From timed component-based systems to time-triggered implementations : a correct-by-design approach

Hela Guesmi

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Hela GUESMI

Thèse dirigée par **Saddek BENSALEM, Professeur, UGA**

préparée au sein du **Laboratoire VERIMAG** et du **CEA LIST**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Des systèmes à base de composants aux implémentations cadencées par le temps : Une approche correcte par conception

From Timed Component-Based Systems to Time-Triggered Implementations : A Correct-by-Design Approach

Thèse soutenue publiquement le **« 27 Octobre 2017 »**,
devant le jury composé de :

Madame, Marie-Laure, POTET
PROFESSEUR, GRENOBLE INP, Présidente
Monsieur, Kamel, BARKAOUI
PROFESSEUR, CNAM - PARIS, Rapporteur
Madame, Claire, PAGETTI
MAITRE DE RECHERCHE, ONERA CENTRE MIDI-PYRENEES, Rapporteur
Monsieur, Eugene, ASARIN
PROFESSEUR, UNIVERSITE PARIS 7, Examinateur
Monsieur, Yamin, AIT-AMEUR
PROFESSEUR, INP TOULOUSE - ENSEEIHT, Examinateur
Monsieur, Saddek, BENSALEM
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Directeur de thèse
Monsieur, Belgacem, BEN HEDIA
INGENIEUR-CHERCHEUR, CEA LIST, Co-Encadrant
Monsieur, Simon, BLIUDZE
CHARGÉ DE RECHERCHE, EPFL LAUSANNE SUISSE, Co-Encadrant

Remerciements

Je dédie ce travail à toutes les personnes ayant rendu cette thèse possible par leur aide, leurs contributions et leurs encouragements.

Je tiens à remercier dans un premier temps Mme Claire Pagetti et Prof. Kamel Barkaoui pour avoir aimablement accepté la lourde tâche de rapporter cette thèse. Je remercie également Prof. Marie-Laure Potet, Prof. Eugene Asarin et Prof. Yamine Ait-Ameur pour avoir accepté d'examiner et de juger mon travail.

Je tiens à témoigner toute ma reconnaissance à mon directeur de thèse Prof. Saddek Bensalem pour la patience et la confiance qu'il m'a accordées au cours de ces quelques années, ainsi que pour ses conseils et ses encouragements.
Je remercie mes encadrants de thèse Belgacem Ben Hedia et Simon Bliudze pour leurs conseils, leurs critiques toujours pertinentes, leur patience et pour l'intérêt constant qu'ils m'ont manifesté tout au long de ma thèse.
Je tiens également à exprimer ma reconnaissance à Jacques Combaz, pour sa disponibilité et ses discussions toujours intéressantes.

Je voudrais également remercier tous mes collègues et amis, notamment les membres du L3S du DACLE et l'équipe DCS de Verimag. Je garderai un bon souvenir des discussions animées au cours des repas et dans nos bureaux de doctorants.
Je n'oublie pas mes amis proches dont les encouragements m'ont permis de finaliser cette recherche. Point n'est besoin de les nommer car je suis sûre qu'ils se reconnaîtront.

Mes plus profonds remerciements vont à mes parents Hedi et Selma. Tout au long de mon cursus, ils m'ont toujours soutenue, encouragée et aidée. Ils ont su me donner toutes les chances pour réussir. Qu'ils trouvent, dans la réalisation de ce travail, l'aboutissement de leurs efforts ainsi que l'expression de ma plus affectueuse gratitude.
Je remercie également mes soeurs Nedia et Manel et mon frère Mohamed pour m'avoir fait partager leur joie de vivre et m'avoir ainsi soutenu dans mes efforts.
Un spécial merci à mon mari Wael de m'avoir tenu la main jusqu'aux dernières lignes de ce manuscrit. Je tiens à le remercier surtout pour sa grande patience et son soutien moral ininterrompu.
Je n'oublie pas ma tante Fatma, mes beaux frères Samir et Bassem, mes cousins et cousines et tous les membres de ma famille que je n'ai pas pu citer. Je vous remercie tous, je vous dois tout ce que je suis.

Je remercie enfin toutes les personnes intéressées par mon travail, en espérant qu'elles puissent trouver dans mon rapport des explications utiles pour leurs propres travaux.

Contents

List of Figures

List of Tables

Abstract

In hard real-time embedded systems, design and specification methods and their associated tools must allow development of temporally deterministic systems to ensure their safety. To achieve this goal, we are specifically interested in methodologies based on the Time-Triggered (TT) paradigm. This paradigm allows to preserve by construction number of properties, in particular end-to-end real-time constraints. However, ensuring correctness and safety of such systems remains a challenging task. Existing development tools do not guarantee by construction specification respect. Thus, a-posteriori verification of the application is generally a must. With the increasing complexity of embedded applications, their a-posteriori validation becomes, at best, a major factor in the development costs and, at worst, simply impossible. It is necessary, therefore, to define a method that allows the development of correct-by-construction systems while simplifying the specification process.
High-level component-based design frameworks that allow design and verification of hard real-time systems are very good candidates for structuring the specification process as well as verifying the high-level model.

The goal of this thesis is to couple a high-level component-based design approach based on the BIP (Behaviour-Interaction-Priority) framework with a safety-oriented real-time execution platform implementing the TT approach (the PharOS Real-Time Operating System). To this end, we propose an automatic transformation process from BIP models into applications for the target platform (i.e. PharOS). The process consists in a two-step semantics-preserving transformation. The first step transforms a generic BIP model coupled to a user-defined task mapping into a restricted one, which lends itself well to an implementation based on TT communication primitives. The second step transforms the resulting model into the TT implementation provided by the PharOS RTOS.

We provide a tool-flow that automates most of the steps of the proposed approach and illustrate its use on an industrial case study for a flight Simulator application and a medium voltage protection relay application. In both applications, we compare functionalities of both original, intermediate and final model in order to confirm the correctness of the transformation. For the first application, we study the impact of the task mapping on the proposed transformation. And for the second application, we study the impact of the transformation on some performance aspects compared to a manually written version.

Résumé

Dans le domaine des systèmes temps-réel embarqués critiques, les méthodes de conception et de spécification et leurs outils associés doivent permettre le développement de systèmes au comportement temporel déterministe afin de garantir leur sûreté de fonctionnement. Pour atteindre cet objectif, on s'intéresse aux méthodologies basées sur le paradigme Time-Triggered(TT). Dans ce contexte, nombre de propriétés et, en particulier, les contraintes temps-réel de-bout-en-bout, se voient satisfaites par construction. Toutefois, garantir la sûreté de fonctionnement de tels systèmes reste un défi. En général, les outils existants n'assurent pas par construction le respect de l'intégralité des spécifications, celles-ci doivent, en général, être vérifiées à posteriori. Avec la complexité croissante des applications embarquées, celle de leur validation devient, au mieux, un facteur majeur dans les coûts de développement et, au pire, tout simplement impossible. Il faut, donc, définir une méthode qui, tout en permettant le développement des systèmes corrects par constructions, structure et simplifie le processus de spécification. Pour cela, on s'intéresse aux plateformes de conception haut niveau basée sur composants et qui permettent aussi la vérification des modèles haut-niveau des systèmes temps-réels.

L'objectif de cette thèse est de coupler une approche de conception haut niveau basée sur composants consistant en la plateforme BIP (Behaviour-Interaction-Priority) et une plateforme d'exécution orientée sureté et basée sur le paradigme TT (le système d'exploitation PharOS). Afin d'atteindre cet objectif, on propose un flot de conception basé sur une approche transformationnelle permettant de générer automatiquement une application PharOS à partir d'un modèle BIP. Cette transformation préserve la sémantique d'origine et consiste en deux étapes majeures. La première étape transforme un modèle BIP et un mapping de tâches défini par l'utilisateur en un modèle BIP plus restreint qui s'approche de l'implémentation en respectant les critères de communication TT. La deuxième étape transforme ce modèle résultant en une implémentation PharOS.

L'approche proposée a été implémentée et intégrée dans la chaîne d'outil BIP. Deux études de cas industriels ont permis de la valider: un simulateur de vol et un relais de protection moyenne tension. Pour les deux applications, on compare les fonctionnalités du modèles d'origine avec le modéle intermédiaire et le modéle final. Et ce afin de confirmer la correction de la transformation. Pour la première application, on étudie l'impact du mapping des tâches sur la transformation proposée. Pour la deuxième application, on étudie l'impact de la transformation sur quelques aspects de performances en comparaison avec une version de la même application écrite manuellement.

Introduction

Challenges in building correct hard real-time systems

Modern societies are being more and more involved with embedded systems. These latter have become a major actor in the daily human life by serving a vast variety of application domains such that home appliances, office automation, aerospace, banking and finance, automotive, medical instruments, avionics, etc. Embedded systems are becoming more and more complex, and their pervasiveness in our everyday lives calls their efficiency and reliability into question.

Real-time systems [56] are systems that undergo a set of "real-time constraints" (e.g. start instants, deadlines, etc.). They are classified into two categories; soft and hard real-time systems. In the former category, respect of timing constraints is important, but the system can still function even if these constraints are occasionally violated. Whereas, a failure of hard real-time systems endangers their original intended mission or the life of the human being. Indeed, the correctness of a result of such systems depends on both the time and the value domains. That is a hard real-time system is correct if it produces the correct result while respecting the specified timing constraints. Despite the existence of different techniques in software engineering for ensuring correctness and reliability such as formal verification, simulation and testing, ensuring value and temporal correctness of hard real-time systems is still a challenging and time-consuming task. With the increasing complexity of such embedded applications, their a posteriori verification becomes, at best, a major factor in the cost of development and, at worst, simply impossible. Sometimes, an error in the specifications is not detectable.

In brief, the main challenges that hard real time systems are facing are their exponentially increasing complexity —and therefore the complexity of their design—and the hard and costly a posteriori verification process intended to prove application correctness. In this context number of approaches and paradigms have been proposed.

Component-based approach

In general, the most basic and intuitive way to tackle complex and large problems is to decompose them into smaller ones. Similarly, the principle of the component-based approach is to build complex systems by assembling a set of building blocks called components. In order to fit into an architecture of the system (i.e. the structure of the system), components require coordination mechanisms allowing to describe how they

are connected and interacting. A component is mainly characterized by its interface, an abstraction that is adequate for composition and reuse. The composition of components is achieved with respect to a notion of "glue" operator. The "Gluing" operation takes, as input, components and their constraints and provides, as output, a complex system. Therefore, the global behavior of the system can be inferred from the behavior of its composing components and its related architecture. Component-based systems provide logical clear descriptions of their behaviors which makes them adequate for a correct-by-construction process. In addition, they allow reuse of components and incremental modification without inferring global changes, which may significantly simplify the verification process.

A variety of component-based frameworks have been proposed in order to allow modelling, simulation and verification of critical embedded applications. Nonetheless, such design frameworks usually provide a capability for automatic generation of C++ or Java code, which has to be compiled for the selected target platform. Thus, guaranteeing hard real-time constraints in the implementation within these frameworks is, at best, difficult.

Implementations based on Real-Time Operating Systems (RTOS) and Time-Triggered (TT) execution model

The Time-Triggered (TT) paradigm was introduced by Kopetz [52]. TT systems are based on a periodic clock synchronization in order to enable TT communications and computations. Each subsystem of a TT architecture is isolated by a so-called *temporal firewall* which consists of a shared memory element for a unidirectional exchange of information between sender and receiver task components. It is the responsibility of the *TT communication system* to transport —by relying on the common global time— the information from the sender firewall to the receiver firewall. In a TT system all communication and computation activities are initiated periodically at predetermined points in time. These statically defined activation instants enforce regularity and make TT systems predictable which makes them well-suited for hard real-time systems.

Developing embedded real-time systems based on the TT paradigm is a challenging task due to the necessity to manage, already in the programming model, the fine-grained temporal constraints and the low-level communication primitives imposed by the temporal firewall abstraction. In this context, a variety of Real-Time Operating System (RTOS) that are based on the TT paradigm, have been provided to guarantee the temporal and behavioural determinism of the executed software. They provide a set of primitive mechanisms for handling communication and timing constraints specifications.

Nonetheless, such TT-based RTOS implementations do not provide high-level programming models that would allow the developers to think on a higher level of abstraction and to tackle the complexity of large safety-critical real-time systems.

Challenge: from component-based model to a TT-based RTOS implementations

The goal of our work is to couple the high-level component-based design approach with a safety-oriented RTOS implementing the TT paradigm. We propose a theory and tools that automatically derive correct TT implementation from the original high-level component-based model of the application. This is achieved, by using correct-by-construction source-to-source transformations techniques. The proposed methodology allows, thus, to combine complementary advantages of both approaches; i.e. tackling the complexity and verifying the model using high-level component-based framework, and guaranteeing determinism of the implementation constraints due to the safety-oriented RTOS implementing. Moreover, the correct-by-construction technique allows avoiding the a posteriori verification of properties that are already verified in the original high-level model.

Our contributions

We present, in this thesis, a methodology to provide automatically correct-by-construction TT implementation starting from a high-level component-based model of the software application.

In order to comply with the correct-by-construction approach, we need to rely on a component-based framework which provides rigorous semantics. BIP (Behavior, Interaction, Priority) is such a formalism for modelling heterogeneous component-based systems [2], developed at Verimag. BIP relies on multi-party interactions for synchronizing components and dynamic priorities for scheduling between interactions. Regarding the target implementation, we consider PharOS [9] framework. It is an extension of the OASIS framework [31, 36, 67, 68] implemented for the automotive applications. Oasis and PharOS implementations comprise a programming language ΨC (Parallel synchronous C), which is an extension of C. This extension allows one to specify TT tasks and their temporal constraints as well as their interfaces.

The proposed transformational approach of this thesis relies on two main semantics-preserving transformations; a model-to-model and a model-to-code transformations. In order to be able to prove formal correctness of the second transformation, we provide the semantics of the PharOS formal model which is at the same level as its ΨC programming language. The transformation has been implemented in two main tools and is proved to be semantics preserving. An overview of the contribution of the thesis is displayed in Figure 1.

Input BIP model

In the proposed transformational approach, we consider input models that are described in BIP. The behavior of a BIP component is modelled using timed automata [5] which is extended with data and C update functions. The component model encompasses only

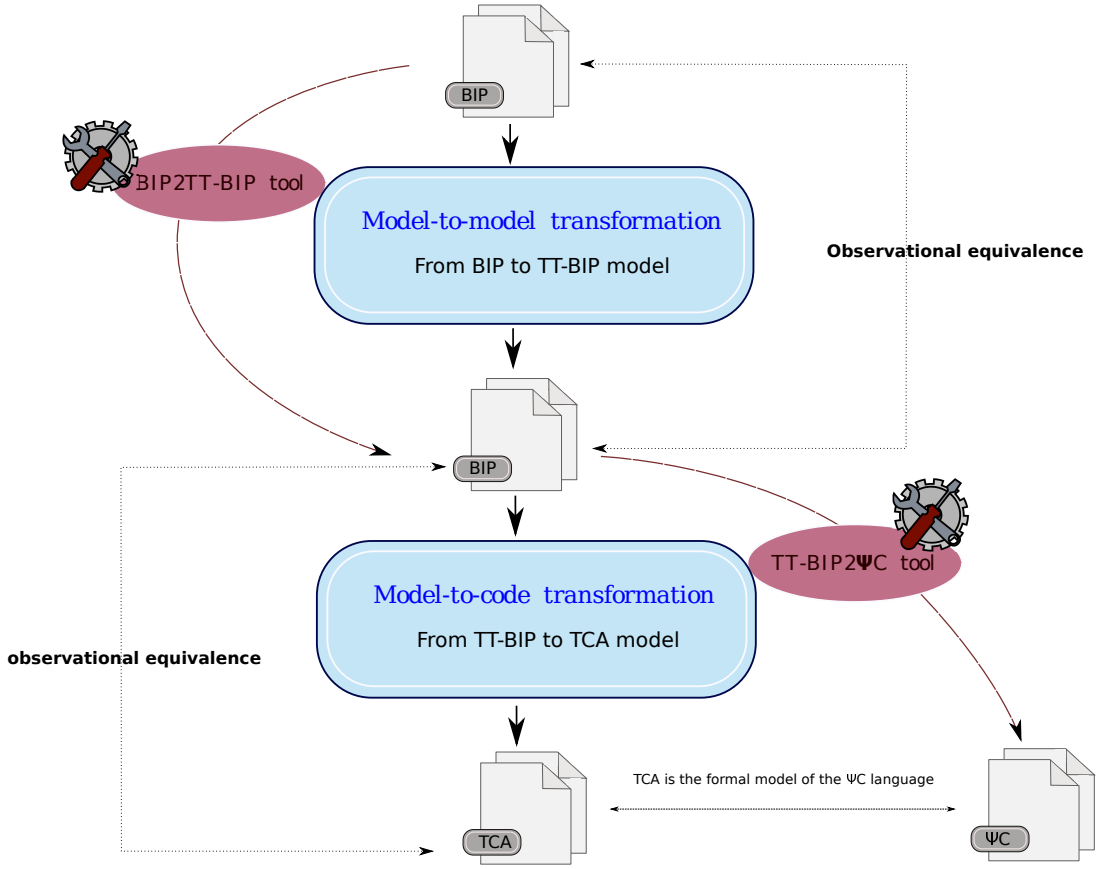


Figure 1: An overview of the methodology presented in this thesis

platform-independent timing constraints consisting in user requirements. Transitions are labelled by ports and are assumed to be timeless. Components are composed using two operators, namely Interaction and Priority. The Interaction operator is parametrized by a set of interactions which synchronize transitions of components. The Priority operator is a partial order on the interactions. In our work, we do not consider the priority operator in BIP input models. The global state semantics of such models is defined by a labelled transition systems LTS where the system can either wait (i.e. when time may progress) or execute the interactions.

From BIP to TT-BIP model: model-to-model transformation

This transformation takes as input a BIP model and produces a more restricted model called TT-BIP model. This transformation is parametrized by a user-defined task mapping. Such transformation allows to obtain a model which is closer to any TT implementation. That is, a model where all intertask interactions are executed by dedicated components and all interactions between its different components correspond to send/receive interactions. These latter provide, on top of synchronization, a unidirectional data

transfer. Another essential criterion for building the transformation rules is the respect of the equivalence to the original model where interactions' conflicts are resolved by the BIP engine. In order to satisfy this criterion, the obtained model contains a component dedicated to conflict resolution and implementing the fully centralized committee coordination algorithm presented in [10].

Formal semantics of the target implementation

In order to be able to formally present the second transformation and provide its related formal correctness proofs, we provide a formal model of the target implementation and define its operational semantics. This model is called the Time Constrained Automata (TCA) model, which is at the same abstraction level as the ΨC language. In this model, a task is an automaton. Its transitions are labelled by triplet-labels specifying release, deadline and synchronization dates. We also define the operational semantics of the provided TCA model by using the notion of labelled transition systems (LTSs).

From TT-BIP to implementation source code: model-to-code transformation

This transformation takes as input the TT-BIP model and produces as output the TCA model. The rules of this transformation aim at transforming each transition of a the original automaton, into a set of successive transitions in TCA model. Different original timing constraints are mapped using deadlines and/or release dates in TCA model. While original communications are mapped using synchronization constraints of the target model. Even if the provided rules are provided as model-to-model transformation rules, this transformation is considered as model-to-code one since the provided TCA model is considered to be at the same level of abstraction as the ΨC language.

Organization of the thesis

This document is composed of two main parts. In the first part, we present the prerequisites of this thesis (Chapter 1 and Chapter 2). In the second part, we present the existing related work and the contribution of the thesis (Chapter 3, Chapter 4, Chapter 5 and Chapter 6). The last chapter (Chapter 6.4) draws the conclusion and future work. The details of all chapters are as follows:

- Chapter 1 introduces the BIP component-based framework. It describes its abstract and concrete models as well its operational semantics.
- Chapter 2 provides necessary background information related to the TT paradigm. It lists some of existing TT implementations. And presents in details PharOS implementation which is the target implementation of the proposed methodology.
- Chapter 3 presents a non exhaustive list of existing transformational approaches that are attempting to establish a link between high-level design frameworks and implementations. This allows to situate and compare our methodology with other related existing approaches.

- Chapter 4 presents a transformational method which starts from a BIP model and a user-defined task mapping. The obtained model—called TT-BIP model—is a structural restriction of BIP model respecting the TT paradigm. First, in this chapter, we present the main challenges of the transformation. Second, we present in details the proposed solution consisting in structuring TT-BIP model under a well-defined architecture which allows the respect of the original model behavior as well as the TT principles. Then, we provide the formal transformation rules. We also provide in this chapter formal correctness proofs of the proposed transformation.

- Chapter 5 presents a method for transforming TT-BIP models into PharOS implementation. Since in the implementation level, the notion of composite process/task does not exist, we present first, in this chapter, the transformation that is applied to the TT-BIP models that are containing composite task components. Second, we propose the Time Constrained Automata (TCA) model as a formal model of TT tasks of a PharOS application and we define its operational semantics by using LTS. Then, we detail different challenges and present the formal transformation rules. Moreover, we prove that the defined transformation preserves the observational equivalence.

- In Chapter 6, we start by presenting an overview of the existing tools that are involved in the BIP framework. Second, we describe the tools developed in this thesis and implementing the methods presented in the previous chapters. Moreover, we describe the used two case study examples and some related experimental results.

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

Part I

Context

1

High-Level Component-Based Models: BIP Framework

In this chapter, we present the BIP (Behavior Interaction Priorities) framework [12, 14, 83]. BIP is a framework for rigorous design, analysis and implementation of complex real-time systems.

These latter are described in BIP as a set of atomic components, composed by a layered application of glue operators. Two glue operators are provided in BIP, namely Interaction and Priority. Interaction describes multi-party interactions between atomic components. Priority is a partial order between interactions.

BIP is thus a model-based framework that describes all software and systems according to a single semantic model. It is also a component-based framework that provides a family of operators for building composite components from basic blocks. These provided operators allow overcoming the poor expressiveness of theoretical frameworks based on a single operator, such as the product of automata. BIP framework guarantees correctness by construction which allows avoiding monolithic a posteriori verification as much as possible.

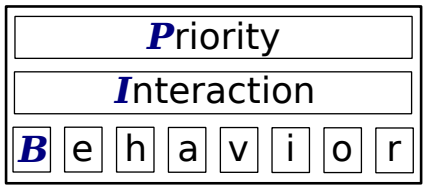


Figure 1.1: Structure of a BIP model

This chapter is structured as follows. Section 1.1 details different kinds of variables of timed systems and their related notions. It also introduces the terminology and notation used in this report. Based only on the clock variables, the abstract models of the three layers of a BIP model

are described in Section 1.2. Section 1.3 represents the concrete model of BIP, based on both data variables and clocks. Section 1.4 represents briefly how a BIP model is executed.

Chapter outline

1.1 Preliminary Notations

The state of any timed system depends on two kinds of variables —data variables and clock variables [7]. In such systems, a value of a data variable is modified explicitly in the system transitions. Values of clock variables —clocks—, are increasing implicitly as time progresses.

Timed automata introduced in [3, 4, 5, 6] are commonly considered as a standard model for real-time systems. They are providing a simple and powerful way to model the behavior of real-time systems by annotating states and/or transitions with guards over different variables of the system.

In our work, the behavior of real-time systems is modeled through timed automata where states are annotated by time progress conditions, and transitions are annotated by guards over data variables and timing constraints over clocks. In the following, we provide preliminary definitions of data variables and clocks. We provide for each variable some related notions (e.g. valuation function, guards, etc.).

1.1.1 Data Variables

Given a variable x, we denote $D(x)$ its domain, i.e. the set of all values possibly taken by x. If x is an integer variable then $D(x) = \mathbb{Z}_+$.

Valuation function

A valuation on a set of variables X is a function $v_x : X \to \bigcup_{x \in X} D(x)$, such that $v_x(x) \in D(x)$, for all $x \in X$. We denote by $\mathcal{V}(X)$ the set of all possible valuations on X.

Guards

Guards are Boolean expressions used to specify when actions of a system are enabled. Given a set of variables X, we denote by $G_X = \mathbb{B}^{\mathcal{V}(X)}$ the set of *Boolean guards* on X.

Update function

An update function $f : \mathcal{V}(X) \to \mathcal{V}(X)$ for variables X is used to assign new values $f(v_x)$ to variables in X from their current values v_x.

1.1.2 Clocks

Time progress is measured by clocks which are integer or real-valued variables increasing synchronously. Each clock can be reset (i.e., set to 0) independently of other clocks. We denote by $\mathbb{R}_+$ the set of non-negative reals, and by $\mathbb{Z}_+$ the set of non-negative integers.

Valuation function

A valuation on a set of clocks C is a function $v_c : C \to \mathbb{R}_+$ such that $v_c(c) \in \mathbb{R}_+$, for all $c \in C$. We denote by $\mathcal{V}(C)$ the set of all possible valuations on the set C, such that $\mathcal{V}(C) \subseteq \mathbb{R}_+$.

Given a subset of clocks $C' \subseteq C$ and a clock value $a \in \mathcal{V}(C)$, we denote by $v_c\,[C' \leftarrow a]$ the valuation defined as follows:

$$v_c[C' \leftarrow a](c) = \begin{cases} a & \text{if } c \in C' \\ v_c(c) & otherwise. \end{cases} \tag{1.1}$$

Timing constraints

Timing constraints are guards over the set of clocks. They are used to specify when actions of a system are enabled regarding system clocks and are defined as follows. Let C be a set of clocks. The associated set G_C of timing constraints tc is defined by the following grammar:

$$tc := True \mid False \mid c \sim a \mid tc \wedge tc \mid tc \vee tc,$$

with $c \in C$, $\sim \in \{\leq, =, \geq\}$ and $a \in \mathbb{Z}_+$.

Thus, any guard tc can be written as:

$$tc := \bigwedge_{c \in C} l_c \leq c \leq u_c, \text{ where } \forall c \in C,\, l_c \in \mathbb{Z}_+, u_c \in \mathbb{Z}_+ \cup \{+\infty\}\,. \tag{1.2}$$

The Boolean value $tc(v_c)$ is the evaluation of the timing constraint tc for the valuation v_c, where each clock c is replaced by its value $v_c(c)$. The notation $v_c + \delta$ where $\delta \in \mathbb{R}_+$ represents a new valuation v'_c defined as $v'_c(c) = v_c(c) + \delta$.

Time progress conditions

Time progress conditions are predicates on clocks used to specify how time can progress at a given state of the system. They are considered as a special case of timing constraint where $\sim$ is restricted to $\{\leq\}$ and operator $\vee$ is disallowed. Formally, time progress conditions are defined by the following grammar:

$$tpc := True \mid False \mid c \sim a \mid tc \wedge tc, \text{ where } c \in C \text{ and } a \in \mathbb{Z}_+$$

Note that any time progress condition tpc can be written as:

$$tpc = \bigwedge_{c \in C} c \leq u_c, \text{ where } \forall c \in C,\, u_c \in \mathbb{Z}_+ \cup \{+\infty\} \tag{1.3}$$

We denote by $TPC(C)$ the set of time progress conditions defined over a set of clocks C. The evaluation of a time progress condition tpc for a valuation v_c is the Boolean value $tpc(v_c)$ obtained by replacing each clock c by its value $v_c(c)$.

1.2 BIP: the Model-based Framework

We provide a formalization of BIP framework through formalization of each layer of BIP models. Respective abstract models of behavior, interaction and priority layers are detailed in this section, by considering only the clock variables.

1.2.1 Modeling Behavior

The basic building block of a BIP abstract model is the behavior unit. A behavior is formally defined as below:

Definition 1.1 (Abstract Behavior)**.** *A behavior B is a timed automaton represented by a tuple (L, P, C, T, tpc) where:*

- *L is a finite set of control locations;*
- *P is a finite set of ports;*
- *C is a finite set of clocks;*
- *$T \subseteq L \times (P \times G_C \times 2^C) \times L$ is a finite set of transitions. A transition $\tau = (l, p, tc, r, l')$ is labelled with a port p, a Boolean guard on clocks tc and a set r of clocks to be reset;*
- *The function $tpc : L \to G_C$ assigns a* time progress condition *to each location, such that, for any $l \in L$, the constraint $tpc(l)$ is a conjunction of constraints of the form $c \leq u_c$.*

The semantics of a behavior B is a Timed Transition System (TTS) consisting of two types of transitions: action transitions and delay transitions. Action transitions correspond to labelled transitions of B. Delay transitions correspond to allowing time to progress in a given state.

A state is described in two parts: the control state (i.e. control location) and the state of the clock variables. Based on this state notion, the definition of a behavior semantics is as follows:

Definition 1.2 (Semantics of a behavior B)**.** *The semantics of a behavior $B = (L, P, C, T, tpc)$ is defined as a Labelled Transition System (LTS) $(Q, \Sigma, \to)$, where:*

- *$Q = L \times \mathcal{V}(C)$ denotes the set of states of B,*
- *$\Sigma = P \cup \mathbb{R}_+$ denotes is the set of labels (ports or time values),*
- *$\to \subset Q \times (P \cup \mathbb{R}_+) \times Q$ is the set of transitions defined as follows. Let (l, v_c) and (l', v'_c) be two states, $p \in P$ and $\delta \in \mathbb{R}_+$.*
  - ***Action transitions**: We have $(l, v_c) \xrightarrow{p} (l', v'_c)$ iff there exists a transition $\tau = (l, p, tc, r, l') \in T$, such that $tc(v_c) = True$ and $v'_c = v_c[r \leftarrow 0]$ for all $c \in r$. The execution of an action transition is timeless.*
  - ***Delay transitions**: We have $(l, v_c) \xrightarrow{\delta} (l, v_c + \delta)$ iff $\forall \delta' \in [0, \delta]$, $tpc(l)(v_c + \delta') = True$, where $(v_c + \delta)(c) \stackrel{def}{=} v_c(c) + \delta$, for all $c \in C$.*

A transition $\tau = (l, p, tc, r, l')$ can be executed from a state (l, v_c) if its timing constraint is met by the valuation v_c. The execution of τ corresponds to moving from control location l to l' while resetting clocks of r. In that case, we say that the port p is *enabled* from the state (l, v_c), and we write $(l, v_c) \xrightarrow{p}$. If p is not enabled (i.e. no transition labeled by p is possible) from that state, we say that p is disabled and we write $(l, v_c) \not\xrightarrow{p}$.

Alternatively, time can progress for a duration $\delta > 0$, if the time progress condition $tpc(l)$ stays *True*. This increases all the clock values by δ. Notice that execution of an action transitions is instantaneous; control location cannot change while time elapses.

An execution sequence of B is a sequence of transitions from different states of the system. It is alternating between action and delay transitions and it is defined as follows:

Definition 1.3. *A finite (resp. infinite) execution sequence of $B = (L, P, C, T, tpc)$ from an initial state (l_0, v_{c_0}) is a sequence that alternates actions and delay transitions:*

$$(l_i, v_{c_i}) \xrightarrow{\sigma_i} (l_{i+1}, v_{c_{i+1}})$$

, where $\sigma_i \in \Sigma$ such that $\Sigma = P \cup \mathbb{R}_+$ and $i \in [1, n]$ such that $n \in \mathbb{Z}_+$.

Example 1.1. *Figure 1.2 depicts a simple behavior sender$= (L, P, C, T, tpc)$, where $L = \{l_0, l_1\}$, $P = \{s, i\}$, $C = c$, $T = \{\tau_1 = (l_0, s, True, \{c\}, l_1), \tau_2 = (l_1, i, (c \geq l), \emptyset, l_0)\}$ and tpc is such that $tpc(l_0) = True$ and $tpc(l_1) = (c \leq u)$. By default, when a time progress condition (resp. timing constraint) is not graphically shown on a location (resp. transition), we consider True as default value.*

Let the initial state of this behavior be $(l_0, 0)$, $tpc(l_0) = True$ means that sender can wait infinitely at the location l_0. Thus, any delay transition $\delta_1 \in \mathbb{R}_+$ is possible from $(l_0, 0)$, i.e. $(l_0, 0) \xrightarrow{\delta_1} (l_0, \delta_1)$. From l_0, only one action transition is possible which is τ_1. The transition τ_1 is labeled by s and having as timing constraint True. It resets the clock c, which means that the reached state of this transition is $(l_1, 0)$.

We have $tpc(l_1) = (c \leq u)$, so sender cannot wait more than u units of time. Therefore, any delay transition labeled by δ_2 such that $\delta_2 \leq u$ is possible from $(l_1, 0)$, i.e. $(l_1, 0) \xrightarrow{\delta_2} (l_1, \delta_2)$. The unique possible action transition from the state (l_1, δ_2), is the transition τ_2 labelled by port i and having as timing constraint $(c \geq l)$. This transition is possible only if δ_2 satisfies $l \leq \delta_2$.

The following is a summary of an execution sequence of the behavior example:

$$(l_0, 0) \xrightarrow{\delta_1} (l_0, \delta_1) \xrightarrow{s} (l_1, 0) \xrightarrow{\delta_2} (l_1, \delta_2) \xrightarrow{i} (l_0, \delta_2)$$

1.2.2 Modeling Interaction Glue

Interaction is a glue operator composing behaviors. Throughout this subsection, we consider n behaviors $\{B_i\}_{i=1}^n$, where $B_i = (L_i, P_i, C_i, T_i, tpc_i)$. Their sets of ports and clocks are assumed to be disjoint, i.e. for all $i \neq j$, we have $P_i \cap P_j = \emptyset$ and $C_i \cap C_j = \emptyset$.

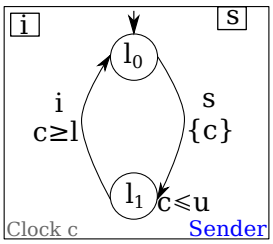


Figure 1.2: An example of Abstract Behavior

Let $P = \{P_i\}_{i=1}^n$ be the set of all ports in the composition. Interactions are defined as subsets of ports.

Definition 1.4 (Interaction Glue)**.** *An interaction between components* $\{B_i\}_{i=1}^n$ *is a non-empty subset* $\alpha \subseteq P$ *of ports, such that* $\forall i \in [1,n]$, $\mid \alpha \cap P_i \mid \leq 1$. *We denote* $\alpha = \{p_i\}_{i \in I}$, *where* $I \subseteq \{1,..,n\}$ *embodies different indexes of components participating in* α, *and* p_i *is the unique port in* $\alpha \cap P_i$.

An interaction glue operator is denoted by a set of interactions $\gamma \subseteq 2^P$. An interaction $\alpha \in \gamma$ can be enabled or disabled. The interaction α is enabled only if, for each $i \in [1,n]$, the port $\alpha \cap P_i$ is enabled in B_i. That is, α is enabled if each port that is participating in this interaction is enabled. The states of components that do not participate in the interaction remain unchanged. Alternatively, α is disabled if there exists $i \in [1,n]$ such that the port $\alpha \cap P_i$ is disabled in B_i.

We denote by $comp(\alpha)$ the set of components that have ports participating in α. $comp(\alpha)$ is formally defined as:

$$comp(\alpha) = \{B_i | i \in [1,n], P_i \cap \alpha \neq \emptyset\} \tag{1.4}$$

Two interactions are *conflicting* at a given state of the system if both are enabled, but it is not possible to execute both from that state (i.e., the execution of one of them disables the other). In fact, the enabledness of interactions only indirectly depends on the current state, through the enabledness of the participating ports. In systems having only the glue of interactions, two interactions α and α' may conflict only if they involve a shared component. In Figure 1.3a, the conflict comes from the fact that α and α' involve two ports p and q of the same component and that these two ports are labelling two transitions enabled from the same location. When reaching the location l_0, the component can execute either transition labelled by p or the one labelled by q but not both. This implies that when α and α' are enabled, only one of them should execute. Figure 1.3b shows a special case of conflict where interactions α and α' are sharing not only a common component but also a common port p.

Below, we define formally conflicting interactions.

Definition 1.5 (Conflicting interactions)**.** *Let* γ *be a set of interactions and* $\{B_i\}_{i=1}^n$ *be a set of BIP behaviors. We say that two interactions* α *and* α' *of* γ *are conflicting, iff,*

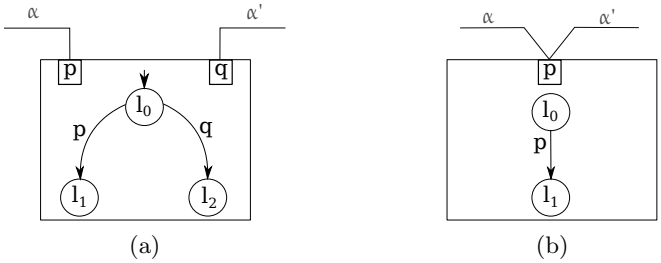


Figure 1.3: Conflicting interactions

there exists an atomic component $B_i \in comp(\alpha) \cap comp(\alpha')$ that has two transitions τ_α and $\tau_{\alpha'}$ having the same source location and labeled respectively by ports p and q such that $p \in \alpha$ and $q \in \alpha'$. We denote the conflict between α and α' by $\alpha \# \alpha'$. If α and α' are not conflicting we say that they are independent. The system is conflict-free if all interactions are pairwise-independent.

1.2.3 Modeling Priority Glue

Several different interactions can be enabled at the same time, thus leading to a certain degree of non-determinism in the product behaviour. This can be avoided by controlling the scheduling of interactions. Priority rules allow choosing one interaction among interactions enabled at a given state. They are expressed as a partial order on the interactions and are formally defined as follows:

Definition 1.6 (Priority Glue)**.** *Given a set γ of interactions defined over a set of components $\{B_i\}_{i=1}^n$, we define a priority as a relation $\pi \subseteq \gamma \times \gamma$. We write $\alpha \pi \alpha'$ for $(\alpha, \alpha') \in \pi$ to state that α has less priority than α'.*

Remark 1.1. *Notice that Definition 1.6 defines static priorities. It could be extended to dynamic priority rules depending on the state of the composition of components (cf. [14] and [77]). In this thesis, we focus only on static priorities.*

1.2.4 Composition of Abstract Models

Given a set of behaviors $\{B_i\}_{i=1}^n$ and a glue GL, the corresponding composite component is denoted by $GL(\{B_i\}_{i=1}^n)$. The glue GL is either limited to interactions (i.e. $GL = \gamma$) or it corresponds to interactions subject to priorities (i.e. $GL = \pi\gamma$). In the following, we define the semantics for both cases.

Definition 1.7 (Semantics of composition with interaction model γ)**.** *Let γ be a set of interactions. We denote by $B \stackrel{def}{=} \gamma(B_1, ..., B_n)$ the composite component obtained by applying γ to the set of behaviors $\{B_i\}_{i=1}^n$ where $B_i = (L_i, P_i, C_i, T_i, tpc_i)$ with semantics $S_{B_i} = (Q_i, \Sigma_i, \rightarrow_i)$. The semantics of B is the transition system $S_\gamma = (Q, \Sigma, \rightarrow_\gamma)$ where:*

- $Q = L \times \mathcal{V}(C)$, *where* $L = L_1 \times ... \times L_n$ *is the set of global locations and* $C = \cup_{i=1}^{n} C_i$ *is the global set of clocks. A global state is of the form* $q = (l, v_c)$. $l = (l_1, ..., l_n)$ *is a global location such that* $l_i \in L_i$ *for all* $i \in [1, n]$. *And* $v_c = (v_{c_1}, ..., v_{c_n})$ *is a global clocks valuation, where* v_{c_i} *is the valuation of clock* C_i *for all* $i \in [1, n]$.

- $\Sigma = \gamma \cup \mathbb{R}_+$,

- $\rightarrow_\gamma$ *is the set of labelled transitions satisfying the following rules:*

  - *Action transitions:*

$$\textsc{Interaction}\ \frac{\alpha = \{p_i\}_{i \in I} \in \gamma \qquad \forall i \in I, q_i = (l_i, v_{c_i}) \xrightarrow{p_i}_i q_i' = (l_i', v_{c_i}') \qquad \forall j \notin I q_j' = q_j}{(l, v_c) \xrightarrow{\alpha}_\gamma (l', v_c')}$$

  - *Delay transitions:*

$$\textsc{Delays}\ \frac{\delta \in \mathbb{R}_+ \qquad l = (l_1, ...l_n) \qquad \forall i \in [1, n], tpc(l_i)(c_{c_i} + \delta)}{(l, v_c) \xrightarrow{\delta}_\gamma (l, v_c + \delta)}$$

In Definition 1.7, action transitions correspond to the execution of interactions. An interaction $\alpha = \{p_i\}_{i \in I} \in \gamma$ is executed from a global state (l, v_c), where $l = (l_1, ..., l_n)$ and $v_c = (v_{c_1}, ..., v_{c_n})$, if for each $i \in I$ the port p_i is enabled from the local state (l_i, v_{c_i}) of the component B_i.

From a global state (l, v_c), a delay transition is executed letting time progress by δ, if it is allowed by respective time progress conditions tpc_i of each location l_i for all $i \in [1, n]$.

Definition 1.8 (Semantics of composition with Interactions γ subject to Priority π)**.** *Let* π *be a set of priority rules and* γ *be a set of interactions. We denote by* $B \stackrel{def}{=} \pi\gamma(B_1, ..., B_n)$ *the composite component obtained by applying the glues* π *and* γ *to the set of behaviors* $\{B_i\}_{i=1}^{n}$. *We define the semantics of* B *as the transition system* $S_\pi = (Q, \Sigma, \rightarrow_\pi)$ *where* $\rightarrow_\pi$ *is a restriction of* $\rightarrow_\gamma$ *defined as follows:*

$$\textsc{Priority}\ \frac{(l, v_c) \xrightarrow{\alpha}_\gamma (l, v_c') \qquad \forall \alpha' \in \gamma, \alpha\pi\alpha' \Rightarrow (l, v_c) \not\xrightarrow{\alpha'}_\gamma}{(l, v_c) \xrightarrow{\alpha}_\pi (l', v_c')}$$

In Definition 1.8, an interaction $\alpha \in \gamma$ is executed from a global state (l, v_c) if it is enabled at that state, i.e. $(l, v_c) \xrightarrow{\alpha}_\gamma (l', v_c')$ and each interaction α' having a higher priority than α (i.e. $\alpha\pi\alpha'$) is not enabled at state (l, v_c), i.e. $(l, v_c) \not\xrightarrow{\alpha'}_\gamma$.

Example 1.2. *Figure 1.4 depicts an example of an abstract model composing four behaviours and denoted* $\pi\gamma(sender_1, receiver_1, receiver_2, sender_2)$. *Behavior* $sender_1$ *(resp.* $sender_2$*) is an instance of the behavior of Figure 1.2 with* $u = 20$ *and* $l = 5$ *(resp.* $l = 6$*).*

The interaction α (resp. α') is a ternary interaction synchronizing ports r_1 and r_2 with port s_1 (resp. port s_2). By the Definition 1.5, these two interactions are conflicting since they are involving the same ports (r_1 and r_2) (same case as in Figure 1.3a). Non-determinism introduced by this conflict, is avoided by the priority π, which states that at each state of the system, the interaction α has less priority than the interaction α'

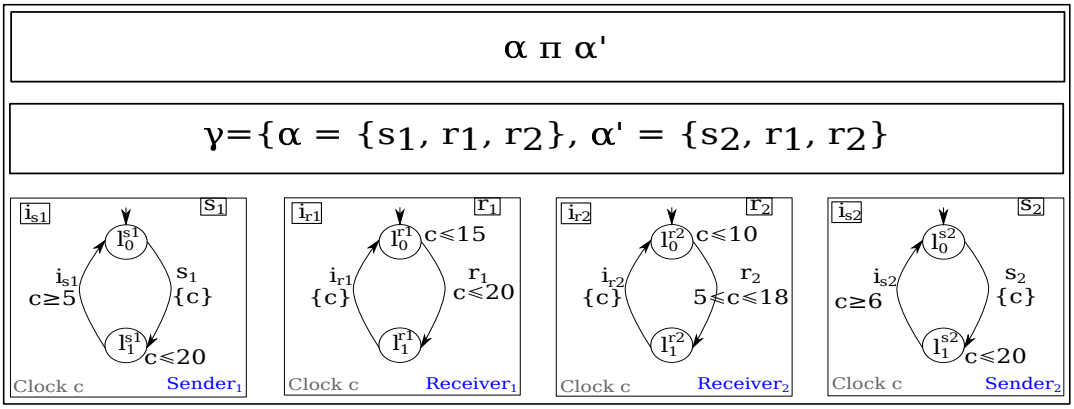


Figure 1.4: Example of abstract composition of two sender behaviors and two receiver behaviors

The execution of interactions in BIP framework is guaranteed by a sequential engine. This latter computes from the states of single components, the set of enabled interactions, applies priority rules and choose an interaction to execute.

Note that BIP framework does not compute the automaton resulting from the composition before execution. But for a better understanding of the composition glue notion, we provide in Figure 1.5, the resulting automaton after composing the components of the example of Figure 1.4 with the interactions α and α'. Note that both transitions α and α' in the obtained automation are having the same timing constraint (i.e. $5 \leq c \leq 18$). Thus, when applying the priority rule $\alpha\pi\alpha'$, α' can never be executed since whenever it is enabled there is a higher priority interaction that is enabled in the same time. Therefore we can chose, in this specific example, not to present it in the resulting global automaton (cf. Figure 1.6). Note that in both Figure 1.5 and Figure 1.6, we choose to graphically duplicate the initial location (l_0^{s1} l_0^{r1} l_0^{r2} l_0^{s2}) in order to simplify the representation of the automata.

We recall that examples provided in Figure 1.5 and Figure 1.6, are provided only to clarify the notion of glues. BIP framework does not compute these automata before execution.

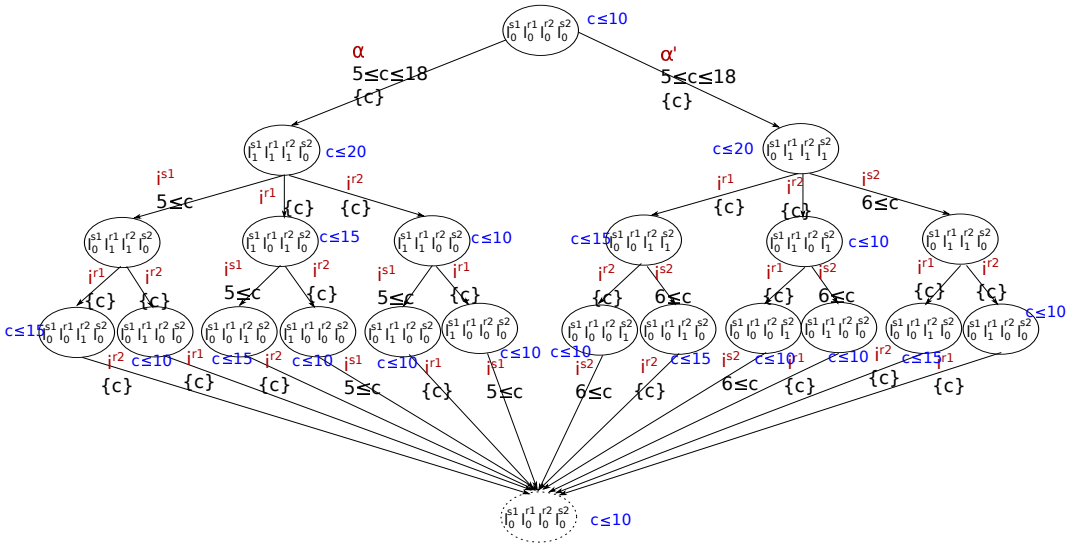


Figure 1.5: The resulting automaton of the composition with interactions of the example of Figure 1.4

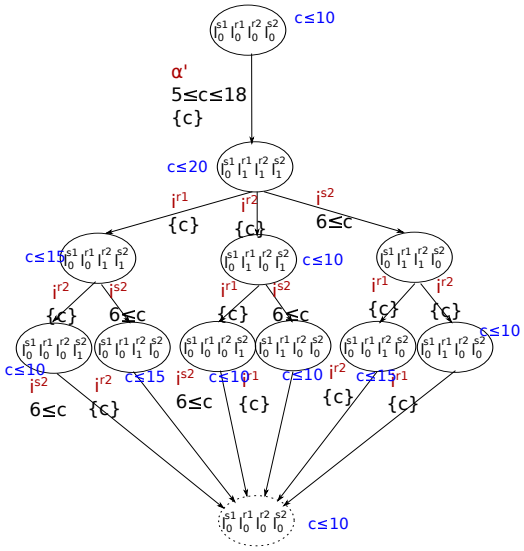


Figure 1.6: The resulting automaton of the composition with priority of the example of Figure 1.4

1.3 BIP: The Component-Based Framework

For each abstract model of the BIP layer (cf. Section 1.2), we provide its concrete model. An abstract model focuses on control while concrete model handles data variables added on top of the abstract model. Handling data variables provides a detailed representation

of complex behavior, for example, by using guards over variables in order to prevent/allow execution of transitions and interactions.

In concrete model, the behavior layer is modeled with atomic components, the interaction layer is modeled with connectors, and finally, Priorities is a mechanism for scheduling interactions.

1.3.1 Ports and Interfaces

Ports are particular names defining communication interfaces for components. They are used to establish interactions between components by using connectors. In BIP, we assume that every port has an associated distinct set of data variables. This set of variables is used to exchange data with other components when interactions take place. A set of ports is called an interface.

Definition 1.9 (Port)**.** *A port p is defined by:*

- *p : The port identifier;*
- *X_p : The set of data variables associated with p.*

Remark 1.2. *A port can be made invisible to other components, and thus label only internal computational transitions. In that case, it is called* internal *port. Symmetrically, ports visible to other components and composing the communication interface of the component are* exported *ports. We may denote* exported *ports in the remainder of this work simply by* "ports".

