } r^{dt} \text{ applies only on } \rho \\ \upsilon + d' \models \mathcal{I}(\ell) \wedge g_\sigma \end{cases}$$

   Notice that $k' = k - \delta + \delta' - \epsilon \in [-\epsilon, 0]$ Then $\upsilon + d + \epsilon \models \mathcal{I}(\ell) \wedge g_\sigma$. We put $d' = d + \epsilon$ and conclude that $q \xrightarrow{d',\sigma} q'$ such $(q^{dt'}, q') \in R$

2. We repeat the same procedure for the rest of the intervals in $[-3\epsilon, 3\epsilon]$

Case 2: $d = 0$ Since $PBS(S^{dt}) \cap Reach(S^{dt}) = \emptyset$, using the same methodology of case 1, we can conclude that $\exists q' \in \mathcal{Q}^g, q \xrightarrow{d',\sigma} q'$, with $d' \in \mathbb{R}_{\geq 0}$, $|d' - d| \leq \epsilon$ and $(q^{dt'}, q') \in R$ This proof shows that we can always find an execution sequence of the standard semantics that is up to $3\epsilon-$similar to any execution sequence of the distributed semantics. □ □

Definition 6.2.2 (Robustness)**.** For a given timed system S, we say that S is robust to clock drifts iff $LTS^{dt*} \sqsubseteq_R^{\epsilon*} LTS^g$ and $Reach(S^{dt})$ is deadlock-free.

6.3 Related Work

Robust reachability has been introduced to check whether a given timed automata model (system) still satisfies the specification when subject to different perturbations such as clocks drift. In [**?**], Puri introduced a model of clock drift for closed timed automata by introducing a parameter $\epsilon > 0$ that bounds the clocks drift rates. This work showed that the standard reachability analysis approach is not correct when clocks drift, even by infinitesimally small amount, and subsequently provide a region based method for calculating $Reach^*(S, q_0)$, the set of reachable states for *every* drift (the limit as $\epsilon \to 0$), that is, $Reach^*(S, q_0) = \cap_{\epsilon>0} Reach(S_\epsilon, q_0)$. Other works [DK06, Dim07] proposed a zone based algorithm for computing this reach-set more efficiently and generalized the approach for open timed automata model [Dim07]. In [WDMR04], another perturbation model was considered. Here, the system model is syntactically modified by *relaxing* the guards through a parametrized enlargement of δ. De Wulf [WDMR04] showed that the notion of robustness defined in [Pur00] and studied in other works [DK06, Dim07] is closely related to the notion of implementability introduced in [WDMR04], that is, whether for some $\delta > 0$, the enlarged system model still satisfies the requirements expressed by the safety properties. This allows to prove that the considered notion of implementability is decidable for timed automata. Furthermore, [SFK08] considers a more realistic model of drifting clocks by considering clock resynchronization, available now in most distributed real-time systems. It was proven that standard zone-based reachability analysis is exact when testing robust safety, provided a uniform strictly positive robustness margin of 1. Finally, [**? ?**] tackled the problem by proposing a timed automata model with independent clocks (icTA). Akshay *et al.* focused more on the untimed language of icTA, where Ortiz *et al.* suggested an alternative semantics for

icTA and introduced an extension to the usual notion of bisimulation (multi-timed bisimulation) to study the behavior of the latter.

Chapter 7

Implementation and Experimentation

Contents

7.1 The BIP Framework **89**
7.2 The BIP Toolbox **93**
7.3 Tools Developed in this Thesis **95**
7.4 Experimentation **97**

7.1 The BIP Framework

BIP [BBB+11b] is a highly expressive model-based and component-based framework for building embedded applications. It is based on three main layers: Behavior, interactions, and priorities as shown in Figure 7.1:

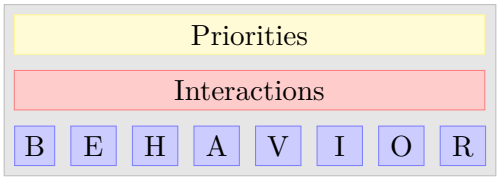


Figure 7.1 – BIP Layers

- Behavior: This layer describes the behavior of each component of a system as a timed transition system. Atomic components are defined as timed automata with well defined interface (ports).
- Interactions: The interaction layer specifies how components interact together and coordinate their action using n-ary synchronization. It restricts thus the global behavior of components together using these synchronizations.
- Priorities: Priorities are used to favor the execution of a subset of enabled interactions called *maximal* interactions. They can be used to resolve conflicts or to express particular scheduling policies.