1.3.2 Atomic Components

An Atomic component is a concrete unit of behavior consisting in the combination of an interface (i.e. a set of ports) and a behavior encapsulated as a timed automaton extended with data and clock variables. Each transition of the automaton is guarded by a predicate on variables and a predicate on clocks, it triggers an update function, resets a subset of clocks and is labelled by a port belonging to the interface. An atomic component is formally defined as follows:

Definition 1.10 (Atomic component)**.** *An atomic component B is defined by $B = (L, P, X, C, T, tpc)$ where:*

- *L is a finite set of locations;*
- *P is a finite set of ports;*
- *X is a finite set of local variables;*
- *C is a finite set of clocks;*

- $T \subseteq L \times (P \times G_X \times G_C \times 2^C \times \mathcal{V}(X)^{\mathcal{V}(X)}) \times L$ *is a finite set of transitions, each labelled with a port, two Boolean guards (on variables and on clocks), a set of clocks to be reset and a function updating a subset of variables of* X*; the function* $tpc : L \to G_C$ *assigns a* time progress condition *to each location, such that, for any* $l \in L$*, the constraint* $tpc(l)$ *is a conjunction of constraints of the form* $c \leq u_c$*.*

Example 1.3. *Figure 1.7 shows a concrete atomic component of the behavior of Figure 1.2. This latter has been extended with the variable* x *associated with the exported port* s*. Before being sent, this variable is modified locally by the transition labeled by the internal port* i*, which executes the update function* f*.*

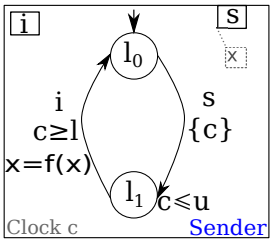


Figure 1.7: An example of an Atomic Component

Defining the operational semantics of an atomic component requires a notion of state. The state of an atomic component is described in three parts: the control state (i.e. control location), the state of the clock variables and the state of the data variables.

Definition 1.11 (Semantics of an atomic component)**.** *The semantics of an atomic component* $B = (L, P, X, C, T, tpc)$ *is defined as a Labelled Transition System (LTS)* $(Q, \Sigma, \to)$*, where:*

- $Q = L \times \mathcal{V}(C) \times \mathcal{V}(X)$ *denotes the set of states of* B*,*
- $\Sigma = P \cup \mathbb{R}_+$ *denotes is the set of labels (ports or time values),*
- $\to \subset Q \times (P \cup \mathbb{R}_+) \times Q$ *is the set of transitions defined as follows. Let* (l, v_c, v_x) *and* (l', v'_c, v'_x) *be two states,* $p \in P$ *and* $\delta \in \mathbb{R}_+$*.*
  - ***Action transitions****: We have* $(l, v_c, v_x) \xrightarrow{p} (l', v'_c, v'_x)$ *iff there exists a transition* $\tau = (l, p, g_X, tc, r, f, l') \in T$*, such that* $tc(v_c) = g_X(v_x) = True$*,* $v'_x = f(v_x)$ *and* $v'_c = v_c[r \leftarrow 0]$ *for all* $c \in r$ *(i.e.* $v'_c(c) = 0$*, for all* $c \in r$*).*
  - ***Delay transitions****: We have* $(l, v_c, v_x) \xrightarrow{\delta} (l, v_c + \delta, v_x)$ *iff* $\forall \delta' \in [0, \delta]$*,* $tpc(l)(v_c + \delta') = True$*, where* $(v_c + \delta)(c) \stackrel{def}{=} v_c(c) + \delta$*, for all* $c \in C$*.*

A component B can execute a transition $\tau = (l, p, g_X, tc, r, f, l')$ from a state (l, v_c, v_x) if its timing constraint is met by the valuation v_c. The execution of τ corresponds to moving from control location l to l', updating variables and resetting clocks of r.

Alternatively, it can wait for a duration $\delta > 0$, if the time progress condition $tpc(l)$ stays *True*. This increases all the clock values by δ. Notice that execution of jump transitions is instantaneous; control location cannot change while time elapses.

1.3.3 Interactions and Connectors

The definition of interactions is extended in the concrete model in order to handle variables. An interaction is mainly a set of ports exporting each a set of variables. An interaction can access all variables exported by its ports. Particularly, it is guarded by a predicate defined on these variables. This predicate, if evaluated to *True*, enables the interaction. This latter also defines a data transfer function which modifies the variables values upon the execution of the interaction.

Remark 1.3. *The definition of conflicting interactions in the concrete model is the same as in the abstract model (cf. Definition 1.5 and Figure 1.3). Note that, when considering data variables, this definition can be an over approximation in some cases. For example when guards of interactions satisfying the Definition 1.5 are always mutually exclusive, these interactions are not really conflicting (i.e. they are never enabled simultaneously).*

Throughout this subsection, we consider n atomic components $\{B_i\}_{i=1}^n$, where $B_i = (L_i, P_i, X_i, C_i, T_i, tpc_i)$. Their sets of locations, ports, clocks and data variables are assumed to be disjoint, i.e. for all $i \neq j$, we have $L_i \cap L_j = \emptyset$, $P_i \cap P_j = \emptyset$, $C_i \cap C_j = \emptyset$ and $X_i \cap X_j = \emptyset$. Let $P = \{P_i\}_{i=1}^n$ be the set of all ports in the composition. Interactions are defined as subsets of ports.

Definition 1.12 (Interaction)**.** *An interaction α between components $\{B_i\}_{i=1}^n$ is a triplet $(P_\alpha, G_\alpha, F_\alpha)$, where:*

- *P_α is a set of ports such that $\mid P_\alpha \cap P_i \mid \leqslant 1$, for all $i \in [1, n]$,*
- *G_α is the set of boolean guards associated to α and defined over a subset of $\cup_{p \in P_\alpha} X_p$.*
- *F_α is the set of the update functions associated to α and defined over $\cup_{p \in P_\alpha} X_p$.*

In the remainder of this report, when no confusion is possible from the context, we may simply denote the port set of the interaction by the interaction name. Thus we may use $p \in \alpha$ instead of $p \in P_\alpha$ and $p \in \{\alpha_1, \alpha_2, \alpha_n\}$ instead of $p \in P_{\alpha_i}$, $\alpha_i \in \{\alpha_1, \alpha_2, \alpha_n\}$.

As in the abstract model, interactions are representing the first layer of glue. In order to avoid an explicit enumeration of all possible interactions between a given set of components, the notion of connector has been introduced. It allows to present sets of related interactions in a compact way. Each connector Γ is defined over a set of ports P_Γ and defines a set of interactions γ_Γ, i.e. a subset of 2^{P_Γ}. A connector can be *atomic* or *hierarchical*. An *atomic* connector (or simply called connector) can export a port that is

used for the construction of hierarchical connectors. A *hierarchical* connector is obtained by combining *atomic* connectors to form a structure acting as a single connector. The ports of the top level connector include the exported port of the low level connector.

An algebraic formalisation of BIP connectors is provided in [19] and [20]. In this thesis, we settle for the following generic definition of a connector. For hierarchical connectors, we only provide some intuitive but representative examples.

Definition 1.13 (Connector)**.** *A connector* Γ *is defined by the triplet* $(P_\Gamma, \gamma_\Gamma, p_\Gamma)$*, where:*

- P_Γ *is the set of ports of* Γ*, i.e. the set of ports of components synchronized by* Γ*,*
- γ_Γ *is the set of interactions,*
- p_Γ *is the exported port by the connector* Γ*.*

For a connector Γ, the set of feasible interactions γ_Γ depends on types of ports of P_Γ. Two types of these latter are available: *trigger* and *synchron* ports. A *trigger* —represented graphically by a triangle—is an active port that can initiate an interaction without synchronizing with other ports. A *synchron* —represented graphically by a circle—is a passive port that needs synchronization with other ports.

A feasible interaction of a connector is a subset of its ports such that either it contains some trigger, or it is maximal, i.e., consisting of all the synchron ports. Thus, by construction, if more than one interaction is possible, then the maximal interaction (i.e. the interaction having the maximal number of ports) is prioritized. Figure 1.8 shows an example of three connectors and their feasible sets of interactions denoted by γ. In Figure 1.8a, the connector consists of three synchron ports p, q and r. The only feasible interaction in this connector is pqr. In Figure 1.8b, the port p is a trigger and can occur alone, even if q and r are not possible. Nevertheless, the occurrence of q and r requires the occurrence of p. Thus, the feasible interactions are p, pq, pr and pqr. In Figure 1.8c, both ports p and q ere trigger ports. Thus, the interactions p and q can occur alone or synchronize with each other through the interaction pq.

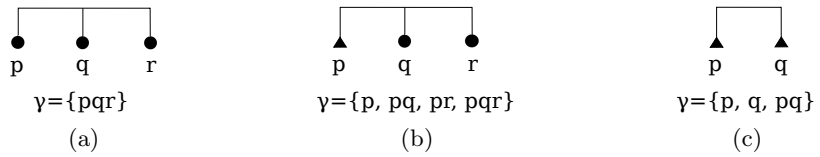


Figure 1.8: Connectors and their feasible interactions

As explained before, types of ports are defined , in order to specify the feasible interactions of a connector. In addition to ports types, connectors sometimes need to be structured, i.e. specifying types associated to groups of ports instead of just one port. This is needed to represent some interactions. Different coordination schemes are depicted in examples of Figure 1.9.

Example 1.4. *Figure 1.9 shows four connectors defined on the same set of ports s, r_1, r_2 and r_3. Each connector shows a different coordination scheme; Figure 1.9a and Figure 1.9b for atomic connectors and Figure 1.9c and Figure 1.9d for hierarchical connectors. Let the port s be the port of a sender component and ports r_i, $i \in \{1, 2, 3\}$ be ports of receiver components, the different synchronization models are the followings:*

- *Rendezvous (cf. Figure 1.9a): Since all ports are synchrons, this synchronization is specified by a single interaction involving all ports. That is, this interaction occurs only if all ports are enabled in their respective components. It means strong synchronization between port s and ports r_i.*
- *Broadcast (cf. Figure 1.9b): It includes one trigger port s and three synchron ports r_i. A trigger port initiates the interaction, independently of the enabledness of other ports. For this reason, this scheme is also called weak synchronization, that is a synchronization involving one trigger port and a (possibly empty) set of synchron ports. This is specified by the set of all interactions containing s, i.e. interactions s, sr_1, sr_2, sr_3, sr_1r_2, sr_1r_1, $sr2_r3$ and $sr_1r_2r_3$.*
- *Atomic broadcast (cf. Figure 1.9c): The bottom connector based on ports r_i is a Rendezvous exporting a synchron port t_1. This connector allows only the maximal interaction $r_1r_2r_3$. The top connector, is a broadcast defined on ports s and t_1, allowing thus interactions s and st_1. Therefore, this hierarchical connector allows interactions s and $sr_1r_2r_3$, which means that either a message is received by all r_i, or by none.*
- *Causal chain (cf. Figure 1.9d): The bottom, intermediate and top connectors are Broadcast connectors. Therefore, this hierarchical connector allows interactions s, sr_1, sr_1r_2 and $sr_1r_2r_3$. That is, for a message to be received by r_i, it has to be received at the same time by all r_j such that $j < i$.*

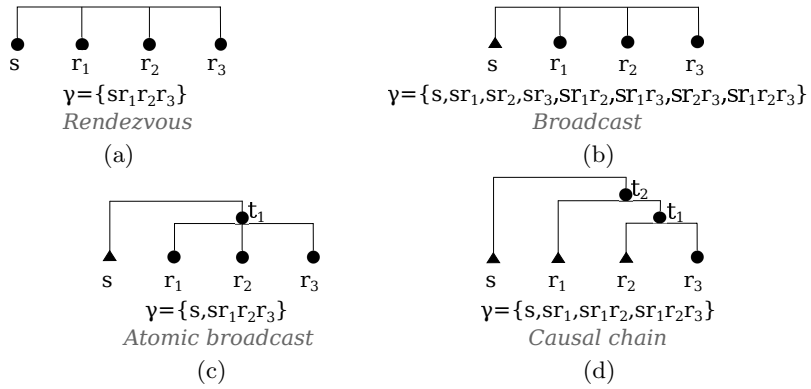


Figure 1.9: Connectors and different coordination schemes

In Definition 1.12, an interaction consists of one or more ports of the connector, a guard on variables associated with these ports and a data transfer function. Connectors provide a mechanism for handling this transfer function. Actually, instead of considering a single data transfer function, this mechanism implies two phases; an upward U and a downward D actions. The upward action —after deciding whether the guard is *True*— updates the connector local variables based on values of variables of ports. The downward action computes the values to return in variables of components ports from the values of connector local variables. This mechanism also allows data transfer in hierarchical connectors.

Example 1.5. *Figure 1.10 shows an atomic (a) and a hierarchical (b) connectors defined on the same set of ports p, q and r exporting respectively variables x, y and z. Both ports allow to compute the maximal value of these variables and return it to the rest of ports. For each connector, a guard G, upward U and downward D transfer functions are displayed. The connector of Figure 1.10a needs all ports variables to be positive before executing interactions. It defines a local variable t. The function U computes the max of variables x, y and z and stores it in variable t. The function D stores back value of t in variables x, y and z.*

In Figure 1.10b, the bottom (resp. top) connector defines a guard G_1 (resp. G_2) stating that the interaction will not be executed until variables y and z (resp. t_1) be positive. It defines a local variable t_1 (resp. t_2), which stores after executing function U_1 (resp. U_2) the maximal value between those of variables y and z (resp. x and t_1). Function D_1 (resp. D_2) stores back value of variable t_1 (resp. t_2) in variables y and z (resp. x and t_1).

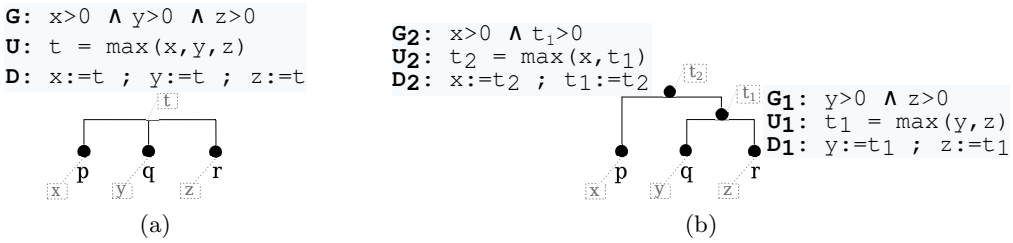


Figure 1.10: An atomic (a) and a hierarchical (b) connectors computing the maximum of exported values

Remark 1.4. *In [27] and [47], authors show that a hierarchical connector can be replaced by an equivalent set of atomic connectors defining interactions as in Definition 1.12. This is established by composing guards of bottom, intermediate and top connectors, in order to obtain a guard for the interaction. The update function is obtained by composing different upward and downward actions. This transformation has been implemented and allows easily to transform a BIP model with hierarchical connectors into a model with only atomic flat connectors. Therefore, in this thesis, we do not consider hierarchical*

connectors. And we assume that all our input models have had their potential hierarchical connectors flattened using this transformation.

Remark 1.5. *Notice also that, intuitively, a connector with trigger ports can be replaced by an equivalent set of connectors defined only on synchron ports. For example, consider the connector of Figure 1.8b defining interactions p, pq, pr and pqr. The set of connectors of Figure 1.11, i.e. the unary connector on p, the binary connectors on p and q and on p and r and the ternary connector on p, q and r are defining the same set of interactions as the connector of Figure 1.8b.*

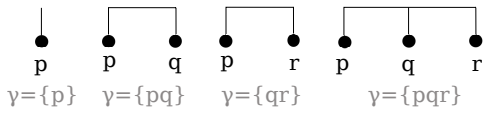


Figure 1.11: Set of connectors based only on synchron ports and equivalent to connector of Figure 1.8b

Taking into account this remark, we consider in this thesis only synchron ports.

Remark 1.6. *The BIP semantics presented in this section assume atomic execution of interactions which provides sequential execution of the system.*

1.3.4 Priority

Priorities assign a partial order between interactions, in order to reduce non-determinism in the system. As mentioned in Section 1.2, we consider static priorities in this thesis. These priorities do not depend on the state of the system including data variables values. Therefore, Definition 1.6 remains available for concrete BIP models.

1.3.5 Composition of Concrete Models

Similarly to the abstract model composition, we denote the composition of atomic components $\{B_i\}_{i=1}^n$ by using the glue GL by $GL(\{B_i\}_{i=1}^n)$. The glue GL is either limited to interactions (i.e. $GL = \gamma$) or it corresponds to interactions subject to priorities (i.e. $GL = \pi\gamma$). Below, we define the semantics of the two models.

Definition 1.14 (Semantics of composition with interaction model γ)**.** *Let γ be a set of interactions and let $\{B_i\}_{i=1}^n$ where $B_i = (L_i, P_i, X_i, C_i, T_i, tpc_i)$ be a set of atomic components. The semantics of the composite component $B = \gamma(B_1, ..., B_n)$ is the transition system $S_\gamma = (Q, \Sigma, \rightarrow_\gamma)$ where:*

- *$Q = L \times \mathcal{V}(C) \times \mathcal{V}(X)$, where $L = L_1 \times ... \times L_n$ is the set of global locations, $C = \cup_{i=1}^n C_i$ is the global set of clocks and $X = \cup_{i=1}^n X_i$ is the global set of variables. A state $q \in Q$ is of the form (l, v_c, v_x) such that $l = (l_1, ..., l_n)$ is the global location, $v_c = (v_{c_1}, ..., v_{c_n})$ is a global clocks valuation and $v_x = (v_{x_i}, ..., v_{x_n})$ is a global data variables valuation.*

- $\Sigma = \gamma \cup \mathbb{R}_+$ *corresponds to the set of labels,*
- $\rightarrow_\gamma$ *is the set of labelled transitions satisfying the following rules:*
  - *Action transitions:*

$$\textsc{Inter}\ \frac{\begin{array}{c}\alpha = (\{p_i\}_{i\in I}, G_\alpha, F_\alpha) \in \gamma \qquad G_\alpha(\{v_{x_i}\}_{i\in I}) \\ \forall i \in I, (l_i, v_{c_i}, v_{x_i}) \xrightarrow{p_i} i \qquad (\{v^*_{x_i}\}_{i\in I}) = F_\alpha(\{v_{x_i}\}_{i\in I}) \\ \forall i \in I, (l_i, v_{c_i}, v^*_{x_i}) \xrightarrow{p_i}_i (l'_i, v'_{c_i}, v'_{x_i}) \qquad \forall i \notin I, (l_i, v_{c_i}, v_{x_i}) = (l'_i, v'_{c_i}, v'_{x_i})\end{array}}{(l, v_c, v_x) \xrightarrow{\alpha}_\gamma (l', v'_c, v'_x)},$$

  - *Delays transitions:*

$$\textsc{Delays}\ \frac{\delta \in \mathbb{R}_+ \qquad l = (l_1, ...l_n) \qquad \forall i[1,n], tpc(l_i)(t+\delta)}{(l, v_c, v_x) \xrightarrow{\delta}_\gamma (l, v_c + \delta, v_x)}$$

The first inference rule of Definition 1.14 specifies that a composite component $B = \gamma(B_1, ..., B_n)$ can execute an interaction $\alpha = (\{p_i\}_{i\in I}, G_\alpha, F_\alpha)$ from a global state $q = (l, v_c, v_x)$ only if (1) each port p_i is enabled in its corresponding component B_i, i.e. $q_i = (l_i, v_{c_i}, v_{x_i}) \xrightarrow{p_i}$, where q_i is the projection of the state q on the component B_i, and (2) the guard G_α defined over variables exported by ports $\{p_i\}_{i\in I}$ is evaluated to *True*. The function F is triggered by the execution of α. It modifies the variables $\{v_{x_i}\}_{i\in I}$ exported by ports $\{p_i\}_{i\in I}$. Obtained new values $\{v^*_{x_i}\}_{i\in I}$ are then processed by their respective components' transitions, which in turn can apply transformations to obtain values $\{v'_{x_i}\}_{i\in I}$. The clock valuation v'_c takes into account clocks that have been reseted by their respective components' transitions. States of components which are not participating in the interaction α remain unchanged.

The second inference rule of Definition 1.14 states that B can execute a delay transition δ from a state $q = (l, v_c, v_x)$, only if respective time progress conditions $\{tpc_i\}_{i\in I}$ of each participating component B_i are evaluated to *True*.

Definition 1.15 (Semantics of composition with Interactions γ subject to Priority π)**.** *Let π be a set of priority rules and γ be a set of interactions. We denote by $B \stackrel{def}{=} \pi\gamma(B_1, ..., B_n)$ the composite component obtained by applying the glues π and γ to the set of atomic components $\{B_i\}_{i=1}^n$. We define the semantics of B as the transition system $S_\pi = (Q, \Sigma, \rightarrow_\pi)$ where $\rightarrow_\pi$ is a restriction of $\rightarrow_\gamma$ defined as follows:*

$$\textsc{Priority}\ \frac{(l, v_c, v_x) \xrightarrow{\alpha}_\gamma (l, v'_c, v'_x) \qquad \forall \alpha' \in \gamma, \alpha\pi\alpha' \Rightarrow (l, v_c, v_x) \not\xrightarrow{\alpha'}_\gamma}{(l, v_c, v_x) \xrightarrow{\alpha}_\pi (l', v'_c, v'_x)}$$

The application of priority π filters out the interactions which are not maximal with respect to the priority order. The inference rule of Definition 1.15 specifies that an interaction $\alpha = (\{p_i\}_{i\in I}, G_\alpha, F_\alpha)$ is executed from a state $q = (l, v_c, v_x)$ only if any other interaction α' having a higher priority is disabled from that state.

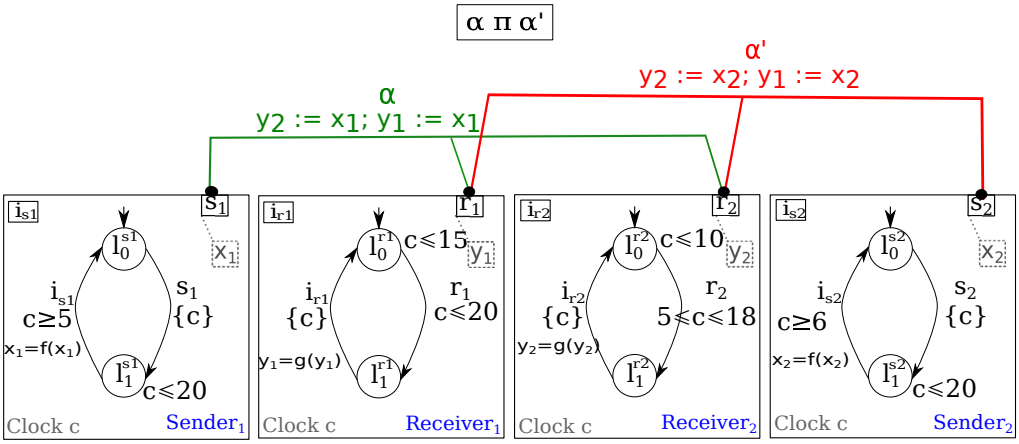


Figure 1.12: Example of concrete composition of two sender components and two receiver components

Example 1.6. *Figure 1.12 depicts a composition of four atomic components* $sender_1$, $receiver_1$, $receiver_2$ *and* $sender_2$. *It extends model of Figure 1.4 with data variables. components* $sender_i$, $i \in \{1, 2\}$ *which have variables* x_i *associated to ports* s_i. *These variables are updated locally by function* f *executed by the occurrence of transitions labeled by the internal ports. Components* $receiver_i$ *define variables* y_i *associated to ports* r_i *and updated by function* g.

Interaction α *(resp.* α'*), transmits the value of variable* x_1 *of component* $sender_1$ *(resp. variable* x_2 *of component* $sender_2$*) to components* $receiver_i$ *that stores it in variables* y_i.

Priority rule π *states that the interaction* α *has less priority than the interaction* α', *when both are possible. A component* $receiver_i$ *can receive a new value through its port* r_i *either from component* $sender_1$ *or component* $sender_2$.

1.4 BIP Execution Platform

The operational semantics is implemented directly by the BIP execution engine. It plays the role of the co-ordinator in selecting and sequentially executing interactions between components with respect to the glue specified in the input component model. It computes the enabled interactions by enumerating over the complete list of interactions in the model.

During the execution, on each iteration of the engine, the enabled interactions are selected from the complete list of interactions, based on the current state of the atomic components. Then, between the enabled interactions, priority rules are applied to eliminate the ones with low priority.

1.5 Conclusion

In this chapter, we have presented the BIP framework, a component-based framework for modeling heterogeneous systems. A BIP model is built by the superposition of three layers. The lower layer describes the behavior of a component as a timed automaton. The intermediate layer is composed of a set of multi-party interactions synchronizing transitions of the Behavior layer. The upper layer describes the priorities characterizing a set of scheduling policies for interactions to reduce non-determinism. Such technique of layering offers a clear separation between components behaviors and the structure of the system (interactions and priorities).

The component-based approach aims at dealing with the complexity of systems. It allows building a complex system by assembling basic blocks (atomic components) in an incremental way. It thus provides important characteristics for system construction such as reuse, incrementality, compositionality, etc. Besides the reuse of components, BIP allows the reuse of known properties of constituent components.

BIP affords for its models, a clean and very well-defined operational semantics based on Labelled Transition Systems (LTS). It is thus a good candidate for model transformations, aiming at preserving observational equivalence.

In the following chapters, we present a method for generating time-triggered implementations from BIP models. We will also show results after applying the proposed method to case studies.

2

Time-Triggered Approach

One of the characterizing features of hard real-time computer systems is the fact that they must provide a particular result at intended points in real-time. That is the functional specifications of such systems must be met within the specified deadlines. It follows that any real-time computer architecture or design methodology of such systems must be concerned with both issues of value and temporal correctness.

Two main design paradigms for implementing real-time systems are identified [56]; the Event-Triggered (ET) and the Time-Triggered (TT) approaches. These approaches differ in the type of the triggering mechanism of communication and processing actions.

- *In the event-triggered approach, actions are initiated whenever a significant event—other than clock interrupts—occurs. Such systems derive temporal control from the environment in an unpredictable manner. The event-triggered approach is not suitable for guaranteeing the respect of requirements of hard real-time systems such as predictability, determinism and guaranteed latencies.*
- *In time-triggered systems, temporal control is derived from the global progression of time, i.e. all actions are initiated at predetermined points in time. There is only one interrupt signal: the ticks generated by the global local periodic clock. These statically defined activation instants enforce regularity and make the TT approach well-suited for hard real-time systems —since it supports predictability and determinism.*

Since our work targets hard real-time systems, we focus, in this chapter, on the TT paradigm. We provide all necessary background information related to this approach and we cite some of existing TT implementations as well as the chosen RTOS-based implementation that we target in our work.

This chapter is structured as follows. Section 2.1 presents key features of the TT paradigm. Section 2.2 provides examples of existing tools implementing the TT paradigm. Section 2.3 focuses

on PharOS, the RTOS-based implementation based on the TT execution model.

Chapter outline

2.1 The Time-Triggered Paradigm

In [52], [53] and [55], Kopetz presents an approach for real-time system design based on the TT paradigm [39]. This latter advocates a set of design principles that support the design of highly dependable hard real-time systems:

The global notion of time:

One of the major features and requirements of TT systems is the global synchronized time. It is established by a periodic synchronized clock in order to enable a TT communication and computation.

In the case of a distributed TT system, each node of the system defines its local periodic clock. Different local clocks synchronization consists in bringing the time of clocks in a distributed network into close relation with respect to each other. The quality of clock synchronization is measured by the *precision* and *accuracy* [61]. Precision is defined as the maximum offset between any two clocks in the network. Accuracy is defined as the maximum offset between any clock and the absolute reference time. This synchronization is compulsory to establish the global time of a cluster.

TT communication system and temporal firewall

The temporal firewall [62] is a special interface for unidirectional data transfer between sender/receiver nodes over a TT communication system [39, 52]. It consists in a shared memory element. The sender memory forms the output firewall of the sender and the receiver memory forms the input firewall of the receiver. It is the responsibility of the TT communication system to transport, with access to the global time, the data from the sender's firewall to the receiver's firewall. The instants at which information is delivered or received are a priori defined in a common periodic communication schedule. This latter is known to all nodes. A sender does not send any control or data signal directly to a receiver. Furthermore, avoidance of interference between concurrent read and write operations on the memory elements is guaranteed by the protocol implemented by the TT communication system.

Figure 2.1 reproduced from [39], depicts a basic data and control transfer —from one sender to one receiver—using a temporal firewall interface.

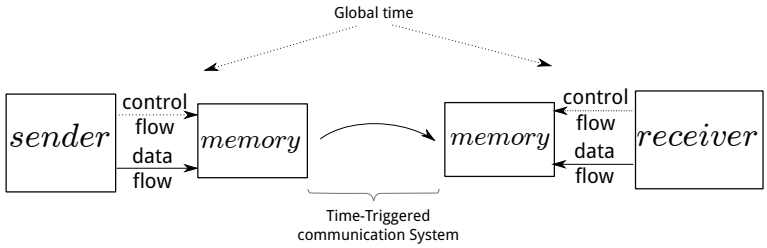


Figure 2.1: Temporal firewall (reproduced from [39])

Composability and dependability

The composability principle covers several aspects in a distributed real-time system design. First, it requires that nodes can be designed independently of each other assuming that the architecture and service have been specified precisely. Secondly, independently developed components can be integrated with minimal integration effort. And finally, if fault tolerance is implemented by the replication of nodes, then the architecture and the nodes must support replica determinism.

The dependability is an overall term that includes availability, safety, maintainability and security [63]. This principle is met only if faults are taken into account. In order to tolerate faults in a time-triggered distributed system two design approaches are supported. The first one is redundancy approach, consisting in introducing redundant components in a system. This redundancy allows to provide the intended service even is presence of faults. The second approach is recovery approach. It consists in designing the system's software which is able to detect and then recover from faults. Compared to the first approach, the recovery approach avoids instantiating extra components but needs to allow time for recovery.

2.2 Time-Triggered Implementations

The principles developed from the MARS (MAintainable Real-Time System) project [59] —ancestor of Time-Triggered Architecture (TTA) [58]—served as the basis for codification of time-triggered principles. A key concept embraced by the MARS project is called "fail-silent", which means that a node either sends the correct message or no message at all. Access to the communications bus is through a simple TDMA scheme with a static schedule.

The Time-Triggered Architecture (TTA) provides a computing infrastructure for the design and implementation of dependable distributed systems. The basic building block of the TTA is a node which consists of a processor with memory, an input-output subsystem, a TT communication controller, an operating system and the application software. Data is exchanged between different nodes using a TT protocol.

2.2.1 Time-Triggered Protocols

In a TT communication system, the sender and receiver(s) agree a priori on a cyclic time-controlled conflict-free communication schedule for the sending of time-triggered messages. This cyclic communication schedule can be expressed in the cyclic model of time, where the send and receive instants of a message, are represented by a period and phase. In every period a message is sent at exactly the same phase.

The literature embraces several protocols that integrate the TT communication. This subsection attempts to briefly outline some of these protocols. Detailed and deep comparisons between these protocols can be found in [38, 54, 84, 82].

TTP

The Time-Triggered Protocol (TTP) [60] —initially named TTP/C—is a high-speed, masterless, multicast and a dual channel 25 Mbit/s [38] field bus communication protocol for safety-critical embedded applications. It is a development from the European Brite-Euram 'X-by-wire' project integrating time-triggered communication.

The TTP communication system autonomously establishes a fault-tolerant global time reference and coordinates all communication activities based on the globally known message schedules specified at the design time. It requires that all communication participants to comply with an exactly specified and rigidly enforced temporal communication schedule that serves as a strict communication interface definition.

A TTP network is composed by a set of nodes consisting in electronic control units (ECUs), connected by two replicated physical communication channels (buses). As a result of redundant buses, TTP tolerates a single bus failure. TTP implements a time division multiple access (TDMA) scheme derived from a global notion of time that avoids collision on the bus. Every active ECU owns a TDMA slot, during which it has the full transmission capacity of the bus for this short period of time. The sequence of TDMA slots in which each ECU sends its frames forms a TDMA round.

Each TTP node consists mainly of a host subsystem and a communications subsystem (see Figure 2.2). The host runs the application software and the communications subsystem is formed by the TTP controller, which executes the TTP protocol and regulates access to the physical bus. The communications interface between the host computer and the TTP/C controller, called the communication network interface (CNI), is a dual-port memory. It acts as a temporal firewall, isolating the host from the network and not allowing any control errors to propagate. It is within the TTP controller that the MEssage Descriptor List (MEDL) resides. The MEDL contains the global static message transmission schedule. that determines when a particular message has to be sent or received. The communication subsystem contains also bus guardians, in order to guarantee that the node would not transmit data during wrong time-slots and eliminates "babbling idiot" problem.

TTE

The Time-Triggered Ethernet (TTE) [57] is an adaptation of the TTP to ethernet-based networks. It expands the protocol to support the standard event-triggered Ethernet traffic and the time-triggered safety-critical traffic. The handling of the event triggered traffic in TT Ethernet is managed with conformance to the existing Ethernet standards of the IEEE. A global synchronized time is established in order to execute a distributed time-triggered communication scheme.

TT Ethernet is intended to handle all kinds of applications; e.g. data acquisition, multimedia systems and also safety-critical real-time control systems etc. .

A TTE network consists of a set of nodes and TTE-switches, which are interconnected using bidirectional communication links (see Figure 2.3 —adapted from [57] Figure 4). TTE-Switches relay the messages and take care that time-triggered messages are not

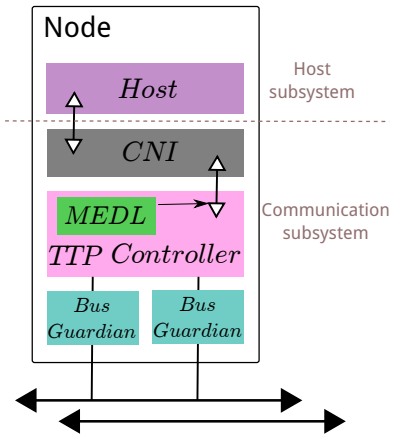


Figure 2.2: TTP Node Architecture

delayed by other messages, i.e. they prioritize all time-triggered traffic over non-time-triggered messages. In order to prevent error propagation from failed components the fault-tolerant TTEthernet network configuration deploys two independent channels for each connection.

Mainly, we distinguish between two types of TTEthernet configurations [57]: (1) standard configuration with standard Ethernet controllers, TT Ethernet controllers, and a single switch; (2) fault-tolerant configuration with a safety-critical TT Ethernet controller containing two ports to two independent switches.

Figure 2.3 —adapted from [57] Figure 4—illustrates examples of a standard and a typical safety-critical TTE network configurations.

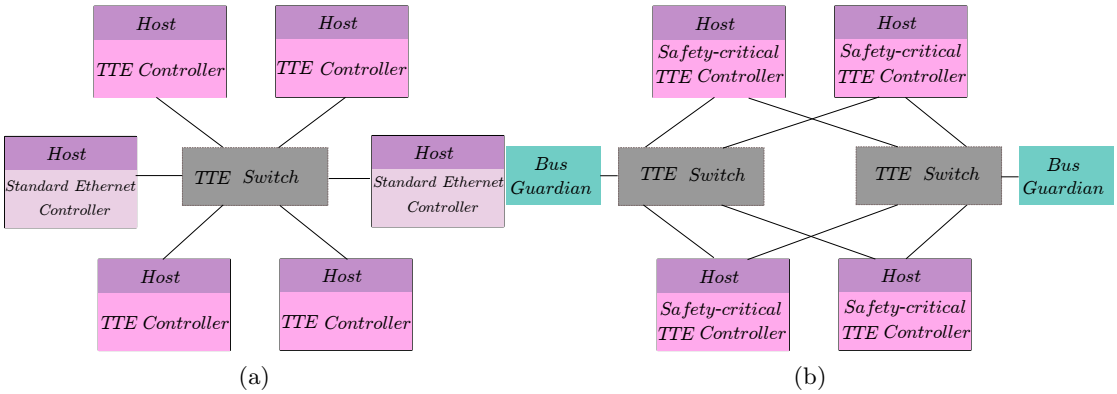


Figure 2.3: Examples of Standard TTE (left) and safety-critical TTE (right) configuration

Flexray

Flexray [1, 35] is a communication protocol for automotive applications such as X-by-wire. Flexray has been developed and is supported by a consortium of automotive manufacturers and suppliers. The FlexRay protocol consists of two parts; a time-triggered part where messages are scheduled according to an a priori defined TDMA schedule which is similar to TTP and an event-triggered part supporting sporadic traffic. FlexRay implements a global synchronized timebase that supports synchronized actions. Flexray can support a communication speed of up to 10 MBit/s.

The building block of a FlexRay network is a node. Each communication node has —similarly to a TTP node—a host with a subordinate communication controller connected through CNI interface. Depending on the network topology (active star or passive bus topologies), one or two bus drivers and bus guardians can connect different nodes of the network. The bus guardian is controlled by the communication controller, while the bus driver controls the power supply.

TTCAN

Time-Triggered Controller Area Network (TTCAN) [64] is the time-triggered extension built on top of the event-triggered CAN protocol [24]. TTCAN is introduced to guarantee a deterministic communication pattern on the communication bus

It establishes a global synchronized time derived from periodically broadcasted synchronization messages sent by a special node, called the time master node. This latter assigns the remaining nodes on the network —slave nodes —with time windows which are the only times available for nodes to transmit.

The TTCAN protocol is implemented in hardware using a dedicated TTCAN controller. The event-triggered part uses the standard CAN arbitration to avoid collisions.

2.2.2 Modelling of Time-Triggered Systems

PBO

The port-based object (PBO) [85] provides a software framework to program reconfigurable robots. A PBO system consists of a set of tasks that communicate with each other and the environment. Tasks—called PBOs—are activated by time periodically and communicate through ports via state variables that are stored in a global table. A PBO receives data from other PBOs via its input ports. It makes its results available to other PBOs through its output ports. And it interacts with the environment via its resource ports.

Each PBO stores in its own local table the needed subset of the data of the global table. Before executing a PBO, the state of the local variables corresponding to input ports are updated from the global table. Upon execution completion, the state variables corresponding to output ports are copied from its local table to the global one. Note that read and write operations are atomic.

For synchronizing access to the global state variable table and ensuring the mutual exclusivity of accesses to the same state variable, the PBO framework provides mechanisms using spin-locks [70]. This is managed outside of the objects, at the operating system level, instead of by the objects themselves.

In the PBO model, the communication between PBOs is not deterministic. In fact, for the same PBO, the execution time may be variable in two different activations of the task. Thus, the time when the outputs are produced and get updated in the global table may vary from one activation to another. Therefore when reading a variable from the global table, a consulting PBO may or may not get the results of the current cycle. Recall that the value of a variable is preserved as long as it is not overwritten, and a new value overwrites the old value even if this latter has not been used by other tasks. Figure 2.4

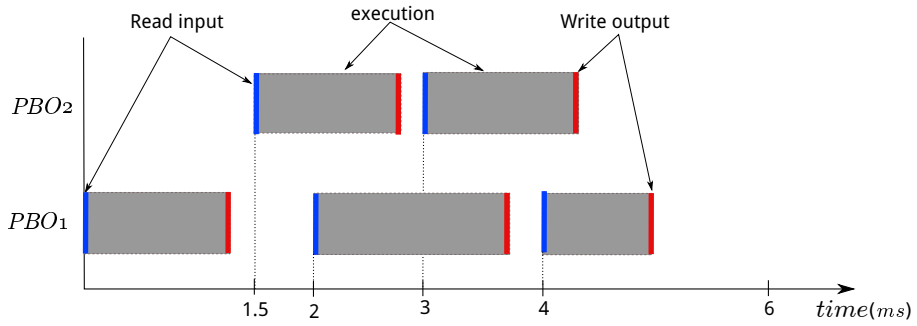


Figure 2.4: Example of two PBOs execution traces

depicts an example of execution traces of two task—denoted PBO1 and PBO2. Task PBO1 is activated every 2 ms. Its output variables are consumed by the task PBO2, which is activated every 1.5 ms. Notice that PBO2, in the first period, reads a fresh output from PBO2 (produced before 1.5 ms). But in the second period of PBO2, output of PBO1 is not yet produced. Therefore, PBO2 will read PBO1 output from the last cycle again (i.e. the value produced before 1.5 ms).

Giotto: TT language

The Giotto [44, 43] language and its associated tools are based on time-triggered execution. It extends the semantics of the TT paradigm to include the time-triggered invocation of tasks, mode switching and message passing.

The Giotto model defines a software architecture of the implementation which specifies its functionality and timing requirements and abstracts away issues related to the target specific platform such as hardware performance and scheduling mechanism.

Giotto introduced the concept of Logical Execution Time (LET) [51], which abstracts from the actual execution time of a real-time program, thereby, from both the execution platform and the communication topology. LET is motivated by the observation that the relevant behavior of real-time programs is not determined by time when programs just execute their computations, but when input is read and output is written. The

inputs of a task are read at the release instant and the newly calculated outputs are written at the termination instant. Between these, the outputs have taken the value of the previous execution. Figure 2.5 —reproduced from the literature—illustrates the LET abstraction compared to the physical execution.

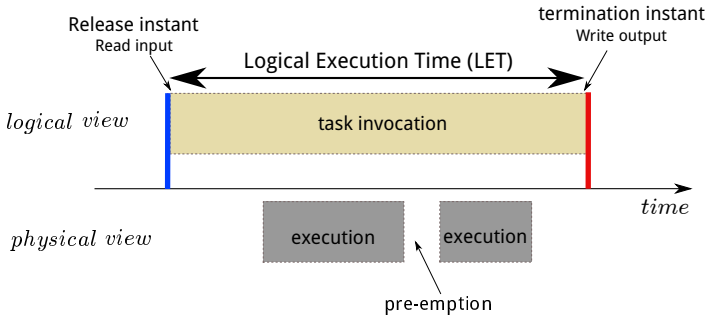


Figure 2.5: Logical Execution Time Abstraction

A programmer's Giotto model consists of:

- **Tasks**: Which are the basic functional entities, implemented by external (Java or C++) code. Tasks are expected to run periodically, with a fixed period per mode. Each task has a start time and an end time. The start time corresponds to the starting time instant when the execution period starts. The end time corresponds to the end time instance when the execution period ends. A task reads all its inputs at the start time and makes its outputs available to other tasks at its end time.

- **Ports**: Which are memory locations (typed variables) facilitating inter-task communication and carrying system state. There are three types of ports in a Giotto program: sensor ports, actuator ports, and task ports. Note that ports stand for the notion of temporal firewall of the TT paradigm.

- **Drivers**: They perform data copying between ports and implement device access (for sensors and actuators). Tasks uses drivers for communication either with other tasks or with sensors and actuators. These latter can have an associated guard condition, which can be evaluated in zero logical time as well. Note that drivers stand for the communication system of the TT paradigm.

- **Modes**: include periodic task invocations and actuator updates with their related driver calls. The transition between modes is possible if the guard condition of a mode switch driver evaluates to *True*. Tasks can be added or removed when switching between modes.

All actions in such applications are triggered by real time, namely the periodic invocation of tasks, the consulting of sensor data, the writing of actuator values, and the

switching between modes. And the communication between tasks is well defined and deterministic. It is computed from the worst case communication time, which represents an upper bound on the time required for broadcasting the value of task port over the network.

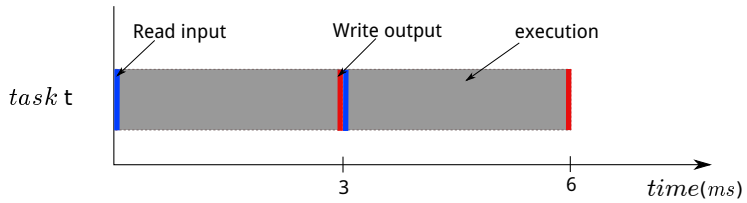


Figure 2.6: Example of a Giotto periodic task invocation

Figure 2.6, depicts an example of execution of a task t, invoked every 3 ms. The task reads inputs upon each invocation (i.e. at instants 0, 3, 6 etc.) and write its output values upon each completion.

TDL

The Timing Definition Language (TDL) [76] is a high-level description language for specifying the explicit timing requirements of an application. TDL is an extension of Giotto. It contains a few additional notions and complies with the time-triggered semantics. It differs from Giotto in using modules which are comparable to components as they support local definition of variables, constants, tasks, modes, and inputs and outputs. Similarly to Giotto, TDL is based on the Logical Execution Time abstraction.

2.2.3 Conclusion

Implementations of the TT paradigm can be classified under two main categories. The first category focuses entirely on the communication networks, e.g. TTP, TTE, Flexray and TTCAN protocols. The second kind of implementations makes assumptions that the network will provide TT behavior, and instead focuses almost entirely on the system modelling and task execution, e.g. Giotto, TDL and PBO frameworks

In our work, we rely on an RTOS implementation based on the TT approach which is part of the second category of the TT paradigm implementations. This implementation is the PharOS platform [9]. Detailed representation of this platform is subject of the next section.

2.3 The PharOS Implementation

2.3.1 Overview of the PharOS Platform

PharOS [9] is an extension of the OASIS framework [31, 36, 67, 68] implemented for the automotive applications. It consists in a framework for safety-critical real-time systems,

based on the time-triggered paradigm. This framework provides methodologies and tools allowing the development of embedded critical software with completely deterministic temporal behavior. Oasis and PharOS implementations comprise a programming language ΨC (Parallel synchronous C), which is an extension of C. This extension allows one to specify tasks and their temporal constraints as well as their interfaces.

An Oasis application is composed of a finite set of communicating and interacting real-time tasks, called agents. An agent is an autonomous execution entity in which external communications are totally defined. An agent is composed of a number of jobs —called also Elementary Actions (EA). These latter are executed sequentially following logical conditions that are expressing their precedence relationships. Each elementary action of each agent has a temporal execution window —i.e. a specific earliest starting date and a deadline—deduced automatically from temporal information of the agent code. This temporal window is specified by the application developers, through specific primitives.

Agents perform computation (through their elementary actions) in parallel on private data. Each data item has exactly one producer (the owner agent) but can have several consumers. Reading of the value of a data item is handled in such a way that the communications are deterministic and in particular independent of the implementation. In fact, a very specific primitive of the ΨC language—for instance the advance primitive—allows the developer to specify, on top of deadline and earliest start instances of jobs, the Temporal Synchronization Points (TSP) which defines instants when tasks can exchange data. At each defined TSP, output variables of elementary action executing before this instant are published to their statically defined consumer tasks, and elementary action starting execution after this TSP read input variables from their respective producer tasks.

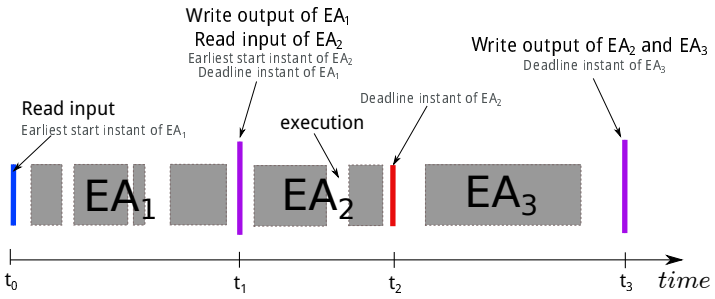


Figure 2.7: Example of elementary actions and their associated time windows and TSP instants.

The example of Figure 2.7, display a set of elementary actions (i.e. EA_1, EA_2 and EA_3) of an agents and their temporal windows and synchronization points. Note that gray boxes represent the effective execution of the elementary action, as it can be preempted by other PharOS agents. The instant t_0 is the earliest start instant of the elementary action EA_1. The instant t_1 defines at the same time the deadline instant of

EA_1, the earliest start instant of EA_2 and a TSP, i.e. at t_1, EA_1 publishes its output variables to their consumer tasks and EA_2 reads its input variables from their owner tasks. The instant t_2 defines the deadline of EA_2, while the instant t_3 defines the deadline of EA_3 and a TSP.

Notice that in PharOS, communication between agents follows a strict observability principle [48]; i.e. an EA can use only temporally visible data, and data that are already visible can not be modified. The visibility date can only be in the future compared to the current date of an agent, i.e. its earliest start date of its current EA.

PharOS and OASIS provide two modes of communication between agents. The first mode uses the exported variables, also called *temporal variables* and the second mode is based on the sending of messages from a sender task to one or more receiver tasks. The new values of a temporal variable are made visible at every synchronization point of its unique producer/owner agent, while messages require explicit definition of visibility dates.

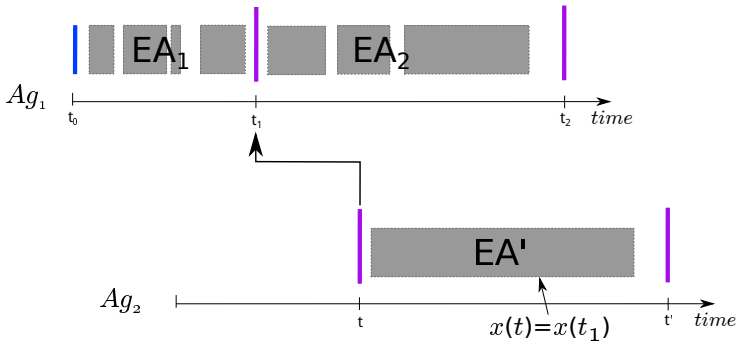


Figure 2.8: Example of two PharOS agents communicating through temporal variable mechanism.

In the first mechanism, each temporal variable defines a real-time data flow which is associated with an internal variable of its owner agent. A real-time clock is associated with each temporal variable. This clock defines the rhythm of adding new values to the flow. If the instant of the clock corresponds to a TSP, the current value of the variable will be at the top of the flow, else the top of the flow will be duplicated. Each temporal variable can be accessed by one or more consumer agents which are statically defined. To access a temporal variable, a consumer agent has to specify the number —i.e. the depth—of the value it needs to consult from the flow.

Consider two agents Ag_1 and Ag_2 of Figure 2.8. And consider a temporal variable x of the agent Ag_1, that is consulted by Ag_2. Regardless of the values between instants t_1 and t_2 of the clock of Ag_1 (i.e. however the value of x is modified by EA_2), the value of the variable x "observed" by the agent Ag_2 at instant t is its past value $x(t) = x(t_1)$. Note that in this example the depth of the observation is 1, i.e. only the last value is consulted.

In the sending message mechanism, the sender agent associates with each message a visibility date, i.e. the date beyond which a message can be accessed by the recipient agent. The latter has queues for receiving messages that are sorted by their visibility dates. For example, consider Figure 2.9 where an agent Ag_s sends a message M with

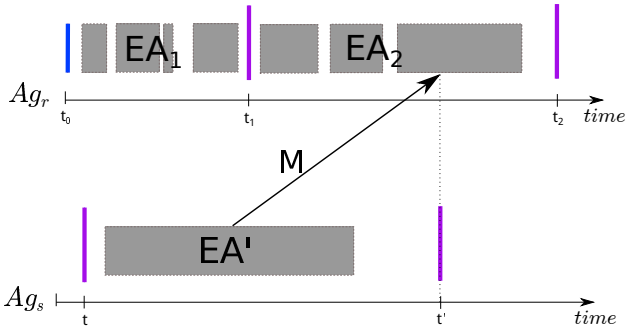


Figure 2.9: Example of two PharOS agents communicating through sending message mechanism.

visibility date t_1 to the agent Ag_r. The message M cannot be observed before the instant t_1 of the clock of Ag_r, since $t' > t_1$.

PharOS platform provides an off-line tool chain responsible for extracting the application's temporal behavior in order to generate a runtime. More specifically, all possible temporal behaviors are computed in order to size communication buffers and to analyse the timing constraints on the execution times. At runtime phase, PharOS applies an Early Deadline First (EDF) algorithm [66], in order to dynamically schedule elementary actions of agents based on their temporal synchronization points.

In our work, we focus only on the temporal variable mechanism. In the next subsection, we provide more details about the ΨC programming language and its syntax, considering only this communication mechanism.

2.3.2 The ΨC Programming Language

ΨC is a programming language designed for specifying different tasks of a PharOS application and their temporal synchronization points. It preserves the operational semantics of C, but adds time constraints to these semantics with the Ψ extension (this extension could be applied to any imperative programming language). C control flow graphs are automata, so C's instructions for control flow can be used to express sequencing of blocks, loops, and choices. The basic Ψ addition to C is the addition of the following instructions: `before`, `after`, and `advance` instructions that respectively add before and after constraints, and temporal synchronization points.

- After instruction (`after`(d)): defines d as the relative release date of the following EA;

- Before node (`before`(d)): defines d as the relative deadline of the preceding EA;
- Advance node (`advance`(d))— also called temporal synchronization point: combines `after`(d) and `advance`(d) instructions. It defines the absolute visibility date of the data produced by the job;

A PharOS application consists of a set of clock definitions followed by a set of task—also called agent—definitions. Recall that, in our work, we consider that the set of parallel agents communicates only through *temporal variables*. PharOS applications are characterised by the following abstract syntax:

$$
\begin{aligned}
\text{Application} &::= \text{Clock}^+.\text{Agent}^+ ,\\
\text{Clock} &::= c = (\phi_c, P_c) ,\\
\text{Agent} &::= \{\texttt{local } variable\}^*.\{\texttt{input } tv\}^*.\{\texttt{output } tv\}^*.\text{Body}^+ ,\\
\text{Body} &::= \{C \ code.[\texttt{after}(n)|\texttt{before}(n)|\texttt{advance}(n)] \ \texttt{with Clock}\}^*.\texttt{next Body} ,
\end{aligned}
$$

where c is a clock, with ϕ_c and P_c being respectively the *phase shift* and *period* of c (see the detailed definition below); tv is a *temporal variable* and $n \in \mathbb{Z}_+$ is a time step w.r.t. to an associated clock.

Clocks

Clocks are variables used to describe the temporal behavior of the application. A clock defines a sequence of periodic instants called *activation instants*. These latter are used by the agents for describing timing constraints and synchronizations. Each clock c has an associated phase shift ϕ_c and a period P_c. Formally, the clock c defines a sequence of instants $(t_i)_{i\geq 0} = (i \cdot P_{BASE} + \phi_{BASE})_{i\geq 0}$.

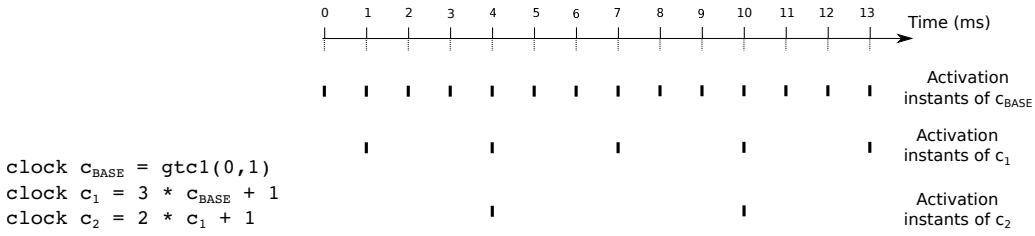
The global clock $c_{BASE} = (\phi_{BASE}, P_{BASE})$ is defined by its phase shift (always default to zero) and period expressed in real time units, such as 1 second, 100 milliseconds etc. The ΨC language provides a set of primitives allowing to define these clocks depending on the unit of their period (i.e. time separating two ticks of the clock). Let P_{BASE} be the period of c_{BASE} which is measured in nanoseconds, the different primitives are as follows:

- clock $c_{BASE} = gtc0(valSec)$, where $P_{BASE} = ((valSec * 1000) * 1000) * 1000ns$;
- clock ${}_{BASE}c = gtc1(valSec, ValMilliSec)$, where $P_{BASE} = ((valSec * 1000 + ValMilliSec) * 1000) * 1000ns$;
- clock $c_{BASE} = gtc2(valSec, ValMilliSec, ValMicroSec)$, where $P_{BASE} = ((valSec * 1000 + ValMilliSec) * 1000 + ValMicroSec) * 1000ns$;
- clock $c_{BASE} = gtc3(valSec, ValMilliSec, ValMicroSec, valNanoSec)$, where $P_{BASE} = ((valSec * 1000 + ValMilliSec) * 1000 + ValMicroSec) * 1000 + valNanoSecns$.

Other clocks $c = (\phi_c, P_c)$, are defined w.r.t. c_{BASE}, by putting $c = P_c * c_{BASE} + \phi_c$. Activation instants $(r_i)_{i\geq 0}$ of c are computed from those of c_{BASE} as follows:

$$r_i = (i \cdot P_c + \phi_c)P_{BASE} + \phi_{BASE}\,. \tag{2.1}$$

The ΨC language also provides—for the designers' convenience—a possibility of defining new clocks in terms of clocks other than c_{BASE}. Figure 2.10b depicts activation instants of the clock c_{BASE} with period of one millisecond, a clock $c_1 = (1, 3)$ derived from c_{BASE} and a clock c_2 derived from c_1. Activation instants of c_1 are $1ms$, $4ms$, $7ms$ etc.. The ΨC code declaring the clocks of this example is shown in Figure 2.10a, where `gtc1` is the ΨC primitive declaring a global clock with a period of one millisecond.



(a) (b)

Figure 2.10: Example of clocks and activation instants

An instant t_i of c_{BASE} (resp. r_j of c) can be referenced by its index i (resp. j). For example, in Figure 2.10, "instant 4 of c_{BASE}" refers to the physical activation instant $t_4 = 4ms$. Similarly, "instant 1 of c_1" refers to the instant $r_1 = 4ms$.

An instant r_i of clock $c = (\phi_c, P_c)$ can be mapped into an instant t_j of clock c_{BASE} by the function $conv^c_{c_{BASE}} : c \to c_{BASE}$, defined by letting

$$conv^c_{c_{BASE}}(r_i) = t_j\,, \quad \text{with} \quad j = i \cdot P_c + \phi_c\,. \tag{2.2}$$

Inversely, a global instant t_i of clock c_{BASE} can be mapped into an instant r_j of a derived clock c by using the function $conv^{c_{BASE}}_c$, defined by letting

$$conv^{c_{BASE}}_c(t_i) = r_j\,, \quad \text{with} \quad j = \left\lfloor \frac{i - \phi_c}{P_c} \right\rfloor\,. \tag{2.3}$$

For example, in Figure 2.10, the instant $r = 1$ of clock c_1 is mapped to the instant $conv^{c_1}_{c_{BASE}}(1) = 4$ of clock c_{BASE}. The instant $t = 5$ of clock c_{BASE} is mapped into instant $conv^{c_{BASE}}_{c_1}(5) = 1$ of clock c_1.

Agent

An agent consists of an interface including declarations of local variables, input and output data flows (*temporal variables*) followed by a body.

The block allowing to define the set of local variables necessary for different computations is named *global* (since these variables local to the agents are global to different bodies of the agent). It consists of C declarations of variables.

The block allowing to define a temporal variable is named *temporal*. It consists of declarations of temporal variables of the agent (i.e. the output variables). Each declaration starts by the C type of the variable (int, float etc.). It is followed by an integer expression defining the depth of the temporal variable i.e. the maximum number of past values to which the agent wants to be able to access. When equal to zero, this means that only the current value of the variable is manipulated by different bodies of the agent. This integer expression is separated by the symbol "$" from the unique identifier of the temporal variable. For example the declaration *int* $0\$x$, defines a temporal variable containing an integer x and allowing access only to its current value.

After defining the output temporal variable, the agent defines the list of agents that access this output temporal variable. This is done through the *display* block. One declaration of this block is of the form $x : agent2$. Which consists in allowing $agentId$ to access to the temporal variable x.

An input temporal variable is specified by the *consult* block. A declaration of this block consists in indicating the identifier of the owner agent followed by an integer expression defining the number of the consulted past values. This integer is separated from the identifier of the temporal variable by the symbol "$". For example $agent1 : 1\$x$ allows the consult the last value of the temporal variable x of the agent $agent1$. Figure 2.11, displays the definition of *agent1* with its four blocks *global*, *temporal*, *display* and *consult*. *Agent1*, defines a local integer variable z necessary for its computations, a temporal variable x which is consulted by *agent2*. The agent *Agent1* consults the last value of the temporal variable y of *agent3*.

The body block within an agent describes the behavior of the agent through a block of timeless C code extended with *after*, *before* and *advance* statements. An `after`(d) (resp. `before`(d), `advance`(d)) with a clock $c = (\phi_c, P_c)$, defines the release (resp. deadline, synchronization) instant corresponding to d units of time after a reference instant. This reference instant corresponds to the absolute instant recording the visit of the last `after` or `advance` node.

Code of Figure 2.12 describes the behavior of a task with four jobs (labelled A to D). In this example, all temporal constraints are defined over the same clock c. The release date of job B is one unit of time after the initial instant or previous advance constraint, i.e. `advance`(3), depending on the execution history. Two units of time later, job B must have ended. After the execution of the job C, communication take place since advance statement is reached. The visibility date of data produced by C is three units of time after the previous visit to the statement `after`(1).

A formal model of ΨC was provided in [65], where the behaviour of a task is specified using a directed graph, where arcs represent the successive jobs and nodes bear the temporal constraints. Nodes of the graph are of four types: After, before, advance and

```
agent agent1() with clock {
  global{
    int z;
  }
  temporal{
    int 0$x;
  }
  display{
    x : agent2;
  }
  consult{
    agent3 : 1$y;
  }
  body start
  {
    ...
  }
  ...
}
```

Figure 2.11: Example of input and output temporal variables declarations in ΨC

```
body start
{
  // Job A
  ComputationA();

  // Job B
  after(1) with c;
  ComputationB();
  before(2) with c;

  // Job C
  ComputationC();
  advance(3) with c;

  // Job D
  ComputationD();
}
```

Figure 2.12: Example of body ΨC code

no-constraint nodes. We believe that this model is not at the same abstraction level as the ΨC language since it does not hold clocks and thus does not provide the possibility of specifying constraints over different clocks. Also operational semantics of this model are not provided.

2.4 Conclusion

This chapter presents a conceptual overview of the time-triggered paradigm and its key features (namely in Section 2.1). Implementations of the TT paradigm are ranging from TT protocols that are focusing entirely on the communication networks, e.g. TTP, TTE, FlexRay and TTCAN protocols (see Section 2.2.1) to system modelling frameworks which are focusing almost only on TT tasks execution, e.g. Giotto, TDL, PBO (cf. Section 2.2.2) and PharOS (cf. Section 2.3) frameworks.

This thesis work targets the RTOS-based implementation which is the PharOS platform, presented in details in Section 2.3. We focus only on the temporal variable communication mechanism of this framework as our PharOS platform version uses exclusively this mode of communication.

Design principles of a TT model (presented in Section 2.1) are guiding elements for the definition of the first part of our transformational approach (as presented in Chapter 4). A formal model of the ΨC language —the programming language of PharOS platform—with explicit operational semantics is extremely necessary to the second part of our transformational approach. For this aim, we elaborated a formal model that is presented in Chapter 5.

Part II

Approach

3

Related Work and Background: Existing Transformational Approaches

The transformation approach presented in this thesis combines advantages of three major features; (1) the source model is a component-based model, (2) the target implementation is an RTOS-based implementation which relies on the time-triggered model, and (3) the transformation is correct-by-construction, due to the well defined operational semantics of its source models.

Based on these three criteria, we tried to situate and compare our approach with other existing transformational approaches. Nevertheless, to the best of our knowledge, no related work has been found with respect to these three criteria at once. There exist, however, several approaches which satisfy one or two of these features. This chapter presents a non-exhaustive list of existing transformational approaches attempting to establish a link between high-level design frameworks and implementations.

In Section 3.1, we list these approaches and evaluate them according to our approach. And in Section 3.2, we present some background concepts (namely conflict resolution protocols) of one of these approaches, that are reused later in the first step of our transformation method.

Chapter outline

3.1 Related Work

Approaches relying on component-based source models

Methods relying on model transformations in order to automatically refine AADL models are presented in [23, 29]. In order to reduce the gap between models used for timing analysis and for code generation, abstract models of computation are first transformed in more precise models, which include the timing characteristics of the execution platform. These refined models are then used for a more precise timing analysis. It is clear that these proposed frameworks have been proposed in order to ease the timing analysis of embedded systems. However, these approaches do not specifically target TT implementations nor rely on well-defined formal semantics allowing to formally prove the correctness of the transformation process.

Another transformational approach, having as source models the AADL models, is presented in [46]. The goal of this work is to propose a rapid prototyping methodology based to develop distributed real-time and embedded systems around the AADL. The proposed design-by-refinement approach is implemented around the Ocarina tool suite The obtained system is assumed to be very close to the final product, where some user functional components have to be completed. Although this approach is claimed to significantly reduce the time needed to specify, prototype, and produce a distributed real-time embedded system, it is not providing formal correctness proofs nor guarantees of determinism for hard real time systems.

Approaches Targeting TT implementations

A design framework based on UML diagrams and targeting the TT Architecture (TTA) [58] is presented in [72]. This approach relies on a decomposition of a system into clusters and nodes to instantiate the communication mechanisms. It assumes the underlying TT protocol to implement the FlexRay standard [74]. Essential features of the underlying architecture and protocol are expressed using the different diagram types and notations of UML. Even if it targets a TT implementation, this framework —unlike our approach—does not support the earlier architectural design phase, nor the verification at model level. It requires a backward association mechanism to link faulty runs obtained at the SystemC level to the UML model.

A code generation tool-chain from SCADE/Lustre [42] to the TT Architecture (TTA) is presented in [30]. In this approach, Lustre has been extended with additional primitives to specify code distribution, timing requirements and deadlines. Another relevant work proposes an automatic transformation from SCADE synchronous language models into OASIS applications. In particular, the paper presents a transformation method preserving the functional semantics of the applications through an optimised arrangement of OASIS logical clocks. These two approaches are both limited to relatively simple temporal behaviors. Their source models define periodic functional behaviour of the system, with the key real-time constraint being the duration of the period. In contrast, in our approach, RT-BIP source models define real-time constraints of arbitrary complexity.

In [75] and [76], authors propose the integration of the TDL methodology with Simulink framework. This approach provides a powerful modelling and simulation environment where TDL components can be modelled and simulated without knowing on which platform they will be executed. The basic idea of this integration is to use standard Simulink blocks to model the LET behavior of TDL tasks. The mapping to a specific platform —distributed or not—is a straight-forward assignment of TDL components to the platform nodes (ECUs).

An extension of Simulink to express designs of the time-triggered Giotto language is also presented in [50, 45]. The proposed tool-chain in this work —demonstrated on a helicopter autopilot system—proposes an automatic generation of Giotto code meant for monitoring the interaction of the functionality code with the physical environment. These extension approaches of Simulink and Ptolemy (with TDL and Giotto) are not presented as formal rule-based transformations. And no formal correctness of the integration is proven.

Similarly, approaches presented in [79] and [41] propose to extend the Ptolemy II framework respectively with TDL and Giotto models of computations. In [41], the code generation framework within Ptolemy II is extended to generate C code for the Giotto programming model (running on the FreeRTOS embedded operating system). While authors of [79] present the TDL domain in Ptolemy II, that is, the add-on Ptolemy software components which allow the specification and simulation of discrete event models with TDL semantics.

Although these two integration approaches are different from the viewpoint of the purpose and the implementation, they are both not presented as a rule-based transformation approaches that are proven to be correct.

Approaches presenting correct-by-construction transformations

Two model transformation approaches for generating distributed implementations from non-real-time BIP models and real-time BIP models, are presented respectively in [21] and [86]. In these approaches, the initial model is transformed into a 3-layer model relying exclusively on simple message-passing interactions, which are implementable using basic message-passing primitives.

Another method for generating a mixed hardware/software system model for many-core platforms from a high-level non-real-time application model and a mapping between software and hardware components are presented in [25].

The above approaches take advantage of the BIP framework to build correct-by-construction implementations based on a single semantic framework. Nevertheless, they do not target the platforms based on TT execution model, thereby falling short of exploiting the strong temporal guarantees provided by the latter.

In [11], authors present a correct-by-construction approach to transformations across design environments. In order to ensure correctness by construction, authors suggest using a common formal model, namely the synchronous reactive model of computation. This formal model is used as the common ground to interpret system specifications

given with different underlying models. Authors chose two tools (ASCET and Simulink) —widely used in the automotive domain—to demonstrate the presented approach. Although this approach is based on a common formal model which allowed authors to present a rule-based transformation, no formal correctness proofs are provided.

In [37], authors present a framework for graph transformation. Semantical correctness is ensured by using the rules for the model transformation also for the transformation of the operational semantics, which is given by graph rules. This allows to compare the behaviour of the source model with the one of the target model. However, even if this paper is presenting formal transformation rules and correctness theorems, it does not consider the time-triggered paradigm as a basis for the target implementation. It is not an approach for designing and implementing a critical real-time application based on the time-triggered model.

In another line of work, authors of [49] propose two methods of certifying model transformations. In the first method, they propose to establish links between the elements in the target model and the elements in the source model. These links will then be checked using a bisimilarity checker tool to prove that the target model is a bisimulation of the source model. The second method requires the translation of the source and target models to an equivalent formal model that is written in the same formal language. The obtained formal models will then be checked for bisimulation. It is clear that the main difference between this work and our approach relies in the purpose. In fact, our main goal is to propose a correct-by-construction transformation in order to obtain a TT implementation. In the contrary, this work target the general purpose of certifying model transformation approaches without any special focus on hard real-time implementation.

3.2 Background

As stated in the previous section, the transformational approaches of [21] and [86] aim at transforming a BIP model into a distributed implementation. During these transformations, authors face the conflict resolution problem and propose a set of solutions. Even though, in our work, we do not aim at targeting especially the distributed implementation, we face the same conflict resolution problem while transforming a BIP model (more details in Chapter 4). For this reason, this work is considered as a background to our work since we reuse their proposed solution for resolving conflicts. In order to present in details this solution, we need to provide a quick overview of their approach. This is the main subject of this section.

3.2.1 Transformation of BIP models into distributed implementations

Transformational approaches of [21] and [86] propose a methodology to provide automatically efficient and correct-by-construction distributed implementations starting from

a high-level model of the software application in non real-time BIP and real-time BIP. A key idea of this methodology is to use a set of correct transformations which preserve functional properties. Furthermore, they take into account extra-functional constraints.

In distributed implementations, primitives available for communication are less powerful than BIP coordination. This latter is achieved through multiparty interactions and scheduling by using dynamic priorities. And its associated semantics is defined on a global state model.

In order to be able to derive distributed application from BIP models, authors propose to transform arbitrary BIP models into Send/Receive BIP models which are directly implementable on distributed execution platforms.

Send/Receive BIP models consist of components coordinated by using asynchronous message passing (Send/Receive primitives). They comply with a three-layer architecture where the bottom layer includes the components of the application software, the second layer includes a set of distributed engines handling each a subset of interactions of the original model and the third layer implements a conflict resolution protocol used to resolve conflicts between engines of the second layer.

The obtained Send/Receive BIP models are proven observationally equivalent to the initial models. They are then used to generate stand-alone C++ implementations using either TCP sockets for conventional communication, or MPI implementation, for the deployment on multi-core platforms.

In the case when engines of the intermediate layer, handle interactions that are conflicting with other engine interactions, the third layer interferes —dynamically—in order to resolve this conflict.

The Conflict Resolution Protocol is implemented using algorithms that solve the committee coordination problem [33]. Authors adapt a variation of the idea of the message-count technique from [10]. This technique is based on counting the number of times that a component executes a communication or a computation step. Each component keeps a counter nb which indicates the current number of participations of the component in interactions or internal computations. The Conflict Resolution Protocol ensures that each participation number is used only once. That is, each component takes part in only one interaction per transition. To this end, in the Conflict Resolution Protocol, for each component B_i, we keep a variable NB_i which stores the latest number of participations of B_i. Whenever the Conflict Resolution Protocol is solicited by the second layer to execute an interaction α where $P_\alpha = \{p_i\}_{i\in I}$, it receives a set of participation numbers $\{nb_i\}_{i\in I}$ for all components involved in α. If for each component B_i, the participation number nb_i is greater than NB_i, then the Conflict Resolution Protocol acknowledges successful reservation through port ok_α and the participation numbers in the Conflict Resolution Protocol are set to values sent by the the second layer. On the contrary, if there exists a component whose participation number is less than or equal to what Conflict Resolution Protocol has recorded, then the corresponding component has already participated for this number and the Conflict Resolution Protocol replies failure

via port $fail_\alpha$.

Authors of [21] and [86], in particular, consider three committee coordination algorithms —all inspired from [10]: (1) a fully centralized algorithm, (2) a token-based distributed algorithm and (3) an algorithm based on reduction to distributed dining philosophers [32].

Regardless the employed algorithm, A CRP handling a set of conflicting interactions follows these restrictions:

- For each component $B_i \in comp(\alpha)$, such that α is handled by the Conflict Resolution Protocol, this latter maintains a variable NB_i indicating the last participation number reserved for B_i.
- For each interaction α where $P_\alpha = \{p_i\}_{i \in I}$ handled by the Conflict Resolution Protocol, are included three ports: rsv_α , ok_α and $fail_\alpha$. The port rsv_α receives reservation requests containing fresh values of variables n_i. The ports ok_α and $fail_\alpha$ accept or reject the latest reservation request. In case of positive response (through port ok_α), variables NB_i are updated.
- Each rsv_α message should be acknowledged by exactly one ok_α or $fail_\alpha$ message.
- Each component of the Conflict Resolution Protocol should respect the message-count properties described above.

In the rest of this section, we explain the behavior —through a representative example—of the CRP component implementing the fully centralized algorithm. Details about the two remaining algorithms as well as formal definitions are provided in [47].

Centralized CRP component behavior

In this paragraph, we present the behavior of the CRP component implementing the fully centralized algorithm through the example of fragment in Figure 3.1 which displays the principle of the CRP behavior that handles an interaction α_1. We assume that α_1 is connecting two components: B_1 and B_2. As depicted in Figure 3.1, the CRP component has variables NB_1 and NB_2 which correspond to reference variables storing the latest number of participations of components B_1 and B_2. The CRP component contains a waiting location $w\alpha_1$, a reservation location r_{α_1} and three ports rsv_{α_1}, ok_{α_1} and $fail_{\alpha_1}$. Time progress condition of the location $w\alpha_1$ is always set to *True*, while the time progress condition of a location r_{α_1} is set to *False*. To the port rsv_{α_1}, are associated variables nb_i such that $B_i \in comp(\alpha_1)$ (in our example nb_1 and nb_2).

The location of the initial state is $w\alpha_1$. Whenever a reservation for executing the interaction α_1 arrives, the location r_{α_1} is reached. From this location, if the guard of the transition labeled by ok_{α_1} is *True*—according to freshly received nb_i and the current values of NB_i—the transition ok_{α_1} can execute to reach back the location $w\alpha_1$. When executing, the transition ok_{α_1} updates reference variables NB_i by copying values of the

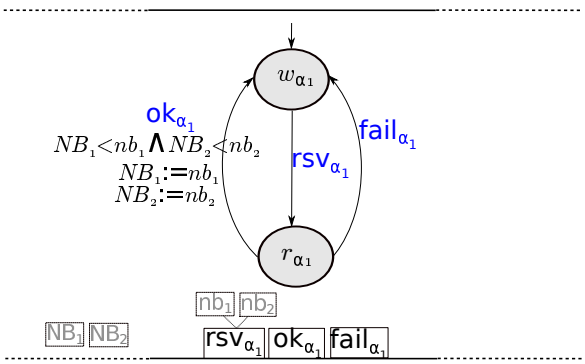


Figure 3.1: Conflict resolution principle

variables nb_i. Therefore after the execution of this transition, its guard becomes *False*. The transition labeled by $fail_{\alpha_1}$ is always possible.

As explained before, the fragment displayed in Figure 3.1 represents a seperate automaton handling only one interaction. A CRP component usually handles two or more interactions. Its behavior is, thus, obtained by composing seperate automata of its handled interactions. For example, we consider a CRP component that handles two conflicting interactions α_1 and α_2. These interactions are connecting, each, two components: B_1 and B_2 for interaction α_1 and B_2 and B_3 for interaction α_2. For simplicity of the representation, this CRP component may be displayed as in Figure 3.2. The

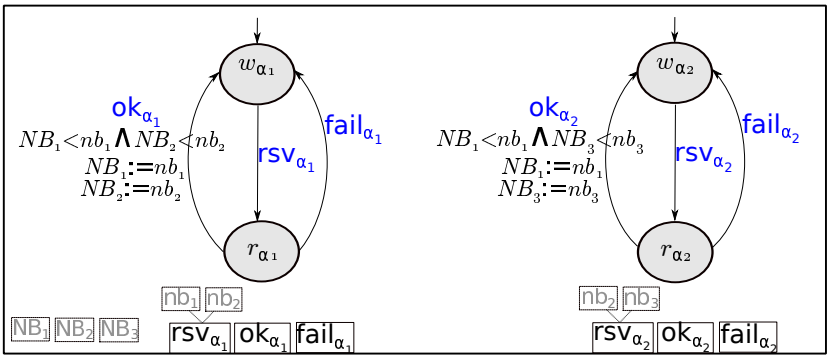


Figure 3.2: An example for the centralized Conflict Resolution Protocol for handling two conflicting interactions α_1 and α_2

composed automaton of this CRP component is as displayed in Figure 3.3. Note that, in the case when the CRP component receives two reservation requests for executing conflicting interactions, one of the two transitions labeled by port ok_{α_k} will be selected and executed. The other *ok* transition will become disabled, since one of its guards becomes *False*, leaving the fail transition be the only possible transition. This latter is then executed allowing to reach back the waiting state.

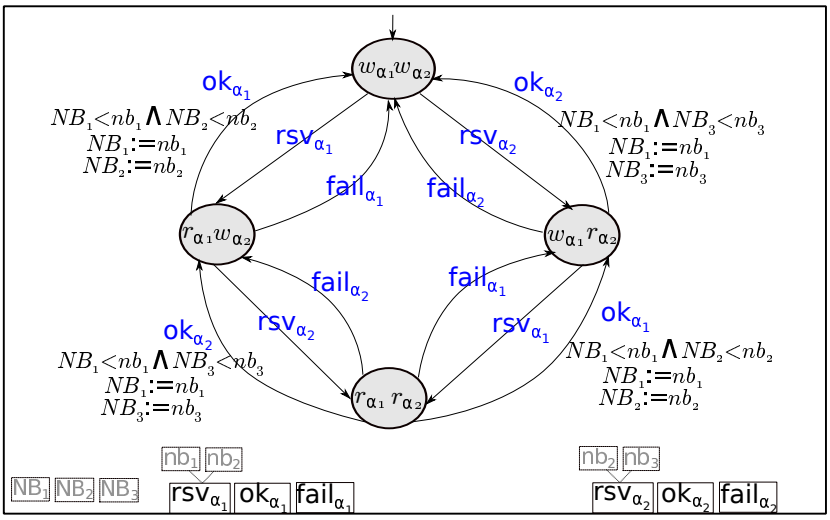


Figure 3.3: Automaton of CRP component of Figure 3.2

3.3 Conclusion

In this chapter, we present some of existing approaches that establish a link between high-level design frameworks and implementations. Among these, we focus on one approach that transforms BIP models to distributed implementation. This work is considered as a background to our work, since we reuse the proposed solution in conflict resolution. In Chapter 4, we detail the first step of our transformation process, and we show how this conflict resolution method is integrated into the target model of our transformation.

4

From High-Level BIP Model to Time-Triggered BIP Model

After presenting the BIP framework (Chapter 1), the time-triggered paradigm (Chapter 2) and after giving an overview of the existing approaches transforming a high-level model to implementations (Chapter 3), we can start to present our approach to transform a BIP model into a TT implementation. Direct transformation is challenging since we need, in a first step, to introduce mode implementation details earlier in the BIP model.

Therefore, in this chapter, we focus on transforming BIP models in such a way that the TT communication system can be explicitly instantiated in the resulting model.

We present a transformational method which starts from a BIP model and a user-defined task mapping (see Figure 4.1) and consists in adapting the initial model to comply with the TT-communication pattern, i.e. tasks communicate only through a communication medium by using unidirectional message passing. The obtained model—called TT-BIP model—is then a structural restriction of BIP model respecting the TT paradigm. This model is needed to be—in a second step—directly transformed into the programming language of the target platform based on the TT execution model.

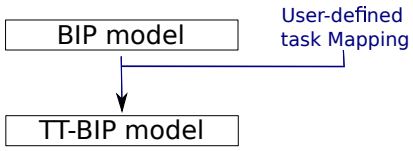


Figure 4.1: Transformation approach

This chapter is structured as follows. Section 4.1 discusses different challenges of the transformation. In Section 4.2, we explain approach allowing to address these challenges, and explain choices leading to the definition of the structure of the target model. In Section 4.3, we detail

restrictions on the input BIP model. In Section 4.4, we formally define the transformation of a high-level BIP model into a TT-BIP model. Section 4.5 deals with correctness proof of the proposed transformation.

Chapter outline

4.1 Problem Statement

Transforming a user-defined task mapping and a high-level model based on multi-party interaction model into an equivalent model where interactions comply with the TT communication pattern, is a challenging task. From one hand, introducing TT settings consists in (1) instantiating tasks in the derived model according to the user-defined task mapping, (2) modelling the TT communication system by introducing dedicated atomic components and (3) restricting the synchronous multiparty inter-task interactions to simple unidirectional communications with the introduced communication components. From the other hand, the derived model is required to be observationally equivalent to the original BIP model.

In order to understand different challenges of such a transformation, consider the BIP model in Figure 4.2.

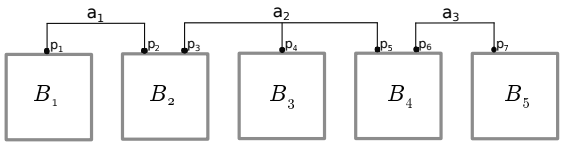


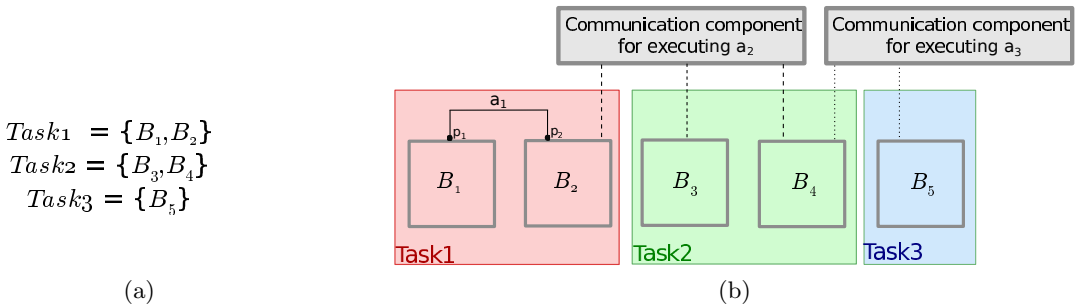
Figure 4.2: High-level BIP model

In Figure 4.2, the model consists of five atomic components B_1,..., B_5 which are synchronizing through rendezvous interactions a_1, ..., a_3. In BIP framework, interactions are executed sequentially and atomically by the BIP engine. Thus, combining the need for respecting the TT settings with the need for providing the transformation correctness, requires the target model to deal with more complex issues:

Decomposition into Tasks

Tasks (processes, threads, etc.) are building blocks of TT applications. In the design phase, designers have the choice to model a TT task using one or more BIP components. This task mapping is needed not only for defining task components but also for defining inter-task interactions that are concerned by the transformation.

For example, if we consider the task mapping displayed in Figure 4.3a for the model of Figure 4.2, then inter-task interactions are interactions a_2 and a_3. Only these two interactions have to be handled by dedicated communication components. Moreover, in the final model, components of a single task are grouped into the same composite component. Figure 4.3b shows a skeleton of the obtained model from the BIP model of Figure 4.2 and task mapping of Figure 4.3a. Dashed and dotted lines in Figure 4.3b display communication between tasks' components and their corresponding communication components. Details about connectors of these communications are provided by answering to the next challenge.



(a) (b)

Figure 4.3: Skeleton of the obtained model according to task mapping

Strong synchronization in BIP interactions Vs. asynchronous message-passing

In order to respect TT communication settings, the derived model should handle each inter-task communication through a dedicated BIP component which stands for the TT communication system. This latter can communicate with tasks only through message-passing. The challenge here is to switch from the high-level BIP model, where multi-party interactions provide component synchronization on top of data transfer, to asynchronous message-passing communications while preserving the models equivalence.

Suppose that the interaction a_2 of the example of Figure 4.2 allows to transfer data from component B_2 to components B_3 and B_4. Note that this interaction is atomic and allows to synchronize components B_2, B_3 and B_4. Suppose also that the dashed lines in Figure 4.3b present three binary connectors allowing B_2 to send data to the communication component and B_3 and B_4 to receive data from that component. Clearly, this option doesn't preserve the synchronisation between these three components ensured by the interaction a_2 in the original model since the atomicity of the original interaction is no more respected. In such a case, the communication component must be designed so that execution of interactions does not introduce behaviors that were not allowed in the initial model.

This issue is addressed by breaking the atomicity of execution of interactions. A task can execute unobservable actions to notify the communication component about their states. If all participating components are ready, the communication component can execute the corresponding interaction.

Resolving conflicts

Suppose interaction a_2 is conflicting with interaction a_1 and/or with interaction a_3. Interaction a_2 shares with interaction a_1(resp. a_3) component B_2 (resp. B_4). Thus, a_2 can not execute concurrently with a_1 and/or with a_3. In high-level BIP model, such conflicts are resolved by the single engine. TT communication components in the derived model must ensure that execution of conflicting interactions is mutually exclusive.

4.2 Proposed Solution

We propose a generic framework for transforming a high-level BIP model into an equivalent model satisfying the TT settings and addressing the previously cited challenges.

The obtained model (1) operates in partial-state semantics, (2) expresses multiparty interactions in terms of asynchronous message passing and (3) is observationally equivalent to the initial model. The target model is structured following a three-layer architecture called TT-BIP architecture:

1. The Task Components Layer consists of a transformation of atomic components corresponding to the behavior layer of the initial model. This layer also depends on a user-defined task mapping. A task component can interfere even in an internal computation, intra-task interaction (i.e. communication between components of the same task) or inter-task interaction (i.e. communication with other tasks). Components within a task that are concerned by the inter-task interaction or participating in an intra-task interaction that is conflicting with an inter-task interaction, operate in partial-state semantics.

2. The communication Layer aims at modelling the TT communication system by hosting inter-task interactions and allowing to resolve their potential conflicts by soliciting the third layer. This layer contains TT communication component (TTCC) hosting each an inter-task interaction of the original model.

   We have essentially two conflict cases involving inter-task interactions; conflict between only inter-task interactions and conflict between inter-task interactions and intra-task interactions or internal computations. By dedicating a third layer for resolving conflicts, the first case of conflicts, if existing, can be directly resolved. Resolving the second conflict case, can not be resolved locally since a task has a partial observability of the system. This needs however, to host the conflicting intra-task interaction or internal computation in the communication layer in order to be resolved by requesting the third layer. Notice also that two conflicting intra-task interactions a_1 and a_2, such that a_2 is conflicting with an inter-task interaction b, need both to be handled in the communication layer. We say that a_2 is *directly* conflicting with b, while a_1 is *indirectly* conflicting with the same interaction.

   Thus, this layer consists of components hosting each either an inter-task interaction or an interaction that is either *directly* or *indirectly* conflicting with another inter-task interaction. For simplifying the notation, all constituent components of the communication layer are denoted by TTCC components.

3. The Conflict Resolution Protocol (CRP) Layer resolves the conflicts requested by the communication layer. In the original model, these conflicts are resolved by the BIP engine. In order to guarantee conflicts resolution in the derived model, we reuse the same solution proposed in [47, 77, 86] which consists in dedicating

a third layer to implement the fully centralized committee coordination algorithm presented in [10].

Cross-layer interactions are send/receive interactions, i.e. providing a unidirectional data transfer from one sender component to one or more receiver(s).

Note that tasks are building blocks of the first layer, which addresses the first challenge. Components within a task that are concerned by the inter-task interaction or a related conflicting one operate in partial-state semantics. This allows tasks to break the atomicity of the original interactions and communicate with the second layer in two steps through the send/receive interactions, which addresses the second challenge. The introduction of the third layer and hosting all interactions that are conflicting with inter-task interactions in the communication layer allows to resolve the third challenge.

4.2.1 TT-BIP: Architecture of the Target Model

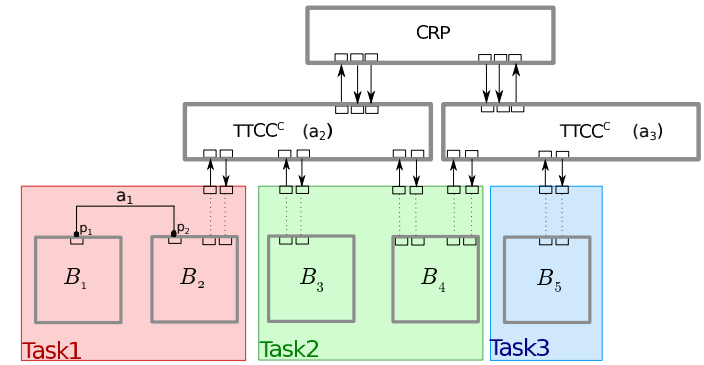
In this subsection, we present in details the TT-BIP architecture. As explained before, it imposes a structure for the target model of the transformation in order to guarantee both its compliance with the TT settings and its observational equivalence with respect to the original BIP model.

A BIP model complies with the TT-BIP architecture if it consists of three layers: Tasks layer, TTCC layer and CRP layer, organized by the following abstract grammar:

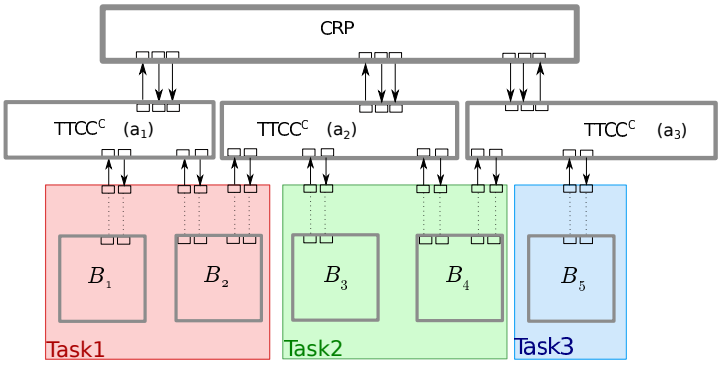
$$
\begin{array}{lll}
\textit{TT-BIP-Model} & ::= & \textit{Task}^{+} \;.\; \textit{TTCC}^{+} \;.\; \textit{CRP} \;.\; \textit{S/R-connector}^{+} \\
\textit{Task} & ::= & \textit{atomic-component}^{+} \;.\; \textit{atomic-talking-component}^{+} \;.\; \textit{connectors}^{+} \\
\textit{TTCC} & ::= & \textit{TTCC}^{NC} \mid \textit{TTCC}^{C}
\end{array}
$$

The TT-BIP model consists of a set of Tasks, TTCC and CRP components. A task component is a composite component consisting of one or more atomic components. Atomic components within a task which interfere in inter-task interactions (via the task interface) are called *atomic-talking-components* (ATC). These latter can only communicate with a TTCC component or a component within the same task. The behavior of a TTCC component depends on whether the interaction it is hosting is conflicting or not. If the interaction is conflicting, the TTCC component is denoted by $TTCC^{C}$ and needs to communicate with the CRP component. Otherwise, it is denoted by $TTCC^{NC}$. Conflicts between different $TTCC^{C}$ components are resolved through CRP component.

Task components (resp. TTCC components) and TTCCs (resp. CRP components) communicate with each other through message-passing, i.e. send/receive interactions. Such interaction is a set of one send port and one or more receive ports. Communications between components inside a task are classic multi-party BIP interactions. Figure 4.4 shows an overview of the TT-BIP model derived from BIP model of Figure 4.2 and the task mapping displayed in Figure 4.3a. Notice that in Figure 4.4a, we assume that the interaction a_2 is conflicting only with the interaction a_3, while in Figure 4.4b a_2 is conflicting with both a_1 and a_3.



(a) a_2 conflicting with a_3



(b) a_2 conflicting with a_1 and a_3

Figure 4.4: Overview of the TT-BIP model of the model of Figure 4.2

Formally, we define a TT-BIP model as follows:

Definition 4.1. *We say that* $B^{TT} = \gamma^{TT}(B_1^{TT}, ..., B_n^{TT})$ *is a TT-BIP model iff we can partition the set of its ports into three sets* P_u, P_s *and* P_r *that are respectively the set of unary ports, send ports and receive ports, such that:*

- *Each interaction* $\alpha \in \gamma^{TT}$ *is either a send/receive interaction with* $P_\alpha = s, r_1, ..., r_k$, $s \in P_s$, $r_1, ..., r_k \in P_r$, $G_\alpha = True$ *and* F_α *copies variables exported by port* s *to variables associated with ports* $r_1, ..., r_k$, *or a unary interaction—called also* external *interaction—where* $P_\alpha = p_\alpha$ *with* $p_\alpha \in P_u$, $G_\alpha = True$ *and* F_α *is the identity function.*

- *Interactions that are relating components of the same task are classic multiparty interactions—called* internal *interaction—.*

- *If* s *is a port in* P_s, *then there exists one and only one send/receive interaction* $\alpha \in \gamma^{TT}$ *with* $P_\alpha = (s, r_1, ..., r_k)$ *and all ports* $r_1, ..., r_k$ *are receive ports. We say that* $r_1, ..., r_k$ *are receive ports of* s,

- *In the TT-BIP model, from the same state, an internal port can be simultaneously enabled only with another internal port. A receive port can be conflicting either with receive or send ports or both. A send port can be conflicting either with send or receive ports.*
- *If defined, update functions of transitions labelled by send ports do not involve data associated to the labelling port (send port).*
- *All transitions that are triggered by receive-ports are associated with timing constraint and guards that are always default to True.*
- *If $\alpha \in \gamma^{TT}$ is a send/receive interaction such that $P_\alpha = (s, r_1, ..., r_k)$ and s is enabled at some global state of B^{TT}, then all its receive ports $r_1, ..., r_k$ are also enabled at that state.*

4.2.2 Discussion

The proposed solution leads out to a 3-layer architecture structuring the target model of the transformation. Although our work does not have the same goal as transformational approaches proposed in [47, 77, 86], but there is some intersection between both target models' architectures. Aiming at deriving distributed implementations from high-level BIP model, these cited approaches propose an intermediate model called send/receive model. This latter is a 3-layer model consisting of atomic components layer, schedulers layer and CRP layer.

As already mentioned in the opening of this chapter and in Chapter 3 Section 3.2, we reuse the third layer of the send/receive model (i.e. the CRP layer) since it is, so far, the unique solution to guarantee the conflicts resolution without requesting the BIP engine. The difference between the send/receive and the TT-BIP architectures lies in the task notion introduced in the TT-BIP architecture. Thus, we build the task layer depending on a user-defined task mapping, and we construct communication components in order to handle inter-task interactions and other conflicting interactions. In the second layer of send/receive models, are introduced schedulers allowing to handle interactions between all atomic components. Also, we introduce one component per external interaction, while a scheduler of send/receive model can handle more than one interaction.

4.3 Input Model Restrictions

In our work we impose the following restrictions on the input model in order to simplify the presentation of the transformation towards TT model:

- We assume that the input model is flat, i.e. it consists only of atomic components and flat connectors. Since all connectors are assumed to be flat, they do not hold an exported port. This restriction is obtained by using the flattening tool from

previous research work [47, 27]. This tool replaces all hierarchical connectors and composite components of a BIP model by an equivalent set of flat connectors and atomic components.

- We also assume that all connectors' ports are synchron ports. This restriction can be met by replacing a connector with a trigger port by an equivalent set of connectors implementing the same set of interactions. For more details see Remark 1.5 of Section 1.2.
- Each port is assumed to labels at most one transition of the component automaton.
- We also assume that the input model contains no priority rules. In previous work [77], it has been shown that any BIP model with priority rules can be transformed into an equivalent model where priority rules are transformed into predicates on interactions. We are convinced that our transformation can be easily adapted to these predicates.

4.4 Transformation of a BIP Model into a TT-BIP Model

In this section, we describe in details our technique for transforming a BIP model $B \stackrel{def}{=} \gamma(B_1, ..., B_n)$ into a TT-BIP model B^{TT} such that $B^{TT} = \gamma^{TT}(B_1^{TT}, ..., B_n^{TT}, TTCC_1, ..., TTCC_m, CRP)$.

One parameter to this transformation is the user-defined task mapping which consists in associating to each task T_k a group of atomic components of the model B. We denote by $\mathcal{B}$ the set of atomic components of model B. The task mapping is formally defined as follows:

Definition 4.2 (Task mapping)**.** *We assume, we have $K \leq n$ tasks and we denote by $\mathcal{T} = \{T_k\}_{k \in K}$ the task set, such that $\mathcal{T}$ is a partition of $\mathcal{B}$: where for all $j, k \in K$ and $j \neq k, T_j \cap T_k = \emptyset$. For all $k \in K$ we have $T_k = \{B_i\}_{i \in I_k}, I_k \subseteq K$ such that $\bigcup_{k \in K} I_k = K$.*

The transformation process is performed in two steps as shown in Figure 4.5. First, depending on the given task mapping, the original model is analysed in order to define the set of components and connectors to be transformed. Then, the BIP model is transformed into a TT-BIP model where only inter-task interactions and other related conflicting interactions are replaced by TTCC components. Non conflicting intra-task interactions remain intact. Components mapped to the same task are gathered in a composite task component.

We first present details about the analysis phase in Section 4.4.1. Then, we explain how concerned atomic components are transformed and how task components are instantiated in Section 4.4.2. Then we show how TTCC components are built in order to coordinate task components in Section 4.4.3. The behavior of the CRP component is detailed in Section 4.4.4. Finally, we define the cross-layer connections in Section 4.4.5.

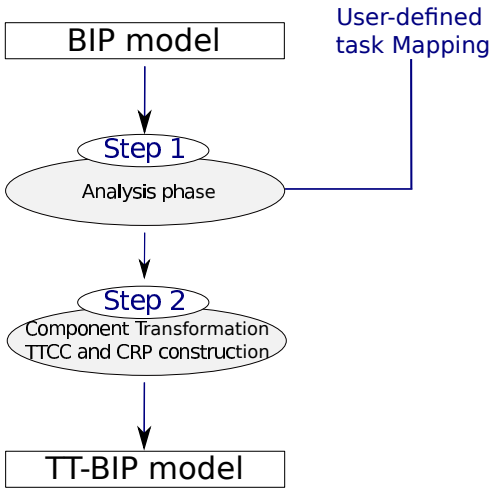


Figure 4.5: A two-step transformation

4.4.1 Analysis phase

We have first to identify internal and external interactions as well as ATC components denoted respectively A_I, A_E and $\mathcal{B}^{ATC}$. These obtained sets are inputs for the transformation of components and connectors of B into B^{TT}.

External interactions

In order to be able to define the set A_E, we need first to define the set of inter-task interactions denoted A_{IT}. An interaction $a \in \gamma$ is an inter-task interaction iff at least two of its participating components belong to two different tasks.
Formally,

$$A_{IT} = \{\alpha \in \gamma \,|\, \exists B_1, B_2 \in comp(\alpha), T_1, T_2 \in \mathcal{T} : B_1 \in T_1, B_2 \in T_2, T_1 \neq T_2\}.$$

We denote intra-task interactions that are either *directly* or *indirectly* conflicting with inter-task ones by $A^{\#}_{IT}$ defined as follows:

$$\begin{aligned} A^{\#}_{IT} = &\{a \in \gamma \,|\, a \notin A_{IT}, \exists \alpha \in A_{IT} : a \# \alpha\} \\ &\cup \{a \in \gamma \,|\, a \notin A_{IT}, \exists b \notin A_{IT}, \exists \alpha \in A_{IT} : a \neq b, a \# b, b \# \alpha\}. \end{aligned}$$

And we denote by A^{p}_{IT} the set of transitions labelled by internal ports and conflicting with interactions of $A^{\#}_{IT} \cup A_{IT}$. It is defined as follows:

$$A^{p}_{IT} = \{p \,|\, \forall a \in \gamma, p \notin P_a, \exists \alpha \in A_{IT} \cup A^{\#}_{IT}, q \in P_\alpha, \exists i \in [1, n], \exists l \in L_i : l \xrightarrow{p}, l \xrightarrow{q}\}.$$

As explained in Definition 4.1, A_E consists of inter-task interactions A_{IT}, intra-task interactions $A^{\#}_{IT}$ and internal transitions A^{p}_{IT} that are either *directly* or *indirectly* conflicting with inter-task ones. Thus, we have:

$$A_E = A_{IT} \cup A^{\#}_{IT} \cup A^{p}_{IT} \tag{4.1}$$

Internal interactions

The set A_I is defined as the set of intra-task interactions (i.e. participating components are belonging to the same task) which are neither *directly* nor *indirectly* conflicting with inter-task components:

$$A_I = \gamma \setminus A_E. \tag{4.2}$$

Atomic talking components (ATC)

$\mathcal{B}^{ATC}$ set is the set of atomic components in $\mathcal{B}$ that are concerned by external interactions A_E. We define:

$$\mathcal{B}^{ATC} = \{B \in \mathcal{B} | A_E \cap P_B \neq \emptyset\}, \tag{4.3}$$

where P_B is the set of ports of the component B.