The Real-Time BIP language [ACS10] extends the BIP language through clocks used to express timing constraints on transitions and locations (locations invariant). It also supports urgency types on transitions [BS00] that provide additional means to constrain the progress of time in a given system. In this thesis, we do not consider priorities or urgency types. However, given a timed system S that includes priority rules on interactions and/or urgency types on transitions, then its underlying timed transition system $\mathtt{T}_S$ is included in the timed transition system representing its abstraction S^* from the latter, that is, $\mathtt{T}_S \subseteq \mathtt{T}_{S^*}$. This means that all the results of this thesis, if they apply on S^*, then they apply on S.

The BIP language defines system as a composition of components using a set of syntactic constructs that specify components behavior and interface as well as the ongoing synchronizations (interactions) and priorities. They consist of the following:

- Atomic component: Atoms represent the simplest entity of a system. An atomic component may include a set of ports, data and clocks. Its behavior is described using an automaton or a Petri net whose transitions are labeled by ports with guards possibly on data and/or clocks. Ports and data can be exported, and thus become accessible at a higher hierarchy level (compound). They define the interface of a component. Additionally, atomic components support the usage of external C/C++ functions on transition guards and may as well trigger the execution of such functions on the execution of a transition.
- Connector: Connectors are stateless entities that characterize the possible interactions between a set of components via their interface ports.
- Priority: Priorities are used to restrict the possible set of enabled interactions.
- Compound: A compound is a composite component that consists of a set of atomic components, the connectors specifying their interactions and a set of priority rules. It may as well export ports and data.
- Package: A BIP package is a compilation unit contained in a single .bip file. It may contain data, external data types, external functions, external operators, port types, atom types, connector types and compound types. It also may use other BIP packages.

Example 7.1.1. Figure 7.2 illustrates the syntax of BIP and presents different elements of the BIP language. The package *ControllerWorker* includes the definition of the port type *Port*, the connector type *Link* for interactions involving two ports of type *Port*, atomic components *Controller* and *Worker*. The atomic component *Controller* consists of a clock x, an internal port *init* and two exported port, a and c, defining its interface. The statement `place lc0, lc1, lc2` defines the places of the timed automaton describing its behavior, where `lc0` is the initial place. Lines 20 to 24 represent the declaration of a transition. It consists of the following elements:

1. The port labeling the transition: **`on`** `init`
2. The source and destination places: **`from`** `lc0` **`to`** `lc1`
3. A possible empty list of guards over data and clocks: **`when`** $x \geq 8$ `second`
4. A possible empty list of clocks to reset: **`reset`** `x`

5. A possible empty transfer function for updating data variable or triggering the execution of external C code: `do{}`

As explained earlier, components are composed using connectors. The Compound Type *System* defines a system composed of two Workers and one Controller. It also defines the interaction between the *controller* component an each *Worker* component ($worker_1$ and $worker_2$) through connectors $ab1$, $cd1$, $ab2$, and $cd2$.

```
1  package ControllerWorker
2
3    port type Port()
4
5    connector type Link (Port p1, Port p2)
6      define p1 p2
7    end
8
9  atom type Worker()
10   clock y
11
12   export port  Port d()
13   export port  Port b()
14
15   place l1, l2
16
17  initial to l1
18
19   on b
20     from l1 to l2
21     when ( y>= 5 )
22     do {}
23
24   on d
25     from l2 to l1
26     reset y
27     do {}
28 end
```

```
29 atom type Controller()
30   clock x
31   export port  Port a()
32   export port  Port c()
33   port Port init()
34
35   place lc0, lc1, lc2
36
37   initial to lc0
38     do {}
39
40   on init
41     from lc0 to lc1
42     when (x >= 8 second)
43     reset x
44     do {}
45
46   on a
47     from lc1 to lc2
48     when (x == 5 second)
49     reset x
50     do {}
51
52   on c
53     from lc2 to lc1
54     reset x
55     do {}
56
57   invariant inv1 at lc1  when (x<= 5 second)
58 end
59
60 compound type System ()
61          component Worker worker1 ()
62          component Worker worker2 ()
63          component Controller controller ()
64
65          connector Link ab1 (worker1.b, controller.a)
66          connector Link ab2 (worker2.b, controller.a)
67          connector Link cd1 (worker1.d, controller.c)
68          connector Link cd2 (worker2.d, controller.c)
69
70     end
71 end
```

Figure 7.2 – A BIP Example

7.2 The BIP Toolbox

The BIP framework provides a rich set of tools that allows to model, verify and execute systems. The BIP toolbox is structured in different categories as shown by Figure 7.3.

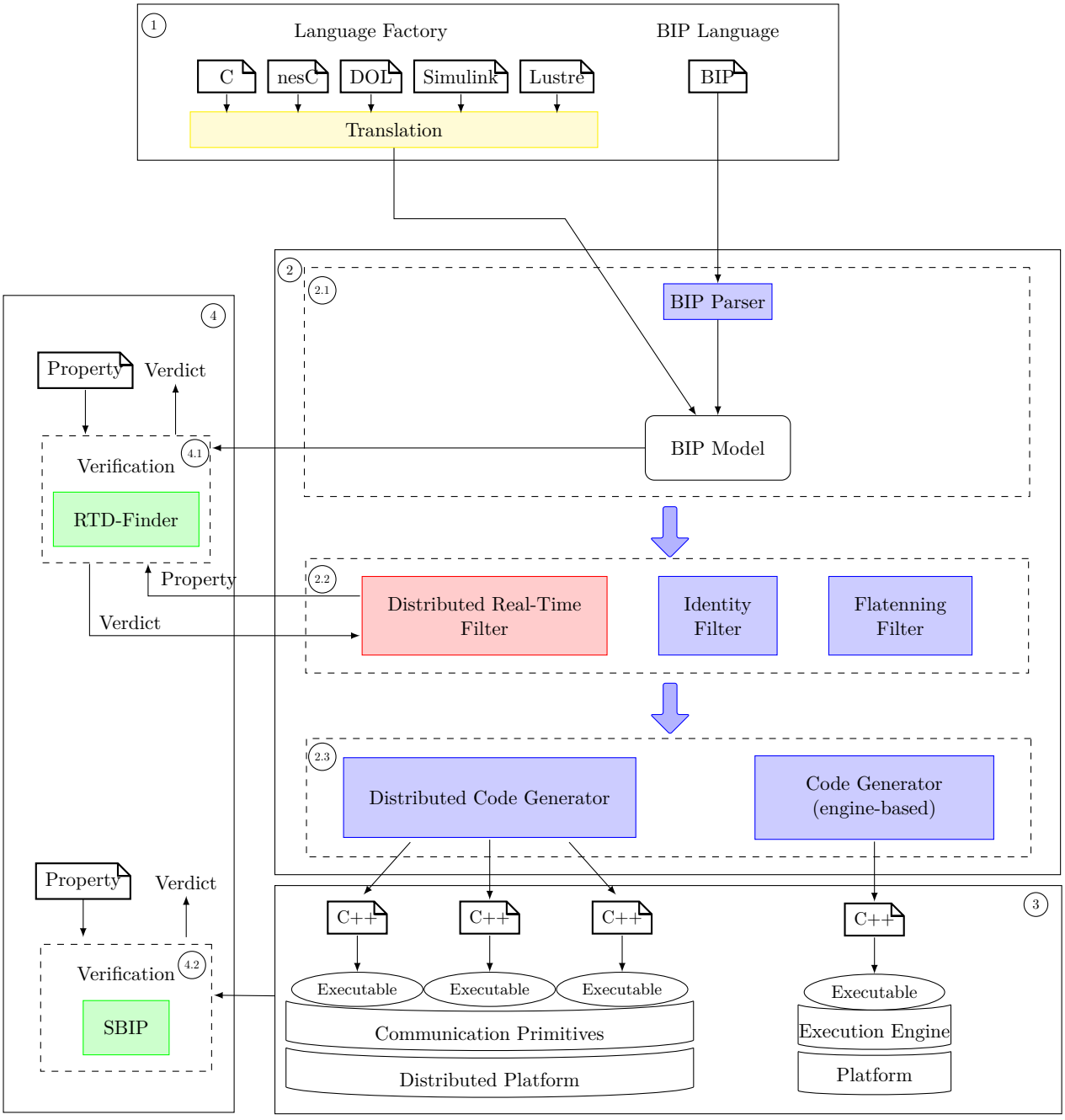


Figure 7.3 – The BIP Toolbox

(1) Language Factory

This category includes *translation of various language or modeling paradigm* in addition to the BIP language. It includes translation of synchronous languages [BSS09, STS+10] as well as languages combining software applications and hardware architecture [CRBS08, BMP+07, BBB+11c], allowing thus automatic generation of BIP models through several translation steps. First, the functional modules of the considered application, along with the necessary data structures and applications functions, are translated into BIP components. Thereafter, connectors representing the interactions between the application modules are added. Finally, priorities may be added to refine the behavior of the obtained BIP model with respect to the expected behavior.

Additionally, in association with the RTD-Finder tool it provides analyses allowing performance evaluation [DCBB17] as well as the actual analyses for the approach presented in Section 5.2. Note that the identity filter is the default filter that given a BIP model return the same BIP model.

(2) The BIP Compiler

The BIP compiler consists of three parts:

1. The front-end : it interacts with the user of the compiler. It reads user input and transforms it in a form suitable for the following process (ie. internal representation).

2. The middle-end : applies operations on the internal representation (eg. optimizations, architectural transformations,etc.). Such operations are contained into small blocks of the middle-end that we will call filter later on. An example of filters that can be found in the middle-end of the BIP compiler are:

   - The identity filter is the default filter that given a BIP model return the same BIP model.
   - The flattening filter transforms a hierarchical system into flattened atomic components synchronized through flattened interaction.
   - The distributed real-time filter includes the transformation of BIP models to Send/Receive models as described in Chapter 3. It also includes our implementation of the methods presented in Chapter 4 and Chapter 5.

   The BIP middle-end can be also associated with the RTD-Finder tool for validation and optimization purposes.

3. The back-end : It consists of a code generator that produces the final result from the internal representation. Usually, the final output is in the form of a source code in a programming language (eg. C++). We distinguish two types of code generators, namely, a self contained distributed code generator and an engine-based generator.

A typical compilation consists of the following steps:*(1)* First, the front-end executes and creates a BIP-EMF model. Then, *(2)* the filters in the middle-end are executed in turn. The result is a possibly modified BIP-EMF model. Finally, *(3)* the wanted back-end is executed and produces the compilation results.