4.4.2 Transformation of Task Components

We transform each ATC atomic component $B_i \in \mathcal{B}^{ATC}$ of a BIP model into a TT ATC component B_i^{TT} that is capable of communicating with $TTCC$ component(s). This transformation consists mainly in decomposing each "atomic" inter-task synchronization into send and receive actions. The synchronization between the ATC component (via the task interface) and the TTCC layer is implemented as a two-phase protocol.

First, B_i^{TT} sends communication *offers* through dedicated send ports. Then, in the second step, it waits for a notification coming from the TTCC component via a receive port. The communication *offer* contains information about the enabledness of the interaction. Each offer is associated to one of the enabled ports of B_i through which the component is ready to interact. An offer consists of a set of variables related to the corresponding enabled port. Let p be such port enabled from a location l (i.e. $l \xrightarrow{p}$). The set of variables of the corresponding offer includes variables initially exported by p since they may be read and written by the interaction. It also includes variables tc_p and tpc_l storing respectively timing constraint of transition labelled by p and enabled from l and the time progress condition of the location l. Another variable g_p is dedicated to store the evaluation of the Boolean guard of the transition labelled by p and enabled from l. The offer contains also a variable f_i storing the update function of the transition labelled by the port p. In order to be able to resolve conflicts, each offer contains the *participation count* variable nb of the component B_i^{TT}. This variable counts the number of interactions B_i^{TT} has participated in.

The notification —received after sending offers—allows the ATC component to execute the transition triggered by the enabled receive port marking the end of the interaction.

Notice that each offer —sent by a component—contains information about only one enabled interaction among the enabled interaction set. Therefore, if in the original model B, more than one interaction involving B_i are enabled, then B_i^{TT} has to send first successive offers before waiting for notification from the TTCC component executing the interaction selected after conflict resolution.

Let a location l, in B_i, from which $p_1, ..., p_n$ are enabled such that at least one of the n ports interferes in an inter-task interaction. In B_i^{TT}, we split such a location l into $n+1$ locations, namely l itself and locations $\{\perp_{p_i}^{l}\}_{i\in[1,n]}$ from which corresponding offers are sent (see Figure 4.6).

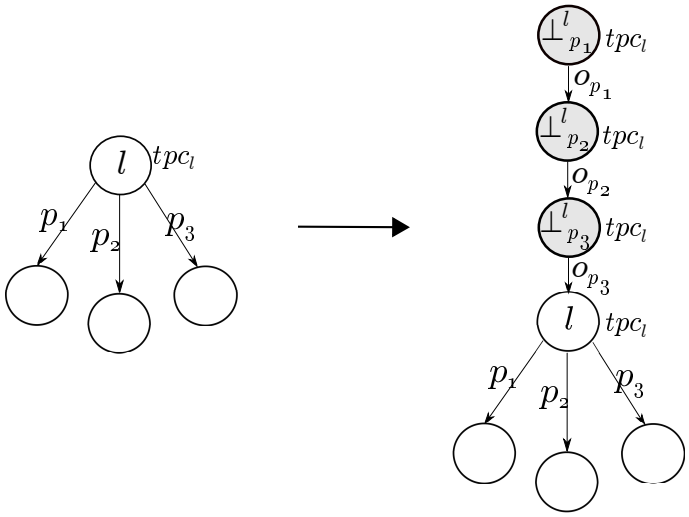


Figure 4.6: Atomic component transformation into an ATC component

Consider the case when, in the original model B_i, time is allowed to progress from location l, i.e. before executing the interaction. In order to enforce the correctness of the target model, time should be able to progress until the interaction is actually executed. Thus we associate to locations $\perp_{p_i}^{l}$ the time progress condition of location l originally defined in the atomic component B_i.

4.4.2.1 Expressing Timing Constraints and Time Progress Conditions over a Common Global Clock

In BIP framework, each atomic component can define its own local set of clocks. These clocks can be reset at any time and are used in definitions of timing constraints and time progress conditions.

In order to execute an external interaction $a = p_i, i \in I$, a TTCC component needs to evaluate the timing constraint of the interaction, i.e. the conjunction of timing constraints of transitions labelled by ports p_i involved in the interaction in the original model. These respective timing constraints are sent by respective ATC components to the TTCC layer within offers. In order to allow the TTCC to compute interactions between tasks components and schedule them correctly, we need to reduce the effort of keeping track of different clocks of participating components. This can be resolved by expressing timing

constraints in terms of a single time scale, that is, a single global clock. Moreover, the global time scale is a key feature of the TT paradigm targeted by the transformation.

For these two reasons, we need to translate all timing constraints and express them over the global clock.
We denote by c^g, the global clock which is initialized to 0 and measures the absolute time elapsed since the system started executing, i.e. c^g is never reset.

We follow a similar approach as in [2] in order to translate selected timing constraints. Here are the different translation steps:

1. for each component $B_i \in \mathcal{B}$ and for each clock $c \in C$, we introduce a variable w_c that stores the absolute time of the last reset of c. The variable w_c is initialized to zero and updated to the absolute time (i.e. the valuation of the global clock c^g) whenever the component executes a transition resetting clock c.

2. Each atomic expressions $lb \leqslant c \leqslant ub$ involved in a timing constraint tc, is rewritten by using the global clock c^g and the variable w_c. Mainly, we have to add to the initial lower and upper bounds the last reset value w_c of the local clock c as follows:

$$lb \leqslant c \leqslant ub \equiv lb + w_c \leqslant c^g \leqslant ub + w_c \tag{4.4}$$

3. Similarly, we rewrite each atomic expressions $c \leqslant ub$ of time progress conditions tpc —defined on all locations from which an external interaction can be enabled—as follows:

$$c \leqslant ub \equiv c^g \leqslant ub + w_c \tag{4.5}$$

Notice that the value of each local clock c can be computed from the current value of the global clock c^g and the variable w_c by using the equality $c = c^g - w_c$. This allows to entirely remove clocks of components B_i, keeping only the clock c^g and variables w_c; $c \in C$.

4.4.2.2 Formal transformation rule

Rule 4.1 (Transforming ATC components)**.** *Each ATC BIP component* $B_i = (L_i, P_i, X_i, C_i, T_i, tpc_i) \in \mathcal{B}^{ATC}$ *is transformed into a TT ATC component* $B_i^{TT} = (L_i^{TT}, P_i^{TT}, X_i^{TT}, C_i^{TT}, T_i^{TT}, tpc_i^{TT})$ *as detailed by the following rules:*

- *Each location* $l \in L_i$*, enabling ports* $\{p_j\}_{j\in[1,n]} \subseteq P_i \cap A_E$*, is split into* $n+1$ *locations. Obtained locations are* l *itself and partial-state locations* $\{\perp_{p_j}^{l}\}_{j\in[1,n]}$*. The time progress conditions of locations* $\perp_{p_j}^{l}$ *and* l *are equal to* $tpc(l)$*,*

- *Each port* $p_j \in P_i \cap A_E$ *such that* $l \xrightarrow{p_j}$ *is split into two ports; receive port* p_j *and send port* o_{p_j}*. A port* $p_j \in P_i^{TT}$ *exports variables* $X_{p_j} \subseteq X_i$ *originally exported by port* $p_j \in P_i$*. A port* o_{p_j} *exports, on top of variables* $X_{p_j} \subseteq X_i$*, variables tpc,* tc_p*,*

g_p, f_p and nb which are respectively the timing constraint variable, the time progress constraint variable, the Boolean guard variable, the update function variable and the participation count variable. These variables store respectively tpc of location l (i.e. $tpc(l)$) expressed on clock c^g, the timing constraint, the update function and the guard of transition enabled from l and labelled by p_j and the number of interactions the component has participated in.

- *For each clock $c \in C_i$, we add a corresponding variable w_c,*
- *For each transition $\tau_{p_j} = (l, p_j, g_{\tau_{p_j}}, tc_{\tau_{p_j}}, r_{\tau_{p_j}}, f_{\tau_{p_j}}, l')$, such that $\forall j \in [1, n]$, $l \xrightarrow{p_j}$ and $p_j \in P_i \cap A_E$, we include, in T_i^{TT}, the corresponding offer transition $\tau_{o_{p_j}}$ and notification transition τ'_{p_j}. The offer transition $\tau_{o_{p_j}}$ is enabled from location $\perp_{p_j}^{l}$. Both its guard and timing constraint are True. Its update function is the identity function and it resets no clock. It reaches location $\perp_{o_{p_k}}$ if $j \neq k$ and the offer o_{p_k} is not yet sent, otherwise it reaches location l. Notification transition τ'_{p_j} is enabled from location l and reaches location l'. As in the offer transition, guard and timing constraint of the notification transition are always True. It resets the same clock set as $r_{\tau_{p_j}}$. The update function $f_{\tau'_{p_j}}$ (1) updates the clock reset variables: $\forall c \in r_{\tau_{p_j}}$, $w_c = v_c(c^g)$, where v_c is the clock valuation function, (2) increments the participation count variable nb and (3) updates variables of offers sent from next reached state.*
- *For each transition $\tau_p = (l, p, g_{\tau_p}, tc_{\tau_p}, r_{\tau_p}, f_{\tau_p}, l')$, such that $p \in P_i \setminus A_E$, we instantiate the transition τ'_p, where only the update function is changed compared to the initial transition τ_p. The update function $f_{\tau'_p}$ (1) applies the original update function f_{τ_p}, (2) updates the clock reset variables: $\forall c \in r_{\tau_{p_j}}$, $w_c = v_c(c^g)$, where v_c is the clock valuation function, (3) increments the participation count variable nb and (4) updates variables of offers sent from next reached state.*
- *In order to update variables of offers that will be sent from its reached location l', a transition needs to execute the following functions:*
  - *$tpc := tpc(l')^{c^g}$, where $tpc(l')^{c^g}$ corresponds to expressing the tpc of l' over the global clock c^g following (4.5),*
  - *$\forall p \in P_i \cap A_E$, such that $\exists \tau_p = (l', p, g_{\tau_p}, tc_{\tau_p}, r_{\tau_p}, f_{\tau_p}, l") \in T_i$, $tc_p := tc_{\tau_p}^{c^g}$, $g_p = g_{\tau_p}$ and $f_p := f_{\tau_p}$, where $tc_{\tau_p}^{c^g}$ corresponds to expressing the timing constraint of τ_p over the global clock c^g following (4.4) and g_{τ_p} is the guard evaluation.*

After applying Rule 4.1, we can formally define the obtained component in function of the original one.

Definition 4.3. *Formally, B_i^{TT} is obtained from B_i as follows:*

- $L_i^{TT} = L_i \cup L_\perp$, *where* $L_\perp = \{\perp_p^l \mid \exists l \in L_i, \exists \tau = (l, p, g, tc, r, f, l') \in T_i, p \in P_i \cap A_E\}$,

- $P_i^{TT} = P_i \cup P_o$, *where* $P_o = \{o_p | p \in P_i \cap A_E\}$. *Each port* o_p *exports the set of variables* $X_{o_p}^{TT} = X_p \cup \{tpc, tc_p, g_p, f_p, nb\}$. *For all ports in* $p \in P_i$, *we have* $X_p^{TT} = X_p$,
- $X_i^{TT} = X_i \cup \{tpc\} \cup \{tc_p, g_p, f_p\}_{p \in P_i \cap A_E} \cup \{w_c\}_{c \in C_i} \cup \{nb\}$,
- $C_i^{TT} = \{c^g\}$,
- $T_i^{TT} = \{\tau_{o_p}\}_{p \in P_i \cap A_E} \cup \{\tau'_p\}_{p \in P_i}$. *Such that for each* $\tau_p = (l, p, g_{\tau_p}, tc_{\tau_p}, r_{\tau_p}, f_{\tau_p}, l') \in T_i$ *we have:*

$$\tau_{o_p} = (\perp_{o_p}^{l}, o_p, True, True, \emptyset, Id, \perp_{o_p}^{\prime l}) \qquad \text{if } p \in P_i \cap A_E$$
$$\tau'_p = (l, p, True, True, r_{\tau_p}, f_{\tau'_p}, l'),$$

  where $\perp_{o_p}^{\prime l}$ *is* l *or* $\perp_{o_q}^{l}$ *such that* $l \xrightarrow{q}$ *and* $f_{\tau'_p}$ *is as described in Rule 4.1.*

- *For places of* $L_\perp$, *the time progress condition* $tpc^{TT}(\perp_{o_p}^{l}) = tpc(l)$.

Example 4.1. *Figure 4.7 illustrates transformation of an ATC component into its corresponding ATC TT component. In this example we consider that ports* p *and* q *are participating in external interactions.*

Once all ATC components are transformed, we instantiate the composite component of each task, which corresponds to gathering all components mapped to that task and exporting send and receive ports of ATC components (see Rule 4.2).

Rule 4.2. *For each* $T_j \in \mathcal{T}$ *we instantiate a composite component* $B_{T_j}^{TT}$ *including:*

- *Component* $B_i \in T_j$ *if* $B_i \notin \mathcal{B}^{ATC}$ *and* B_i^{TT} *if* $B_i \in \mathcal{B}^{ATC}$,
- *Interactions* $\{\alpha \in \gamma \cap A_I \,|\, \forall p \in \alpha, \exists B_i \in T_j : p \in P_i\}$, *where* P_i *is the set of ports of* B_i.
- *The set of exported ports* $\{(p, o_p) \,|\, \exists B_i \in \mathcal{B}^{ATC} \cap T_j : p \in P_i \cap A_E\}$.

4.4.3 Building TTCC Components

As explained before, a TTCC component layer is introduced initially in order to handle intertask interactions and thus model the TT communication system. By considering the need for operational equivalence (i.e. keeping the same original behavior), and in order to be able to resolve all conflicts of the target model interactions, the TTCC layer handles, on top of intertask interactions, other interactions that are conflicting *directly or indirectly* with these latter. Recall that all interactions of the original model, that are handled in the TTCC layer are called external interactions.

Initially, all components are doing their initial computations and the TTCC layer does not know their state or their enabled communication ports until they send offers.

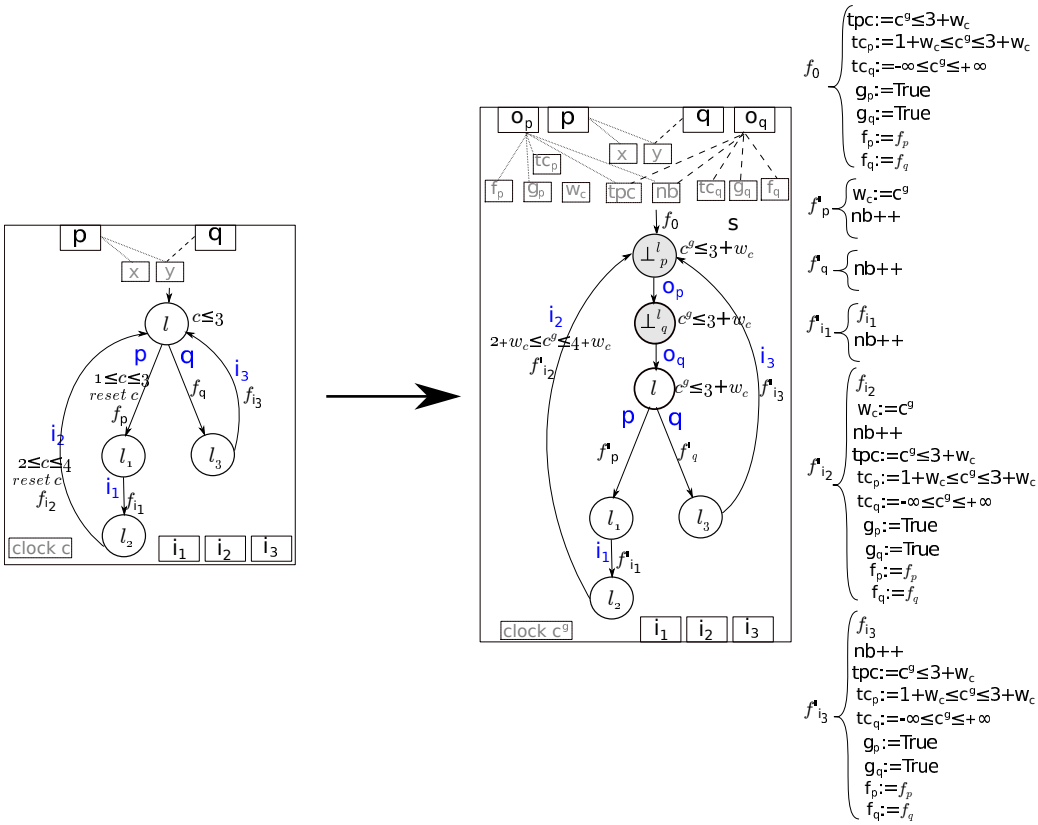


Figure 4.7: Example of transformation of an ATC component

Handling only one external interaction, a TTCC can execute this latter only when all participating tasks' components have sent their offers and are ready to execute the interaction.

Since in the input model we assume that no priority rules can be established between external interactions, a TTCC component does not need to connect with tasks participating in interactions other the one it is handling. Since the enabledness of its interaction only depends on offers received from its participating tasks components. When the interaction is conflicting with another external interaction, the TTCC has to communicate, after checking the enabledness of the interaction, with the CRP in order to get the permission or not to execute. We call this communication a reservation mechanism.

To summarize, the behavior of a TTCC component handling an interaction $a = (a, G_a, F_a) \in \gamma$ is made of three steps: (1) it waits for offers from its participating task components, (2) once all offers are received —regardless their order, the TTCC component takes a decision by either executing the interaction upon synchronization (i.e., conjunction of received guards and G_a evaluates to *True*) if a is a non-conflicting interaction or soliciting the CRP component to find out if the conflicting interaction a can be executed and (3) finally it writes on appropriate task components by sending a

notification.

Figure 4.8 shows a representative part of a TTCC automaton, where we can distinguish the three steps. From location *wait*, the TTCC is waiting for respective offers from its participating components. Since these offers can be received in a random order, the TTCC is designed in such a way to allow all possible combination from location *wait*. Once all offers are received, the location *read* is reached. From this location, the TTCC starts the second step in order to execute the interaction depending on whether it is conflicting or not. Once the TTCC executes the interaction, the automaton reaches location *send* from which it executes a transition allowing to notify participating components and reaches back the location *wait*. All transitions of the first step are triggered

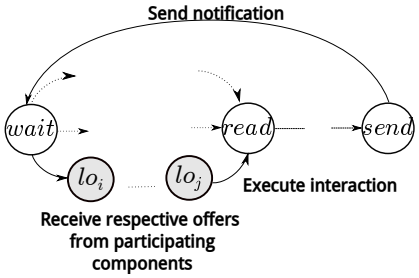


Figure 4.8: Skeleton of a TTCC automaton

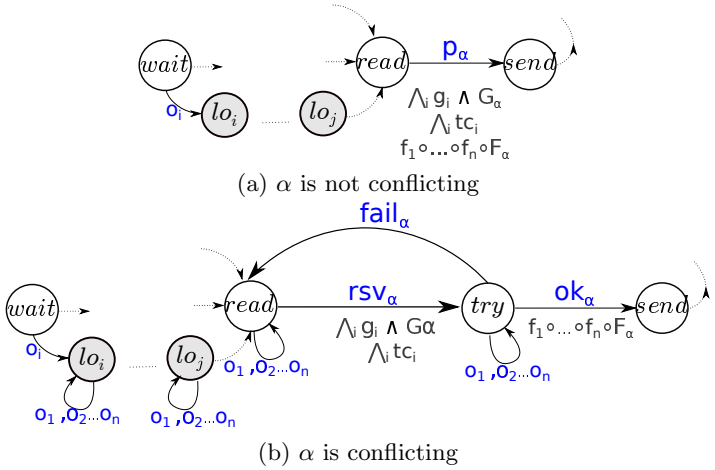
by receive ports corresponding to respective offers. The transition of the third step is triggered by a send port. Behaviour and ports triggering transitions of the second step are detailed later.

Let a TTCC component handling an external interaction $\alpha = (P_\alpha, G_\alpha, F_\alpha) \in \gamma \cap A_E$. We denote by n the number of components related to TTCC, i.e. the number of participating components of α, i.e. $n = |comp(\alpha)|$.

In the case when α is a non-conflicting interaction, the execution of this latter is performed without requesting the CRP component. As shown in Figure 4.9a, the TTCC executes a transition from location *read* to *send* labelled by a unary port denoted p_α. Its update function executes the update function F_α of the interaction α and then respective update functions that are received in offers. The transition p_α is guarded by the conjunction of the guard G_α and respective guards and timing constraints received in offers. If the conjunction of these guards evaluates to *True*, the interaction is executed and the TTCC sends a notification to participating components.

In the case when α is conflicting with another interaction, the TTCC goes through a reservation mechanism (cf. Figure 4.9b). If the interaction is enabled, i.e. the conjunction of the guard G_α and respective guards and timing constraints received in offers evaluates to *True*, the TTCC executes transition rsv_α from location *read*. This transition reaches location *try*. By the execution of rsv_α, a reservation request is sent to the CRP component. This reservation contains different values of participation count variables of α participating components. Based on these participation counters, the CRP

decides whether to allow or disallow the interaction execution. It notifies the TTCC component either through port ok_α in the case when the reservation succeeds or through port $fail_\alpha$ if the reservation can not be made. While waiting for CRP notification, the TTCC occupies the location *try*. If the port ok_α is enabled, then it executes the transition reaching location *send* from which notification to components are ready to be sent. Note that update function F_α composed with those of received offers is associated with the transition labelled by the ok_α port. If the port $fail_\alpha$ is enabled, the TTCC reaches back the location *read* in order to proceed again for the reservation.



(a) α is not conflicting

(b) α is conflicting

Figure 4.9: Mechanisms for execution of interaction $\alpha = (P_\alpha, G_\alpha, F_\alpha)$

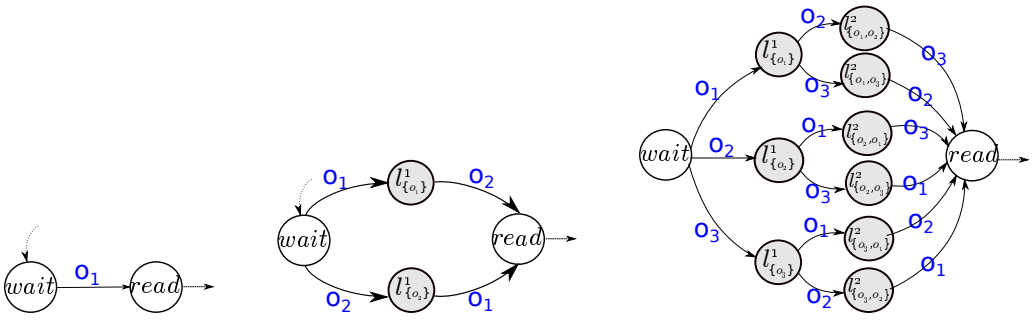
When an ATC component is participating in two conflicting interactions α_1 and α_2, it sends successively offers to each of the corresponding TTCC components $TTCC_{\alpha_1}$ and $TTCC_{\alpha_2}$ and waits from a notification from one of them. After resolving the conflict by requesting the CRP, suppose $TTCC_{\alpha_1}$ will notify the component after successfully executing the interaction α_1, while $TTCC_{\alpha_2}$ reaches back its location *read* in order to proceed to a new reservation attempt. The component is able to continue execution of its next transitions. And it may reach again the location allowing to send again offers to $TTCC_{\alpha_1}$ and $TTCC_{\alpha_2}$. Both TTCC components should be ready to receive the offers. For that, we add loop transitions in TTCC automata labelled by offers receive ports over locations *read* and *try*. Furthermore, such an ATC component may need to resend an offer to a TTCC even before this latter receives other offers from the rest of its participating components. This is resolved by adding loop transitions labelled by offer receive ports over locations that are placed between location *wait* and *read* (cf. Figure 4.9b). These added loop transitions allow to respect the last point of Definition 4.1 stating that whenever a send port is activated, all its receive ports are enabled as well.

4.4.3.1 Formal Transformation rule

In the following, we explicit the transformation rule allowing to instantiate a TTCC component for each external interaction.

Rule 4.3. *Each external interaction* $\alpha = (P_\alpha, G_\alpha, F_\alpha) \in \gamma \cap A_E$, *such that* $P_\alpha = \{p_i\}_{i\in[1,n]}$, *and* $comp(\alpha) = \{B_i\}_{i\in[1,n]}$, *is transformed into a TTCC component* $TTCC = (L^{TTCC}, P^{TTCC}, X^{TTCC}, C^{TTCC}, T^{TTCC}, tpc^{TTCC})$:

- *Ports and variables:*
  - *For each port* $p_i \in P_\alpha$, *we include in* P^{TTCC} *a receive port* o_{p_i}. *For each port* o_{p_i} *we associate a local copy of the set of variables* X_{p_i} *initially exported by port* p_i *of component* B_i. *We associate also to* o_{p_i} *the time progress condition variable* tpc_i, *the timing constraint variable* tc_{p_i}, *the Boolean guard variable* g_{p_i}, *the update function variable* f_{p_i} *and the participation count variable* nb_i.
  - *We include also one send port* p_s^α *in* P^{TTCC}. *To the port* p_s^α, *we associate sets of local variables* X_{p_i}, $p_i \in P_\alpha$.
  - *If* α *is not conflicting, then we include a unary port denoted* p_α, *which allows to label the transition executing the interaction. Otherwise, we include in* P^{TTCC} *one send port* rsv_α *and two receive ports* ok_α *and* $fail_\alpha$. *Only port* rsv_α *has associated variables, which are participation count variables* nb_i *for all* $i \in [1, n]$, *i.e. all participation count variables of participating components* $\{B_i\}_{i\in[1,n]}$
- *Clock: As explained before, the TTCC component defines only one clock which is the global clock denoted* c^g.
- *Locations:*
  - *We include in* L^{TTCC} *location* wait *marking thee beginning of offer reception, location* read *marking the reception of all offers and the location* send *marking the end of interaction execution. If* $n \geq 2$, *we include —between location* wait *and* read*—the set of intermediate waiting locations* $L_\perp$ *allowing reception of offers in any order. Let* $\mathcal{O} == \{o_{p_i} \,|\, p_i \in P_\alpha, i \in [1, n]\}$ *be the set of all offers received by TTCC. The set* $L_\perp$ *is constructed as follows;* $L_\perp = \{l^k_{O_k} \,|\, k \in [1, n-1], O_k \in \mathcal{P}_{k(\mathcal{O})}\}$, *where* $\mathcal{P}_k(\mathcal{O})$ *is the k-permutation of* $\mathcal{O}$, *allowing to indicate the ordered subset of offers sent before reaching the location* $l^k_{O_k}$. *Note that the cardinality of* $L_\perp$ *is* $|L_\perp| = \sum_{k=1}^{n-1} \frac{n!}{(n-k)!}$. *Figure 4.10 shows how intermediate waiting locations (displayed in gray) are constructed for* $n = 2$ *and* $n = 3$. *Its shows also the case when* $n = 1$, *where no intermediate waiting location is needed.*
  - *If* α *is conflicting, we introduce in* L^{TTCC} *the location try allowing the reservation mechanism.*



(a) n=1 (b) n=2 (c) n=3

Figure 4.10: Intermediate waiting locations

- *The time progress condition of location* wait *is set to True. The time progress condition of location send is False. In the case of a conflicting TTCC, the time progress condition of its try is True. For location* read*, the time progress condition is set to the conjunction of time progress conditions received in the offers. That is, after receiving offers from participating components, we require that the TTCC component executes its interaction before different time progress conditions of participating components become False.*

- *Transitions:*

  - *In order to receive offers from task components B_i, we include receiving transition, we have three classes of receiving transitions; the n transitions starting from location wait and labelled each by an offer port, transitions between locations $L_\perp$ and transitions reaching the location read. They are respectively as follows:*

$$\begin{aligned}
\tau_{o_{p_i}} &= (wait, o_{p_i}, True, True, \emptyset, Id, l^1_{O_1}), && \forall O_1 \in \mathcal{P}_1(\mathcal{O}) :\ o_{p_i} \in O_1,\\
\tau_{o_{p_i}} &= (l^k_{O_k}, o_{p_i}, True, True, \emptyset, Id, l^{k+1}_{O_{k+1}}), && \forall k \in [1, n-2] :\ O_k \subsetneq O_{k+1}, o_{p_i} \in O_{k+1} \setminus O_k,\\
\tau_{o_{p_i}} &= (l^{n-1}_{O_{n-1}}, o_{p_i}, True, True, \emptyset, Id, read), && \forall O_{n-1} \in \mathcal{P}_{n-1}(\mathcal{O}) :\ o_{p_i} \notin O_{n-1}.
\end{aligned}$$

    These transitions' guards and timing constraints are default to True, their update functions are the identity function and they does not reset clocks.

  - *If α is conflicting, the set of transitions includes loop waiting transitions as already explained, for each $l^k_{O_k} \in L_\perp$, we include k loop transitions labelled each by an offer port $o_{p_i} \in O_k$. That is, for each $l^k_{O_k} \in L_\perp$, and for each $o_{p_i} \subsetneq O_k$, we include the transition $\tau^{l^k_{O_k}}_{o_{p_i}} = (l^k_{O_k}, o_{p_i}, True, True, \emptyset, Id, l^k_{O_k})$. we add also loop transitions on locations* read *and* try*, i.e. for each $o_{p_i} \in \mathcal{O}$, we add $\tau^{read}_{o_{p_i}} = (\text{read}, o_{p_i}, True, True, \emptyset, Id, \text{read})$ and $\tau^{try}_{o_{p_i}} = (\text{try}, o_{p_i}, True, True, \emptyset, Id, \text{try})$. These transitions allow components participating in conflicting interactions that have already sent their offer to be able to send it again.*

- *To notify task components after executing the interaction α, we include the transition $\tau_{send} = (send, p_s^\alpha, True, True, Identity, \emptyset, wait)$.*
- *If α is not conflicting, we include the transition $\tau_\alpha = (read, p_\alpha, G^*, TC^*, \emptyset, F^*, write)$, where the port p_α is a unary port, $G^* = G_\alpha \bigwedge(\bigwedge_{i=1}^{n} g_{p_i})$, $TC^* = \bigwedge_{i=1}^{n} tc_{p_i}$, $F^* = f_{p_1} \circ ... \circ f_{p_n} \circ F_\alpha$ such that G_α and F_α are respectively the guard and the update function of the initial interaction α, g_{p_i}, tc_{p_i} and f_{p_i} are respectively the guard, the timing constraint and the update function of offer o_{p_i}.*
- *If α is conflicting, we include transitions allowing the reservation mechanism:*
  $\tau_{rsv} = (read, rsv, G^*, TC^*, \emptyset, Id, try)$,
  $\tau_{ok} = (try, ok, True, True, \emptyset, F^*, send)$,
  $\tau_{fail} = (try, fail, True, True, \emptyset, Id, read)$, where G^*, TC^* and F^* are as detailed in the previous item.

Example 4.2. *In Figure 4.11 (resp. Figure 4.12), we illustrate transformation of a conflicting (resp. non conflicting) external interactions α into its corresponding TTCC component. In these examples we consider that ports p and q of the interaction α are exporting respectively variables x_p and x_q.*

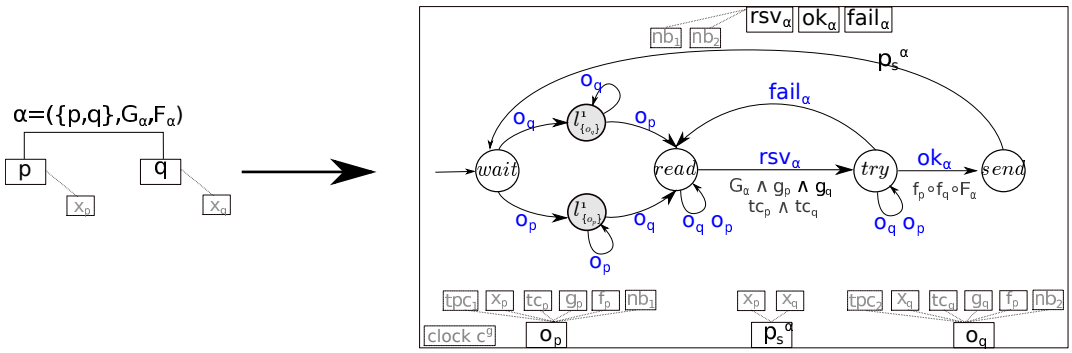


Figure 4.11: Example of transformation of a conflicting external interaction into a TTCC component

4.4.4 Conflict Resolution Protocol Component

The conflict resolution protocol (CRP) that we use in our work is the same CRP used in [47, 77, 86]. It is, so far, the unique solution to guarantee the resolution of conflicts without requesting the BIP execution engine. It accommodates the algorithm proposed in [10]. It uses message counts to ensure synchronization and reduces the conflict resolution problem to dining or drinking philosophers [32]. Its main role is to check the

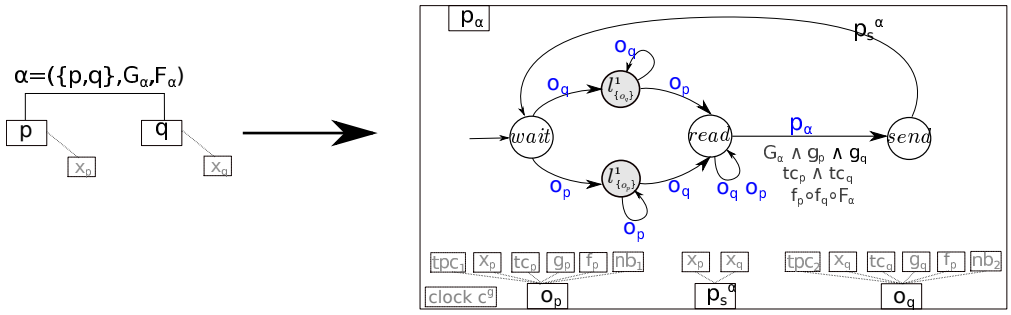


Figure 4.12: Example of transformation of a non-conflicting external interaction into a TTCC component

freshness of requests received for an interaction, that is, to check that no conflicting interactions have been already executed using the same request. In each request, an interaction sends the participation numbers of its components, i.e. number of interactions each ATC component has participated in. This ensures that two conflicting interactions cannot execute with the same request. Mutual exclusion is ensured using participation numbers. To this end, the conflict resolution protocol keeps the last participation number NB_i of each component B_i and compares it with the participation number nb_i provided along with the reservation request from TTCC components. If each participation number from the request is greater than the one recorded by the conflict resolution protocol ($nb_i > NB_i$), the interaction is then granted to execute and NB_i is updated to nb_i. Otherwise, the interaction execution is disallowed.

4.4.4.1 Formal Transformation rule

As explained in Chapter 3 Section 3.2, the CRP behaviour is expressed by a set of parallÃÍle automata handling each an interaction (cf. Figure 3.2).

In the following, we explicit the rule allowing to instantiate a CRP component based on this same formalism.

Rule 4.4. *Given the model* $B \stackrel{def}{=} \gamma(B_1, ..., B_n)$*, we instantiate the component* $CRP = (L^{CRP}, P^{CRP}, X^{CRP}, C^{CRP}, T^{CRP}, tpc^{CRP})$ *where:*

- X^{CRP} *contains the last used offer variable* N_i *for each* $B_i \in comp(\alpha)$ *where* $\alpha \in A_E$*,*
- $C^{CRP} = c^g$*,*
- *For each externally conflicting* $\alpha \in A_E$*,*
  - L^{CRP} *contains the waiting place* w_α *where* $tpc(w_\alpha) = True$ *and the reservation place* r_α *where* $tpc(r_\alpha) = False$*,*

- P^{CRP} *contains the ports* rsv_α, ok_α *and* $fail_\alpha$,
- X^{CRP} *contains the participation numbers* $\{nb_i^\alpha \mid B_i \in comp(\alpha)\}$. *These variables are associated to the port* rsv_α. *Ports* ok_α *and* $fail_\alpha$ *do not have associated variables.*
- T^{CRP} *contains the following three transitions;* $\tau_{rsv_\alpha} = (w_\alpha, rsv_\alpha, r_\alpha)$, $\tau_{ok_\alpha} = (r_\alpha, ok_\alpha, w_\alpha)$ *and* $\tau_{fail_\alpha} = (r_\alpha, fail_\alpha, w_\alpha)$. *The transitions* τ_{rsv_α} *and* τ_{fail_α} *has no guard, no timing constraint and no update function. The transition* τ_{ok_α} *has no timing constraint but is guarded by* $G_{\tau_{ok_\alpha}} = \wedge_{B_i \in comp(\alpha)} nb_i^\alpha > NB_i$. *Its update function sets the variables* NB_i *of components* $B_i \in comp(\alpha)$ *to the values of corresponding participation numbers* nb_i^α*: i.e. for each* $B_i \in comp(\alpha)$, *it performs* $NB_i := nb_i^\alpha$.

4.4.5 Cross-layer interactions

In this section, we define the interactions between the task components and the TTCC layer and between this latter and the CRP component. Tasks and TTCC components exchange offers and notifications. Communication between TTCC components and the CRP component involves the transmission of messages corresponding to rsv, ok and $fail$ (cf. Rule 4.5). In the following rule, and for clarity of presentation, we use the notation $B.p$ to denote the port p of the component B.

Rule 4.5. *Let* $B \stackrel{def}{=} \gamma(B_1, ..., B_n)$ *be a BIP model,* $\mathcal{T}$ *be a task mapping. We define the obtained model after transformation as* $B^{TT} = \gamma^{TT}(B_1^{TT}, ..., B_n^{TT}, TTCC_1, ..., TTCC_m, CRP)$. *The send/receive interactions of* γ^{TT} *are defined as follows:*

- *For each task component* $B_{T_j}^{TT}$ *such that* $T_j \in \mathcal{T}$, *for each port* $B_{T_j}^{TT}.o_p$ *and each* $TTCC_\alpha$ *such that* $p \in \alpha$, *we include in* γ^{TT} *the offer interaction based on ports* ($B_{T_j}^{TT}.o_p$, $TTCC_\alpha.o_p$). *Its guard is set to True. And its update function copies variables associated with* $B_{T_j}^{TT}.o_p$ *to those of the receive port* $TTCC_\alpha.o_p$.

- *For each* $TTCC_\alpha$, *and all* $\{B_{T_j}^{TT}\}_{j \in J}$, *such that for all* $j \in J$, $T_j \cap comp(\alpha) \neq \emptyset$, *we include the notification interaction based on ports* ($TTCC_\alpha.p_s^\alpha$, $\{B_{T_j}^{TT}.p_j\}_{j \in J}$), *where for all* $j \in J$, $p_j \in \alpha$. *Its guard is set to True. And its update function copies variables associated with* $TTCC_\alpha.p_s^\alpha$ *to those of the receive ports* $B_{T_j}^{TT}.p_j$.

- *For each interaction* $\alpha \in \gamma$ *that is not conflicting, we include the unary interaction having as unique port* ($TTCC_\alpha.p_\alpha$), *where* $TTCC_\alpha$ *is the TTCC component handling the interaction* α. *Its guard is set to True. And its update function is the identity function.*

- *For each interaction* $\alpha \in \gamma$ *that is conflicting, we include a triplet of interactions having respectively the following sets of ports:* ($TTCC_\alpha.rsv_\alpha$, $CRP.rsv_\alpha$),

($CRP.ok_\alpha$, $TTCC_\alpha.ok_\alpha$) and ($CRP.fail_\alpha$, $TTCC_\alpha.fail_\alpha$). All their guards are set to True. The update function of the former interaction copies variables of ports $TTCC_\alpha.rsv_\alpha$ to port $CRP.rsv_\alpha$. Since ports $CRP.ok_\alpha$ and $CRP.fail_\alpha$ do not have any associated variables, the update function of the last two interactions is the identity function.

4.5 Transformation Correctness

In this section, we show that the described transformation is correct, that is the obtained TT-BIP model is observationally equivalent to the original BIP model. Before proving the observational equivalence, we show that the final model is a valid TT-BIP model.

4.5.1 Validity of the Obtained Model

Proposition 4.1. *Given a BIP model $B = \gamma(B_1, ..., B_n)$ and a task mapping $T = \{T_1, ..., T_k\}$, the model $B^{TT} = \gamma^{TT}(B_1^{TT}, ..., B_n^{TT}, TTCC_1, ..., TTCC_m, CRP)$ obtained by transformation of Section 4.4 meets the properties of Definition 4.1.*

Proof. **Points 1-3 of Definition 4.1**
The first three criteria of Definition 4.1 are syntactic, namely only allowed interactions are either classic multiparty interactions or send/receive interactions or unary interactions and each send port participates in exactly one Send/Receive interaction. These criteria are met by the previous definition.

Point 4 of Definition 4.1
The fourth point of Definition 4.1, enumerates all conflict cases of a TT-BIP model. The first case states that an internal port can only be conflicting with a similar port. By construction of the transformation, internal ports are instantiated only in task components (cf. Rule 4.1). If an internal transition is originally conflicting with a similar transition then this conflict is preserved, since these transitions remain intact after transformation. If in the original model, an internal transition is conflicting with an external transition then this port will be replaced by a send and receive ports. Therefore, the original conflict is no more existing in TT-BIP.

The second case involves receive ports. In task components, by construction of the transformation (cf. Rule 4.1), a receive port can be only conflicting with receive port. In TTCC component, receive transitions are offer transitions or *ok/fail* transitions. *Ok* transitions and *fail* transitions have the same source location. Similarly, offer transitions can be also enabled from the same location (in the case of conflicting TTCC component). They also can be conflicting with a send transition labelled by an rsv_α port (cf. Rule 4.3). In CRP component, receive transitions are *rsv* transitions which are enabled from the initial location only simultaneously with other rsv transitions. Therefore, in

all components, a receive transition can be enabled simultaneously either with a receive port or with a send port or both.

The third case involves send ports. In task components send ports are *offer* ports and by construction of the transformation (cf. Rule 4.1) only one send port is enabled from one location. In TTCC components, send ports are either p_s^α ports (sending notifications to task components) or rsv_α ports. The former has no conflicting port (i.e. no other port is enabled from its source location) while the latter is enabled from the same location as receive ports (offer ports) (cf. Rule 4.3). In CRP component, send ports are *ok* or *fail* ports. Note that these ports are enabled from the same location. Therefore we deduce that a send port can have the same source location as a receive or other send ports.

Point 5 of Definition 4.1
The fifth point of Definition 4.1 states that the update function of a transition labelled by a send port does not involve variables exported by this port. In task components, send ports are *offer* ports and they trigger transitions whose update functions are the identity function (cf. Rule 4.1). In TTCC components, the send port is either a p_s^α or a rsv_α port. In both cases, it labels a transition with an identity update function (cf. Rule 4.3). In the CRP component, send port can be either an *ok* or *fail* port. In the first case, the port labels a transition whose update function applies on NB_i variables which are not exported. In the second case, the port labels a transition with an identity update function.

Point 6 of Definition 4.1
The second-last point in Definition 4.1 states that a transition labelled by a receive port always has a timing constraint and guards that are default to *True*. In the layer of task components, receive ports label only notification transitions which, by construction, are associated with a timing constraint and guard equal to *True* (cf. Rule 4.1). In the TTCC layer, receive ports label either *offer* transitions or *ok/fail* transitions. These latter are also associated with a timing constraint and guard always default to *True*(cf. Rule 4.3). In the third layer (i.e. the CRP component), receive ports label rsv transitions, which are also associated with timing constraint and guard always equal to *True*.

Point 7 of Definition 4.1
The last criterion of Definition 4.1 states that whenever a send port is enabled, the associated receive ports will unconditionally become enabled within a finite number of transitions in the receiver component. Intuitively, this holds since communications between tasks and TTCC components, and between TTCC components and CRP component follow a request/acknowledgement pattern. Whenever a component sends a request (via a send port) it enables the receive port to receive acknowledgement.

In the following, we detail different configuration cases:

- Communications between a task component B_i^{TT} and a $TTCC_j$ component, for all interactions α involving a component B_i. We denote by $l_{B_i^{TT}}$ the enabled location of B_i^{TT} and by l_{TTCC_j} the active place of $TTCC_j$. We distinguish the following cases:

Case 1: $l_{B_i^{TT}} = \perp_p^l$ where p is exported by B_i and $l_{TTCC_j} \in \{wait\} \cup L_\perp$.

In this configuration, the only enabled send-port involved in a send/receive interaction is the offer port o_p of B_i^{TT}. Note that the initial state allowing a send/receive interaction between tasks and TTCC components falls in that case. By definition of the configuration, all associated receive ports are also enabled (the $TTCC_j$ component can only execute transitions labelled by receive ports).

Case 2: $l_{B_i^{TT}} = l$ where l is a place of B_i and $l_{TTCC_j} = \{read\}$.

This configuration is reached from the first one by executing offer transitions. From this configuration, no send/receive interaction with the task components can be enabled (i.e. no send port is enabled). To send offers, the task component should be in a $\perp_p^l$ location which is not the case.

Case 3: $l_{B_i^{TT}} = l$ where l is a place of B_i and $l_{TTCC_j} = \{send\}$.

In this case, the component B_i^{TT} is still in a place l that is not a busy location, and the $TTCC_j$ component is in the *send* place. From that configuration, the enabled send-port that is involved in a send/receive interaction with B_i^{TT} is the port p_s^α of the TTCC component. By definition of the configuration, the receive port associated to this send-port is the one activated from place l of component B_i^{TT}. Thus, the property holds in that configuration as well. Note that after executing the send/receive interaction with the component B_i^{TT}, the first configuration is reached back.

- Communications between a conflicting $TTCC_j^C$ component with the CRP component, for all conflicting interaction α involving a component B_i. We denote by $l_{TTCC_j^C}$ the enabled location of $TTCC_j^C$ and by l_{CRP} the active set of marked places of CRP. We distinguish the following cases:

  Case 1: $l_{TTCC_j^C} = read$ and $l_{CRP} \ni \{w_\alpha\}$.

  In this case, the unique enabled send-port is the port rsv_α of the component $TTCC_j^C$. And by definition of the configuration, the associated receive port of this send-port is enabled, i.e. the port rsv_α of component CRP is enabled from place w_α. Thus, the property holds in that configuration as well.

  Case 2: $l_{TTCC_j^C} = try$ and $l_{CRP} \ni \{r_\alpha\}$.

  This case is reached by executing the reservation interaction from the previous configuration. In this case, two send-ports are active, ok_α and $fail_\alpha$ of the component CRP. From the enabled location of $TTCC_j^C$ component, the corresponding receive ports associated to these two send-ports are enabled as well. Thus, the property holds by-construction in that configuration as well.

□

This proof ensures that any component ready to perform a transition labelled by a send-port will not be blocked by waiting for the corresponding receive-ports.

4.5.2 Observational Equivalence Between B and B^{TT}

We denote by $B = \gamma(B_1, ..., B_n)$ the initial model and by $B^{TT} = \gamma^{TT}$ $(B_1^{TT}, ..., B_n^{TT}, TTCC_1, ..., TTCC_m, CRP)$ the resulting model of the first step of the transformation.

In order to prove the correctness of the transformation from B to B^{TT}, we have to show that their corresponding semantic LTSs are observationally equivalent. We denote by $G(B)$ and $G(B^{TT})$ successively the LTSs of B and B^{TT} (see Definition 1.14).

We define observational equivalence between transition systems based on the classical notion of weak bisimilarity [69], where some transitions are considered unobservable.

We will use the following notation. Consider a binary relation $R \subseteq X \times Y$. For $x \in X$, we denote $R(x) \stackrel{def}{=} \{y \in Y \mid (x, y) \in R\}$.

Definition 4.4. *(LTS relations) Let $A = (Q_A, P_A, \underset{A}{\longrightarrow})$ and $B = (Q_B, P_B, \underset{B}{\longrightarrow})$ be two LTS. Given a relation $\beta \subseteq P_A \times P_B$, we write $q \xrightarrow[A]{\beta} q'$, for $q \in Q_A$, iff there exists $a \in P_A$, such that $q \xrightarrow[A]{a} q'$ and a is* not *related by β to any label in P_B, i.e. $\beta(a) = \emptyset$. The notation $q \xrightarrow[B]{\beta} q'$, for $q \in Q_B$, is defined symmetrically.*

A weak simulation *over A and B, is a pair of relations $R \subseteq Q_A \times Q_B$ and $\beta \subseteq P_A \times P_B$, such that:*

$$\forall (q, r) \in R, \forall a \in P_A, \left(\beta(a) \neq \emptyset \wedge q \xrightarrow[A]{a} q' \Longrightarrow \exists (a, b) \in \beta : \exists (q', r') \in R : r \xrightarrow[B]{\beta^* b \beta^*} r' \right)$$

and

$$\forall (q, r) \in R, \left(q \xrightarrow[A]{\beta} q' \Longrightarrow \exists (q', r') \in R : r \xrightarrow[B]{\beta^*} r' \right),$$

where β^* denotes zero or more successive β transitions (i.e. transitions whose label is not related by the relation β).