(3) Simulation and Execution

As stated above, the back-end of the BIP compiler generates the final representation of a BIP model as source code in a programming language such as C++. The resulting source code is either self contained and can be directly compiled and deployed for execution, or it can be linked with an engine that computes the corresponding execution sequences according to the BIP semantics. Usually, the representation used is a C++ software that is linked against the engine's runtime to create an executable software. Typically, engines target one or more of the following main goals:

- *Execution* of the model corresponds to the computation of a single execution sequence that is intended to be executed on the target platform. In this case, the engine realizes the connection between the model and the platform in order to ensure a correct behavior of the execution.
- *Simulation* of the model corresponds to the computation of a single execution sequence that is intended to be executed on the host machine for simulation purpose, that is, time is interpreted in a logical way.
- *Exploration* of the model corresponds to the computation of several execution sequences corresponding to multiple simulations of the model.

(4) Verification Tools

The BIP toolset includes two verification tools intended for system validation and performance evaluation.

1. The RTD-Finder [RBBC16] tool is a compositional verification tool that allows to verify a given system against a set of *safety properties* such as deadlock freedom, mutual exclusion or bounded response time. It is based on the computation of invariants representing over-approximations of the reachable states of a system.
2. SBIP [MNB+18] is a statistical model-checker that supports multiple modeling formalism ranging from DTMCs to CTMCs and GSMPs. It includes a single integrated environment where one can edit models, compile, simulate, and perform SMC analysis.

7.3 Tools Developed in this Thesis

The methods presented in Chapter 4 and Chapter 5 have been implemented in the distributed real-time filter of the BIP compiler. The latter also includes the transformation of BIP models to Send/Receive models as described in Chapter 3. Figure 7.4 depicts an overview of the distributed real-time filter. It includes the following modules:

- **Analyser**: The Analyser creates internal abstractions of the input BIP model. Particularly, it includes:
    1. Component Info: It encompasses atomic component information such as a map indicating for each port a list of source and destination locations matching transitions labeled by this port, with the corresponding timing constraints. It also includes urgency predicate for each component.

2. Interaction Info: This unit builds for each interaction the set of its participating components, the involved port for each components, as well as all the possible combinations (locations configurations and timing constraints) based on component info. It also includes for each interaction α all the predicates $Enabled(\alpha)$, $Enabled\not\swarrow(\alpha)$ and $Enabled\not\swarrow_l^u(\alpha)$. The latter is constructed based on the *Planning Horizons* files provided as input.
3. Compound Info: The *Compound Info* unit combines the aforementioned info in order to build a given system abstraction. In addition to components and interactions info, it construct the set of potential conflicts based on the *Interaction Partition* file and all the necessary elements required for building Send/Receive models of Chapter 3, and for the generation of expressions presented in Chapter 4 and Chapter 5.

- **Property Generator**: The *Property Generator* module builds all the necessary expressions for the optimization of conflicts detection and the deadlock freedom verification of the local planning semantics. It interacts with the RTD-Finder tool in order to achieve these tasks.

- **Send/Receive Transformation**: Using the system abstraction provided by the *Analyser* and based on the result of the verification results obtained from the RTD-Finder tool, the *Send/Receive Transformation* module applies the transformations presented in Chapter 3.

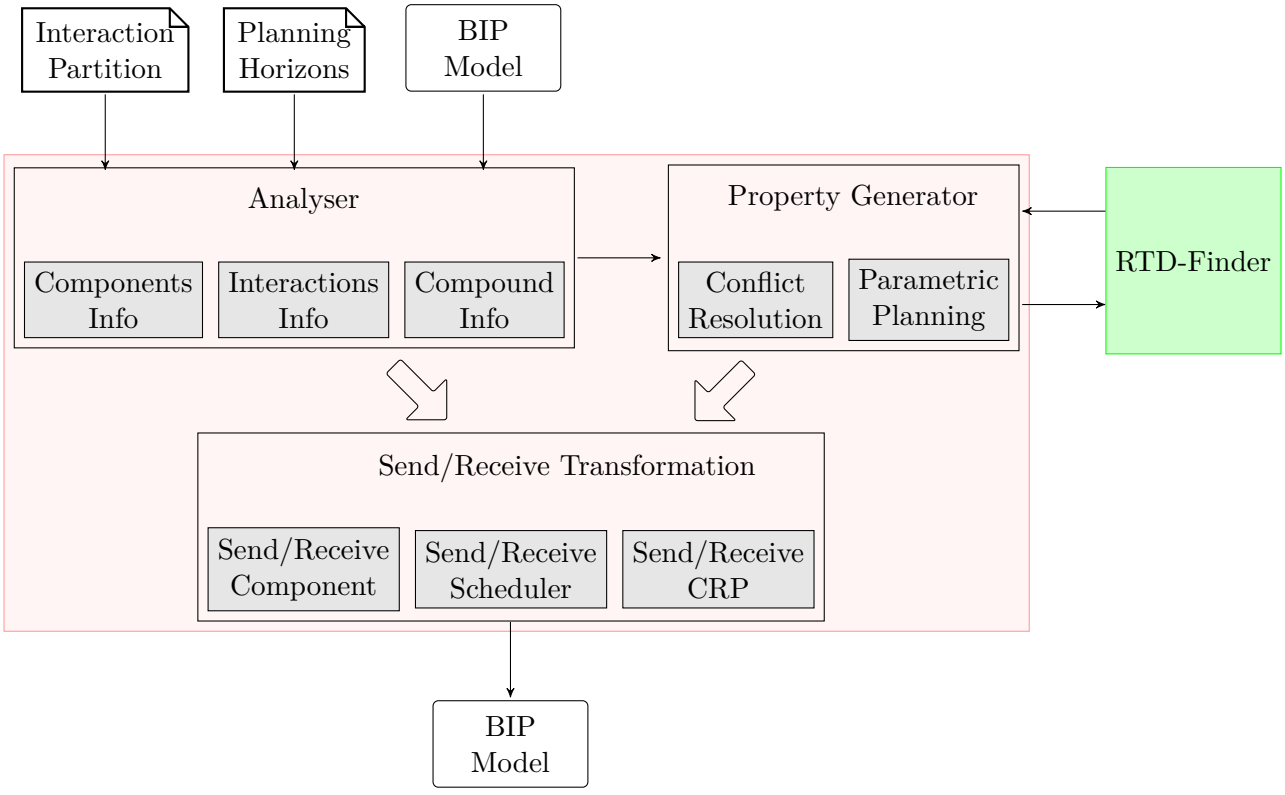


Figure 7.4 – The Distributed Real-Time BIP Filter

7.4 Experimentation

Part III

Conclusion

Chapter 8

Conclusion and Perspectives

List of Figures

1.1 Overal Design Flow . . . 13
1.2 Giotto Workflow . . . 14
1.3 Example of a Giotto Task Execution . . . 15
1.4 Execution of a Processing Operation in Oasis . . . 16
1.5 PTIDES Code Generation and Analysis Workflow . . . 16
1.6 A Lustre Program . . . 17

2.1 Example of a Timed Component . . . 27
2.2 Backward and Forward Operators on a Guard . . . 28
2.3 Task Manager . . . 32
2.4 Example of an Extended Timed Component . . . 34

3.1 High Level Representation of the Target Architecture . . . 38
3.2 Potential Conflict Between Interactions α_1 and α_2 . . . 39
3.3 High Level Representation of a Decentralized Send-Receive Model of the Task Manager Example . . . 41
3.4 Offer Construction . . . 42
3.5 Representation of an Urgent Locatoion . . . 42
3.6 Send/Receive Transformation of the Controller Component From Figure 2.3 . . . 43