A weak bisimulation *over A and B is a pair of relations $R \subseteq Q_A \times Q_B$ and $\beta \subseteq P_A \times P_B$, such that both (R, β) and (R^{-1}, β^{-1}) are weak simulations. Recall that $R^{-1} \subseteq Q_B \times Q_A$ and $\beta^{-1} \subseteq P_B \times P_A$ are the symmetric relations of R and β.*

We say that A and B are weakly bisimilar w.r.t. *$\beta \subseteq P_A \times P_B$, denoted $A \sim_\beta B$, if there exists $R \subseteq Q_A \times Q_B$ total on both Q_A and Q_B, such that (R, β) is a weak bisimulation.*

First, we need to establish correspondence between labels of $G(B)$ (ranging over the set $\gamma \cup \mathbb{R}_+$) and those of $G(B^{TT})$ (ranging over the set $\gamma^{TT} \cup \mathbb{R}_+$). Therefore, we define the relation β as follows:

$$\beta = \left\{ (\alpha, \alpha) \mid \alpha \in \gamma \cap A_I \right\} \cup \left\{ (\alpha, p_s^\alpha) \mid \alpha \in \gamma \cap A_E \right\}, \tag{4.6}$$

where p_s^α is the send port of the TTCC component allowing to send notifications to its related components.

Note that by this relation, we can say that each transition $\alpha \in \gamma$, is represented in γ^{TT} either by the transition α itself if it is internal, or by p_s^α if it is external. Transitions of B that are not related by the relation β are only delay transitions. And transitions of B_{TT} that are not related by the relation β are offer , reserve, fail, ok and p_α transitions. These transitions are denoted by β transitions.

We may use later in this proof the following notations $fail_\alpha$ and ok_α (resp. rsv_α) to denote the fail and ok (resp. reservation) interactions between the CRP and the TTCC component handling interaction α in B^{TT} model.

Theorem 4.1. *The LTSs $G(B)$ and $G(B^{TT})$ are weakly bisimilar w.r.t. β, i.e. $G(B) \sim_\beta G(B^{TT})$.*

Proof. Let $G(B) = (Q_B, P, \xrightarrow[B]{})$ and $G(B^{TT}) = (Q_{B_{TT}}, P_{B_{TT}}, \xrightarrow[B^{TT}]{})$. Recall (Definition 1.11) that state spaces Q_B and $Q_{B_{TT}}$ have each three components: control location, clock and variable valuations. For a given state q, we will denote $v_c(q)$ (resp. $v_x(q)$) its clock (resp. variable) valuation component. Similarly, we denote $l(q)$ the location of a state q.

Below, we will use variables q_B, r_B, ranging over Q_B, and $q_{B_{TT}}, r_{B_{TT}}$, ranging over $Q_{B_{TT}}$ and denote their respective components as follows:

$$q_B = \big(l, v_x(q_B), v_c(q_B)\big)\,, \qquad r_B = \big(l', v_x(r_B), v_c(r_B)\big)\,,$$
$$q_{B_{TT}} = \big(l_{TT}, v_x(q_{B_{TT}}), v_c(q_{B_{TT}})\big)\,, \qquad r_{B_{TT}} = \big(l'_{TT}, v_x(r_{B_{TT}}), v_c(r_{B_{TT}})\big)\,.$$

For clarity reasons, for each state $q_{B_{TT}}$, we detail the control location l_{TT} by using the triplet $(l_{TT}^B, l_{TT}^{TTCC}, l_{TT}^{CRP})$ where l_{TT}^B denotes the tuple of active locations of the tasks layer components, l_{TT}^{TTCC} contains the tuple of active locations of all TTCC components of the TTCC layer, and l_{TT}^{CRP} contains enabled locations of the CRP. We recall also that a place l of a model $B = \gamma(B_1, ..., B_n)$ is written $l = (l_1, .., l_n)$. The place l_{TT}^B of the tasks components layer of the model B^{TT} is written $l_{TT}^B = (l_1^{TT}, .., l_n^{TT})$. The place l_{TT}^{TTCC} of the TTCC components layer is written as follows $l_{TT}^{TTCC} = (l_1^{TTCC}, ..., l_m^{TTCC})$ while the place l_{TT}^{CRP} of the CRP component is written as $l_{TT}^{TTCC} \in \{w_\alpha, r_\alpha\}$.

We define the relation $R \subseteq Q_B \times Q_{B_{TT}}$ as follows:

$$R = \left\{ (q_B, q_{B_{TT}}) \;\middle|\; \begin{array}{c} l_{TT}^B \in \{l_i, \bot_{p_i}^{l_i}\}^n, \text{ where } l_i \xrightarrow[B_i]{p_i}, \\ v_c(q_B) = v_c(q_{B_{TT}})\,, \\ v_x(q_B) = v_x^*(q_{B_{TT}}) \end{array} \right\} \tag{4.7}$$

where v_x^* is the restriction of v_x to the variables X of the original model B. That is the valuation function v_x^* is defined only over variables which are common between B

and B_{TT}. We recall that the notation $l_i \xrightarrow[B_i]{p_i}$ means that port p_i is enabled from place l_i of the component B_i.

Note that in the definition (4.7) of the relation R, there is no restriction to the location of TTCC and CRP components. This means that we consider all states of these components in the defined equivalence class. That is q_B is equivalent with $q_{B_{TT}}$ whose location is a combination of any location of TTCC and CRP components with the locations l_i or $\perp_{p_i}^{l_i}$ of components B. That is $\forall j \in [1,m]\,, l_j^{TTCC} \in \{wait, l_{o_p}, .., read, try, send\}$ and $l_{TT}^{CRP} \in \{w_\alpha, r\alpha\}$.

Thus,the following four assertions prove that (R, β) is a weak bisimulation:

(i) $\forall (q_B, q_{B_{TT}}) \in R\,,$

$$q_B \xrightarrow[B]{\beta} r_B \implies \exists (r_B, r_{B_{TT}}) \in R : q_{B_{TT}} \xrightarrow[B_{TT}]{\beta^*} r_{B_{TT}}\,,$$

(ii) $\forall (q_B, q_{B_{TT}}) \in R\,,$

$$q_{B_{TT}} \xrightarrow[B_{TT}]{\beta} r_{B_{TT}} \implies \exists (r_B, r_{B_{TT}}) \in R : q_B \xrightarrow[B]{\beta^*} r_B\,,$$

(iii) $\forall (q_B, q_{B_{TT}}) \in R\,, \forall \alpha \in \gamma\,,$

$$\beta(\alpha) \neq \emptyset \wedge q_B \xrightarrow[B]{\alpha} r_B \implies \exists (\alpha, \alpha') \in \beta : \exists (r_B, r_{B_{TT}}) \in R : q_{B_{TT}} \xrightarrow[B_{TT}]{\beta^*\alpha'\beta^*} r_{B_{TT}}\,,$$

(iv) $\forall (q_B, q_{B_{TT}}) \in R\,, \forall k \in K\,,$

$$\beta^{-1}(k) \neq \emptyset \wedge q_{B_{TT}} \xrightarrow[B_{TT}]{k} r_{B_{TT}} \implies \exists (p, k) \in \beta : \exists (r_B, r_{B_{TT}}) \in R : q_B \xrightarrow[B]{p} r_B\,.$$

Hereafter, we detail proofs of each of these four points:

(i) In definition (4.6) of the relation β, only interactions of γ are related to interactions of γ^{TT}. That is for each $\alpha \in \gamma$, $\beta(\alpha) \neq \emptyset$. Therefore if $q_B \xrightarrow[B]{\beta} r_B$, then this transition corresponds to a transition that is not related by the relation β. Therefore, by definition (4.6) of the relation β, the corresponding transition is not an interaction of γ. It is then a transition labelled by a real number representing a delay transition.

By Definition 1.14, there is a tpc constraint on location l in B, $tpc(l) = (c^g \leq v)$. That is the tpc constraint of each partial location l_i of each component B_i of the model B (such that $l = (l_1, .., l_n)$) must satisfy this same condition. Therefore, we have:

$$\begin{gathered} q_B = \big(l, v_x(q_B), v_c(q_B)\big)\,, \quad r_B = \big(l, v_x(r_B), v_c(r_B)\big)\,, \\ v_x(r_B) = v_x(q_B), \quad \text{and} \quad v_c(r_B) = v_c(q_B) + \delta\,, v_c(q_B) + \delta \leq v\,. \end{gathered} \tag{4.8}$$

Note that, depending on the nature of interactions enabled from r_B, two cases should be considered. In the first case, only an internal interaction $\alpha_I \in A_I$ can be enabled from state r_B once β executed. In the second case, only external interactions $\alpha_E \in A_E$ are enabled from r_B.

By construction of the definition (4.7) of R, we have $q_B = \big(l, v_x(q_B), v_c(q_B)\big)$, such that

$$v_c(q_B) = v_c(q_{B_{TT}}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{B_{TT}})\,. \tag{4.9}$$

By construction of the transformation (Rule 4.3 and Rule 4.1) the same tpc constraint is mapped in the first case to the place l_{TT} where $l_{TT} = l$. In the second case, the same tpc constraint is mapped to the places l_i and $\perp_{p_i}^{l_i}$ where $p_i \in \alpha_E$ as well as to the place *read* of the corresponding TTCC (handling the interaction α_E). Thus, after executing the β transition corresponding to the mapped tpc in the B_{TT} model, components do not change their places. And there exists a transition $q_{B_{TT}} \xrightarrow[B_{TT}]{\delta} r_{B_{TT}}$ in B_{TT} where $r_{B_{TT}} = (l'_{TT}, v_x(r_B), v_c(r_B))$ such that:

$$l'^{B}_{TT} = l\,, \quad v_c(q_B) = v_c(r_B) + \delta \quad \text{and} \quad v_x(q_B) = v_x(r_B)\,. \tag{4.10}$$

Combining (4.8), (4.9) and (4.10), we obtain that $v_c(r_{B_{TT}}) = v_c(r_B)$ and $v_x^*(r_{B_{TT}}) = v_x(r_B)$. And we deduce that by definition (4.7) of the relation R, we have $(r_B, r_{B_{TT}}) \in R$.

(ii) If $(q_B, q_{B_{TT}}) \in R$, $q_{B_{TT}} \xrightarrow[B_{TT}]{\beta} r_{B_{TT}}$, then this transition is not related to any transition in γ by the relation β. Therefore and by definition (4.6) of the relation β, the transition β is either labelled by a real number representing a delay transition or by a send/receive interaction other than the notification transition or a p_α transition. That is, β corresponds either to a rsv_α, $fail_\alpha$, offer, ok_α, p_α interaction or to a delay step.

Case 1: $\beta \in \{rsv_\alpha, fail_\alpha\}$**.**

By Definition 1.14, there is a transition $l_{TT} \xrightarrow{\beta \in \{rsv_\alpha, fail_\alpha\}} l'_{TT}$ in B_{TT}, such that:

$$\begin{aligned} q_{B_{TT}} &= \big(l_{TT}(q_{B_{TT}}), v_x(q_{B_{TT}}), v_c(q_{B_{TT}})\big)\,, \\ r_{B_{TT}} &= \big(l'_{TT}(r_{B_{TT}}), v_x(r_{B_{TT}}), v_c(r_{B_{TT}})\big)\,, \\ v_x(r_{B_{TT}}) = v_x(q_{B_{TT}}), &\quad \text{and} \quad v_c(r_{B_{TT}}) = v_c(q_{B_{TT}})\,. \end{aligned} \tag{4.11}$$

Note that both rsv_α and $fail_\alpha$ define no update function nor a guard or timing constraints (see Rule 4.5).

By definition of the transformation rules (Rule 4.3 and Rule 4.4), in the case of a rsv_α (resp. $fail_\alpha$) interaction, the corresponding TTCC component is in a *read* (resp. *try*) place and the CRP component is in w_α (resp. r_α) place. After executing this rsv_α (resp. $fail_\alpha$) transition, the TTCC component reaches place

try (resp. *read*) and the place r_α (resp. w_α) is activated in the CRP. Note that, in both cases, places of other components remain intact. That is, the reached place $l'^{B}_{TT} = l^{B}_{TT} = l$. Thus, we have :

$$l'^{B}_{TT} = l = (l_1, .., l_n)\,, \tag{4.12}$$

By construction (4.7) of R, we have $q_B = \big(l, v_x(q_B), v_c(q_B)\big)$, such that

$$v_c(q_B) = v_c(q_{B_{TT}}) \quad \text{and} \quad v_x(q_B) = v^*_x(q_{B_{TT}})\,. \tag{4.13}$$

Combining (4.11) and (4.13) we obtain that $v_c(r_{B_{TT}}) = v_c(q_B)$ and $v^*_x(r_{B_{TT}}) = v_x(q_B)$. Combining this to (4.12), we deduce that by definition (4.7) of the relation R, we have $(q_B, r_{B_{TT}}) \in R$.

Case 2: β is an offer interaction.

By Definition 1.14, there is a transition $l_{TT} \xrightarrow{\beta} l'_{TT}$ in B_{TT}, where β allows sending an offer from port p_i of component B_i to the corresponding TTCC component, such that:

$$\begin{aligned} q_{B_{TT}} &= \big(l_{TT}, v_x(q_{B_{TT}}), v_c(q_{B_{TT}})\big)\,, \\ r_{B_{TT}} &= \big(l'_{TT}, v_x(r_{B_{TT}}), v_c(r_{B_{TT}})\big)\,, \\ v_x(r_{B_{TT}}) = v_x(q_{B_{TT}}), &\quad \text{and} \quad v_c(r_{B_{TT}}) = v_c(q_{B_{TT}})\,. \end{aligned} \tag{4.14}$$

Note that the offer transition defines no update function nor a guard or timing constraint (see Rule 4.5).

By definition of the transformation rules (Rule 4.3 and Rule 4.4), after executing this β transition, the TTCC component reaches a place l_{o_i} and the component B_i reaches a place $\perp^{l_i}_{p'_i}$ if another offer is likely to be sent, otherwise it reaches the place l_i. Note that this β transition does not change the location of the CRP component. Thus, we have :

$$l'^{B}_{TT} \in \{l_i, \perp^{l_i}_{p_i}\}^n\,. \tag{4.15}$$

By construction (4.7) of R, we have $q_B = \big(l, v_x(q_B), v_c(q_B)\big)$, such that

$$v_c(q_B) = v_c(q_{B_{TT}}) \quad \text{and} \quad v_x(q_B) = v^*_x(q_{B_{TT}})\,. \tag{4.16}$$

Combining (4.14) and (4.16) we obtain that $v_c(r_{B_{TT}}) = v_c(q_B)$ and $v^*_x(r_{B_{TT}}) = v_x(q_B)$. Combining this to (4.15), we deduce that by definition (4.7) of the relation R, we have $(q_B, r_{B_{TT}}) \in R$.

Case 3: $\beta \in \{ok_\alpha, p_\alpha\}$

By Definition 1.14, there is a transition $l_{TT} \xrightarrow{\beta} l'_{TT}$ in B_{TT}, where β is labelled either by the port ok_α or p_α. The transition p_α changes only location of the TTCC component (from *read* to *send* location). Whereas the transition ok_α changes the location of the TTCC component (from *try* to *send*) and the location of the CRP (from r_α to w_α). In both cases, locations of other components are intact. We denote G^*, TC^* and F^* respectively the guard, timing constraint and update function of the transition β. Therefore, we have:

$$\begin{aligned}
q_{B_{TT}} &= \big((l^B_{TT}, l^{TTCC}_{TT}(q_{B_{TT}}), l^{CRP}_{TT}(q_{B_{TT}})), v_x(q_{B_{TT}}), v_c(q_{B_{TT}})\big)\,,\\
r_{B_{TT}} &= \big((l'^B_{TT}, l'^{TTCC}_{TT}(r_{B_{TT}}), l'^{CRP}_{TT}(r_{B_{TT}})), v'_x(r_{B_{TT}}), v_c(r_{B_{TT}})\big)\,,\\
G^*(v_x(q_{B_{TT}})) &= \mathit{True}\,,\\
TC^*(v_c(q_{B_{TT}})) &= \mathit{True}\,,\\
v_c(r_{B_{TT}}) &= v_c(q_{B_{TT}})\\
v_x(r_{B_{TT}}) &= F^*(v_x(q_{B_{TT}}))\,,\,,
\end{aligned} \tag{4.17}$$

In the before last equality of (4.17), we have $v_c(r_{B_{TT}}) = v_c(q_{B_{TT}})$ since transition is instantaneous. For the last equality of (4.17), notice that, F^* operates only on variables that are local to the TTCC component. Therefore this function does not update variables of the components B_i^{TT} that are common with the model B. Therefore the execution of this update function does not change the valuation v_x^*. Thus, we have:

$$v_x^*(r_{B_{TT}}) = v_x^*(q_{B_{TT}})\,. \tag{4.18}$$

By definition of the transformation rules (Rule 4.3 and Rule 4.4), after executing this β transition, the TTCC component reaches the place *send* and the CRP component reaches back the place *wait*. The component B_i^{TT} does not change its location. Thus, we have :

$$l'^B_{TT} = l^B_{TT}\,. \tag{4.19}$$

By construction (4.7) of R, we have $q_B = \big(l, v_x(q_B), v_c(q_B)\big)$, such that

$$l^B_{TT} \in \{l_i, \bot^{l_i}_{p_i}\}^n \quad , \quad v_c(q_B) = v_c(q_{B_{TT}}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{B_{TT}})\,. \tag{4.20}$$

Combining (4.17), (4.18), (4.19) and (4.20) we obtain that $v_c(r_{B_{TT}}) = v_c(q_B)$, $v_x^*(r_{B_{TT}}) = v_x(q_B)$ and $l'^B_{TT} = l^B_{TT} \in \{l_i, \bot^{l_i}_{p_i}\}^n$. Thus, we deduce that by definition (4.7) of the relation R, we have $(q_B, r_{B_{TT}}) \in R$.

Case 4: β **is a delay step labelled by** $\delta \in \mathbb{R}_+$**.**

By Definition 1.14, there is a tpc constraint on location l_{TT} in B_{TT}, $tpc(l_{TT}) = (c^g \leq v)$. That is the tpc condition of each partial location of each component of

the B_{TT} model that is composing the global location l_{TT} must satisfy this same condition. Therefore, we have:

$$\begin{aligned} q_{B_{TT}} = \big(l_{TT}, v_x(q_{B_{TT}}), v_c(q_{B_{TT}})\big)\,, \quad r_{B_{TT}} = \big(l_{TT}, v_x(r_{B_{TT}}), v_c(r_{B_{TT}})\big)\,, \\ v_x(r_{B_{TT}}) = v_x(q_{B_{TT}}), \quad \text{and} \quad v_c(r_{B_{TT}}) = v_c(q_{B_{TT}}) + \delta\,, v_c(q_{B_{TT}}) + \delta \leq v\,. \end{aligned} \tag{4.21}$$

Note that, by construction of the transformation (Rule 4.3), this delay transition is only possible if at least one conflicting TTCC component is not occupying the *send* place, i.e. $l_{TT}^{TTCC^C} \neq \{send\}^k$. After executing this β transition, the TTCC component does not change the global place nor the variables valuation, only the clock valuation is augmented by δ. Thus, we have :

$$l'^{B}_{TT} = l\,. \tag{4.22}$$

By construction of the definition (4.7) of R, we have $q_B = \big(l, v_x(q_B), v_c(q_B)\big)$, such that

$$v_c(q_B) = v_c(q_{B_{TT}}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{B_{TT}})\,. \tag{4.23}$$

By definition of the transformation (see Rule 4.3), the tpc constraints of the TTCC component is the conjunction of time progress conditions received in the offers from participating components. Thus there exist a transition $q_B \xrightarrow[B]{\delta} r_B$ in B where $r_B = (l, v_x(r_B), v_c(r_B))$ such that:

$$v_c(q_B) = v_c(r_B) + \delta \quad \text{and} \quad v_x(q_B) = v_x(r_B)\,. \tag{4.24}$$

Combining (4.21), (4.23) and (4.24), we obtain that $v_c(r_{B_{TT}}) = v_c(r_B)$ and $v_x^*(r_{B_{TT}}) = v_x(r_B)$. Combining this to (4.22), we deduce that by definition (4.7) of the relation R, we have $(r_B, r_{B_{TT}}) \in R$.

(iii) Let $(q_B, q_{B_{TT}}) \in R$ such that $q_B \xrightarrow[B]{\alpha} r_B$. If $\beta(\alpha) \neq \emptyset \wedge q_B \xrightarrow[B]{\alpha} r_B$, then by definition (4.6) of the relation β, $\alpha \in \gamma$ and can be either an internal ($\alpha \in A_I$) or an external interaction ($\alpha \in A_E$).

Case 1: $\alpha \in \gamma \cap A_I$**.**

By Definition 1.14, there is a transition $l \xrightarrow{\alpha} l'$ in B, where α is guarded by G^*, the timing constraint TC^* and having as transfer function F^*, such that:

$$\begin{aligned} q_B = \big(l, v_x(q_B), v_c(q_B)\big)\,, \quad r_B = \big(l', v_x(r_B), v_c(r_B)\big)\,, \\ TC^*(v_c(q_B)) = \mathit{True}, \quad G^*(v_x(q_B)) = \mathit{True}, \\ v_x(r_B) = F^*(v_x(q_B)), \quad \text{and} \quad v_c(r_B) = v_c(q_B)\,, \end{aligned} \tag{4.25}$$

where the update function $F* = f_i \circ ... \circ f_j \circ F_\alpha$, where f_i corresponds to the update function of the transition labelled by port $p_i \in P_\alpha$ in the component $B_i \in \mathit{comp}(\alpha)$. By construction (4.7) of R, we have $q_{B_{TT}} = \big(l_{TT}, v_x(q_{B_{TT}}), v_c(q_{B_{TT}})\big)$, such that

$$v_c(q_B) = v_c^*(q_{B_{TT}}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{B_{TT}})\,. \tag{4.26}$$

By definition of the transformation (Rule 4.3 and Rule 4.1), this interaction remains intact in the obtained B_{TT} model. Therefore, by Definition 1.14, we also have $q_{B_{TT}} \xrightarrow[B_{TT}]{\alpha} r_{B_{TT}}$, where $r_{B_{TT}} = \big(l'_{TT}, v_x(r_{B_{TT}}), v_c(r_{B_{TT}})\big)$ such that:

$$\begin{aligned} l'^{B}_{TT} &= l' \,, \\ v_c(r_{B_{TT}}) &= v_c(q_{B_{TT}}) \,, \\ v^*_x(r_{B_{TT}}) &= F^*\big(v^*_x(q_{B_{TT}})\big) \,. \end{aligned} \tag{4.27}$$

In the second equality of (4.27), we have $v_c(r_{B_{TT}}) = v_c(q_{B_{TT}})$ since transition α is instantaneous. For the last equality of (4.27), notice that, v^*_x operates only on common variables between models B and B_{TT}.

Combining (4.25), (4.26) and (4.27) we obtain that l_{TT} satisfies $l'^{B}_{TT} = l'$, $v^*_c(r_{B_{TT}}) = v_c(r_B)$ and $v^*_x(r_{B_{TT}}) = v_x(r_B)$. Thus, we have $q_{B_{TT}} \xrightarrow[B_{TT}]{\alpha} r_{B_{TT}}$ such that $(\alpha, \alpha) \in \beta$ since $\alpha \in \gamma \cap A_I$. By definition (4.7) of the relation R, we obtain $(r_B, r_{B_{TT}}) \in R$.

Case 2: $\alpha \in \gamma \cap A_E$**.**

By Definition 1.14, there is a transition $l \xrightarrow{\alpha} l'$ in B, where α is guarded by G^*, the timing constraint TC and having as transfer function F^*, such that:

$$\begin{aligned} q_B = \big(l, v_x(q_B), v_c(q_B)\big) \,, &\quad r_B = \big(l', v_x(r_B), v_c(r_B)\big) \,, \\ TC^*(v_c(q_B)) = \mathit{True}, &\quad G^*(v_x(q_B)) = \mathit{True}, \\ v_x(r_B) = F^*(v_x(q_B)), &\quad \text{and} \quad v_c(r_B) = v_c(q_B) \,, \end{aligned} \tag{4.28}$$

where the update function $F* = f_i \circ ... \circ f_j \circ F_\alpha$, where f_i corresponds to the update function of the transition labelled by port $p_i \in P_\alpha$ in the component $B_i \in \mathit{comp}(\alpha)$. By construction (4.7) of R, we have $q_{B_{TT}} = \big(l_{TT}, v_x(q_{B_{TT}}), v_c(q_{B_{TT}})\big)$, such that

$$v_c(q_B) = v^*_c(q_{B_{TT}}) \quad \text{and} \quad v_x(q_B) = v^*_x(q_{B_{TT}}) \,. \tag{4.29}$$

By definition of the transformation (Rule 4.3 and Rule 4.1), the interaction α of the original model B is held by a dedicated TTCC component that we denote here $TTCC_\alpha$ in the obtained B_{TT} model. It may be mapped to the following successive transitions in the B_{TT} model:

- If the component l^B_{TT} of the global place l_{TT} contains a partial place $l^{TT}_i = \perp^{l_i}_{p_i}$, where $B_i \in \mathit{comp}(\alpha)$ and $p_i \in P_\alpha$, then a sending offer interaction may be enabled, note that by definition of β, this interaction is a β transition. If the component l^B_{TT} of the global place l_{TT} is equal to l (i.e. $l^B_{TT} = (l_1, .., l_n)$), no offer transition is enabled.

- Once all offers of components $B_i \in comp(\alpha)$ are send to $TTCC_\alpha$, then this latter reaches the place *read*. If initially, α is not conflicting, then from the reached global location, after sending offers, the transition labelled by the unary interaction p_α is enabled. This transition has the guard G^*, the timing constraint TC^* and executes the function F^*. Note that by definition of β, $\beta(p_\alpha) = \emptyset$. If α is initially a conflicting interaction, then from the reached global location, after sending offers, the enabled transition is the $rsv\alpha$ interaction. This interactions has the guard G^* and the timing constraint TC^*. By definition of β, $\beta(rsv_\alpha) = \emptyset$, it is then a β transition. From the reached location by the rsv_α interaction, two interactions are possible, $fail_\alpha$ or ok_α. $\beta(fail_\alpha) = \emptyset$ and $\beta(ok_\alpha) = \emptyset$. If the $fail_\alpha$ interaction is enabled then the $TTCC_\alpha$ component is reaching back the state enabling again the $rsv\alpha$ interaction until the $ok\alpha$ is enabled. From this reached global location a loop of rsv_α and $fail_\alpha$ may be enabled before the ok_α interaction is enabled. This latter reaches a state where the $TTCC_\alpha$ is in place *send*. The ok_α as well as the p_α transition applies the update function F^* to the local variables that are local to the TTCC. Note that these variables are not concerned by the valuation v_x^*.
- Note that after the previously executed interaction the components $B_i \in comp(\alpha)$ do not change their locations. The $TTCC_\alpha$ component reaches the *send* location. From this new reached global state, the notification interaction is enabled. It relates the port p_s^α of the $TTCC_\alpha$ to ports p_i of components B_i, such that $p_i \in P_\alpha$. Note that $\beta(p_s^\alpha) \neq \emptyset$. This notification interaction updates variables of components B_i according to their copies in the component $TTCC_\alpha$. Note that these copies have been transformed by F^* in the previous β transition. The reached location of the notification interaction in a component B_i is l_i' or $\perp_{p_i'}^{l_i'}$, where $l_i' \xrightarrow{p_i'}$.

Notice that in the previously cited cases of possible interactions, we consider only β interactions in which the $TTCC\alpha$ participates. For clarity reasons, we do not detail different other possible β transitions involving other TTCC components and potential offer sending requests. Not considering them, does not invalidate this proof since they always satisfy the property $l_{TT}^B \in \{l_i, \perp_{p_i}^{l_i}\}^n$, are instantaneous and do not hold any update function (i.e. they do not impact the location property, nor the clock and variables valuations).

Therefore, by Definition 1.14, we have:

$$q_{B_{TT}} \xrightarrow[B_{TT}]{\beta^*} q'_{B_{TT}} \xrightarrow[B_{TT}]{p_s^\alpha} r_{B_{TT}} ,$$

where

$$q'_{B_{TT}} = \big((l^B_{TT}, l^{TTCC}_{TT}(q'_{B_{TT}}), l'^{CRP}_{TT}(q'_{B_{TT}})), v_x(q'_{B_{TT}}), v_c(q'_{B_{TT}})\big)\,,$$
$$r_{B_{TT}} = \big((l'^B_{TT}, l^{TTCC}_{TT}(r_{B_{TT}}), l^{CRP}_{TT}(r_{B_{TT}})), v_x(r_{B_{TT}}), v_c(q'_{B_{TT}})\big)\,,$$

with

$$\begin{aligned} l'^B_{TT} &\in \{l'_i, \perp^{l'_i}_{p'_i}\}^n\,,\\ v_c(r_{B_{TT}}) &= v_c(q'_{B_{TT}}) = v_c(q_{B_{TT}})\,,\\ v^*_x(q'_{B_{TT}}) &= v^*_x(q_{B_{TT}})\,,\\ v^*_x(r_{B_{TT}}) &= F^*\big(v^*_x(q'_{B_{TT}})\big)\,, \end{aligned} \tag{4.30}$$

For the last equality of (4.30), notice that, v^*_x operates only on common variables between models B and B_{TT}. And F^* has been first applied to local variables of the TTCC component in the β transition preceding the p^α_s transition. These variables are not concerned by the v^*_x valuation, thus, the equality $v^*_x(q'_{B_{TT}}) = v^*_x(q_{B_{TT}})$. The transition p^α_s copies values of TTCC variables to those of B_i components. Thus the function F^* is indirectly applied to variables of B_i. Which explains the equality $v^*_x(r_{B_{TT}}) = F^*\big(v^*_x(q'_{B_{TT}})\big)$.

Combining (4.28), (4.29) and (4.30), we obtain that l'_{TT} satisfies $l'^B_{TT} \in \{l'_i, \perp^{l'_i}_{p'_i}\}^n$, $v^*_c(r_{B_{TT}}) = v_c(r_B)$ and $v^*_x(r_{B_{TT}}) = v_x(r_B)$. Thus, we have $q_{B_{TT}} \xrightarrow[B_{TT}]{\beta^* p^\alpha_s} r_{B_{TT}}$ such that $(\alpha, p^\alpha_s) \in \beta$. By definition (4.7) of the relation R, we obtain $(r_B, r_{B_{TT}}) \in R$.

(iv) Let $(q_B, q_{B_{TT}}) \in R$ such that $q_{B_{TT}} \xrightarrow[B_{TT}]{\alpha_{TT}} r_{B_{TT}}$. If $\beta^{-1}(\alpha_{TT}) \neq \emptyset \wedge q_{B_{TT}} \xrightarrow[B_{TT}]{\alpha_{TT}} r_{B_{TT}}$, then by definition (4.6) of the relation β,

$$\alpha_{TT} \in (\gamma \cap A_I) \cup \{p^\alpha_s \in \gamma_{TT} \,|\, \alpha \in \gamma \cap A_E\}$$

Case 1: $\alpha_{TT} = \alpha \in \gamma \cap A_I$**.**

By Definition 1.14, there is a transition $l_{TT} \xrightarrow{\alpha_{TT}} l'_{TT}$ in B_{TT}, where the transition α_{TT} has a guard G^*, a timing constraint TC^* and an update function F^*, such that:

$$\begin{aligned} q_{B_{TT}} &= \big((l, l^{TTCC}_{TT}(q_{B_{TT}}), l^{CRP}_{TT}(q_{B_{TT}})), v_x(q_{B_{TT}}), v_c(q_{B_{TT}})\big)\,,\\ r_{B_{TT}} &= \big(l', l'^{TTCC}_{TT}(r_{B_{TT}}), l'^{CRP}_{TT}(r_{B_{TT}}))v_x(r_{B_{TT}}), v_c(r_{B_{TT}})\big)\,,\\ G^*(v_x(q_{B_{TT}})) &= \mathit{True}\,,\\ TC^*(v_c(q_{B_{TT}})) &= \mathit{True}\,,\\ v_x(r_{B_{TT}}) &= F^*(v_x(q_{B_{TT}})\,,\\ v_c(r_{B_{TT}}) &= v_c(q_{B_{TT}})\,. \end{aligned} \tag{4.31}$$

By definition of the transformation (Rule 4.3 and Rule 4.1), the transition $\alpha_{TT} = \alpha$ is exactly the same as in the model B which corresponds to the following transition $l \xrightarrow{\alpha} l'$ in B, which is guarded by G^*, TC^* and has the update function F^*.

By construction (4.7) of R, we have $q_B = \big(l, v_x(q_B), v_c(q_B)\big)$, such that

$$v_c(q_B) = v_c(q_{B_{TT}}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{B_{TT}})\,. \tag{4.32}$$

Therefore, By Definition 1.14, we also have $q_B \xrightarrow[B]{\alpha} r_B$,, where

$$r_B = \big(l', v_x(r_B), v_c(r_B)\big)\,,$$

with

$$\begin{aligned} G^*(v_x(q_B)) &= \mathit{True}\,,\\ TC^*(v_c(q_B)) &= \mathit{True}\,,\\ v_c(r_B) &= v_c(q_B)\,,\\ v_x(r_B) &= F^*(v_x(q_B))\,. \end{aligned} \tag{4.33}$$

Combining (4.31), (4.32) and (4.33), we obtain that l'_{TT} satisfies $l'^{B}_{TT} = l' \in \{l_i, \perp^{l_i}_{p_i}\}^n$, $v_c(r_{B_{TT}}) = v_c(r_B)$ and $v_x^*(r_{B_{TT}}) = v_x(r_B)$. Thus, we have $q_B \xrightarrow[B]{\alpha} r_B$ and, by definition (4.7) of the relation R, $(r_B, r_{B_{TT}}) \in R$.

Case 2: $\alpha_{TT} = p_s^{\alpha}\,, \alpha \in \gamma \cap A_E$**.**

By Definition 1.14, there is a transition $l_{TT} \xrightarrow{\alpha_{TT}} l'_{TT}$ in B_{TT}. The transition α_{TT} has no guard.

By construction of the transformation (Rule 4.3 and Rule 4.1), this α_{TT} transition is always preceded by a β transition consisting in p_α if α is not conflicting and in ok_α if α is conflicting. These latter execute an update function F^* that updates variables local to the TTCC component. These variables are local copies of variables of B_i. When receiving offers, values of variables of the TTCC component are the same as their remote copies in B_i components. And then, they are updated by using the function F^* of transition ok_α or p_α.

The notification transition is not guarded and have an update function which copies values of local variables of the TTCC to their corresponding copies in the participating B_i components. Therefore the function F^* is indirectly applied to variables of B_i components. These variables are concerned by the v_x^* valuation.

Note that this α_{TT} transition, changes the location of the TTCC component to its initial *wait* location and allows to reach location l'_i or $\perp^{l'_i}_{p'_i}$, where $l'_i \xrightarrow{p'_i}$ and $p'_i \in A_E$.

Therefore, we have $l_{TT} \xrightarrow{\alpha_{TT}} l'_{TT}$, such that:

$$\begin{aligned} q_{B_{TT}} &= \big((l^B_{TT}(q_{B_{TT}}), l^{TTCC}_{TT}(q_{B_{TT}}), l^{CRP}_{TT}(q_{B_{TT}}), v_x(q_{B_{TT}}), v_c(q_{B_{TT}})\big)\,, \\ r_{B_{TT}} &= \big(l'^B_{TT}(q_{B_{TT}}), l'^{TTCC}_{TT}(r_{B_{TT}}), l'^{CRP}_{TT}(r_{B_{TT}}) v_x(r_{B_{TT}}), v_c(r_{B_{TT}})\big)\,, \\ v^*_x(r_{B_{TT}}) &= F^*(v^*_x(q_{B_{TT}}))\,, \\ v_c(r_{B_{TT}}) &= v_c(q_{B_{TT}})\,, \end{aligned} \tag{4.34}$$

such that

$$l'^B_{TT} \in \{l'_i, \perp^{l'_i}_{p'_i}\}^n\,. \tag{4.35}$$

By definition of the transformation (Rule 4.3 and Rule 4.1), there exist a corresponding transition $l \xrightarrow{\alpha} l'$ in B, which is having as transfer function F^*.

By construction (4.7) of R, we have $q_B = \big(l, v_x(q_B), v_c(q_B)\big)$, such that

$$l^B_{TT}(q_{B_{TT}}) \in \{l_i, \perp^{l_i}_{p_i}\}^n \quad , \quad v_c(q_B) = v_c(q_{B_{TT}}) \quad \text{and} \quad v_x(q_B) = v^*_x(q_{B_{TT}})\,. \tag{4.36}$$

Therefore, By Definition 1.14, we also have $q_B \xrightarrow[B]{\alpha} r_B$,, where

$$r_B = \big(l', v_x(r_B), v_c(r_B)\big)\,,$$

with

$$\begin{aligned} v_c(r_B) &= v_c(q_B)\,, \\ v_x(r_B) &= F^*(v_x(q_B))\,. \end{aligned} \tag{4.37}$$

Combining (4.34), (4.35), (4.36) and (4.37), we obtain that l'_{TT} satisfies $l'^B_{TT} \in l'_i, \perp^{l'_i\ n}_{p'_i}$, $v_c(r_{B_{TT}}) = v_c(r_B)$ and $v^*_x(r_{B_{TT}}) = v_x(r_B)$. Thus, we have $q_B \xrightarrow[B]{\alpha} r_B$ and, by definition (4.7) of the relation R, $(r_B, r_{B_{TT}}) \in R$.

□

4.6 Conclusion

In this chapter, we have presented a model to model transformational method allowing to explicit TT communication settings in the obtained model. The obtained model is structured following the TT-BIP architecture. It consists of tasks layer, communication layer and the conflict resolution layer. The first layer is obtained after transforming components participating in external interactions depending on a user-defined task mapping. Each TTCC component of the second layer is dedicated to handle one external interaction and communicate with tasks of the layer underneath in two steps; it receives offers

and sends notification after executing the interaction. The third layer is responsible for resolving conflicts between different interactions handled by the second layer.

The obtained model is based on one global clock, implements multiparty interactions through dedicated communication media (i.e. TTCC components) and ensures communication between different layers by using message passing interactions (i.e. Send/receive interactions). Even though the obtained model satisfies the TT settings described in the opening of Section 4.1, it is yet still far from being intuitively translatable to the programming language of a target platform which is based on the TT execution model.

In the next chapter, we present a method for generating TT implementation from the obtained TT-BIP model.

5

From Time-Triggered BIP Model to Time-Triggered Implementation

In the previous chapter, we presented how a BIP model is transformed in order to comply with the TT communication pattern. In the obtained model, multiparty communication/interactions are handled by using dedicated communication components and different layers communicate only via send/receive interactions. Also, components mapped to the same task are gathered under a composite component presenting the corresponding task.

In this chapter, we present how to transform the obtained TT-BIP model into ΨC code. On the implementation level, the notion of composite process/task does not exist. Even though keeping atomic components under the composite task can facilitate component reuse, it is necessary—in this step of our transformation—to transform each composite task component into an atomic one. The obtained model after composition —denoted TT-BIP model—is the input model of the translation into the ΨC language process (cf. Figure 5.1).*

Also, to be able to prove correctness of the transformation from TT-BIP to ΨC, we must first provide a target formal model for the implementation and define its operational semantics.

This chapter is organized as follows. In Section 5.1, we define the transformation that is applied to the TT-BIP model with composite task components in order to obtain the TT-BIP model which encompasses only atomic components. Section 5.2 proposes the Time Constrained Automata (TCA) model as a formal model of TT tasks of a PharOS application. Section 5.3 deals with challenges of the transformation. The formal rules that define the transformation from TT-BIP* to TCA are presented in Section 5.4. Correctness of this transformation is proved in Section 5.5 and Section 5.6.*

Chapter outline

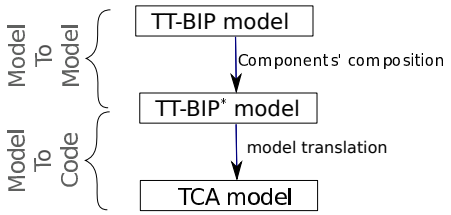


Figure 5.1: Transformation approach

5.1 Component Composition

We define in this section the transformation allowing to obtain atomic task components from composite ones. This composition is needed to prepare for the code generation.

In [47], a BIP model transformation is presented, which transforms untimed BIP models containing composite components into models with only atomic components. We present here the definition of composition for timed BIP models.

Intuitively, as shown in Figure 5.2, the composition operation consists in replacing transitions from atomic components that are synchronized through an internal interaction by a single transition, labelled by an internal port. Guards of synchronized transitions are obtained by conjuncting the guard of the interaction and individual guards. Similarly, the timing constraint of the obtained transition is obtained by conjuncting the timing constraints of the composing transitions. The update function of the transition, is defined as the sequential composition of the update function of the interaction followed by the individual update functions of composing transitions in an arbitrary order since they operate on independent data variables.

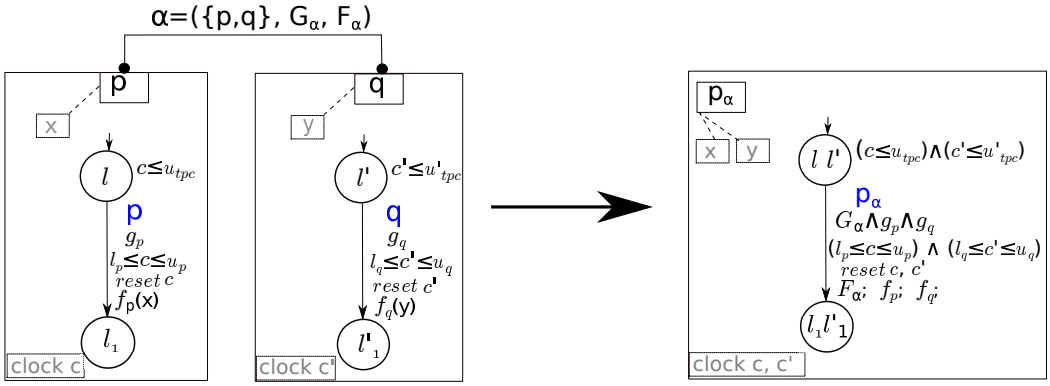


Figure 5.2: Component composition

In the following, we provide the formal rules:

Rule 5.1 (Component Composition)**.** *Let* $\{B_i\}_{i\in[1,n]}$*, where* $B_i = (L_i, P_i, X_i, C_i, T_i, tpc_i)$*, be the set of atomic components constituting the composite component. And let* γ *be the set of interactions* $\alpha = (P_\alpha, G_\alpha, F_\alpha)$ *composing these atomic components. The atomic component* $B = (L, P, X, C, T, tpc)$ *corresponding to the composition* $\gamma\{B_i\}_{i\in[1,n]}$ *is built as follows:*

- *the set of locations* $L = L_1 \times L_2 \times ... \times L_n$,
- *the set of ports* $P = (\bigcup_{i=1}^{n}(P_i) \setminus \bigcup_{\alpha\in\gamma} P_\alpha) \cup \{p_\alpha \mid \alpha \in \gamma\}$,
- *the set of variables* $X = (\bigcup_{i=1}^{n} X_i)$,

- *The set of transitions T is built as follows:*
  - *for each interaction $\alpha \in \gamma$ such that there exists a set of interacting transitions $\{\tau_1, \tau_2, ..., \tau_m\} \subseteq \bigcup_{i=1}^{n} T_i$, such that $\forall k \in [1,m]$, $\tau_k = (l_{\tau_k}, p_{\tau_k}, g_{\tau_k}, tc_{\tau_k}, r_{\tau_k}, f_{\tau_k}, l'_{\tau_k})$ and $\{p_{\tau_k}\}_{i=1}^{k} = P_\alpha$. We include in T the transition composing all these interacting transitions defined by $\tau_\alpha = (l_\alpha, p_\alpha, g_\alpha, tc_\alpha, r_\alpha, f_\alpha, l'_\alpha) \in T$ where:*
    - $l_\alpha = (l_{\tau_1} \times l_{\tau_2} \times ... \times l_{\tau_m})$,
    - $l'_\alpha = (l'_{\tau_1} \times l'_{\tau_2} \times ... \times l'_{\tau_m})$,
    - *the guard* $g_\alpha = G_\alpha \bigwedge(\bigwedge_{k=1}^{m} g_{\tau_k})$,
    - *the timing constraint* $tc_\alpha = \bigwedge_{k=1}^{m} tc_{\tau_k}$,
    - *the update function f_α executes first the function F_α then executes functions f_{τ_k} for all $k \in [1,m]$.*
  - *For each transition $\tau_i = (l_{\tau_i}, p_{\tau_i}, g_{\tau_i}, tc_{\tau_i}, r_{\tau_i}, f_{\tau_i}, l'_{\tau_i}) \in T_i$ of one of the constituent components B_i, such that $\forall \alpha \in \gamma$, $p_{\tau_i} \notin P_\alpha$, and for each $l = (l_1, l_2, ...l_i, ..., l_n) \in L$ such that $l_i = l_{\tau_i}$ we introduce the transition $\tau_i^l = (l, p_{\tau_i}, g_{\tau_i}, tc_{\tau_i}, r_{\tau_i}, f_{\tau_i}, l') \in T$ where $l' = (l_1, l_2, ..., l'_{\tau_i}, ..., l_n)$,*
- $\forall l = (l_1, ..., l_n) \in L$, $tpc(l) = \bigwedge_{i=1}^{n} tpc(l_i)$.

5.2 Formal Model of the ΨC Language

To define a formal translation from TT-BIP* to PharOS application and to prove its correctness, we need to provide a formal definition of operational semantics of the target formalism. Moreover, we need the latter to be at the same abstraction level as the ΨC code, i.e. to specify constraints (release, deadline and synchronization shifts) over different clocks.

In this subsection, we present the Time-Constrained Automata (TCA) model as the formal model of PharOS applications (Definition 5.1). The presented model is an extension of the TCA model presented in [65]. The extension consists mainly in the presentation of timing constraints on edges instead of nodes and in handling the multi-clock timing constraints. Then we provide its operational semantics (Definition 5.2).

A TCA automaton describes the behavior of a task, where nodes represent only the control locations and arcs are labelled by the triplets of constraints defining respectively the release, deadline and synchronization instants. Each component of such a triplet is either $(-1, \perp)$, or a pair of a shift constraint and a clock over which this shift is defined. We denote by $M \stackrel{def}{=} (\mathbb{Z}_+ \times C) \cup \{(-1, \perp)\}$ the set of all such labels. When a label is $(-1, \perp)$, the corresponding constraint is not defined.

Let $x \in \mathbb{Z}_+$ be a shift over a clock c, and λ be a reference instant over the global clock c_{BASE}. To shift the instant λ of the clock c_{BASE} by x along the clock c, we take the instant of the c, corresponding to λ, add x, then convert back to c_{BASE}. We denote by $shift^{c}_{c_{BASE}} : c_{BASE} \times \mathbb{Z}_+ \to c_{BASE}$ the function computing the global instant corresponding to the desired shift as follows:

$$shift^{c}_{c_{BASE}}(\lambda, x) = conv^{c}_{c_{BASE}}\big(conv^{c_{BASE}}_{c}(\lambda) + x\big), \tag{5.1}$$

where $conv^{c}_{c_{BASE}}$ and $conv^{c_{BASE}}_{c}$ are defined in (2.2) and (2.3), respectively. In Figure 5.3, we explain graphically how this shift function is computed.

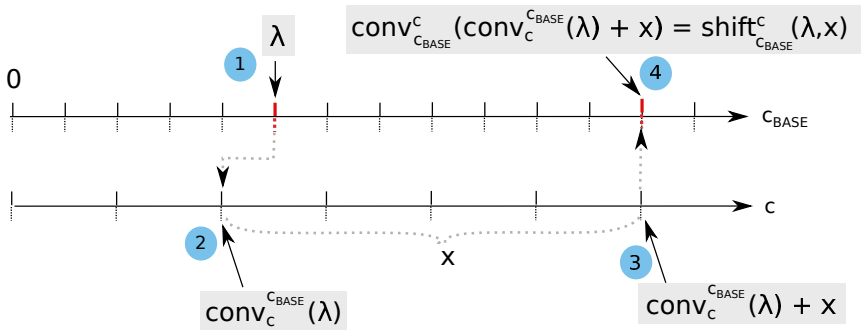


Figure 5.3: Graphical explanation of the shift function (5.1)

Definition 5.1 (TCA). *A TCA is a tuple (N, K, X, C, T) where N is a finite set of* nodes*, K is a finite set of* jobs*, X is a set of* local variables*, C is a set of clocks, comprising a real-time global clock c_{BASE} and other clocks derived from c_{BASE}, and $T = N \times G_X \times M^3 \times K \times \mathcal{V}(X)^{\mathcal{V}(X)} \times N$ is a set of transitions. Thus, a transition is a tuple $\tau = (n, g_X, m, k, f, n') \in T$ where:*

- *$g_X \in G_X$ is a Boolean guard on X;*
- *$m = \big((r, c_r), (d, c_d), (s, c_s)\big) \in M^3$, is a triplet defining respectively, the release shift over clock c_r, the deadline shift over clock c_d and the synchronization shift over clock c_s:*
  - *If $(s, c_s) \neq (-1, \perp)$, then $(d, c_d) = (s, c_s)$,*
  - *If $(r, c_r) \neq (-1, \perp)$, then the release instant over the clock c_{BASE} is defined by $shift^{c_r}_{c_{BASE}}(\lambda, r)$ where $\lambda \in c_{BASE}$ is a reference instant referring to the last absolute release or synchronization instant.*
  - *If $(d, c_d) \neq (-1, \perp)$ (resp. $(s, c_s) \neq (-1, \perp)$) and $(r, c_r) \neq (-1, \perp)$ then the deadline (resp. synchronization) instant over the clock c_{BASE} is defined by $shift^{c_r}_{c_{BASE}}(\lambda, r) + conv^{c_d}_{c_{BASE}}(d)$ (resp. $shift^{c_r}_{c_{BASE}}(\lambda, r) + conv^{c_s}_{c_{BASE}}(s)$) where $\lambda \in c_{BASE}$ is a the reference instant referring to the last absolute release or synchronization instant.*

  - *If* $(d, c_d) \neq (-1, \perp)$ *and* $(r, c_r) = (-1, \perp)$*, then the deadline instant over the clock* c_{BASE} *is defined by* $shift^{c_d}_{c_{BASE}}(\lambda, d)$.