3.7 A Simple Petri Net with Two Succesive Markins . . . 45
3.8 Scheduling Mechanism . . . 45
3.9 Internal Representation of Scheduler Sch_1 from Figure 3.3 . . . 49
3.10 Sub-Part of the Timed Component for the Centralized CRP of Figure 3.3 Handeling Interactions of γ_1 . . . 50

4.1 Example of a History Clock for Action a . . . 61
4.2 Refined Version of the Conflict Resolution Layer from Figure 4.2 . . . 63
4.3 Refined Version of Scheduler Sch_1 from Figure 4.3 . . . 63

5.1 Planning as a Timed Automaton . . . 75
5.2 Discretizing Planning Horizons for Interaction . . . 76
5.3 Planning Automata for the Task Manager Example . . . 78
5.4 Sample of the Output Strategy from UPPAAL-Tiga . . . 79

7.1 BIP Layers . . . 89
7.2 A BIP Example . . . 92
7.3 The BIP Toolbox . . . 93

7.4 The Distributed Real-Time BIP Filter 96

List of Tables

1.1 Classification Of Real-Time Systems 10

Bibliography

[ACS10] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-based implementation of real-time applications. In *Proceedings of the 10th International conference on Embedded software, EMSOFT 2010, Scottsdale, Arizona, USA, October 24-29, 2010*, pages 229–238, 2010.

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

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[Bag89] Rajive L. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Trans. Software Eng.*, 15(9):1053–1065, 1989.

[BBB+11a] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.

[BBB+11b] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.

[BBB+11c] Paraskevas Bourgos, Ananda Basu, Marius Bozga, Saddek Bensalem, Joseph Sifakis, and Kai Huang. Rigorous system level modeling and analysis of mixed HW/SW systems. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*, pages 11–20, 2011.

[BBBL13] Saddek Bensalem, Marius Bozga, Benoît Boyer, and Axel Legay. Incremental generation of linear invariants for component-based systems. In *13th International Conference on Application of Concurrency to System Design, ACSD 2013, Barcelona, Spain, 8-10 July, 2013*, pages 80–89, 2013.

[BBC+00] Felix Bachmann, Len Bass, Gary Chastek, Patrick Donohoe, and Fabio Peruzzi. The architecture based design method. Technical Report CMU/SEI-2000-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000.

[BBJ+12] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 25(5):383–409, 2012.

[BCP07] Albert Benveniste, Benoît Caillaud, and Roberto Passerone. A generic model of contracts for embedded systems. *CoRR*, abs/0706.1456, 2007.

[BFM+08] Luca Benvenuti, Alberto Ferrari, Leonardo Mangeruca, Emanuele Mazzi, Roberto Passerone, and Christos Sofronis. A contract-based formalism for the specification of heterogeneous systems (invited). In *Forum on specification and Design Languages, FDL 2008, September 23-25, 2008, Stuttgart, Germany, Proceedings*, pages 142–147, 2008.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[BMP+07] Ananda Basu, Laurent Mounier, Marc Poulhiès, Jacques Pulou, and Joseph Sifakis. Using BIP for modeling and verification of networked systems – A case study on tinyos-based networks. In *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007), 12 - 14 July 2007, Cambridge, MA, USA*, pages 257–260, 2007.

[BS00] Sébastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Inf. Comput.*, 163(1):172–202, 2000.

[BSS09] Marius Bozga, Vassiliki Sfyrla, and Joseph Sifakis. Modeling synchronous systems in BIP. In *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*, pages 77–86, 2009.

[CCDA07] Sylvain Camier, Damien Chabrol, Vincent David, and Christophe Aussaguès. OASIS formal approach for distributed safety-critical real-time system design. In *ISoLA 2007, Workshop On Leveraging Applications of Formal Methods, Verification and Validation, Poitiers-Futuroscope, France, December 12-14, 2007*, pages 167–178, 2007.

[CDA+05] Damien Chabrol, Vincent David, Christophe Aussaguès, Stéphane Louise, and Frédéric Daumas. Deterministic distributed safety-critical real-time systems within the oasis approach. In *International Conference on Parallel and Distributed Computing Systems, PDCS 2005, November 14-16, 2005, Phoenix, AZ, USA*, pages 260–268, 2005.

[Cha09] Robert N. Charette. This car runs on code. *IEEE Spectrum*, 2009.

[CRBS08] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. Translating AADL into BIP - application to the verification of real-time systems. In *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, pages 5–19, 2008.

[DCBB16] Mahieddine Dellabani, Jacques Combaz, Marius Bozga, and Saddek Bensalem. Local planning of multiparty interactions with bounded horizons. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, pages 199–216, 2016.

[DCBB17] Mahieddine Dellabani, Jacques Combaz, Saddek Bensalem, and Marius Bozga. Knowledge based optimization for distributed real-time systems. In *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, pages 751–756, 2017.

[DCBB18a] Mahieddine Dellabani, Jacques Combaz, Marius Bozga, and Saddek Bensalem. Local planning semantics : a semantics for distributed real-time systems. *Leibniz Transactions on Embedded Systems*, 2018.

[DCBB18b] Mahieddine Dellabani, Jacques Combaz, Marius Bozga, and Saddek Bensalem. Revesiting robustness of timed automata under clock drifts. Technical Report TR-2018-6, Verimag Research Report, 2018.

[Dim07] Catalin Dima. Dynamical properties of timed automata revisited. In *Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007, Proceedings*, pages 130–146, 2007.

[DK06] Conrado Daws and Piotr Kordy. Symbolic robustness analysis of timed automata. In *Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS 2006, Paris, France, September 25-27, 2006, Proceedings*, pages 143–155, 2006.

[DOL03] Mark Denford, Tim O'Neill, and John Leaney. Architecture-based design of computer based systems. In *10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003), 7-10 April 2003, Huntsville, AL, USA*, pages 39–46, 2003.

[DS02] Manfred Droste and R. M. Shortt. From petri nets to automata with concurrency. *Applied Categorical Structures*, 10(2):173–191, 2002.

[ELM+12] John C. Eidson, Edward A. Lee, Slobodan Matic, Sanjit A. Seshia, and Jia Zou. Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE*, 100(1):45–59, 2012.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, pages 1–16, 1998.

[HHK03] 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.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Software Eng.*, 18(9):785–793, 1992.

[HLY91] Uno Holmer, Kim Guldstrand Larsen, and Wang Yi. Deciding properties of regular real time processes. In *Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings*, pages 443–453, 1991.

[JCL11] Jeff C. Jensen, Danica H. Chang, and Edward A. Lee. A model-based design methodology for cyber-physical systems. In *Proceedings of the 7th International Wireless Communications and Mobile Computing Conference, IWCMC 2011, Istanbul, Turkey, 4-8 July, 2011*, pages 1666–1671, 2011.