- $k \in K$ *is a job;*
- $f \in \mathcal{V}(X)^{\mathcal{V}(X)}$ *is an update function on variables in* X.
- *let* $S = \{\tau_i\}_{i=1}^{n \cup +\infty} \subseteq T$ *be a sequence of transitions* $\tau_i = (n_i, g_{X\,i}, m_i, k_i, f_i, n'_i)$. *Let* $\tau_j = (n_j, g_{X\,j}, m_j, k_j, f_j, n'_j)$ *and* $\tau_l = (n_l, g_{X\,l}, m_l, k_l, f_l, n'_l)$ *be two transitions in* S *such that* $j < l$, $m_j = \big((r_j, {c_r}^j), (d_j, {c_d}^j), (-1, \perp)\big)$, $m_l = \big((-1, \perp), (-1, \perp), (s_l, {c_s}^l)\big)$ *and* $\forall k \in]j, l[\,, m_k = \big((r_k, {c_r}^k), (-1, \perp), (-1, \perp)\big)$.

  The deadline d_j *should satisfy the following property:*

$$conv^{{c_d}^j}_{c_{BASE}}(d_j) \leq conv^{{c_s}^l}_{c_{BASE}}(s_l) + \Sigma_{k \in]j,l[} conv^{{c_r}^k}_{c_{BASE}}(r_k)\,,$$

- *Similarly, let* $\tau_j = (n_j, g_{X\,j}, m_j, k_j, f_j, n'_j)$ *and* $\tau_l = (n_l, g_{X\,l}, m_l, k_l, f_l, n'_l)$ *be two transitions in* S *such that* $j < l$, $m_j = \big((-1, \perp), (d_j, {c_d}^j), (-1, \perp)\big)$, $m_l = \big((-1, \perp), (d_l, {c_d}^l), (-1, \perp)\big)$ *and* $\forall k \in]j, l[$, $m_k = \big((r_k, {c_r}^k), (-1, \perp), (-1, \perp)\big)$. *The deadlines* d_j *and* d_l *should satisfy the following property:*

$$conv^{{c_d}^j}_{c_{BASE}}(d_j) \leq conv^{{c_d}^l}_{c_{BASE}}(d_l) + \Sigma_{k \in]j,l[} conv^{{c_r}^k}_{c_{BASE}}(r_k)\,.$$

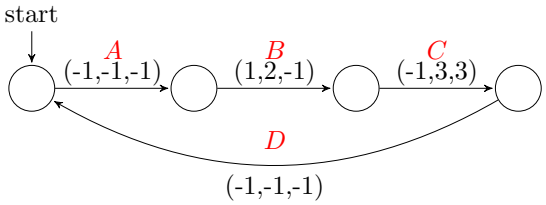
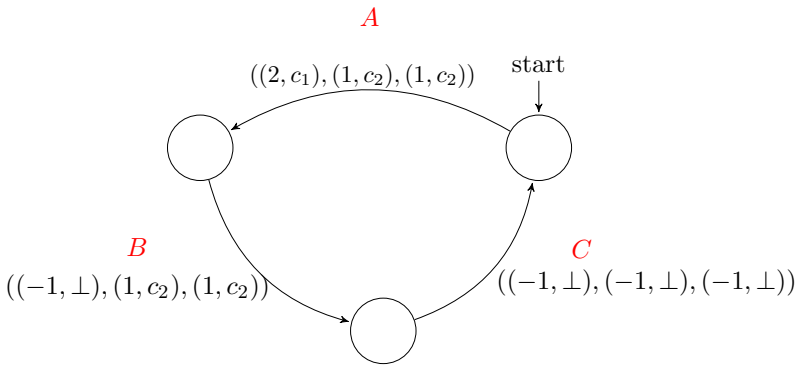


Figure 5.4: Alternative representation of the task behavior of Figure 2.12

Note that ΨC is essentially a syntactic representation of the TCA formal model. The transformations from a ΨC code of a task behavior to a TCA automaton or vice versa are straightforward. In the following we explain how we obtain ΨC code of a TCA automaton. Each job of a TCA automaton corresponds to either a seperate body or a part of a body in the ΨC code level. Its translation starts by an "$after(r)$ *with* c_r" statement if the first component (r, c_r) of the triplet-label of the job is different from $(-1, \perp)$. The body executes, then, the update function of the job. It ends by an "$advance(s)$*with* c_s" statement if the component (s, c_s) is different from $(-1, \perp)$. Otherwise it ends by a "$before(d)$ *with* c_d" statement if the component (d, c_d) of the triplet-label is different from $(-1, \perp)$.

Example 5.1. *For example, Figure 5.4 shows the automaton representing the task behavior of the task of Figure 2.12. Since in the model of Figure 2.12 all constraints are*

defined over the same clock, we only show the first component, i.e. the shift, of each pair of the triplet-label. This triplet-label depends on timing instructions encompassing the job in the original body code. The label of the job A is $\big((-1,\perp),(-1,\perp),(-1,\perp)\big)$ *since in the original code, it is not preceded by an* **after** *instruction, nor succeeded by a* **before** *or* **advance** *instruction. Notice that, in the labels of job C, the deadline shift coincides with the corresponding synchronization shift, reflecting the fact that in the original behavior code, this job is succeeded by an* **advance** *instruction.*



(a) TCA automaton

```
body
{
  // Job A
  after(2) with c1;
  ComputationA();
  advance(1) with c2;

  // Job B
  ComputationB();
  advance(1) with c2;

  // Job C
  ComputationC();
}
```

(b) Code ΨC

Figure 5.5: An example of a TCA task with two clocks and its ΨC code

Example 5.2. *The example of Figure 5.5a shows a TCA automaton where constraints are defined over two clocks; the clock* c_1 *and the clock* c_2*. Its corresponding* ΨC *code is displayed in Figure 5.5b. In the triplet-label of the job A, the release instant is defined over the clock* c_1 *while the synchronization instant is defined over the clock* c_2*. In the corresponding* ΨC *code of Figure 5.5b, actions of job A are executed between an "***after***(2) with* c_1*" instruction and an "***advance***(1) with* c_2*". The job B defines in its triplet-label*

only a synchronization instant over the clock c_2, which is represented in the ΨC code by an instruction "`advance`*(1) with c_2". The job C does not define constraints in its triple-label. Thus, in the ΨC code, its actions are executed after the previously instantiated instruction.*

Defining the operational semantics of TCA automaton requires a notion of state. The state of a TCA automaton is described in four parts: the occupied node, the valuation of the data variables, the valuation of the clock variables and the valuation of a reference variable that stores the valuation of the global clock in the last defined release or synchronization instants. This reference variable is needed for absolute constraints computation (cf. Section 5.3 for further details). Based on this notion of state, TCA semantics can be defined as a labelled transition system as described by the following definition:

Definition 5.2 (Semantics of TCA)**.** *The semantics of a time-constrained automaton (N, K, X, C, T) is defined as a labelled transition system $(Q, K, \rightarrow)$, where $Q = N \times \mathcal{V}(X) \times \mathcal{V}(C) \times \mathbb{R}_{\geqslant 0}$ and $\rightarrow \subseteq Q \times K \times Q$ is the set of transitions, defined as follows. We denote by v the valuation function, and by $v(X)$ (resp. $v(C)$) its restriction to the set of variables X (resp. the set of clocks C). Let $(n, v(X), v(C), v(\lambda_{ref}))$ and $(n', v'(X), v'(C), v'(\lambda_{ref}))$ be two states, such that $v(c) \leq v'(c)$ for all $c \in C$. We have $(n, v(X), v(C), v(\lambda_{ref})) \xrightarrow{k} (n', v'(X), v'(C), v'(\lambda_{ref}))$ iff there exists a transition $\big(n, g_X, ((r, c_r), (d, c_d), (s, c_s)), k, f, n'\big) \in T$ such that :*

- $g_X(v(X)) = True$,
- $v'(X) = f(v(X))$,
- $v(c) \leq v'(c)$ *for all* $c \in C$,
- *if* $(r, c_r) \neq (-1, \perp)$, *then* $\forall c \in C \setminus \{c_{BASE}\}$, $shift^{c_r}_{c_{BASE}}(v(\lambda_{ref}), r) \leq conv^{c}_{c_{BASE}}\big(v'(c)\big)$,
- *if* $(d, c_d) \neq (-1, \perp)$ *and* $(r, c_r) = (-1, \perp)$, *then* $\forall c \in C \setminus \{c_{BASE}\}$, $conv^{c}_{c_{BASE}}\big(v'(c)\big) \leq shift^{c_d}_{c_{BASE}}(v(\lambda_{ref}), d)$,
- *if* $(d, c_d) \neq (-1, \perp)$ *and* $(r, c_r) \neq (-1, \perp)$, *then* $\forall c \in C \setminus \{c_{BASE}\}$, $conv^{c}_{c_{BASE}}\big(v'(c)\big) \leq shift^{c_d}_{c_{BASE}}(v(\lambda_{ref}), d) + conv^{c_r}_{c_{BASE}}\big(r\big)$,
- $v'(\lambda_{ref})$ *is updated to the value of the synchronization instant s, or the release instant r, shifted to the clock c_{BASE}. If none of these instants are defined, the valuation $v'(\lambda_{ref})$ is unchanged:*

$$v'(\lambda_{ref}) = \begin{cases} shift^{c_s}_{c_{BASE}}(v(\lambda_{ref}), s)\,, & \text{if } s \neq -1 \text{ and } r = -1, \\ shift^{c_r}_{c_{BASE}}(v(\lambda_{ref}), r) + conv^{c_s}_{c_{BASE}}(s)\,, & \text{if } s \neq -1 \text{ and } r \neq -1, \\ shift^{c_r}_{c_{BASE}}(v(\lambda_{ref}), r)\,, & \text{if } s = -1 \text{ and } r \neq -1, \\ v(\lambda_{ref})\,, & \text{if } s = -1 \text{ and } r = -1. \end{cases}$$

An execution sequence of a TCA is defined as follows:

Definition 5.3 (Execution Sequence)**.** *An execution sequence of a time-constrained automaton* (N, K, X, C, T) *from an initial state* $(n^0, v^0(X), v^0(C), v^0(\lambda_{ref}))$ *is a sequence of transitions:*

$$\{\big(n^i, v^i(X), v^i(C), v^i(\lambda_{ref})\big) \xrightarrow{k_i} \big(n^{i+1}, v^{i+1}(X), vi+1(C), v^{i+1}(\lambda_{ref})\big)\}_{i=1}^{n}\,,$$

where $k_i \in K$*, for all* $i \in [1, n]$*, and* $n \in \mathbb{Z}_+ \cup \{\infty\}$*. An execution sequence is* finite *if* $n \in \mathbb{Z}_+$*, it is* infinite *if* $n = \infty$*.*

5.3 Transformation Challenges

Transforming a TT-BIP* model into a PharOS application requires addressing several challenges.

Moving from absolute to relative constraints.
In TT-BIP, all constraints are defined in terms of absolute clock values. On the contrary, TCA and ΨC bear only relative constraints, i.e. as an increment to the last release instant of a preceeding triplet-label (corresponding to the previous `after` statement in ΨC) or the last preceeding synchronization instant (corresponding to the previous `advance` statement in ΨC).

In order to address this issue, we make use of the variable λ_{ref}. It is initiated to zero and updated whenever a TCA transition is holding in its triplet-label a release or synchronisation constraint (i.e. the second or the third components of the triplet-label is different from $(-1, \perp)$). In terms of ΨC code, the variable λ_{ref} is updated whenever an `after` or an `advance` statement is instantiated. Thus, λ_{ref} stores the valuation of the global clock in the last defined release or synchronization instants (i.e. the last visited `after` or `advance` statement is ΨC). Relative constraint $d_{relative}$ is computed from its corresponding absolute constraint $d_{absolute}$ following this formula:

$$d_{relative} = d_{absolute} - \lambda_{ref}\,. \tag{5.2}$$

Mapping of timing constraints.
Both BIP and TT-BIP models are based on an abstract notion of time. In particular, actions that correspond to the computational steps (jump transition) of the system are considered to be atomic and have zero execution times. Thus, only start instants of these actions have associated timing constraints. However, in TCA models, actions do not always have a zero execution time. They are considered to have both a release and a deadline instants. These instants can be easily specified by using `after` and `before` instructions of the ΨC language, which correspond to a release and deadline components of the triplet-labels in the TCA model presented in Section 5.2.

This issue can be addressed by applying the timing constraint of the original TT-BIP transition —applying originally to the start instant of the transition —to both the release and the deadline instants of the job in the obtained TCA automaton. Note that by doing so, the equivalence with the BIP model is preserved —since the transition is guaranteed to finish before the original timing constraint becomes *False*.

We figured out two options for transforming computational steps and delay steps into TCA jobs. The first option is the intuitive mapping solution while the second option presents a more elaborated solution. Both options are designed in such a way to allow the TCA jobs to have a non zero execution time. In both options, release and deadline instants of the obtained TCA jobs are mapped from the timing constraints of the original TT-BIP* transitions:

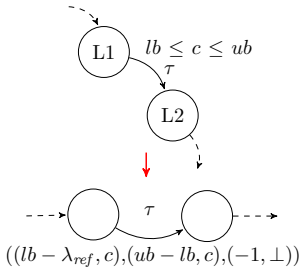
- **Option 1:** Let l be a location in the original TT-BIP* model such that $tpc(l) = (c \leq v)$ and let τ be the transition outgoing from l and having a timing constraint of the form $lb \leq c \leq ub$ in TT-BIP*. According to BIP semantics, τ has only its start instant constrained —since it is considered to have a zero execution time. It is supposed to start at any instant between the specified lower bound lb and the upper bound ub.

  For the computational step τ of the original model, we can include in the final TCA a job having lb and ub respectively as absolute release and deadline instants. This job has the same update action as the original transition τ and holds the following triplet-label $((lb - \lambda_{ref}, c), (ub - lb, c), (-1, \perp))$. This ensures that the instantiated job will start and end at an instant respecting the constraint of τ. The actions of the original transition τ are executed either within this job or in a new job depending on whether the original transition corresponds to an internal computation or a communication. The example in Figure 5.6a illustrates the mapping rule of a transition having a constraint of the form $lb \leq c \leq ub$.

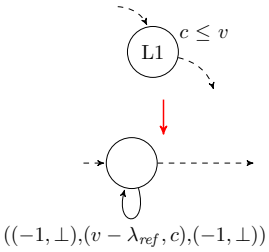
  In BIP semantics, delay steps can be constrained by timing progress conditions of the form $c \leq v$ indicating whether time can progress at a given state of the system. In TCA, this condition can be encoded by a loop job labelled by $((-1, \perp), (v - \lambda_{ref}, c), (-1, \perp))$, since in the original model the start instant of the delay step is not specified and only its deadline is defined (cf. Figure 5.6b).

- **Option 2:** The first option considers only the transition τ and omits other transitions enabled from the same location l. In this option, all the states of the original system are considered by taking into account all timing constraints of all outgoing transitions from the place l. We order all the bounds of these timing constraints and the tpc constraint. After ordering these bounds, we define computational steps that are enabled from l in each sub-interval separating two successive bounds. This is illustrated by the example of Figure 5.7.

  In this example, we consider a location l in the original TT-BIP* model such as $tpc(l) = (c \leq v)$. The transitions τ and τ' are two transitions outgoing from l



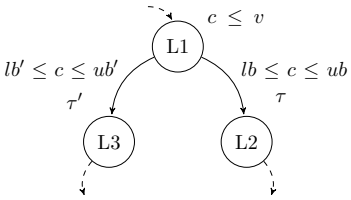
(a) Computational step constraint



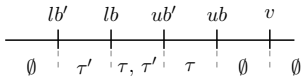
(b) Delay step constraint

Figure 5.6: Mapping of constraints: option 1

and having the respective timing constraints $lb \leq c \leq ub$ and $lb' \leq c \leq ub'$ (cf. Figure 5.7a). We assume that $lb' < lb < ub' < ub < v$. We define for each sub-interval the corresponding enabled transitions as displayed in Figure 5.7b.



(a) 0riginal transitions enabled from l



(b) Sub-intervals and their corresponding enabled transitions

Figure 5.7: Defining sub-intervals and their corresponding enabled transitions: option 2

In Figure 5.8, we show how the original transitions of Figure 5.7a are transformed. Each gray node and its outgoing jobs model one of the sub-intervals described previously. For example, when the system occupies the upper gray node N, the clock valuation is always inferior to lb'. Once the clock valuation reaches the instant lb', the system moves the next gray node—which models the next sub-interval. The loop job on the gray node allows to wait until lb' is reached. To do so, it defines the triplet-label $((1, c), (lb' - \lambda_{ref} - 1, c), (-1, \perp))$, where the release instant is the next instant over the clock c and the absolute deadline is lb'. The second job outgoing from the node N and leading to the next gray node marks the end of the current sub-interval by defining the triplet-label $((lb' - \lambda_{ref}, c), (0, c), (-1, \perp))$. Its absolute release and deadline instants are both the instant lb'. That is, once the absolute instant lb' is reached, the system should immediately move to the next gray node which models the next sub-interval.

From each gray node, we instantiate jobs corresponding to the original computational steps that are enabled in the current sub-interval —following Figure 5.7b.

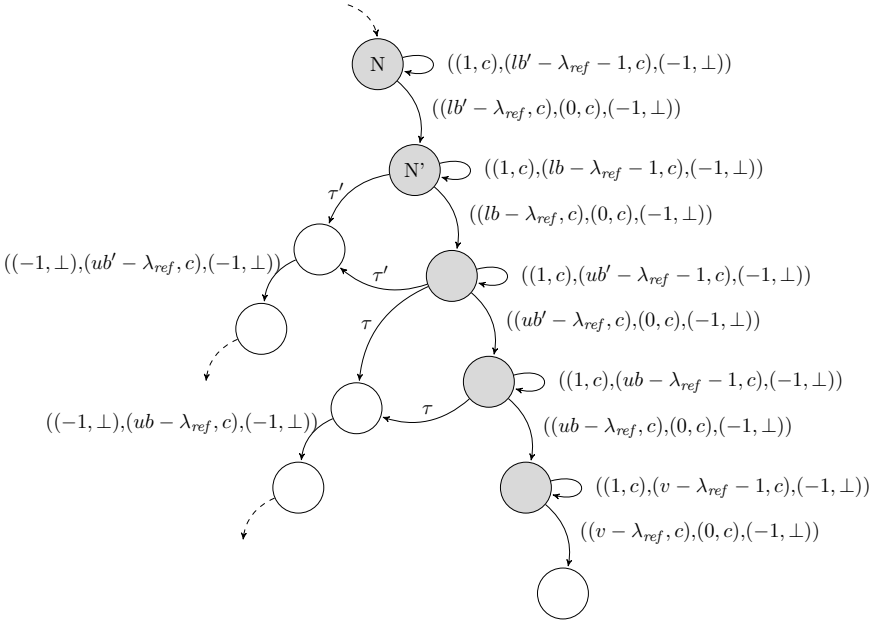


Figure 5.8: Mapping of constraints of Figure 5.7a: option2

Of course in this option, it should be ensured that each job starts and finishes in the same timing constraint as in the original model. For this reason, after each job corresponding to the computational step, a job defining a triplet-label that holds the original deadline is instantiated. For example, in Figure 5.8, the job instantiated after the job τ', hold the triplet-label $((-1, \perp), ((ub' - \lambda_{ref}, c), (-1, \perp))$ where ub' is the deadline of the transition τ' in the original TT-BIP* model of Figure 5.7a.

Note that the first option is the intuitive mapping solution which focuses on transforming the model transition by transition. The second option focuses rather on the state of the system and takes into account all potentially enabled transitions.

In order to avoid ad hoc solutions, we follow the mapping principles of the second option in the proposed transformation.

Communication mapping.

In the previous paragraph, we focused only on the temporal aspect of the transition omitting the fact that transitions are involved in communication. In this paragraph, we consider this aspect, and we detail how the communication is mapped in the TCA formalism.

In TT-BIP, all tasks are related to communication components via send/receive interactions, which provide unidirectional data transfer and synchronization between sending and receiving actions of, respectively, the sender and the receiver components. In TCA, the communication is performed through the temporal variable model. New values of

temporal variables are made visible at each of the synchronization points of the sender. These new values are consulted when the current time of receivers is greater than or equal to the visibility date of the new values. In our transformation two requirements need to be satisfied:

1. the receiver must consult an updated temporal variable (i.e. the receive job of the receive task must execute after the send job of the sender task) and

2. we need to respect communication semantics of the initial model, i.e. the synchronisation between send and receive jobs.

We generate TCA synchronization points (`advance` instructions in ΨC language) that depend on whether the TT-BIP* transition is triggered by a send, receive or an internal port. For each communicating transition in the original model, we instantiate —after jobs guaranteeing respect of timing constraint (cf. Figure 5.6) —a job containing, in its triplet-label, the synchronization component $(1, c_{fg})$, where c_{fg} is a fine-grained clock. Consider —in the original model —a sender and a receiver components having the same clock c. Suppose they are meant to communicate in the same instant t in TT-BIP* model. We can define a finer-grained clock c_{fg}, allowing the instantiation of synchronisation points (send and receive at $t + \epsilon$).

Example 5.3. *For example, consider the time line in Figure 5.9, where clock c_{fg} is n times finer-grained than the clock c, with $n > 2$. The visibility instant of the sender data is $n * t + 1$ of the clock c_{fg}. The receiver will consult these data in the instant $n * t + 2$ of the clock c_{fg}. In this example both requirements cited above are satisfied: (1) the sender updates the variable before the receiver consults it and (2) when considering the original clock c over which the synchronization instant t was defined, these send and receive instants can be approximated to t since the instant $t+1$ over c is still not reached.*

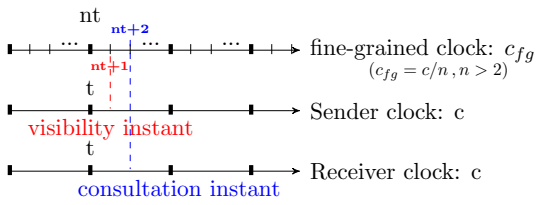


Figure 5.9: Example of `advance` nodes defined over c_{fg}

In order to address this challenge for an arbitrary original TT-BIP* model without resorting to ad hoc solutions, we proceed as follows:

- We define a fine-grained clock $c_{fg} = c^g/4$ where c^g is the unique global clock of the TT-BIP* model (as well as the TT-BIP model, cf. Section 4.4). All synchronization points (i.e. the third component of the triplet-label) are defined over this new clock.

- To each sending action, we associate a job labelled by $\big((-1,\perp),(1,c_{fg}),(1,c_{fg})\big)$ (i.e. `advance`(1) instruction defined over the clock c_{fg} in the ΨC code). Note that this job is instantiated after guaranteeing the respect of timing constraint of the original model (i.e. after instantiating jobs as in Figure 5.6). We add a Boolean flag in each transferred message, which will allow testing the freshness of the message. The sender automaton changes the state of this flag whenever a sending transition is executed. The receiver automaton has a local flag used as reference. The value of that flag is set to the value of the flag of the last received message.

- To each receiving transition, we associate a job labelled by $\big((-1,\perp),(2,c_{fg}),(2,c_{fg})\big)$ corresponding to successive reception attempts until the message is detected to be fresh. That is until the value of the local flag is different from the value of the flag of the message.

  Note that since in the TT-BIP model, all the receive-ports of an interaction are enabled if the send port is enabled (cf. the last property of Definition 4.1 of the TT-BIP model), we can be sure that the receiving job in the obtained TCA automaton will occur at latest one instant after the sending one over the clock c_{fg}. The synchronization requirement over the clock c^g is thus satisfied.

Notice that in the obtained TCA automaton, we have only two clocks, the clock c^g of the original TT-BIP component and the clock c_{fg} over which synchronisation points of the TCA automaton are exclusively defined.

Remark 5.1. *Note that an alternative solution would consist in defining a fine-grained clock $c_{fg} = c^g/3$ and a job labelled by $\big((-1,\perp),(1,c_{fg}),(1,c_{fg})\big)$ corresponding to sending actions and to each reception attempts. Nevertheless, in this solution a receiving task should execute more reception attempts than in the chosen solution.*

5.4 Transformation of a TT-BIP Model into TCA Models

In this section, we describe in details our technique for transforming a TT-BIP model. As explained in the third challenge of Section 5.3, connectors relating different layers are transformed into temporal variables, and different components are transformed into TCA automata. Note that each temporal variable is updated by only one TCA automaton (its owner).

Since in the TCA model, all communication constraints are taken into account, and the communication consists only in copying variables of the consulted temporal variable, we do not need to provide the formal composed model of TCA automata and all temporal variables.

Thus, in this section, we focus only on the formal transformation of each TT-BIP* component into a TCA automaton. Notice that the transformation of connectors into temporal variables is trivial and straightforward; It simply consists in instantiating a

temporal variable within the owner agent, and in defining its consulting agents. Its integration is, however, described in the next chapter when describing the implemented tool. Here, we present the rules of the transformation of a TT-BIP* component into a TCA automaton, while addressing challenges presented in Section 5.3.

The behavior of each TT-BIP* component $B = (L, P, X, \{c^g\}, T, tpc)$ with $P = P_i \cup P_s \cup P_r$ is transformed into a TCA automaton $TCA_B = (N, K, X_{TCA}, C_{TCA}, T_{TCA})$. The respective sets C_{TCA}, X_{TCA} and K are built from the original model following Rule 5.2:

Rule 5.2 (Instantiating sets of clocks, variables and job labels)**.**

- $C_{TCA} = \{c^g, c_{fg}\}$,
- $X_{TCA} = X \cup \{flag^p \,|\, p \in P_s \cap P_r\} \cup \{\lambda_{ref}\} \cup Y$, *where* Y *denotes the set of variables allowing to make local copies of variables of* X *after communication,*
- $K = P \times \{send, receive, internal\}$.

Before detailing rules for instantiating the set of nodes N and the set of transitions T_{TCA}, we need first to specify the rule allowing to order different bounds of timing constraints of transitions outgoing from each original location (cf. Rule 5.3).

Rule 5.3 (Ordering bounds of timing constraints)**.** *For each* $l \in L$*:*

- *we define the set* B_l^{bounds} *that includes lower and upper bounds of constraints of transitions* τ_p *triggered by port* p *such that* $p \in P_l$ *and the upper bound of the time progress condition* $tpc(l) = (c \leq v)$*. Note that we consider only finite bounds. The set* B_l^{bounds} *is defined as follows*

$$B_l^{bounds} = \{v \,|\, tpc(l) = (c \leq v)\} \cup \{lb_p, ub_p \,|\, p \in P_l \,, lb_p \leq v \,, ub_p \leq v\}.$$

- *we define* $sort(B_l^{bounds})$ *as the unique non decreasing sequence* $B_l = \{b_j\}_{j=0}^{n-1}$ *where duplicated elements are not preserved.* B_l *satisfies:*

$$|B_l| = n \quad , \quad n \leq |B_l^{bounds}| \,,$$
$$\forall 0 \leq j \leq |B_l| - 1 \,, b_j \in B_l^{bounds} \qquad and \qquad \forall 0 \leq j \leq |B_l| - 2 \,, b_j < b_{j+1}$$

After, defining the set $|B_l|$ for each $l \in L$, we include in N and T_{TCA} nodes and transitions allowing to model different intervals separating two successive bounds of $|B_l|$ as explained in Section 5.3.

Rule 5.4 (Introducing nodes and jobs corresponding to different sub-intervals)**.** *For each location* $l \in L$*:*

- *we include, in the set N, the nodes $\{N_l^j\}_{j=0}^n$, where n denotes the cardinality of the set B_l,*
- *and for each $j \in [0, n[$, we include in T_{TCA} the following transitions:*
  - *$\tau_{(l,b_j)}^{loop}$: It is a loop transition on the location N_l^j. It has the temporal constraints defined by the following triplet-label $((1, c^g), (b_j - \lambda_{ref}^{c^g} - 1, c^g), (-1, \perp))$ which allows waiting as long as the absolute instant b_j is not reached. Its release shift $(1, c^g)$ allows to increment $\lambda_{ref}^{c^g}$ by 1 at each execution of this transition. The job $\tau_{(l,b_j)}^{loop}$ is not guarded and does not execute an update function. It is labelled by $(p, internal) \in K$.*
  - *$\tau_{(l,b_j)}$: It is introduced to mark the end of the current sub-interval as explained before. It starts from location N_l^j and reaches the location N_l^{j+1}. And it is labelled by the triplet-label $((b_j - \lambda_{ref}^{c^g}, c^g), (0, c^g), (-1, \perp))$ which defines b_j as the absolute release and deadline instant. The transition $\tau_{(l,b_j)}$ is not guarded and does not execute an update function. It is labelled by $(p, internal) \in K$.*

Once bounds are ordered (cf. Rule 5.3) and sets of nodes and transitions allowing to define corresponding sub-intervals are defined (cf. Rule 5.4), we need to map enabled ports to each sub-interval. Rule 5.5 presents in details how we associate to each node N_l^j its enabled set of ports.

Rule 5.5 (Computing enabled ports for each defined sub-interval)**.** *Let $l \in L$, we denote by $P_l = \{p \in P \mid l \xrightarrow{p}\}$ the set of ports enabled in l. For each $l \in L$ and $j \in [0, |B_l|[$, we define a mapping function $\mu_l : [0, |B_l|] \to 2^{P_l}$. The function μ_l is a mapping that associates the set of enabled ports for each node N_l^j. Recall that lb_p and ub_p denote respectively the lower and the upper bounds of the timing constraint of the transition τ_p that is triggered by the port p, such that $p \in P_l$. The mapping μ_l is defined as follows:*

$$\mu_l(j) = \begin{cases} p\,, \text{ such that } p \in P_l \text{ and } lb_p < b_j \leq ub_p\,, & \text{if } j \in [0, |B_l| - 1], \\ p\,, \text{ such that } p \in P_l \text{ and } lb_p < b_{j-1} < ub_p\,, & \text{if } j = |B_l| \text{ and } b_j \neq v. \\ \emptyset\,, & \text{if } j = |B_l| \text{ and } b_j = v. \end{cases}$$

After defining the set of enabled ports from each node N_l^j, we detail how to map original computational steps. This transformation depends strongly on the type of port (i.e. internal, send or receive port). For example internal ports can be mapped into only one transition in the TCA, while a send port needs to be presented by more than one transition. Detailed transformation of each of either ports is presented in Rule 5.6.

Rule 5.6 (Mapping of computational steps)**.** *Let $l \in L$, we denote respectively by P_l^s, P_l^r and P_l^i the sets of send, receive and internal ports enabled from l, i.e. respectively the sets $P_l \cap P_s$, $P_l \cap P_r$ and $P_l \cap P_i$, where P_l is the set of ports enabled in l.*

- *Case 1: Internal port. For each $l \in L$, for each $j \in [0, |B_l|[$ and for each $p \in \mu_l(j) \cap P_l^i$ such that p is the trigger-port of the transition $\tau_p = (l, p, g_X, tc_p, r, f, l') \in T$ and $tc_p = (lb_p \leq c^g \leq ub_p)$, we include the transition $\tau^j_{(l,p)}$ in T_{TCA}. The transition $\tau^j_{(l,p)}$ is introduced to execute the original actions within the original constraints. Since the release instant is constrained by the bound b_j which is guaranteed to respect the original constraints, we only need to specify the original deadline of the original transition in the triplet-label of $\tau^j_{(l,p)}$. Thus, its triplet-label is as follows $((-1, \perp), (ub_p - \lambda^{c^g}_{ref}, c^g), (-1, \perp))$. The transition $\tau^j_{(l,p)}$ starts from location N^j_l and reaches the location $N^0_{l'}$. It is guarded by g_X and executes the update function f of the original transition τ_p. It is labelled by the label $(p, internal) \in K$ since $p \in P^i_l$,*

- *Case 2: Send port. For each $l \in L$ and for each $p \in \mu_l(j) \cap P^s_l$ such that p is the trigger-port of the transition $\tau_p = (l, p, g_X, tc_p, r, f, l') \in T$ and $tc_p = (lb_p \leq c^g \leq ub_p)$:*

  - *we include the node $N_{l'}$ in N,*
  - *for each $j \in [0, |B_l|[$, we include the transition $\tau^j_{(l,p)}$ in T_{TCA}. This transition is introduced in order to allow communication via a synchronization point. As defended in the third challenge of Section 5.3, sending actions are executed through a synchronisation on the next instant over the clock c_{fg} (corresponding to an* `advance`(1) *instruction in the ΨC language). Therefore, the triplet-label of the transition $\tau^j_{(l,p)}$ is the following: $((-1, \perp), (1, c_{fg}), (1, c_{fg}))$. The transition $\tau^j_{(l,p)}$ starts from location N^j_l and reaches the location $N_{l'}$. It is guarded by g_X and is labelled by the label $(p, send) \in K$. In order to prepare for the communication, it executes the update function f^p_{flag} which flips the message flag. Recall that the update function f is guaranteed to operate on variables that are not originally exported by the port p (cf. the fifth point of Definition 4.1). Therefore, we choose to execute the original update function f within the transition $\tau^j_{(l,p)}$,*
  - *we include the transition $\tau'_{(l,p)}$ in T_{TCA}. This transition is introduced to allow the time to progress until the original deadline is reached. It is, thus, labelled by the triplet-label $((-1, \perp), (ub_p - \lambda^{c^g}_{ref}, c^g), (-1, \perp)$. It has as source and target locations respectively $N_{l'}$ and $N^0_{l'}$. It is not guarded and defines no update function. It is labelled by the label $(p, internal) \in K$.*

- *Case 3: Receive port. By construction of the TT-BIP model, all transitions that are triggered by receive ports always carry timing constraints and guards that are default to True (cf. sixth point of Definition 4.1). It is also worth noticing, that by construction of the transformation and in contrary to send ports, several transitions labelled by receive ports can have the same source location (cf. the fourth point of Definition 4.1). Therefore, putting a synchronization point for reception (i.e. an*

advance(2) *instruction in the ΨC language) does not tell on which receive port, the current automaton is communicating. We add a flag that is tested on each received message in order to detect its freshness. For each* $l \in L$ *such that* $P_l^r \neq \emptyset$, *for each* $j \in [0, |B_l|[$:

- *we include the loop transition* $\tau^j_{(l,r)}$ *in* T_{TCA}. *This transition is introduced to allow the synchronization (communication) via a synchronization point. It consists in a loop transition on location* N_l^j, *in order to guarantee the reception of at least one message. Its triplet-label is equal to* $((-1,\perp),(2,c_{fg}),(2,c_{fg}))$ *(corresponding to an* *advance*(2) *instruction in the ΨC language). Its guard is the conjunction* $\bigwedge_{p\in P_l^r} \neg g^p_{fresh}$, *where* g^p_{fresh} *is the guard allowing —when evaluated to True—to detect the freshness of the received message through the receive port p. More details about all communication encoding (*f^p_{flag} *and* g^p_{fresh}*) are provided in the next paragraph. The transition* $\tau^j_{(l,r)}$ *does not define an update function. It is labelled by* $(p, internal) \in K$.
- *for each* $p \in P_l^r$ *such that p is the trigger-port of the transition* $\tau_p = (l, p, g_X, tc_p, r, f, l') \in T$, *we include the transition* $\tau^j_{(l,p)}$ *in* T_{TCA}. *This transition is introduced in order to execute actions of the original transition* $\tau_p \in T$ *after synchronization. It starts from location* N_l^j *and reaches the location* $N_{l'}^0$. *It has the triplet-label* $((-1,\perp),(-1,\perp),(-1,\perp))$, *is guarded by* g^p_{fresh} *and executes the update function* f^p_{update} *before executing the update function f of the original transition* $\tau_p \in T$. *Note that* f^p_{update} *is in charge of making local copies of variables of the received message. The transition* $\tau^j_{(l,p)}$ *is labelled by* $(p, receive) \in K$.

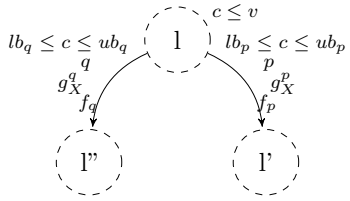
The transformation rules —that are detailed in Rule 5.2, Rule 5.3, Rule 5.4, Rule 5.5 and Rule 5.6—cover all conflict cases of TT-BIP and TT-BIP* models (cf. fourth point in Definition 4.1). Figure 5.10, Figure 5.11, Figure 5.12 and Figure 5.13 illustrate different conflict scenarios and display the sets of transitions of the obtained TCA automata.

In the following paragraph, we provide more details about encoding of f^p_{flag} and g^p_{fresh}. And we show how $\tau^j_{(l,r)}$ and $\tau_{(l,p)^j}$, $p \in P_l^r$ allow the reception of the actual message.

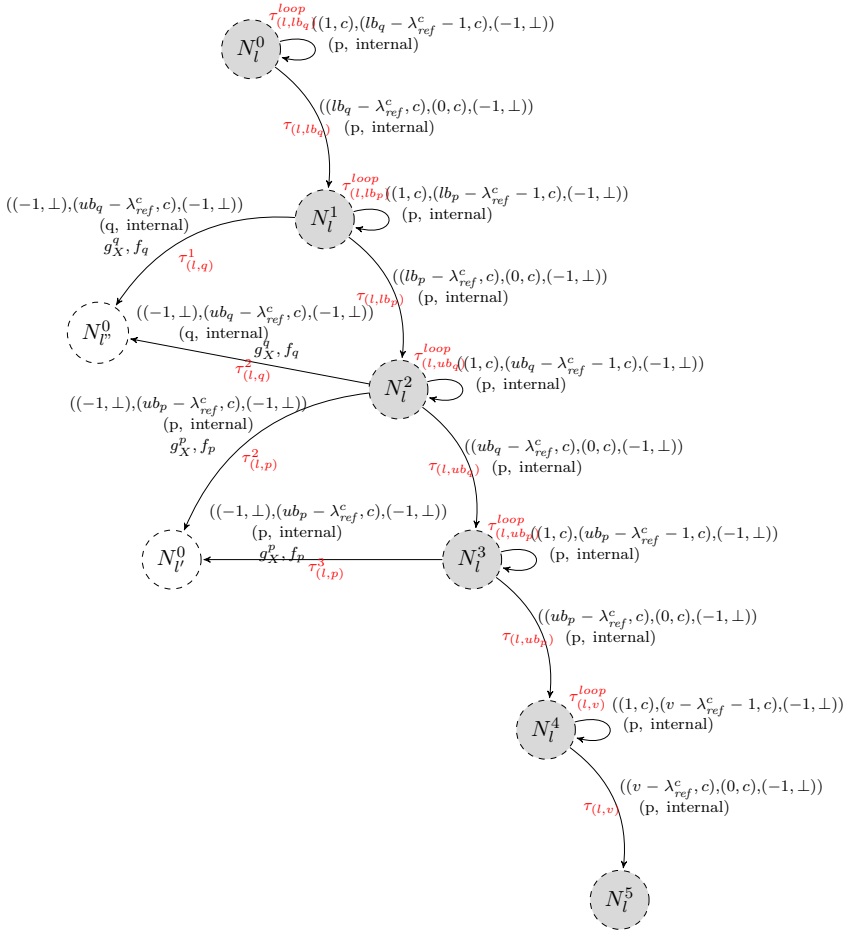
Encoding of communication details.
Consider a receive port p of a TT-BIP* component B and the local Boolean variable $flag^p$ in the corresponding TCA automaton TCA_B. Denote the Boolean flag of the message received through p by $flag^{msg}$. The guard g^p_{fresh} is defined by putting

$$g^p_{fresh} \stackrel{def}{=} (flag^p \neq flag^{msg}).$$

By construction, $flag^p$ and $flag^{msg}$ are initialized to zero. Thus, initially, we have $g^p_{fresh} =$ *False* and the loop transition $\tau^j_{(l,r)}$ is enabled (cf. Figure 5.12). This transition will



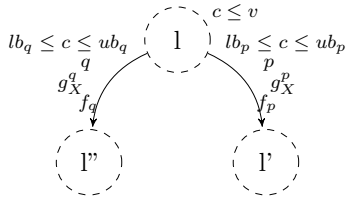
(a) Original TT-BIP transitions



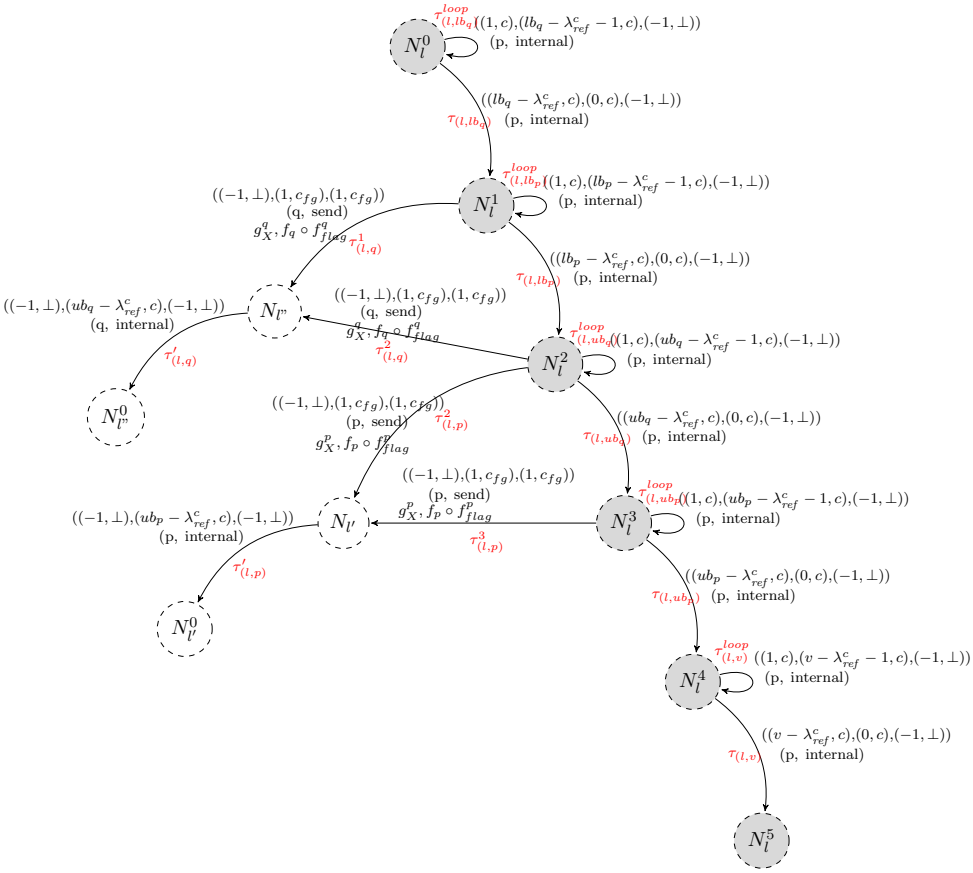
(b) Port $p, q \in P_i$ and $lb_q \leq lb_p \leq ub_q \leq ub_p \leq v$

Figure 5.10: Example of transformation of two conflicting transitions triggered by internal ports

perform a communication attempt (through the triplet label $((-1,\perp),(2,c_{fg}),(2,c_{fg}))$) with no actions on local variables. Each communication attempt leads to the implicit update of the guard g^p_{fresh} depending on the flag of the received message. If the sender has sent a new message—through its corresponding transition $\tau^{j'}_{(l',p')}$, labelled by its send



(a) Original TT-BIP transitions



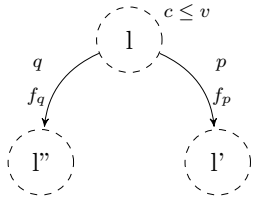
(b) Port $p\,, q \in P_s$ and $lb_q \leq lb_p \leq ub_q \leq ub_p \leq v$

Figure 5.11: Example of transformation of two conflicting transitions triggered by send ports

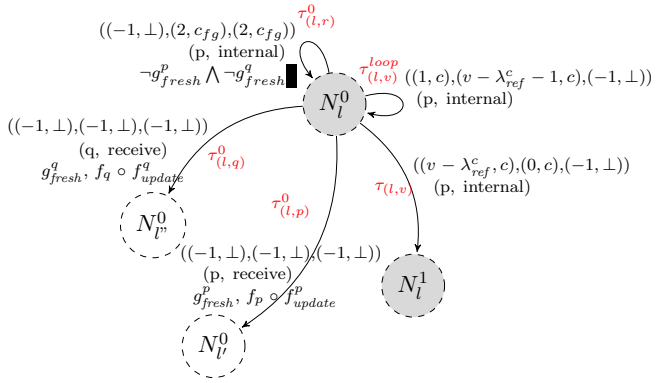
port $p' \in P^s_{l'}$—it should have performed the function $f^{p'}_{flag}$ in order to change the value of $flag^{msg}$ with:

$$f^{p'}_{flag} = (flag^{p'} := \neg flag^{p'})\,.$$

Recall that $flag^{p'}$ is a local variable of the sending component, whereof the value is incorporated into the message. Upon reception of the message by the receiving component, we denote this value by $flag^{msg}$. Thus, upon reception of the message g^p_{fresh} evaluates to



(a) Original TT-BIP transitions



(b) Port $p\,, q \in P_r$

Figure 5.12: Example of transformation of two conflicting transitions triggered by receive ports

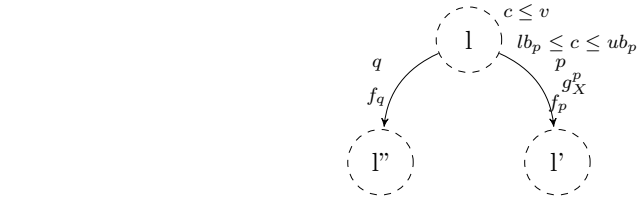
True, enabling the transition $\tau^j_{(l,p)}$ in the receiver automaton. Otherwise, if the sender did not send the new message yet, g^p_{fresh} evaluates to *False* and the transition $\tau^j_{(l,r)}$ (of the receiver automaton) is again enabled.

Notice also that among the values contained in the message, only $flag^{msg}$ is tested after execution of transition $\tau^j_{(l,r)}$. This value is only used to evaluate the freshness of each received message.

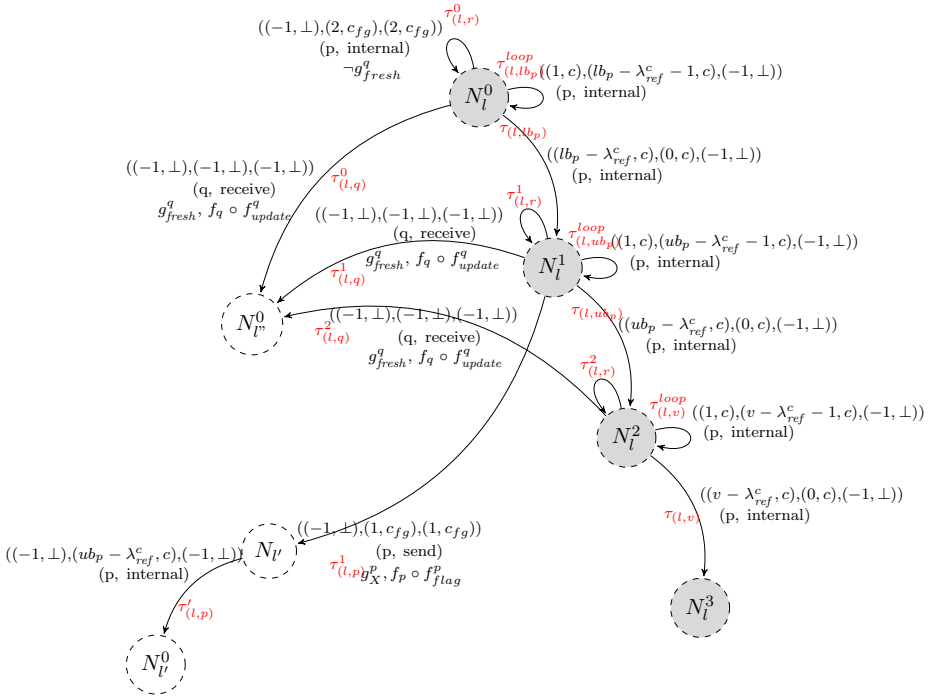
Since the transition $\tau^j_{(l,p)}$ of the receiver automaton is executed when the received message is fresh, it is in charge of making local copies of message variables through the function f^p_{update} before executing the function f of the initial transition. The function f^p_{update} copies also the value of $flag^{msg}$ into $flag^p$, thereby also changing the value of g^p_{fresh} from *True* to *False*.

Example 5.4. *We take as an example a task component having as a unique component the ATC component of Figure 4.7. In Figure 5.14, we show the TCA automaton obtained after transforming this task component behavior. Note that, for the sake of simplicity of the presentation of transitions $\tau^1_{(l_2,i_2)}$ and $\tau^0_{(l_3,i_3)}$ which loop back to the location $N^0_{\perp^l_p}$, we duplicate this latter (displayed in light gray) at the bottom of the TCA automaton.*

To summarise, the TCA automaton obtained from a given TT-BIP* component (by rules Rule 5.2, Rule 5.3, Rule 5.4, Rule 5.5 and Rule 5.6) can be formally defined as



(a) Original TT-BIP transitions



(b) Port $p \in P_s, q \in P_r$ and $lb_p \leq ub_p \leq v$

Figure 5.13: Example of transformation of two conflicting transitions triggered respectively by a send and a receive port

follows:

Definition 5.4. *Let $B = (L, P, X, \{c^g\}, T, tpc)$ be a TT-BIP* component with $P = P_i \cup P_s \cup P_r$. Recall that for $l \in L$, we denote by $P_l = \{p \in P \,|\, l \xrightarrow{p}\}$ the set of ports enabled in l. We denote respectively by P^s_l, P^r_l and P^i_l the sets of send, receive and internal ports enabled from l, i.e. respectively the sets $P_l \cap P_s$, $P_l \cap P_r$ and $P_l \cap P_i$. Recall also that B_l denotes the set of bounds of constraints of transitions enabled from l (cf. Rule 5.3). We denote by $\tau_p = (l, p, g_X, tc_p, r_p, f_p, l'_p) \in T$ an outgoing transition from a location $l \in L$, such that $tc_p = (lb_p \leq c^g \leq ub_p)$ and $tpc(l) = (c^g \leq v)$.*

The TCA corresponding to B is defined by putting $TCA_B = (N, K, X \cup X' \cup Y \cup \{\lambda_{ref}\}, \{c^g\} \cup \{c_{fg}\}, T')$, with $N = \{N^j_l \,|\, l \in L, j \in [0, |B_l|]\} \cup \{N_{l'_p} | l'_p \in L, p \in P_l \cap$

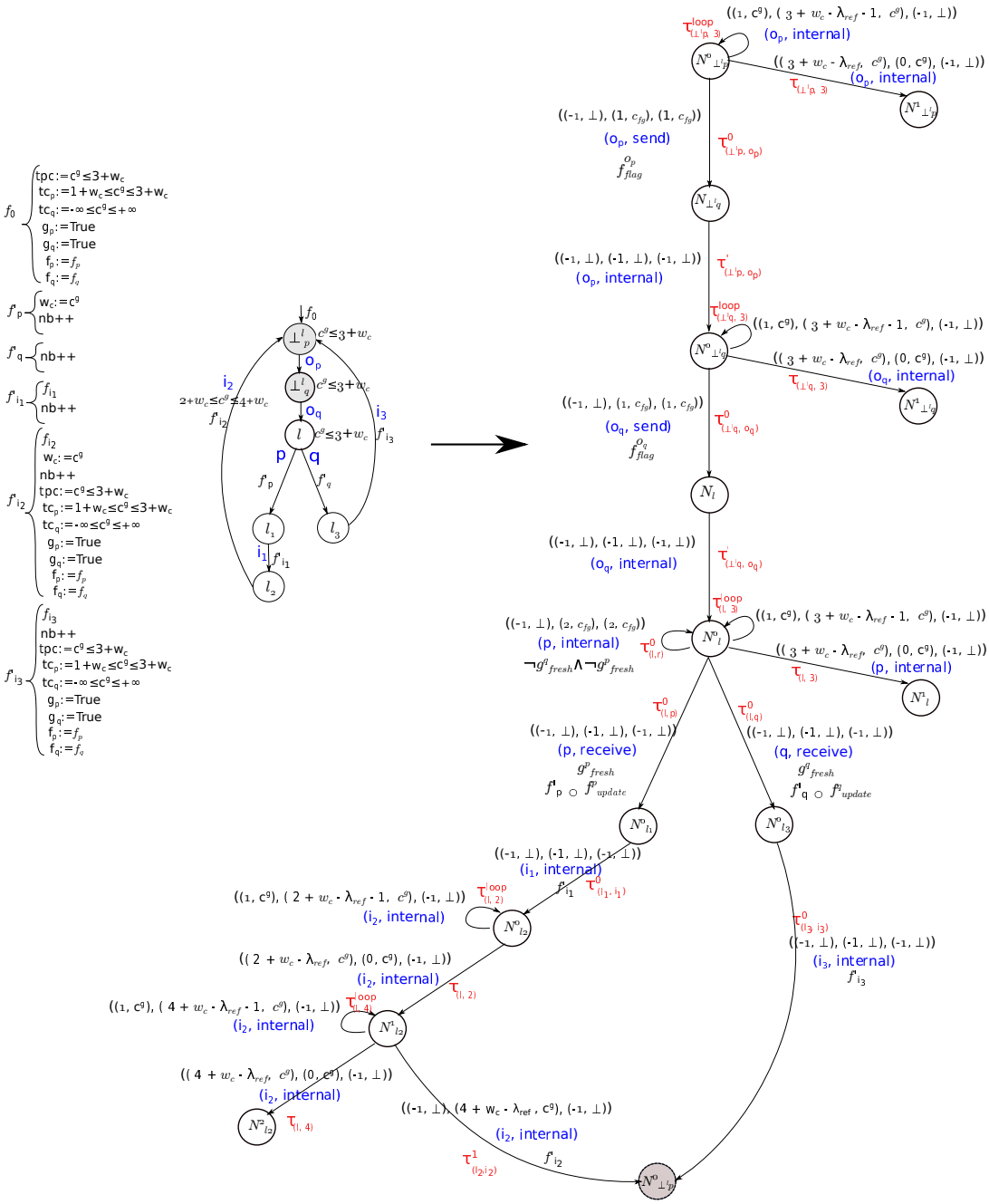


Figure 5.14: TCA model obtained after transforming task components of Figure 4.7

$P_s\}, K = P \times \{send, receive, internal\}, X' = \{flag^p \mid p \in P_s \cup P_r\}$ *and* Y *are the sets of flags and variables used for managing communication. The set of transitions* T' *contains*

the following transitions:

$$\tau^{loop}_{(l,b_i)} = \left(N^j_l,\, True, ((1,c^g),(b_i-\lambda^{c^g}_{ref}-1,c^g),(-1,\bot)),(p,internal),id,N^j_l\right), \qquad \forall l\in L,\forall j\in[0,|B_l|[,$$

$$\tau_{(l,b_i)} = \left(N^j_l,\, True, ((b_i-\lambda^{c^g}_{ref},c^g),(0,c^g),(-1,\bot)),(p,internal),id,N^{j+1}_l\right), \qquad \forall l\in L,\forall j\in[0,|B_l|-1[,$$

$$\tau^j_{(l,r)} = \left(N^j_l,\, \bigwedge_{p\in P^r_l}\neg g^p_{fresh}, ((-1,\bot),(2,c_{fg}),(2,c_{fg})),(p,internal),id,N^j_l\right), \qquad \forall l\in L, P^r_l\neq\emptyset,\forall j\in[0,|B_l|[,$$

$$\tau'_{(l,p)} = \left(N_{l'},\, True, ((-1,\bot),(ub_p-\lambda^{c^g}_{ref},c^g),(-1,\bot)),(p,internal),id,N^0_{l'}\right), \qquad \forall l\in L,\forall p\in P^s_l,$$

$$\tau^j_{(l,p)} =$$

$$\begin{cases} \left(N^j_l, g^p_{fresh}, ((-1,\bot),(-1,\bot),(-1,\bot)),(p,receive),f_p\circ f^p_{update},N^0_{l'}\right), & \forall l\in L,\forall j\in[0,|B_l|[,\forall p\in P^r_l,\\ \left(N^j_l, g_X, ((-1,\bot),(1,c_{fg}),(1,c_{fg})),(p,send),f_p\circ f^p_{flag},N_{l'}\right), & \forall l\in L,\forall j\in[0,|B_l|[,\forall p\in\mu_l(j)\cap P^s_l,\\ \left(N^j_l, g_X, ((-1,\bot),(ub_p-\lambda^{c^g}_{ref},c^g),(-1,\bot)),(p,internal),f_p,N^0_{l'}\right), & \forall l\in L,\forall j\in[0,|B_l|[,\forall p\in\mu_l(j)\cap P^i_l, \end{cases}$$

where $\mu_l : [1,|B_l|] \to 2^{P_l{}^s}$ is a mapping that associates the set of activated ports for each state of the system (cf. Rule 5.5), id is the identity function, $f^p_{flag} : \mathcal{V}(X') \to \mathcal{V}(X')$ is the function that flips the value of the Boolean variable $flag^p$ before sending a message, g^p_{fresh} is the guard verifying whether the value of $flag^p$ is different from that contained in the received message, $f^p_{update} : \mathcal{V}(X\cup Y) \to \mathcal{V}(X)$ is the function updating local variables according to received values if $p \in P_r$ and c_{fg} is the clock having one fourth of the period of the TT-BIP model clock c^g (cf. Section 5.3).

Notice that the domain and co-domain of the function f in the transition τ above are given by $f : \mathcal{V}(X) \to \mathcal{V}(X)$. Hence the composition $f \circ f^p_{update}$ is well-defined.

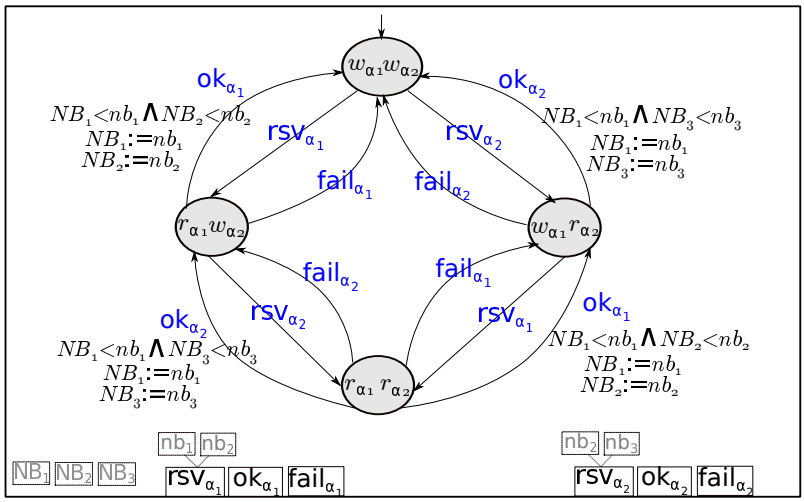
In Figure 5.15b, we display the obtained TCA automaton after transformation of the CRP automaton of Figure 5.15a. Originally, from the location $w_{\alpha_1}w_{\alpha_2}$, two receive transitions are conflicting (i.e. the rsv_{α_1} and rsv_{α_2} transitions). These transitions are transformed as shown in the pattern of Figure 5.12b. From the locations $r_{\alpha_1}w_{\alpha_2}$ (resp. $w_{\alpha_1}r_{\alpha_2}$) there is no conflict between receive transitions. Therefore, the enabled receive transition rsv_{α_2} (resp. rsv_{α_1}) is mapped to the loop transition $\tau^0_{(r_{\alpha_1}w_{\alpha_2},r)}$ (resp. $\tau^0_{(w_{\alpha_1}r_{\alpha_2},r)}$) and the transition $\tau^0_{(r_{\alpha_1}w_{\alpha_2},rsv_{\alpha_2})}$ (resp. $\tau^0_{(w_{\alpha_1}r_{\alpha_2},rsv_{\alpha_1})}$). From locations $r_{\alpha_1}w_{\alpha_2}$, $w_{\alpha_1}r_{\alpha_2}$ and $r_{\alpha_1}r_{\alpha_2}$, send transitions are conflicting. Transformation of these latter follows the pattern of Figure 5.11b.

5.5 Transformation Correctness

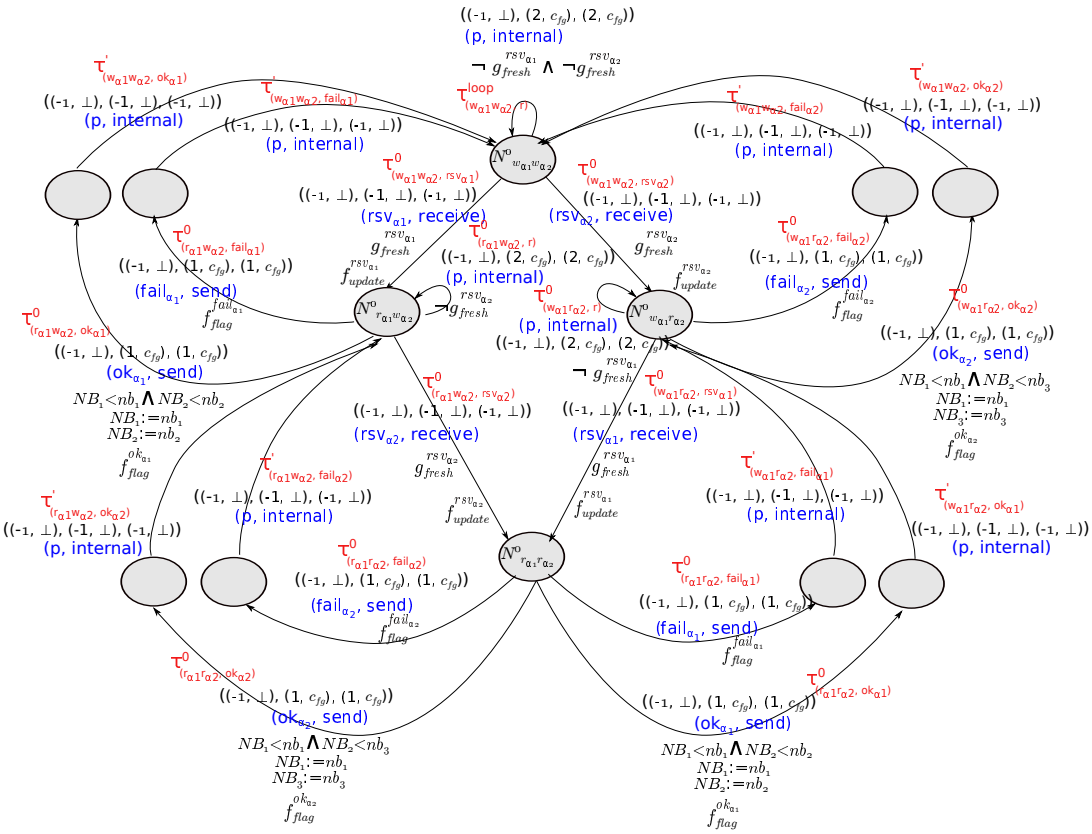
In order to prove the correctness of the transformation from TT-BIP* to TCA, we have to show that the corresponding semantic LTS are equivalent. This is illustrated in Figure 5.16, where F denotes the transformation from TT-BIP to TCA (Definition 5.4), G_1 and G_2 denote the corresponding LTS semantics.

We define observational equivalence between transition systems based on the classical notion of weak bisimilarity [69], where some transitions are considered unobservable.

We will use the same notations as in Section 4.5.2.



(a) Original BIP automaton of CRPcomponent



(b) Obtained TCA automaton after transformation of the CRP

Figure 5.15: Transformation of the CRP component

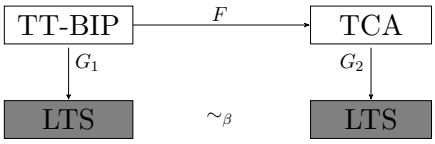


Figure 5.16: Translation functions

Let $B = (L, P, X, C, T, tpc)$ be a TT-BIP* component. We need to prove equivalence between $G_1(B)$ and $G_2(F(B))$. To this end, we define the following relation on labels of the two LTSs:

$$\beta = \left\{ \big(p, (p, send)\big) \,|\, p \in P_s \right\} \cup \left\{ \big(p, (p, receive)\big) \,|\, p \in P_r \right\}. \tag{5.3}$$

Theorem 5.1. *The LTSs $G_1(B)$ and $G_2(F(B))$ are weakly bisimilar w.r.t. β, i.e. $G_1(B) \sim_\beta G_2(F(B))$.*

Proof. Let $G_1(B) = (Q_B, P, \underset{B}{\longrightarrow})$ and $G_2(F(B)) = (Q_{TCA}, K, \underset{TCA}{\longrightarrow})$. Recall (Definition 1.11) that state space Q_B has three components: control location, clock and variable valuations while the state space Q_{TCA} (Definition 5.2) has an extra fourth component—besides the three components previously cited—consisting in the valuation of the reference instant λ_{ref}. For a given state q, we will denote $v_c(q)$ (resp. $v_x(q)$) its clock (resp. variable) valuation component. For a given state $q \in Q_{TCA}$, we will denote $v_{\lambda_{ref}}(q)$ the corresponding valuation of λ_{ref}.

Below, we will use variables q_B, r_B, ranging over Q_B, and q_{TCA}, r_{TCA}, ranging over Q_{TCA} and denote their respective components as follows:

$$\begin{aligned}
q_B &= \big(l, v_x(q_B), v_c(q_B)\big), \\
r_B &= \big(l', v_x(r_B), v_c(r_B)\big), \\
q_{TCA} &= \big(n, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\big), \\
r_{TCA} &= \big(n', v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\big).
\end{aligned}$$

We define the relation $R \subseteq Q_B \times Q_{TCA}$ as follows:

$$R = \left\{ (q_B, q_{TCA}) \;\middle|\; \begin{array}{c} n \in \{N_l^j\}_{j=0}^{|B_l|} \cup \{N_l\}, \\ v_c(q_B) = v_c^*(q_{TCA}), \\ v_x(q_B) = v_x^*(q_{TCA}) \end{array} \right\} \tag{5.4}$$

where v_c^* (resp. v_x^*) is the restriction of v_c (resp. v_x) to the unique clock c of model TT-BIP (resp. variables X). That is the valuation function v_c^* (resp. v_x^*) is defined only over the clock (resp. variables) which are common between B and $F(B)$, i.e. excluding clock c_{fg} (resp. variables $X' \cup Y$) of $F(B)$.

In order to show that (R, β) is a weak bisimulation, we have to prove the following four assertions:

(i) $\forall (q_B, q_{TCA}) \in R\,,$

$$q_B \xrightarrow[B]{\beta} r_B \implies \exists (r_B, r_{TCA}) \in R : q_{TCA} \xrightarrow[TCA]{\beta^*} r_{TCA}\,,$$

(ii) $\forall (q_B, q_{TCA}) \in R\,,$

$$q_{TCA} \xrightarrow[TCA]{\beta} r_{TCA} \implies \exists (r_B, r_{TCA}) \in R : q_B \xrightarrow[B]{\beta^*} r_B\,,$$

(iii) $\forall (q_B, q_{TCA}) \in R\,, \forall p \in P\,,$

$$\beta(p) \neq \emptyset \wedge q_B \xrightarrow[B]{p} r_B \implies \exists (p,k) \in \beta : \exists (r_B, r_{TCA}) \in R : q_{TCA} \xrightarrow[TCA]{\beta^* k \beta^*} r_{TCA}\,,$$

(iv) $\forall (q_B, q_{TCA}) \in R\,, \forall k \in K\,,$

$$\beta^{-1}(k) \neq \emptyset \wedge q_{TCA} \xrightarrow[TCA]{k} r_{TCA} \implies \exists (p,k) \in \beta : \exists (r_B, r_{TCA}) \in R : q_B \xrightarrow[B]{\beta^* p \beta^*} r_B\,.$$

Hereafter, we detail proofs of each of these four points:

(i) If $q_B \xrightarrow[B]{\beta} r_B$, then by definition (5.3) of the relation β, the underlying transition is either labelled by an internal port or by a real number representing a delay transition. Note that if β corresponds to an internal port $p \in P_l^i$, by definition (5.4) of the relation R we have $n \in \{N_l^j\}_{p \in \mu(j)}$ (cf. Rule 5.5), $v_c(q_B) = v_c^*(q_{TCA})$ and $v_x(q_B) = v_x^*(q_{TCA})$. And if β corresponds to a real number, we have $n \in \{N_l^j\}_{p \in \mu(j)} \cup \{N_l\}$

Case 1: β corresponds to an internal port $p \in P_l^i$ and $n = N_l^j$ such that $p \in \mu_l(j)$.

By Definition 1.11, there is a transition $l \xrightarrow{(p, g_X, g_C, \emptyset, f)} l'$ in B (recall that no clocks are reset in TT-BIP models) where $g_C = lb_p \leq c^g \leq ub_p$ with

$$\begin{gathered} q_B = \big(l, v_x(q_B), v_c(q_B)\big)\,, \quad r_B = \big(l', v_x(r_B), v_c(r_B)\big)\,, \\ g_X(v_x(q_B)) = g_C(v_c(q_B)) = \mathit{True}, \quad v_x(r_B) = f(v_x(q_B)), \quad \text{and} \quad v_c(r_B) = v_c(q_B)\,. \end{gathered} \tag{5.5}$$

By definition of F (Definition 5.4), there is a corresponding transition $\tau^j_{(l,p)}$, such that $p \in \mu(j)$:

$$N_l^j \xrightarrow{(\mathit{True}, \big((-1,\perp),(ub_p - \lambda^c_{ref}, c),(-1,\perp)\big), (p, \mathit{internal}), f)} N_{l'}^0$$

in $F(B)$.
By construction (5.4) of R, we have
$q_{TCA} = \big(N_l^j, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\big)$, such that

$$v_c(q_B) = v_c^*(q_{TCA}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{TCA})\,. \tag{5.6}$$

Therefore, by definition of G_2 (Definition 5.2), we also have $q_{TCA} \xrightarrow[TCA]{(p,internal)} r_{TCA}$, where $r_{TCA} = \big(N_{l'}^0, v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\big)$, with

$$v_c(r_{TCA}) = v_c(q_{TCA}), \text{ and } v_x^*(r_{TCA}) = f\big(v_x^*(q_{TCA})\big)\,. \tag{5.7}$$

For the first equality of (5.7), we have $v_c(r_{TCA}) = v_c(q_{TCA})$ since it satisfies the constraint $v_c(r_{TCA}) \leq ub_p + v_{\lambda_{ref}}$ (since we have $g_C(v_c(q_B)) = \mathit{True}$) which respects semantics of Definition 5.2. For the last equality of (5.7), notice that, for internal ports $p \in P_i$, the function f in the transition $\tau^j_{(l,p)}$ only operates on variables in X, but not on those in $X' \cup Y$.

Combining (5.5), (5.6) and (5.7), we obtain that $v_c^*(r_{TCA}) = v_c(r_B)$ and $v_x^*(r_{TCA}) = v_x(r_B)$. Thus, we have $q_{TCA} \xrightarrow[TCA]{\beta} r_{TCA}$ and, by (5.4), $(r_B, r_{TCA}) \in R$.

Case 2: β is a delay $\delta \in \mathbb{R}$.

By Definition 1.11, there is a time progress constraint on location l in B, $tpc(l) = (c^g \leq v)$. Therefore:

$$\begin{gathered} q_B = \big(l, v_x(q_B), v_c(q_B)\big)\,, \quad r_B = \big(l, v_x(r_B), v_c(r_B)\big)\,, \\ v_x(r_B) = v_x(q_B), \quad \text{and} \quad v_c(r_B) = v_c(q_B) + \delta \leq v\,. \end{gathered} \tag{5.8}$$

By definition of F (Definition 5.4), there is a set of corresponding successive transitions $\tau'_{(l^*,p^*)}$, $\tau^{loop}_{(l,b_0)}$, $\tau_{(l,b_0)}$, $\tau^{loop}_{(l,b_1)}$, ..., $\tau^{loop}_{(l,m)}$:

$$N_l \xrightarrow{(\mathit{True},\big((-1,\perp),(ub_{p^*}-\lambda^{c^g}_{ref},c^g),(-1,\perp)\big),(p^*,\mathit{internal}),id)} N_l^0$$

$$N_l^0 \xrightarrow{(\mathit{True},\big((1,c^g),(b_0-\lambda^{c^g}_{ref}-1,c^g),(-1,\perp)\big),(p,\mathit{internal}),id)} N_l^0$$

$$N_l^0 \xrightarrow{(\mathit{True},\big((b_1-\lambda^{c^g}_{ref}-1,c^g),(0,c^g),(-1,\perp)\big),(p,\mathit{internal}),id)} N_l^1$$

$$N_l^1 \xrightarrow{(\mathit{True},\big((1,c^g),(b_1-\lambda^{c^g}_{ref}-1,c^g),(-1,\perp)\big),(p,\mathit{internal}),id)} N_l^1$$

$$N_l^1 \xrightarrow{\cdots} N_l^n \xrightarrow{(\mathit{True},\big((1,c^g),(v-\lambda^{c^g}_{ref}-1,c^g),(-1,\perp)\big),(p,\mathit{internal}),id)} N_l^n$$

in $F(B)$, such that N_l and $\tau'_{(l^*,p^*)}$ exist only if $p^* \in P_s$ such that $l^* \xrightarrow[B]{p^*} l$.

By construction (5.4) of R, we have:

$$q_{TCA} = \big(N_l, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\big) \text{ if } N \ni N_l$$

or

$$q_{TCA} = \big(N_l^0, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\big) \text{ if } N \not\ni N_l\,.$$

In both case, we have

$$v_c(q_B) = v_c^*(q_{TCA}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{TCA})\,. \tag{5.9}$$

Therefore, by definition of G_2 (Definition 5.2), we also have

$$q_{TCA} \xrightarrow[TCA]{(p,internal)} \ldots \xrightarrow[TCA]{(p,internal)} \ldots \xrightarrow[TCA]{(p,internal)} r_{TCA},$$

where $r_{TCA} = \big(N_l^m, v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\big)$, with

$$v_c^*(r_{TCA}) = v_c^*(q_{TCA}) + \delta \text{ and } v_x^*(r_{TCA}) = v_x^*(q_{TCA})\,. \tag{5.10}$$

Note that by (5.9), we obtain $v_c^*(q_{TCA}) + \delta = v_c(q_B) + \delta$ and by (5.8), we have $v_c^*(q_{TCA}) + \delta \leq v$. Therefore, by (5.10), we have

$$v_c^*(r_{TCA}) \leq v.$$

Note that the latter inequality respects semantics of definition of G_2 (Definition 5.2).

Combining (5.8), (5.9) and (5.10), we obtain that $v_c^*(r_{TCA}) = v_c(r_B)$ and $v_x^*(r_{TCA}) = v_x(r_B)$. Thus, we have $q_{TCA} \xrightarrow[TCA]{\beta^*} r_{TCA}$ and, by (5.4), $(r_B, r_{TCA}) \in R$.

(ii) If $(q_B, q_{TCA}) \in R$, $q_{TCA} \xrightarrow[TCA]{\beta} r_{TCA}$, then by definition (5.3) of the relation β, the transition β is neither labelled by $(p, send)$ nor $(p, receive)$. It can be labelled only by $(p, internal)$. Applying this to the definition (5.4) of the relation R, we deduce that this transition can be enabled only from nodes N_l^j and N_l if it exists (cf. Definition 5.4). Thus this β transition corresponds in $F(B)$ to one of these transitions; $\tau^{loop}_{(l,b_j)}$, $\tau_{(l,b_j)}$, $\tau^j_{(l,r)}$ if $P_l^r \neq \emptyset$ and $\tau_{(l^*,p^*)}$ such that $p^* \in P_s$ and $l^* \xrightarrow[B]{p^*} l$

Case 1: β corresponds to $\tau^{loop}_{(l,b_j)}$ in $F(B)$, for some $l \in L$.

By definition of G_2 (Definition 5.2), there is a transition $\tau^{loop}_{(l,b_j)}$:

$$N_l^j \xrightarrow{(True,\big((1,c^g),(b_j-\lambda^{c^g}_{ref}-1,c^g),(-1,\perp)\big),(p,internal),id)} N_l^j$$

in $F(B)$ with

$$\begin{aligned} q_{TCA} &= \big(N_l^j, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\big)\,, \\ r_{TCA} &= \big(N_l^j, v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\big)\,, \\ v_c(r_{TCA}) &= v_c(q_{TCA}) + \delta \leq b_j\,, \\ \text{and } v_x(r_{TCA}) &= v_x(q_{TCA})\,. \end{aligned} \tag{5.11}$$

By definition of F (Definition 5.4), we have either $tpc(l) = True$ or $tpc(l) = (c^g \leq v)$ in B such that $b_j \leq v$. By construction (5.4) of R, we have $q_B = \big(l, v_x(q_B), v_c(q_B)\big)$, such that

$$v_c(q_B) = v_c^*(q_{TCA}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{TCA})\,. \tag{5.12}$$

Therefore, by definition of G_1 (Definition 1.11), we have $q_B \xrightarrow[B]{\delta} r_B$, where $r_B = \big(l, v_x(r_B), v_c(r_B)\big)$, with

$$v_c(r_B) = v_c(q_B) + \delta, \quad \text{and} \quad v_x(r_B) = v_x(q_B)\,. \tag{5.13}$$

Note that by (5.12), we obtain $v_c(q_B) + \delta = v_c^*(q_{TCA}) + \delta$ and by (5.11), we obtain $v_c(q_B) + \delta \leq b_j$. If $tpc(l) = (c^g \leq v)$, we have $b_j \leq v$. Therefore, we have

$$v_c(q_B) + \delta \leq v.$$

This satisfies the constraint $v_c(r_B) \leq v$ of definition of G_1 (Definition 1.11).

Combining (5.11), (5.12) and (5.13), we obtain that $v_c^*(r_{TCA}) = v_c(r_B)$ and $v_x^*(r_{TCA}) = v_x(r_B)$. Thus, we have $q_B \xrightarrow[B]{\beta} r_B$ and, by (5.4), $(r_B, r_{TCA}) \in R$.

Case 2: β corresponds to $\tau_{(l,b_j)}$ in $F(B)$, for some $l \in L$.

By definition of G_2 (Definition 5.2), there is a transition $\tau_{(l,b_j)}$

$$N_l^j \xrightarrow{(True, \big((b_j - \lambda_{ref}^{c^g}, c^g), (0, c^g), (-1, \perp)\big), (p, internal), id)} N_l^{j+1}$$

in $F(B)$, such that

$$\begin{aligned} q_{TCA} &= \big(N_l^j, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\big)\,, \\ r_{TCA} &= \big(N_l^{j+1}, v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\big)\,, \\ v_c(r_{TCA}) &= v_c(q_{TCA}), \quad \text{and} \quad v_x(r_{TCA}) = v_x(q_{TCA})\,. \end{aligned} \tag{5.14}$$

By construction (5.4) of R, we have $q_B = \big(l, v_x(q_B), v_c(q_B)\big)$, such that

$$v_c(q_B) = v_c^*(q_{TCA}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{TCA})\,. \tag{5.15}$$

Combining (5.14) and (5.15), we obtain that $v_c^*(r_{TCA}) = v_c(q_B)$ and $v_x^*(r_{TCA}) = v_x(q_B)$. Thus, we have $q_B \xrightarrow[B]{} q_B$ and, by (5.4), $(q_B, r_{TCA}) \in R$.

Case 3: β corresponds to $\tau_{(l,r)}^j$ in $F(B)$ for some $l \in L$ such that $P_l^r \neq \emptyset$.

By definition of G_2 (Definition 5.2), there is a transition $\tau_{(l,r)}^j$:

$$N_l^j \xrightarrow{\big(\bigwedge_{p \in P_l \cap P_r} \neg g_{fresh}^p, \big((-1, \perp), (2, c_{fg}), (2, c_{fg})\big), (p, internal), id\big)} N_l^j$$

in $F(B)$ for some $j \in [0, |B_l|[$, with

$$\begin{aligned} q_{TCA} &= \left(N_l^j, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\right), \\ r_{TCA} &= \left(N_l^j, v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\right), \\ v_c^*(r_{TCA}) &= v_c^*(q_{TCA}), \\ v_x(r_{TCA}) &= v_x(q_{TCA}). \end{aligned} \tag{5.16}$$

The second-last equality of (5.16) (i.e. , $v_c^*(r_{TCA}) = v_c^*(q_{TCA})$) is explained by the following. By construction of the transformation F, the last property of Definition 4.1 is held since it is a state property, i.e. all TCA receiver tasks are enabling a receiving transition when a TCA sender task is enabling a sending transition. Thus, by construction, and as explained in the third paragraph of Section 5.3, the receiving transition of the receiving TCA automaton will occur at least one instant after the sending one over the clock c_{fg}.

Thus, after one execution of the loop transition $\tau^j_{(l,r)}$ in $F(B)$, the guard $\bigwedge_{p \in P_l \cap P_r} \neg g^p_{fresh}$ becomes *False*. That is, there exist $p \in P_l^r$, such that $g^p_{fresh} = True$

Notice that one execution of the transition $\tau^j_{(l,r)}$ increments the valuation of the clock c_{fg} by 2 units. Since the clock c_{fg} is having as granularity one fourth of the period of clock c^g, the valuation of this latter remains unchanged. Recall that the clock c_{fg} is excluded by the valuation v_c^*, which justifies the second-last equality of (5.16).

By construction (5.4) of R, we have $q_B = \left(l, v_x(q_B), v_c(q_B)\right)$, such that

$$v_c(q_B) = v_c^*(q_{TCA}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{TCA}). \tag{5.17}$$

Combining (5.16) and (5.17), we obtain that $v_c^*(r_{TCA}) = v_c(q_B)$ and $v_x^*(r_{TCA}) = v_x(q_B)$. Thus, we have $q_B \underset{B}{\longrightarrow} q_B$ and, by (5.4), $(q_B, r_{TCA}) \in R$.

Case 4: β corresponds to $\tau_{(l^*,p^*)}$ in $F(B)$ for some $l, l^* \in L$ and $p^* \in P_s$ such that $l^* \xrightarrow[B]{p^*} l$.

By definition of G_2 (Definition 5.2), there is a transition $\tau_{(l^*,p^*)}$

$$N_l \xrightarrow{\left(True, \left((-1,\perp),(ub_{p^*} - \lambda^{c^g}_{ref}, c^g),(-1,\perp)\right),(p,internal),id\right)} N_l^0,$$

in $F(B)$, such that

$$\begin{aligned} q_{TCA} &= \left(N_l, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\right), \\ r_{TCA} &= \left(N_l^0, v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\right), \\ v_c(r_{TCA}) &= v_c(q_{TCA}) + \delta \leq ub_{p^*}, \\ v_x(r_{TCA}) &= v_x(q_{TCA}). \end{aligned} \tag{5.18}$$

By definition of F (Definition 5.4), we have either $tpc(l) = True$ or $tpc(l) = (c^g \leq v)$ in B such that $ub_{p^*} \leq v$ (cf. Rule 5.5). By construction (5.4) of R, we have $q_B = \big(l, v_x(q_B), v_c(q_B)\big)$, such that

$$v_c(q_B) = v_c^*(q_{TCA}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{TCA})\,. \tag{5.19}$$

Therefore, by definition of G_1 (Definition 1.11), we have $q_B \xrightarrow[B]{\delta} r_B$, where $r_B = \big(l, v_x(r_B), v_c(r_B)\big)$, with

$$v_c(r_B) = v_c(q_B) + \delta, \quad \text{and} \quad v_x(r_B) = v_x(q_B)\,. \tag{5.20}$$

Note that by (5.19), we obtain $v_c(q_B) + \delta = v_c^*(q_{TCA}) + \delta$ and by (5.18), we obtain $v_c(q_B) + \delta \leq ub_{p^*}$. If $tpc(l) = (c^g \leq v)$, we have $ub_{p^*} \leq v$ by construction of the model B. Therefore, we have

$$v_c(q_B) + \delta \leq v.$$

This satisfies the constraint $v_c(r_B) \leq v$ of definition of G_1 (Definition 1.11).

Combining (5.18), (5.19) and (5.20), we obtain that $v_c^*(r_{TCA}) = v_c(r_B)$ and $v_x^*(r_{TCA}) = v_x(r_B)$. Thus, we have $q_B \xrightarrow[B]{\beta} r_B$ and, by (5.4), $(r_B, r_{TCA}) \in R$.

(iii) Let $(q_B, q_{TCA}) \in R$ such that $q_B \xrightarrow[B]{p} r_B$. If $\beta(p) \neq \emptyset \wedge q_B \xrightarrow[B]{p} r_B$, then by definition (5.3) of the relation β, $p \in P_l^r \cup P_l^s$).

By Definition 1.11, there is a transition Therefore, we have $lb_p \leq v_c(q_{TCA}) \leq ub_p$. This implies that, if $p \in P_l^s$, the node n is a node N_l^j such that $p \in \mu(j)$ (which respects the previous inequatily). And if $p \in P_l^r$, the node n is a node N_l^j, for all $j \in [0, |B_l|[$. Therefore, we deduce that $n \in \{N_l^j\}_{j=0}^{|B_l|}$.

Case 1: $p \in P_l^s$**.**

By Definition 1.11, there is a transition $l \xrightarrow{p, g_X, g_C, \emptyset, f_p} l'$ in B (recall that no clocks are reset in TT-BIP models), where $g_C = (lb_p \leq c^g \leq ub_p)$, such that

$$\begin{aligned} q_B = \big(l, v_x(q_B), v_c(q_B)\big)\,, \quad r_B = \big(l', v_x(r_B), v_c(r_B)\big)\,, \\ g_X(v_x(q_B)) = True, \quad lb_p \leq v_c(q_B) \leq ub_p, \\ v_x(r_B) = f_p(v_x(q_B)) \quad \text{and} \quad v_c(r_B) = v_c(q_B)\,. \end{aligned} \tag{5.21}$$

Note that we have $v_x(r_B) = f_p(v_x(q_B))$, since $p \in P_l^s$. The interaction, through which the component is communicating, does not define an update function on variables of the send port p (all interactions copy variable associated with the send port to the ones of the receive ports, cf. Section 4.4.5 of Chapter 4). By definition (5.4) of the relation R, we have $v_c(q_B) = v_c^*(q_{TCA})$. Therefore, we have

$lb_p \leq v_c(q_{TCA}) \leq ub_p$. By definition of Rule 5.5, we deduce that the node n of the state q_{TCA} is a node N_l^j such that $p \in \mu(j)$ (which respects the inequality $lb_p \leq v_c(q_{TCA}) \leq ub_p$).

By definition of F (Definition 5.4), there is a corresponding transition $\tau^j_{(l,p)}$,

$$N_l^j \xrightarrow{\left(g_X,\left((-1,\perp),(1,c_{fg}),(1,c_{fg})\right),(p,send),f_p\circ f^p_{flag}\right)} N_{l'}$$

in $F(B)$.

By construction (5.4) of R, we have $q_{TCA} = \big(N_l, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\big)$, such that

$$v_c(q_B) = v_c^*(q_{TCA}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{TCA})\,. \tag{5.22}$$

Therefore, by definition of G_2 (Definition 5.2), we also have

$$q_{TCA} \xrightarrow[TCA]{k} r_{TCA}\,,$$

where $k = (p, send)$ and

$$r_{TCA} = \big(N_{l'}, v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\big)\,,$$

with

$$\begin{aligned} v_c^*(r_{TCA}) &= v_c^*(q_{TCA})\,,\\ v_x^*(r_{TCA}) &= f_p\big(v_x^*(q_{TCA})\big)\,. \end{aligned} \tag{5.23}$$

In the first equality of (5.23), we have $v_c^*(r_{TCA}) = v_c^*(q_{TCA})$ since the transition $\tau^j_{(l,p)}$ increments by one unit only valuation of clock c_{fg} which is excluded by the valuation v_c^*.

For the last equality of (5.23), notice that, for send ports $p \in P_l^s$, the function f^p_{flag} in the transition $\tau^j_{(l,p)}$ operates on variables of X' which are excluded by the valuation v_x^*. The function f_p only operates on variables of X, but not on those of $X' \cup Y$.

Combining (5.21), (5.22) and (5.23) we obtain that $v_c^*(r_{TCA}) = v_c(r_B)$ and $v_x^*(r_{TCA}) = v_x(r_B)$. Thus, we have

$$q_{TCA} \xrightarrow[TCA]{k} r_{TCA}\,,$$

where $k = (p, send)$. By (5.4), $(r_B, r_{TCA}) \in R$.

Case 2: $p \in P_l^r$

By Definition 1.11, there is a transition $l \xrightarrow{(p,True,True,\emptyset,f_p)} l'$ in B. Recall that no clocks are reset in TT-BIP models and that all receive transitions carry timing

constraints and guards that are default to *True* (cf. sixth point of Definition 4.1). We have

$$\begin{aligned} q_B = \big(l, v_x(q_B), v_c(q_B)\big)\,, \quad r_B &= \big(l', v_x(r_B), v_c(r_B)\big)\,, \\ v_x(r_B) = f_p^*(v_x(q_B)), v_c(r_B) &= v_c(q_B)\,, \end{aligned} \tag{5.24}$$

where $f_p^* = f \circ f'^p_{update}$ is the composition of the function f with a function f'^p_{update} of the interaction through which the component is communicating via the port p (cf. Definition 1.14). By construction of the TT-BIP* models, we know that all cross-layer interactions are send/receive interactions which have as update function, the function copying variables of the send port to those of the receive ports (cf. Section 4.4.5 of Chapter 4). Thus, f'^p_{update} copies values of variables of the sender ports to those of the port $p \in P_r$. By definition of F (Definition 5.4), there is a corresponding transition $\tau^j_{(l,p)}$:

$$N_l^j \xrightarrow{\left(g^p_{fresh}, \big((-1,\perp),(-1,\perp),(-1,\perp)\big), (p,receive), f_p \circ f^p_{flag}\right)} N_{l'}^0$$

in $F(B)$.

By construction (5.4) of R, we have

$$q_{TCA} = \big(N_l^j, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\big),$$

such that

$$v_c(q_B) = v_c^*(q_{TCA}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{TCA})\,. \tag{5.25}$$

Therefore, by definition of G_2 (Definition 5.2), we also have

$$q_{TCA} \xrightarrow[TCA]{k} r_{TCA},$$

where $k = (p, receive)$, with

$$r_{TCA} = \big(N_{l'}^0, v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\big)\,,$$

with

$$\begin{aligned} v_c^*(r_{TCA}) &= v_c^*(q_{TCA})\,, \\ &\text{and}\,, \\ v_x^*(r_{TCA}) &= f_p \circ f^p_{update}\big(v_x^*(q_{TCA})\big)\,. \end{aligned} \tag{5.26}$$

For the first equality of (5.26), we have $v_c^*(r_{TCA}) = v_c^*(q_{TCA})$ since the instantaneous execution of the transition $\tau^j_{(l,p)}$ is possible (since it respects semantics of Definition 5.2). For the last equality of (5.26), notice that, for receive ports $p \in P_r$, in the transition $\tau^j_{(l,p)}$, the function f^p_{update} operates on variables in $X \cup Y$, but the valuation v_x^* is defined only on variables X. Thus, we have $f \circ f^p_{update}\big(v_x^*(q_{TCA})\big) = f \circ f'_{update}\big(v_x^*(q_{TCA})\big) = f^*\big(v_x^*(q_{TCA})\big)$.

Combining (5.24), (5.25) and (5.26) we obtain that $v_c^*(r_{TCA}) = v_c(r_B)$ and $v_x^*(r_{TCA}) = v_x(r_B)$. Thus, we have

$$q_{TCA} \xrightarrow[TCA]{k} r_{TCA}\,,$$

where $k = (p, receive)$ and, by (5.4), $(r_B, r_{TCA}) \in R$.

(iv) If $(q_B, q_{TCA}) \in R$ and $k \in K$ such that $q_{TCA} \xrightarrow[TCA]{k} r_{TCA}$, then by definition (5.3) of the relation β, $k = (p, send)$ or $k = (p, receive)$. By definition of F (Definition 5.4), we deduce that this transition can be enabled only from nodes N_l^j. Thus, if $k = (p, send)$ it corresponds to $\tau_{(l,p)}^j$ such that $p \in P_l^s \cap \mu(j)$. If $k = (p, receive)$, it corresponds to $\tau_{(l,p)}^j$ for all $j \in [0, |B_l|[$ such that $p \in P_l^r$.

Case 1: $k = (p, send)$ and $n = N_l^j$, for some $l \in L$ and j such that $p \in \mu(j)$.

By definition of G_2 (Definition 5.2), there is a transition $\tau_{(l,p)}^j$:

$$N_l^j \xrightarrow{\left(g_X, \left((-1,\perp),(1,c_{fg}),(1,c_{fg})\right), (p,send), f_p \circ f_{flag}^p\right)} N_{l'}$$

in $F(B)$, with

$$\begin{aligned} q_{TCA} &= \left(N_l^j, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\right), \\ r_{TCA} &= \left(N_{l'}, v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\right), \\ v_c(r_{TCA}) &= v_c(q_{TCA})\,, \\ v_x^*(r_{TCA}) &= f_p(v_x^*(q_{TCA}))\,. \end{aligned} \tag{5.27}$$

For the last equality of (5.27), notice that, for send ports $p \in P_l^s$, the function f_{flag}^p in the transition $\tau_{(l,p)}^j$ operates on variables of X' which are excluded by the valuation v_x^*. The function f_p only operates on variables of X, but not on those of $X' \cup Y$.

By definition of F (Definition 5.4), there is a corresponding transition

$$l \xrightarrow{(g_X, g_C, p, \emptyset, f_p)} l'\,,$$

in B.

By construction (5.4) of R, we have $q_B = \left(l, v_x(q_B), v_c(q_B)\right)$, such that

$$v_c(q_B) = v_c^*(q_{TCA}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{TCA})\,. \tag{5.28}$$

Therefore, by definition of G_1 (Definition 1.11), we also have $q_B \xrightarrow[B]{p} r_B$,, where

$$r_B = \left(l', v_x(r_B), v_c(r_B)\right),$$

with

$$\begin{aligned} g_X(v_x(q_B)) &= g_C(v_c(q_B)) = \mathit{True}\,, \\ v_c(r_B) &= v_c(q_B)\,, \\ v_x(r_B) &= f_p(v_x(q_B))\,. \end{aligned} \tag{5.29}$$

Combining (5.27), (5.28) and (5.29), we obtain that $v_c^*(r_{TCA}) = v_c(r_B)$ and $v_x^*(r_{TCA}) = v_x(r_B)$. Thus, we have $q_B \xrightarrow[B]{p} r_B$ and, by (5.4), $(r_B, r_{TCA}) \in R$.

Case 2: $k = (p, receive)$ and $n = N_l^j$, for some $l \in L$ and $j \in [0, |B_l|[$.

By definition of G_2 (Definition 5.2), there is a transition $\tau_{(l,p)}^j$:

$$N_l^j \xrightarrow{\left(g_{fresh}^p, \left((-1,\perp),(-1,\perp),(-1,\perp)\right), (p,receive), f_p \circ f_{update}^p\right)} N_{l'}^0$$

in $F(B)$, with

$$\begin{aligned} q_{TCA} &= \left(N_l^j, v_x(q_{TCA}), v_c(q_{TCA}), v_{\lambda_{ref}}(q_{TCA})\right), \\ r_{TCA} &= \left(N_{l'}^0, v_x(r_{TCA}), v_c(r_{TCA}), v_{\lambda_{ref}}(r_{TCA})\right), \\ v_c(r_{TCA}) &= v_c(q_{TCA})\,, \\ v_x^*(r_{TCA}) &= f_p \circ f_{update}^p(v_x^*(q_{TCA}))\,. \end{aligned} \tag{5.30}$$

Notice that even if the actual reception was performed in the β transition $\tau_{(l,r)}$ preceding this k transition, the update of local variables according the received message is only performed via the execution of the k transition (via f_{update}^p). The function f_{update}^p applies to variables of $X \cup Y$.

By definition of F (Definition 5.4), there is a corresponding transition

$$l \xrightarrow{\mathit{True}, \mathit{True}, p, \emptyset, f_p} l'$$

, in B. By construction (5.4) of R, we have $q_B = \left(l, v_x(q_B), v_c(q_B)\right)$, such that

$$v_c(q_B) = v_c^*(q_{TCA}) \quad \text{and} \quad v_x(q_B) = v_x^*(q_{TCA})\,. \tag{5.31}$$

Therefore, by definition of G_1 (Definition 1.11), we also have $q_B \xrightarrow[B]{p} r_B$, where

$$r_B = \left(l', v_x(r_B), v_c(r_B)\right),$$

with

$$\begin{aligned} v_x(r_B) &= f^*(v_x(q_B))\,, \\ v_c(r_B) &= v_c(q_B)\,, \end{aligned} \tag{5.32}$$

where $f^* = f_p \circ f'^p_{update}$ is the composition of the function f_p with a function f'^p_{update} of the interaction through which the component is communicating via the

port p (cf. Definition 1.14). By construction of the TT-BIP* models, we know that all cross-layer interactions are send/receive interactions which have as update function, the function copying variables of the send port to those of the receive ports (cf. Section 4.4.5 of Chapter 4). Thus, f'^{p}_{update} copies variables of the send port to those of the receive port p. Knowing that, f'^{p}_{update} is defined over X while f^{p}_{update} is defined over $X \cup Y$, we have

$$\begin{aligned} f'^{p}_{update} &= f^{p}_{update_{|X}} \\ v^{*}_{x}(r_{TCA}) &= f_p \circ f'^{p}_{update}(v^{*}_{x}(q_{TCA})) = f^{*}(q_{TCA}) \end{aligned} \tag{5.33}$$

Combining (5.30), (5.31), (5.32) and (5.33), we obtain that $v^{*}_{c}(r_{TCA}) = v_c(r_B)$ and $v^{*}_{x}(r_{TCA}) = v_x(r_B)$. Thus, we have $q_B \xrightarrow[B]{p} r_B$ and, by (5.4), $(r_B, r_{TCA}) \in R$.

□

5.6 Compatibility with the Composition Correctness

In Section 5.5, we prove that the transformation of individual TT-BIP* components into TCA automata is semantics-preserving. In this section, we explain why the composition of all obtained TCA automata is equivalent to the initial TT-BIP* model.
Both glues of TCA automata and TT-BIP* components provide the same unidirectional transfer of data. The unique difference is that in TT-BIP*, interactions provide synchronisation on top of data transfer while in TCA the communication is asynchronous. Constraints, necessary to make synchronizations possible, are reflected in the time constraints of individual components of the TT-BIP* model. The transformation from TT-BIP* to TCA —described in Section 5.2—ensures that these synchronization constraints are respected in the obtained automaton. Asynchronous (sending and receiving) actions between interacting TCA automata are ensured to happen at instants over a finer-grained clock as described in third paragraph of Section 5.3. With respect to the clock over which the original synchronization date is defined, these actions are happening at the same instant.

Hence, the correctness of the TCA composition (after step 2 transformation) follows from the correctness of the transformation of individual components of the TT-BIP* model.

5.7 Conclusion

In the thesis, we show that it is possible to propose an automatic and cost effective method for developing TT implementations by combining advantages of component-based rigorous design and time-triggered RTOS-based implementations. For this purpose, the applied method is based on the use of:

1. A high-level component-based modelling platform; timed BIP. This platform is based on well-defined operational semantics and is prone for expressing structured coordination between components. Behavior of each of the atomic components of a BIP model is described by using timed automata. Composite components are descriped as the composition of atomic components by using connectors and priorities. Verification and analysis of component-based BIP models are possible by using tools such as RTD-Finder [8] for compositional verification.

2. A safety-oriented Real-Time Operating Systems (RTOS); PharOS [9] implementing the TT approach. This framework provides a language to describe a TT application consisting of communicating TT tasks (called agents). It provides low-level primitives allowing to specify timing constraints of different computations and communication actions of TT tasks. PharOS ensures, by principle, some important safety properties as the coherence of the data and determinism of real-time behavior [36].