[KJ86] Fritz Krückeberg and Michael Jaxy. Mathematical methods for calculating invariants in petri nets. In *Advances in Petri Nets 1987, covers the 7th European Workshop on Applications and Theory of Petri Nets, Oxford, UK, June 1986*, pages 104–131, 1986.

[Kop04] Hermann Kopetz. An integrated architecture for dependable embedded systems. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil*, pages 160–161, 2004.

[Kop10] Hermann Kopetz. Component-based design of embedded systems. In *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, page 1, 2010.

[Kop11] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer, 2011.

[KSLB03] Gabor Karsai, Janos Sztipanovits, Ákos Lédeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[Lee99] Edward A. Lee. Modeling concurrent real-time processes using discrete events. *Ann. Software Eng.*, 7:25–45, 1999.

[LS11] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. Component-based design for the future. In *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*, page 1029, 2011.

[LXL01] Xiaojun Liu, Yuhong Xiong, and Edward A. Lee. The ptolemy II framework for visual languages. In *2002 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2001), September 5-7, 2001 Stresa, Italy*, page 50, 2001.

[Mar11] Peter Marwedel. *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems, Second Edition*. Embedded Systems. Springer, 2011.

[MNB+18] Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, Mahieddine Dellabani, Jacques Combaz, Axel Legay, and Saddek Bensalem. Sbip 2.0: Statistical model checking stochastic real-time systems. Technical Report TR-2018-5, Verimag Research Report, 2018.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[OG76] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[PCT04] José Antonio Pérez, Rafael Corchuelo, and Miguel Toro. An order-based algorithm for implementing multiparty synchronization. *Concurrency - Practice and Experience*, 16(12):1173–1206, 2004.

[Pur00] Anuj Puri. Dynamical properties of timed automata. *Discrete Event Dynamic Systems*, 10(1-2):87–113, 2000.

[RAB+15] Souha Ben Rayana, Lacramioara Astefanoaei, Saddek Bensalem, Marius Bozga, and Jacques Combaz. Compositional verification for timed systems based on automatic invariant generation. *Logical Methods in Computer Science*, 11(3), 2015.

[RBBC16] Souha Ben Rayana, Marius Bozga, Saddek Bensalem, and Jacques Combaz. Rtd-finder: A tool for compositional verification of real-time component-based systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 394–406, 2016.

[SFK08] Mani Swaminathan, Martin Fränzle, and Joost-Pieter Katoen. The surprising robustness of (closed) timed automata against clock-drift. In *Fifth IFIP International Conference On Theoretical Computer Science - TCS 2008, IFIP 20th World Computer Congress, TC 1, Foundations of Computer Science, September 7-10, 2008, Milano, Italy*, pages 537–553, 2008.

[SSM03] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Petri net analysis using invariant generation. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, pages 682–701, 2003.

[Sta88] John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, 1988.

[STS+10] Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga, and Joseph Sifakis. Compositional translation of simulink models into synchronous BIP. In *IEEE Fifth International Symposium on Industrial Embedded Systems, SIES 2010, University of Trento, Italy, July 7-9, 2010*, pages 217–220, 2010.

[TBCB15] Ahlem Triki, Borzoo Bonakdarpour, Jacques Combaz, and Saddek Bensalem. Automated conflict-free concurrent implementation of timed component-based models. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 359–374, 2015.

[TFL10] Claus R. Thrane, Uli Fahrenberg, and Kim G. Larsen. Quantitative analysis of weighted transition systems. *J. Log. Algebr. Program.*, 79(7):689–703, 2010.

[Tri99] Stavros Tripakis. Verifying progress in timed systems. In *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS'99, Bamberg, Germany, May 26-28, 1999. Proceedings*, pages 299–314, 1999.

[Tri15] Ahlem Triki. *Distributed Implementations of Timed Component-based Systems. (Implémentations distribuées des systèmes temps-réel à base de composants).* PhD thesis, Grenoble Alpes University, France, 2015.

[WDMR04] Martin De Wulf, Laurent Doyen, Nicolas Markey, and Jean-François Raskin. Robustness and implementability of timed automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, pages 118–133, 2004.

[ZLL07] Yang Zhao, Jie Liu, and Edward A. Lee. A programming model for time-synchronized distributed real-time systems. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2007, April 3-6, 2007, Bellevue, Washington, USA*, pages 259–268, 2007.