3. Semantics-preserving transformation process. It allows to generate automatically correct-by-construction PharOS implementation from a BIP model. Thus, all properties that are satisfied by the original model, are satisfied by-construcion by the obtained implementation. A posteriori verification of these properties is thus unnecessary. And the determinism of the application is guaranteed by the PharOS platform. This process is defined in two steps:

    - Step 1: A model-to-model transformation. It transforms an original BIP model into a restricted one (TT-BIP model) with respect to a user-defined task mapping. We assume that the source model of the transformation consists only of flat connectors and atomic components. This assumption can not be considered as a restriction, since an arbitrary BIP model with hierarchical connectors and composite components can be transformed into a flat model where all connectors are flat and components are atomic as shown in [47]. Although BIP provides a rich set of interactions, we only considered rendezvous interactions, as it is possible to transform trigger interactions into rendezvous. The aim of the step1 transformation is to obtain a model wich is closer to any TT implementation. That is, to obtain a model where all inter-task interactions are executed by dedicated components and all interactions between these communication components and task components are send/receive interactions. These latter provide, on top of the synchronization, a unidirectional data transfer. Another essential criterion for building the transformation rules is the respect of the equivalence to the original model where interactions' conflicts are resolved by the BIP engine. In order to satisfy this criterion, the obtained model contains a component dedicated to conflict resolution and implementing the fully centralized committee coordination algorithm presented in [10].

- Step 2: A model-to-code transformation. It generates automatically TT implementation from the intermediate model specified in step 1. The generated code is a ΨC code (the programming language of PharOS applications). The input model of this transformation is first adapted into a model where all task components are flattened, i.e. all atomic components of the same task are composed. the adapted model is called TT-BIP* model.

  In order to be able to provide formal correctness proofs of the transformation from TT-BIP* to ΨC, we provided a formal model of the target implementation and defined its operational semantics. This model is called the TCA model, which is in the same abstraction level as the ΨC language. In this model, a task is an automaton, where nodes present states and transitions allow to model actions. These latter are labelled by triplet-labels specifying release, deadline and/or synchronization dates. The transformation rules aim at transforming each transition of the original component automaton, into a set of successive transitions in TCA model. Time progress conditions and timing constraints are mapped using deadlines and/or release dates in TCA model, while communicating transition (i.e. transitions labelled by send or receive ports), are transformed into a set of transitions, among labels of which we find a synchronization constraint.

  Since the semantics of the proposed TCA model are defined as LTSs, the correctness proof of the transformation is based on the notion of the bi-simulation between single LTSs of TT-BIP* components and their corresponding TCA tasks. The equivalence between the obtained application and the initial TT-BIP* model follows from the equivalence between single components and tasks, since communication (i.e. data transfer) in both models is guaranteed by construction to happen in the same instant over the original clock.

6

Tools Implementation and Experimental Results

This chapter aims at presenting the implemented tools and experimental results obtained from case study examples.

We discuss the followed method for implementing transformations allowing to derive TT implementation from high-level BIP model and a user-defined task mapping. This implementation was performed using the BIP tool-chain. Thus, this chapter starts first by presenting in Section 6.1 the existing BIP tools. Section 6.2 focuses on the tools implementing the methods presented in the previous chapters. And Section 6.3 describes the case study examples and some experimental results.

Chapter outline

6.1 The BIP Tool-chain

In this section, we present the BIP tool-chain available with the BIP framework.

The BIP tool-chain is conceived to use BIP as a common semantic model along the design flow. It consists of a set of tools for modelling, executing, verifying and transforming BIP models.

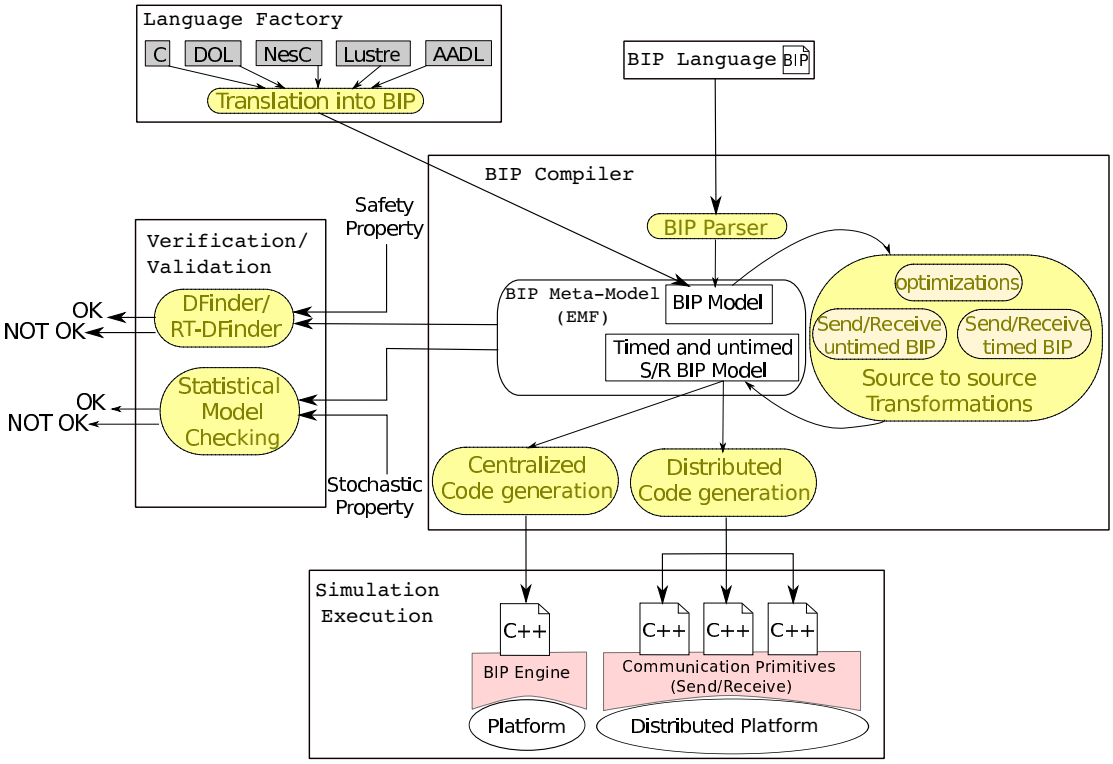


Figure 6.1: Overview of the existing BIP tool-chain

Figure 6.1 shows an overview of the BIP tool-chain. This latter includes five main tools; the Real-time BIP language, the Language factory, the compiler, verification and execution/simulation. We now detail each of these tools.

6.1.1 Real-Time BIP Language

The Real-Time BIP language offers primitives and different syntactic constructs for modelling and composing complex behavior from atomic components by using interactions and priorities. It thus allows to represent component-based models presented in Section 1.3. Note first that for practical reasons, expressions, types of data variables, update and data transfer functions are written in C language. The basic constructs of the BIP language are the following:

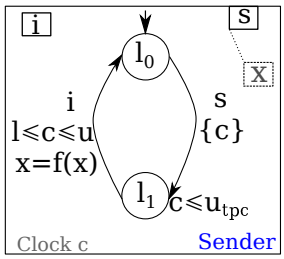
- Atomic components: consisting in communicating timed automata. Transitions are labelled with sets of ports, guards, timing constraints and update functions. A port is either exported or internal.
- Connectors: coordinating between components ports, and having an associated guard and data transfer action.
- Priority: imposing a restriction on the possible interactions
- Composite components: obtained from sub-components by specifying connectors and priorities.
- Model: specifying the entire system and encapsulating the definition of components. It defines, thus, the top level instance of the system.

A model code written in the Real-Time BIP language starts by defining types for components, ports and connectors. These types are instantiated later in order to describe the model architecture.

To introduce briefly how to define and instantiate BIP types, we rely on the model example displayed in Figure 1.4.

Atomic Component Type

Components $Sender_1$ and $Sender_2$ are instances of the same atomic component type *Sender*. This atomic component type is parametrized by lower and upper bounds of timing constraint labelled by port i and by the time progress condition of the location l_0. Similarly, atomic components $Receiver_1$ and $Receiver_2$ are instances of the same atomic component type *Receiver*. Figure 6.2 displays the *Sender* atomic component type and its real-time BIP code. Based on that example, we detail main constructs allowing to define an atomic component type in real-time BIP language. The description of Figure 6.2b starts with the declaration of two types of ports; *IntPort* and *Internal* types. the *IntPort* type defines ports to which we associate an integer variable a. The type *Internal* is an event port type and it is not associated with any variable. The Sender atomic type description starts with the declaration of variables, ports, clocks and locations. Declared ports and their potential associated variables should have types that match those in the port type definition. Instantiated ports can be either exported (e.g. the port s) or internal (e.g. the port i). To each declared clock, we can define a unit (e.g. 1 second, 1 millisecond etc.). A location may define a time progress condition, by declaring the expression after the keyword *while* (e.g. the place l_1 of figcode). The construct *initial to* is used to define the initial transition and potential initialization update functions. Each transition of the described behavior is declared by (1) a port (after the construct *on*), (2) an initial (after the construct *from*) and a final (after the construct *to*) locations, (3) a Boolean guard (after the construct *provided*), (4) the set of clocks to be reset (after the construct *reset*), (5) a timing constraint (after the construct *when*) and (6) an update function (after the construct *do*). Variables of bounds of timing



(a) The sender atomic component type

```
port type IntPort (int a)
port type Internal()
atomic type Sender(int l,int u,int utpc)
   data int x
   export port IntPort s(x)
   port Internal i()
   clock c unit 1 second
   place l0
   place l1 while (c≤utpc)
   initial to l0 do {x = 0;}
   on s from l0 to l1
      provided True
      reset {c}
      do {}
   on i from l1 to l0
      provided True
      when (c≥l && c≤u)
      do{x=f(x);}
end
```

(b) Real-time BIP code

Figure 6.2: BIP code of the sender atomic component type

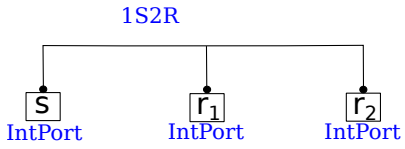
constraints and time progress conditions are defined in the component type parameters. Guards, expressions of timing constraints, time progress conditions and update functions are written using a subset of the C syntax.

Connector Type

Based on the example of Figure 1.4, we now present the connector types description. Both connectors of that model are instances of the same connector type $1S2R$. This connector type defines only one interaction (since all ports are synchrons) and defines transfer functions allowing to copy the sender variable value into receivers ones.

Figure 6.3a shows this connector type graphically, and Figure 6.3b displays its related real-time BIP code.

A connector type is parametrized with a list of port types that defines its port set. The construct *define* specifies the type of ports, trigger of synchron and indirectly the set of interactions allowed by the connector (cf. Section 1.3.3). In the example of Figure 6.3b, we have only three synchrons. A trigger would be specified by appending a quote to the port name. For each interaction, a guard and an update function can be provided. The update function of a given interaction is defined with the *down* construct. The dotted notation, i.e. *port.var* is used to access the variable *var* associated to the port *port*, as defined in the port type declaration. Here all ports are synchrons, thus



(a) The $1S2R$ connector type

```
connector type 1S2R(IntPort s, IntPort r1, IntPort r2)
   define s r1 r2
   on s r1 r2
   provided True
   down {r1.a=s.a; r2.a=s.a;}
end
```

(b) Real-time BIP code

Figure 6.3: BIP code of the $1S2R$ connector type

only one interaction is defined. Its guard is set to *True*. And its transfer function copies the variable $s.a$ into $r_1.a$ and $r_2.a$.

The connector described in Figure 6.3b does not allow hierarchical composition as presented in Section 1.3.3. Recall that hierarchical composition requires the connector to export a port. This can be done through an *export* construct in the connector definition.

Compound Component Type

A compound component is nothing more than a set of instances of existing components and connectors types joined with priority rules. A compound offers the same interface as an atomic component, hence externally there is no difference between a compound and an atomic component. We display, in Figure 6.4, the compound type that corresponds to Figure 1.4, assuming that the atomic component type *Receiver* has been already defined.

```
compound type Application
   component Sender sender1
   component Sender sender2
   component Receiver receiver1
   component Receiver receiver2

   connector 1S2R alpha(sender1.s, receiver1.r, receiver2.r)
   connector 1S2R alpha'(sender2.s, receiver1.r, receiver2.r)

   priority π alpha < alpha'
end
```

Figure 6.4: BIP code of the compound component type of the model of Figure 1.4

A compound type starts by creating instances of each atomic component type. It, then, relates instantiated components by using new instances of connectors types and defines priority between connectors. When creating connectors, the port set of each connector is specified using the dotted notation *comp.port*. This notation denotes the port *port* of the instantiated component *comp*. Furthermore, each priority rule must be given a different name. In order to be able to execute a BIP model, one must add a top level instance of the main compound type.

The compound type of the above example creates first two instances of each atomic component type, i.e. instances $sender1$ and $sender2$ of type $Sender$ and instances $receiver1$ and $receiver2$ of $Receiver$ type. Then, it defines two connectors α and α' by instantiating the connector type $1S2R$. The defined connectors have the respective port set $(sender1.s, receiver1.r, receiver2.r)$ and $(sender2.s, receiver1.r, receiver2.r)$. Since they are sharing ports $receiver1.r$ and $receiver2.r$, a priority rule π is defined to state that the interaction of connector *alpha* is less prior than the one of *alpha'*.

6.1.2 Language Factory

Language factory tools translate various existing languages into BIP models. The input language can model the application software, the hardware architecture, or both of them. Among existing tools in the language factory, we can cite the tool implementing transformations from synchronous languages, i.e. transformations from Lustre [28] and Simulink [81]. These transformations target synchronous BIP [80], that is an extension of BIP dealing efficiently with synchronous models.

Other existing tools focus on languages mixing both the application software and the hardware architecture. These models can be transformed either into two separate models: one dedicated to the software and the other to the architecture or into a single model including both of them, called *system model*. Regarding transformations to hardware model, they often rely on a library of hardware components (e.g. such as memories, buses, processors,etc.) that are modeled in BIP. BIP models can be generated from the Architecture Analysis and Design Language (AADL) [34], from nesC/TinyOS [15] and from the Distributed Operation Layer (DOL) [26].

6.1.3 Verification

On top of the language and the Factory tools, the BIP tool-chain provides tools for verification and validation of BIP models. These tools are very interesting to our approach. They can be used to verify properties on high-level BIP models. And when applying our transformational approach to these models, the already verified properties do not need to be reverified on the obtained implementation due to the correctness by construction of our approach. Therefore, no a posteriori verification is needed.

D-Finder

D-Finder [17, 18] is a verification tool targeting safety properties, e.g. deadlock freedom or mutual exclusion of untimed BIP models. Untimed BIP models are models that have no timing features (clocks, timing constraints, time progress conditions). D-Finder relies on invariants used to approximate the set of reachable states of the target system, hence the method is sound but not complete: it may not be able to prove a property even if it is satisfied by the system. Invariants are computed following the architecture of the system, that is, it generates invariants for components and interactions. The approach is compositional and can be applied incrementally, allowing to better scale to large systems than traditional verification techniques.

RTD-Finder

RTD-Finder [8] extends the D-Finder tool to allow verification of BIP models with timed features. RTD-Finder is based on the approach of D-Finder with the use of auxiliary clocks that help to capture the constraints induced by the time synchronizations between components.

Statistical Model-Checker

In addition to the verification tools, the BIP tool-chain includes the statistical model-checker SMC-BIP [73]. This latter checks stochastic properties described with Probabilistic Bounded Linear Temporal Logic (PBLTL) formulas. These properties refer to the traces of the model. The model has to be expressed using Stochastic BIP (SBIP). Given an SBIP model, a PBLTL formula and confidence parameters, SMC-BIP tool computes execution sequences until the formula can be proven with the target degree of confidence. This tool is well-suited for evaluating quantitative properties including system performance related metrics.

6.1.4 BIP Compiler

The BIP compiler is developed with eclipse, and it uses some eclipse technologies (in particular, EMF). The compiler relies on a modular approach and is composed of three main parts; the front-ends, the middle-ends and the back-ends. These parts can be combined to form a chain, corresponding to a path in the design flow.

The front-end defines the BIP meta-model, the grammar and the rules to build a BIP-EMF model from a BIP source (i.e. the parser). It allows as well to interact with the user and instantiate all parts of the compiler and bind them together to form a coherent compiler. The middle-end is designed to ensure maximal reuse. It contains the needed mechanics allowing to build filters applying BIP-EMF to BIP-EMF transformations. The back-end contains transformations from BIP-EMF to some source codes in a given language. Code generation from BIP-EMF is based on acceleo templates. In the back-end part of the BIP tool-chain compiler, two code generators are provided for generating respectively BIP and C++ code.

In the remainder of this section, we focus of existing transformations in the middle-end part of the compiler. These transformations can be partitioned into optimization transformations and transformations into distributed systems. Optimization transformations are developed for untimed BIP models while transformations for distributed systems are developed for both timed and untimed BIP models.

Source to source optimizations

These transformations are presented in [27, 47]. Their related tools were developed for improving the efficiency of the generated code. They consist mainly of two types of tools; the flattening and merging tools. Flattening tools allow to (1) flatten a component by replacing the hierarchy of components by a set of hierarchically structured connectors which relates atomic components and (2) replace a set of hierarchical connectors by an equivalent set of flat connectors (cf. Remark 1.4). Flattening a connector is performed by composing data transfer functions (e.g. the flat and the hierarchical connectors of Figure 1.10 are equivalent).

Merging tool allows to transform a set of interaction untimed BIP components into a single component with the same behaviour and interface as the composition of the original components.

As explained in Section 4.3, in this thesis all input BIP models are considered to be flat models. The flattening tools are therefore strongly required in the tool-chain since it allows to execute a pre-transformation to any model and obtain flat connectors.

Although needed, the merging tool is not directly usable in our work since it applies only to untimed BIP models. As explained in Section 5.1, the merge transformation is easily adapted to Timed BIP models.

Source to source transformation for untimed distributed model

Aiming at deriving distributed implementations from high-level untimed BIP model, several tools have been integrated into the BIP tool-chain [47, 21, 77]. A global overview of the different options to generate an untimed distributed (i.e. a 3-layer Send/Receive model) model from an untimed BIP model —with or without priorities—is shown in Figure 6.5. Recall that untimed BIP model and untimed Send/Receive models do not contain timing features (clocks, timing constraints and time progress conditions). These tools are parametrized by an interactions partition and a conflict resolution protocol and they consist of the followings:

- UBip2SrBip: This tool generates a 3-layer Send/Receive untimed BIP model from a high-level untimed BIP model. Priority glue is not supported by this tool.
- UBip2Bic: In order to be able to consider the priority glue, this tool was introduced to transform an untimed BIP model into a untimed BIC model. Untimed BIC is an untimed BIP model where priorities are rewritten as Condition predicates [22]. That is each priority is transformed into a predicate on interactions. This predicate

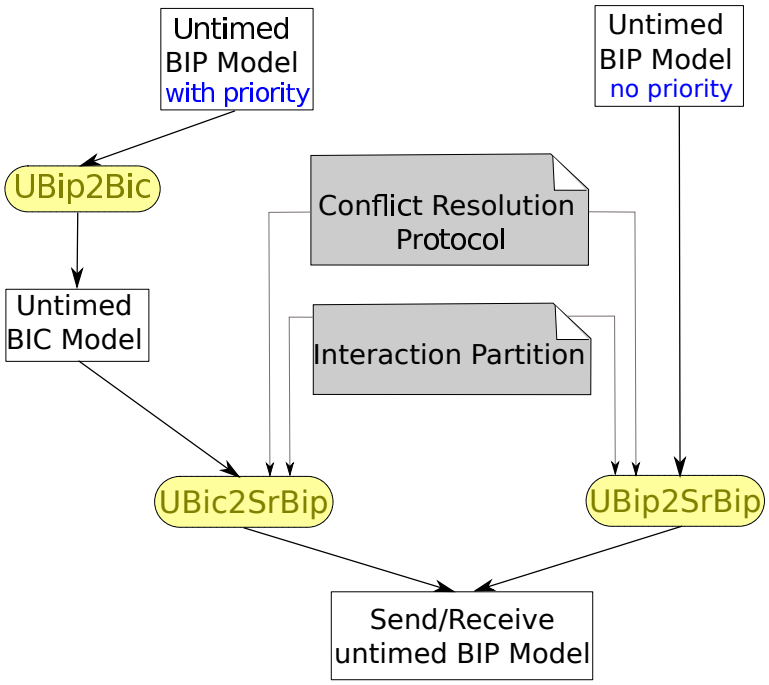


Figure 6.5: Transformation of untimed BIP model into Send/Receive model

helps in characterizing the system state where an interaction could execute, i.e. where no interaction with a higher priority is enabled.

- UBic2SrBip: This tool is an extension of the UBip2srBip tool. It is implemented to support the condition predicate. Its input is a BIC model, i.e. the model resulting from the application of Bip2Bic tool transformation to a BIP model.

Source to source transformation for timed distributed model

These transformation tools, presented in [86], extend the previously mentioned tools in order adapt them to the BIP models and take into account all time features. Figure 6.6 shows an overview of tools targeting timed Send/Receive models. This tool does not support priority glue, and it is parametrized by an interactions partition and a conflict resolution protocol. Although not explicitly displayed in the figure, two versions of the tool were implemented:

- Bip2SrBip: This tool generates a timed Send/Receive model from a high-level BIP model. The implemented approach assumes that communications between components are instantaneous i.e. no communication delays are considered.
- Optimized Bip2SrBip: This tool aims at extending the Bip2SrBip in order to consider communication delays i.e. cross-layer interactions are not instantaneous.

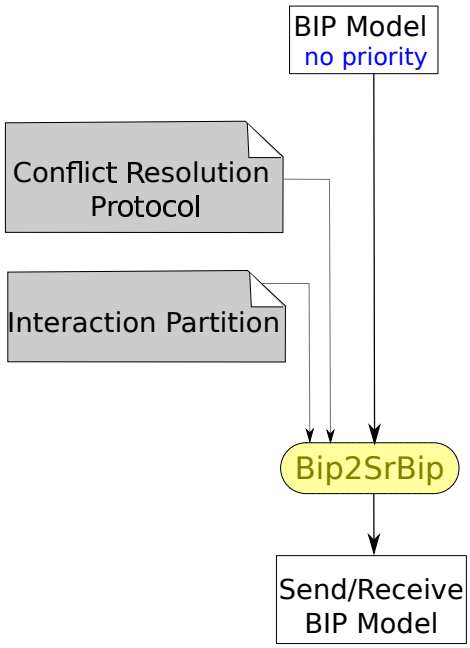


Figure 6.6: Transformation of Timed BIP model into Send/Receive model

6.1.5 Execution/Simulation

The Compiler code generator provides C++ code for either simulation or execution. This code corresponds to both atomic components and glue. One possible scenario to execute the generated code is to use the centralized BIP execution Engine, which directly implements the BIP operational semantics. It plays the role of the coordinator in selecting and executing interactions between different components while respecting the priority rules. Executing an untimed BIP model with the use of a centralized Engine can be performed in two different modes: the single-thread and the multi-thread modes. While for BIP models with timing features, only the single-thread mode is provided by the BIP tool-chain.

For single-thread mode, the Engine and all atomic components execute in a single thread. This execution mode ensures sequential execution of BIP models. During one execution iteration of the Engine, the enabled ports and interactions are selected from the complete list of interactions, based on the current state of the atomic components. Then, priority rules are applied to eliminate interactions with low priority among the enabled ones. An execution iteration starts and finished by the global state of the system. It consists of the following steps:

1. The engine computes Enabled ports after receiving current states of different atomic components (i.e. their current locations and valuations of clocks and variables),

2. The Engine enumerates on the list of interactions in the model and selects the enabled ones based on the current states of the atomic components,

3. The Engine eliminates interactions having low priority,

4. Among the filtered enabled interactions, the Engine selects randomly one interaction, executes its data transfer function and notifies the involved atomic component the transition to execute.

In multi-thread mode, we assign a different thread to each atomic component. The Engine is executed in another thread. Contrarily to the single-thread mode, the global state is unknown to the Engine as an atomic component performing an internal computation has an undefined state. Therefore the Engine executes according to a partial state semantics [13], that takes into account the fact that the state of some components may be unknown. An execution iteration of the multi-thread Engine is very similar to the execution of the single-thread one. Nevertheless, it starts by a partial state. Checking enabledness of an interaction is, thus, not enough to ensure that it can execute, since a higher priority interaction may be enabled. To avoid that, the multi-thread Engine relies on an oracle that must be *True* for the interaction to execute. This means that atomic components can not be ready to execute a higher priority interaction. An execution iteration of the multi-thread Engine consists of the following steps:

1. Components that are ready to interact inform the Engine about their enabled ports,

2. Based on the received information, the Engine filters interactions having an oracle evaluated to *True*,

3. Among the filtered interactions, the Engine randomly selects an interaction to execute, executes its data transfer function and notifies the involved atomic component the transition to execute.

For distributed implementations of untimed and timed BIP models, the BIP toolchain provides code generators in order to generate C++ code for each Send/Receive component of the corresponding 3-layer Send/Receive models. Send/Receive interactions between components of distinct layers are replaced by using the message-passing primitives available on the target platform.

6.2 Tools Developed in This Thesis

In this section, we present how the methods presented in this thesis have been implemented through a set of tools. In Chapter 4 and Chapter 5, we presented a two-step method to derive a TT implementation from a high-level BIP model. We have developed tools for generating such implementation from a given high-level BIP model. Figure 6.7,

shows an overview of the developed tools as well as the input and outputs of each one of them. Different developed tools are integrated within the existing BIP tool-chain.

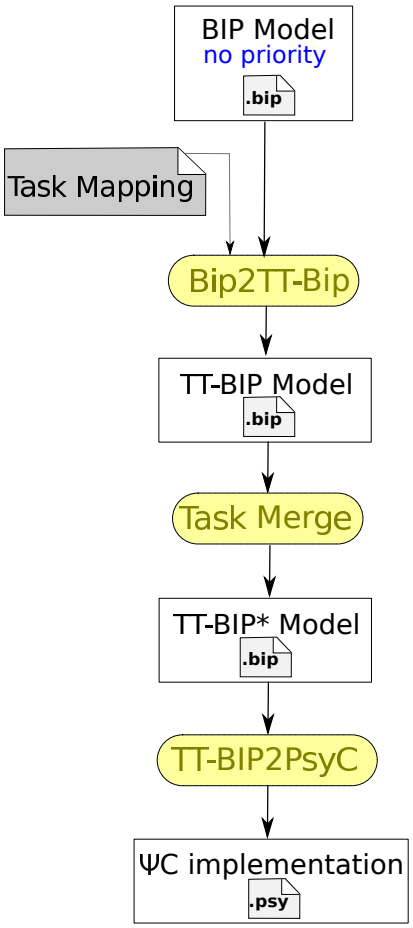


Figure 6.7: Overview of developed tools

Figure 6.8 shows the BIP tool-chain including the new-developed tools.

6.2.1 BIP2TT-BIP Tool

Parametrized by a user-defined task mapping, this tool implements the transformation described in Chapter 4. It allows to transform parsed BIP models into TT-BIP models. The BIP2TT-BIP tool is written in *Java* and is currently integrated as a filter in the middle-end part of the BIP tool-chain compiler. Algorithm 1 displays the pseudo code of the part of the developed tool allowing to transform components of the original BIP model.

Algorithm 1 Transformation of a components of BIP model

Input: Original component B
Output: Obtained component B^{TT}

```
    B^TT = newComponent()
    if B ≠ ATC-component then
 3:   B^TT = B
    else
      // Ports
 6:   portsOf(B^TT).add( portsOf(B) )
      for p ∈ portsOf(B) and p ∈ A_E do
        portsOf(B^TT).add( offer-port )
 9:   end for
      // Locations
      locationsOf(B^TT).add( locationsOf(B) )
12:   for t ∈ transitionsOf(B) do
        if portOf(t) ∈ A_E then
          locationsOf(B^TT).add( offer-location )
15:     end if
      end for
      // Transitions
18:   for l ∈ locationsOf(B^TT) do
        if l = offer-location then
          transitionsOf(B^TT).add( offer-transition )
21:     else if P_l ⊂ A_I then
          transitionsOf(B^TT).add( internal-transition )
        else
24:       transitionsOf(B^TT).add( notification-transition )
        end if
      end for
27: end if
```

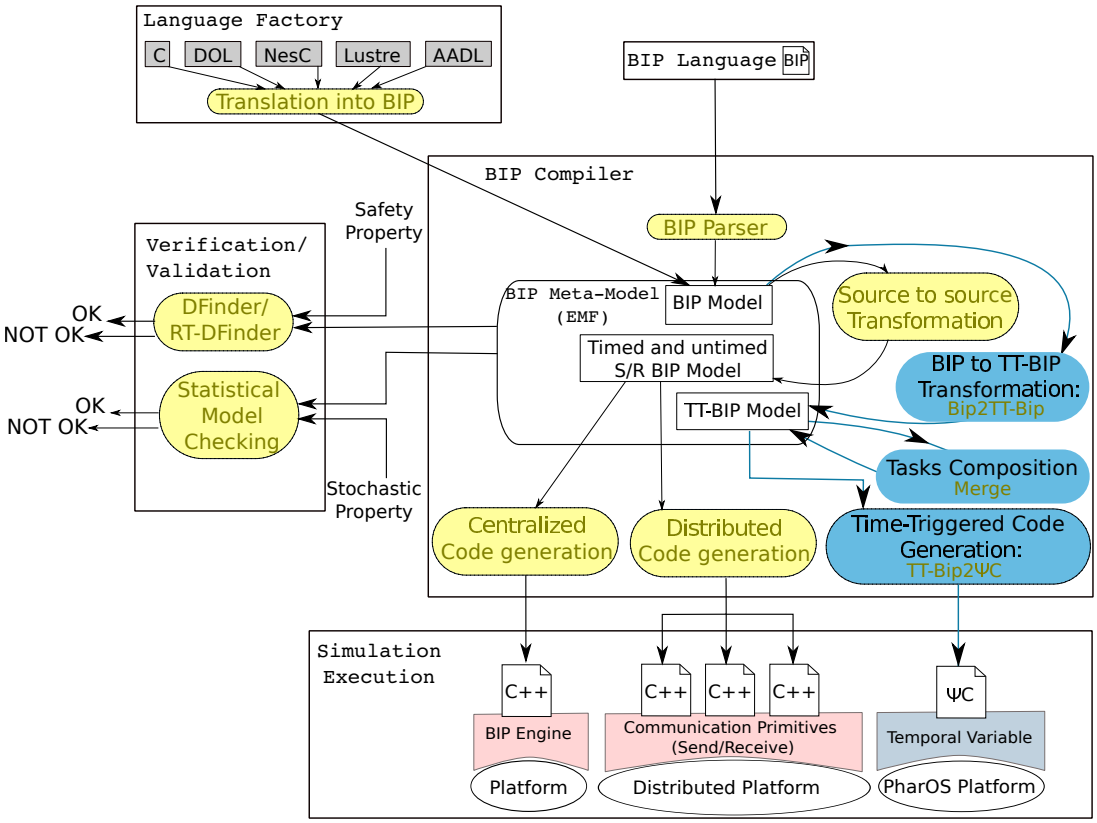


Figure 6.8: Tools developed within the existing BIP tool-chain

6.2.2 Merge Tool

Since tasks in the obtained TT-BIP model from the previous tool may be a composite component, this tool allows to compose components within a task. It implements the formal transformation presented in Section 5.1 of Chapter 5 and consisting in transforming a set of atomic components and a set of flat connectors into an equivalent atomic component. The obtained model after this transformation is denoted TT-BIP* model. This tool is still under construction.

6.2.3 TT-BIP2ΨC Tool

This tools consists in a code generator which implements the translation of a TT-BIP* model into ΨC code as presented in Chapter 5. This tool is implemented as a back-end part of the BIP compiler using the Model To Text (M2T) tools of eclipse, e.g. Acceleo [71] (cf. Figure 6.9).

An Acceleo project consists of different module files (i.e. .mtl files) which include only one main module. These files implement transformation rules that are described

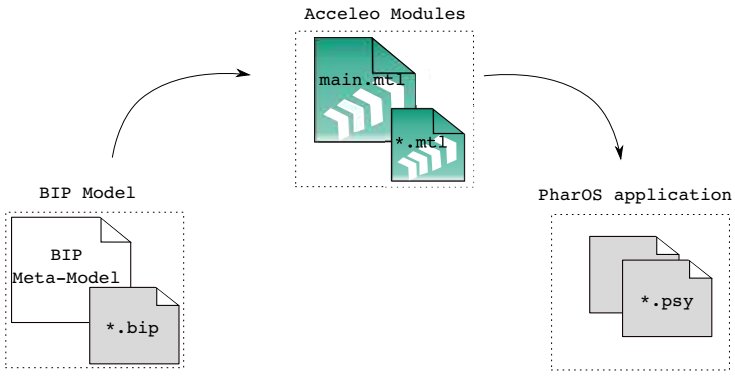


Figure 6.9: Model to Text transformation using Acceleo templates

in Chapter 5. A module file is made up of several templates describing the necessary parameters to generate source code from the meta-model and/or queries used to extract information from manipulated models. A module can depend on other modules for its execution. Templates and queries use the Object Constraint Language (OCL) [87].

As described in Chapter 5, this tool generates an agent for each component and a temporal variable for each send/receive connector of the TT-BIP* model. In Chapter 5, we presented the TCA formalism as the formal model of agents in PharOS application. In this subsection, we detail the transformation of the TT-BIP* model by focusing on the generated code of the behavior (syntactic presentation of the TCA automata), as well as clocks and temporal variables instantiations.

Clocks' generation

By construction, we know that in TT-BIP* models, there is only one global clock c^g. By construction of the transformation described in Chapter 5, the obtained ΨC implementation contains the clock c^g and a finer-grained clock c_{fg}. From the ΨC point of view, the global clock of the application is the clock c_{fg} since it is finer-grained than the clock c^g (the global clock of models TT-BIP and TT-BIP*). Thus, in the generated code, the clock c_{fg} is specified by using primitives (gtc0, gtc1, gtc2, etc.). And the clock c^g is specified based on the clock c_{fg} as displayed in Figure 6.10.

Agents' data and temporal variables generation

For each component in the TT-BIP* model, we instantiate an agent which defines a block of its internal/local variables, a block of its output temporal variables followed by the *display* block and a block of its input variables (cf. Section 2.3.2).

All variables of the original TT-BIP* component that are not associated with send ports, are declared in the *global* block.

For each send port of the TT-BIP* component, the corresponding agent generates a declaration in the *temporal* block. The corresponding temporal variable is a structure encompassing all variables associated with the port in the original TT-BIP* component

```
clock cfg = gtc1(0,1)
clock cg = 3* cfg

Application Example;

  agent Ag1{
   ...
  }
  agent Ag2{
   ...
  }
```

Figure 6.10: Clock instantiation in the generated ΨC code

and variable $flag^{msg}$ that is added by the transformation (cf. Section 5.4 of Chapter 5). Note that if originally the send port is not associated with any variable (which is the case of ports *fail* and *ok* of the CRP component), its corresponding temporal variable corresponds to the Boolean variable $flag^{msg}$. Once *temporal* blocks are defined, we instantiate in each agent the corresponding *display* blocks. For each temporal variable (originally corresponding to a send port), the block *display* defines a declaration of consulting agents which originally correspond to TT-BIP* components that contain a receive port which is related to the send port corresponding to the temporal variable.

For each receive port of the TT-BIP* component, the corresponding agent generates a declaration in the *consult* block. This declaration contains the name of the remote agent corresponding to the TT-BIP component to the send port of which this receive port is related in the TT-BIP* model.

In our work, all depth values are default to zero (resp. to one) in the *temporal* (resp. *consult*) block —since the original TT-BIP* model do not manipulate past values.

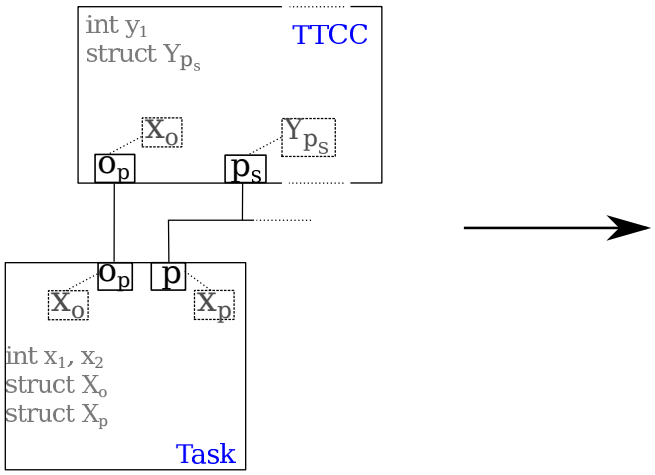
Figure 6.11, displays an example of instantiation of *global*, *temporal*, *display* and *consult* blocks corresponding to a task component communicating with a TTCC component through an offer sending and notification interactions. The structure X_o is the temporal variable of the agent Task, and it is displayed to/consulted by the TTCC. The structure Y_{p_s} is the temporal variable of the agent TTCC, and it is displayed to/consulted by the agent Task.

We display in Algorithm 2, the algorithm that allows to instantiate the blocks of local and temporal variables of an agent starting from the original model. In Algorithm 2, functions *AddVarDeclaration*(), *AddTvDeclaration*(), *AddDisplayDeclaration*() and *AddConsultDeclaration*() are functions that take in parameter corresponding variables or owner/consultant agents of the temporal variable and have as output the declaration code following the syntax of each block. And functions *Start*(*block*) (resp. *End*(*block*)), allow to write the appropriate code to declare the start and the end of each block.

Algorithm 2 Instantiation of global, temporal, display and consult blocks

Input: Original model $TT - BIP* = Components + Connectors$
Output: Obtained ΨC application

```
    for B ∈ Components do
      Start(agent Ag)
 3:   Start(global-block)
      for x ∈ internalVariablesOf(B) do
        AddVarDeclaration(x)
 6:   end for
      End(global-block)
      Start(agent Ag)
 9:   Start(temporal-block)
      for pSend ∈ sendPortsOf(B) do
        AddTvDeclaration(variablesOf(pSend))
12:   end for
      End(temporal-block)
      Start(display-block)
15:   for pSend ∈ sendPortsOf(B) do
        for C ∈ Connectors do
          if portsOf(C) include pSend then
18:         for pReceive ∈ receivePortsOf(C) do
              AddDisplayDeclaration(agentOf(pReceive), variablesOf(pSend)
            end for
21:       end if
        end for
      end for
24:   End(display-block)
      Start(consult-block)
      for pReceive ∈ receivePortsOf(B) do
27:     for C ∈ Connectors do
          if portsOf(C) include pReceive then
            for pSend ∈ sendPartsOf(C) do
30:           AddConsultDeclaration(agentOf(pSend), variablesOf(pSend)
            end for
          end if
33:     end for
      end for
      End(consult-block)
36: end for
```




```
Application Example;

  agent Task{
    global{
      int x1, x2;
    }
    temporal{
      struct 0$Xo;
    }
    display{
      Xo : TTCC;
    }
    consult{
      TTCC : 1$Yps;
    }
    body start{
      ...
    }
   ...
  }

  agent TTCC{
    global{
      int y1;
    }
    temporal{
      struct 0$Yps;
    }
    consult{
      Task : 1$Xo;
    }
    body start{
      ...
    }
   ...
  }
```

Figure 6.11: Example of temporal variables instantiation

Agents' behaviors generation

In the ΨC language, the behavior of an agent can be described using successive body items. We choose in our generation tool, to instantiate one body per original transition. That is each body executes the set of transitions of the TCA automaton corresponding to the image of the original transition of the TT-BIP* component.

Note that in the ΨC code level, no indeterminism is allowed. Thus, at the end of each body we need to specify its corresponding next body through the instruction `next`

`body`. Therefore, each body tests respective guards of next jobs (of the corresponding TCA automaton) and only the body of the job with the guard evaluated to *True* is enabled. When two transitions are enabled, we may in the generated code choose one of them to execute. While transforming the CRP component of Figure 5.15a, we obtain the

```
global {
  int NB1 = 0;
  int NB2 = 0;
  int NB3 = 0;
  int nb1, nb2, nb3;

  bool flagRef_Rsv_alpha_1 = 0;
  bool flagRef_ok_alpha_1 = 0;
  bool flagRef_fail_alpha_1 = 0;
  bool flagRef_Rsv_alpha_2 = 0;
  bool flagRef_ok_alpha_2 = 0;
  bool flagRef_fail_alpha_2 = 0;
 }

 temporal {
  //OkFail structures
  CRP_ok_alpha_1 0$vt_CRP_ok_alpha_1;
  CRP_fail_alpha_1 0$vt_CRP_fail_alpha_1;
  CRP_ok_alpha_2 0$vt_CRP_ok_alpha_2;
  CRP_fail_alpha_2 0$vt_CRP_fail_alpha_2;
}

 display {
  vt_CRP_ok_alpha_1 : TTCC_alpha_1;
  vt_CRP_fail_alpha_1 : TTCC_alpha_1;
  vt_CRP_ok_alpha_2 : TTCC_alpha_2;
  vt_CRP_fail_alpha_2 : TTCC_alpha_2;
 }

 consult{
  TTCC_alpha_1: 1$vt_TTCC_Rsv_alpha_1;
  TTCC_alpha_2: 1$vt_TTCC_Rsv_alpha_2;
 }
```

(a) Obtained global, temporal consult and display blocks after transformation of the CRP

```
#ifndef TYPES_H_
#define TYPES_H_

//Reservation structure
typedef struct {
 int nb1;
 int nb2;
 bool flag;
} TTCC_Rsv_alpha_1;

typedef struct {
 int nb1;
 int nb3;
 bool flag;
} TTCC_Rsv_alpha_2;


//OK structures
typedef struct {
 bool flag;
} CRP_ok_alpha_1;

typedef struct {
 bool flag;
} CRP_ok_alpha_2;


//FAIL structures
typedef struct {
 bool flag;
} CRP_fail_alpha_1;

typedef struct {
 bool flag;
} CRP_fail_alpha_2;

#endif
```

(b) Definitins of structures of temporal variables

Figure 6.12: Generated variables and temporal variables of the CRP component of Figure 5.15a

blocks of variables and temporal variables displayed in Figure 6.12a. In Figure 6.12b, we display the type file that contains the definitions of structures corresponding to different temporal variables. In Figure 6.13, we display the ΨC code corresponding to the behavior of the CRP components of Figure 5.15a. This source code maps the transitions of the TCA automaton of Figure 5.15b. Different comments display the name of the transition mapped to the succeeding lines of code. As you can notice in this example, the non-determinism is resolved by imposing an order for execution of conflicting jobs. That is when the CRP has received already a reservation, originally it can either receive the second reservation or send an OK or a fail notification. This conflict can not be allowed in the ΨC code level. Therefore, we choose to prioritize the ok notification if its guard is *True*, otherwise the fail notification can be sent. In this choice, the CRP can not receive

two successive reservations. This behaviour is included in the original behavior, since in the original BIP model. This resolution of non-determinism does not jeopardise the correctness property of the transformation.

```
body start{
  //Tau_(w_alpha1 w_alpha2,r)^loop
  advance(2) with c_fg;
  while(flagRef_Rsv_alpha_1 == TTCC_alpha_1`1$vt_TTCC_Rsv_alpha_1.flag
  &&  flagRef_Rsv_alpha_1 == TTCC_alpha_1`1$vt_TTCC_Rsv_alpha_1.flag)
  {
   advance(2) with c_fg;
  }
  //Tau_(w_alpha1 w_alpha2,rsv_alpha1)^0
  if (flagRef_Rsv_alpha_1 != TTCC_alpha_1`1$vt_TTCC_Rsv_alpha_1.flag){
   flagRef_Rsv_alpha_1 = TTCC_alpha_1`1$vt_TTCC_Rsv_alpha_1.flag;
   nb1 =  TTCC_alpha_1`1$vt_TTCC_Rsv_alpha_1.nb1;
   nb2 =  TTCC_alpha_1`1$vt_TTCC_Rsv_alpha_1.nb2;
   next Ok_Fail_alpha_1;
  }
 //Tau_(w_alpha1 w_alpha2,rsv_alpha2)^0
  if (flagRef_Rsv_alpha_2 != TTCC_alpha_2`1$vt_TTCC_Rsv_alpha_2.flag){
   flagRef_Rsv_alpha_2 = TTCC_alpha_2`1$vt_TTCC_Rsv_alpha_2.flag;
   nb1 =  TTCC_alpha_2`1$vt_TTCC_Rsv_alpha_2.nb1;
   nb3 =  TTCC_alpha_2`1$vt_TTCC_Rsv_alpha_2.nb3;
   next Ok_Fail_alpha_2;
  }
}

body Ok_Fail_alpha_1{
 //Tau_(r_alpha1 w_alpha2,ok_alpha1)^0
  if (nb1 > NB1 && nb2> NB2){
   vt_CRP_ok_alpha_1.flag = !vt_CRP_ok_alpha_1.flag;
   NB1 = nb1;
   NB2 = nb2;
   advance(1) with c_fg;
  }
  else{
 //Tau_(r_alpha1 w_alpha2,fail_alpha1)^0
   vt_CRP_fail_alpha_1.flag = !vt_CRP_fail_alpha_1.flag;
   advance(1) with c_fg;
  }
  next start;
}

body Ok_Fail_alpha_2{
 //Tau_(w_alpha1 r_alpha2,ok_alpha2)^0
  if (nb1 > NB1 && nb3> NB3){
   vt_CRP_ok_alpha_2.flag = !vt_CRP_ok_alpha_2.flag;
   NB1 = nb1;
   NB3 = nb3;
   advance(1) with c_fg;
  }
 //Tau_(w_alpha1 r_alpha2,fail_alpha2)^0
  else{
   vt_CRP_fail_alpha_2.flag = !vt_CRP_fail_alpha_2.flag;
   advance(1) with c_fg;
  }
  next start;
}
```

Figure 6.13: Generated code of the behavior of the CRP component of Figure 5.15a

6.3 Case Study Examples and Experimetal Results

In this section, we describe industrial case study examples and present experimental results obtained after testing of different developed transformation tools.

6.3.1 Flight Simulator

The first case study is the Flight Simulator (FS) application [16] dedicated to the navigation of DIY radio controlled planes. The original application is written in Modelica [40]. This application provides a simulation of the physics of a plane and an automatic pilot who tries to reach given way-points on a map. The simulation of the Modelica model gives a display of the road followed by the plane (specifically the trajectories of left and right wingtips).

The Modelica model consists of a set of six communicating sub-models (cf. Figure 6.15): autopilot, fly-by-wire, route planner, servo (i.e. the actuator), simulator and sensor. The autopilot models the pilot commands in function of the flight state. It has three main functionalities: flight state reception from sensor component, execution of the route planner and execution of fly-by-wire. The route sub-model receives the flight state from the autopilot and sends information to fly-by-wire after computing distance to current waypoint and changing route towards next waypoint if necessary. It operates in low frequency: every 15 seconds. The fly-by-wire sub-model allows course correction by setting roll attitude and ailerons and elevator. These modifications form the command to be sent to the servo sub-model. The fly-by-wire sub-model operates in high frequency: every 5 seconds. The servo refers to the actuation on plane's flight control surfaces. Servo component receives command from the fly-by-wire sub-model and transfers it to simulator component. Some filtering (e.g. low-pass, delay) could be added to mimic realistic actuators. The flight simulator simulates flight dynamics computation of plane and wing tips position based on received commands from the servo (i.e. new values of roll, pitch and throttle). The sensor refers to the autopilot's perception of real world data. Sensor sub-model receives data about flight state from simulator component and resends them to the autopilot. The sensor can add some noise (e.g. delay, etc.) to mimic realistic data acquisitions. But in our example, it stands for copying the state computed by the simulator.

These sub-models are communicating through Modelica connectors. The software architecture of the original Modelica model is shown in Figure 6.14.

We have first modelled the FS application in BIP language. This latter —coupled with different task mapping strategies— is the input of transformation tools displayed in Figure 6.7. We also simulate the initial BIP model, the TT-BIP model (the output of the TT2TT-BIP tool) and the ΨC code (the output of the TT-BIP2ΨC tool) in order to compare their respective performances.

6.3.1.1 BIP modelling

Each sub-model of the Modelica model is modelled as a BIP component, communication between different components is modeled using BIP connectors. Figure 6.15 displays the overall architecture of the BIP model. Automata of different components are displayed in Figure 6.16.

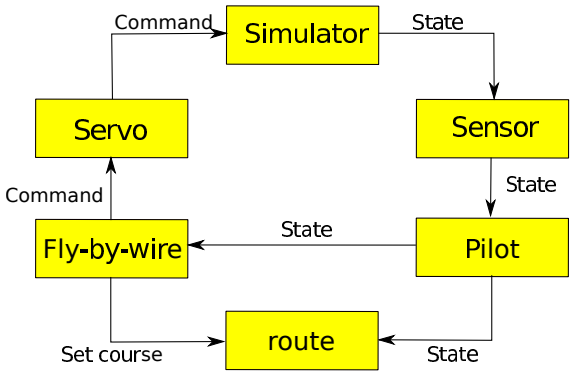


Figure 6.14: Software Architecture of the Modelica Model of the Flightsim Application

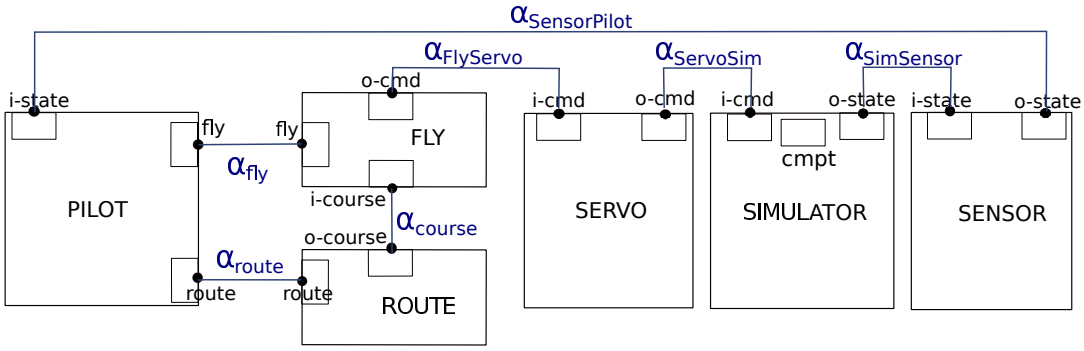


Figure 6.15: Initial Flightsim BIP model

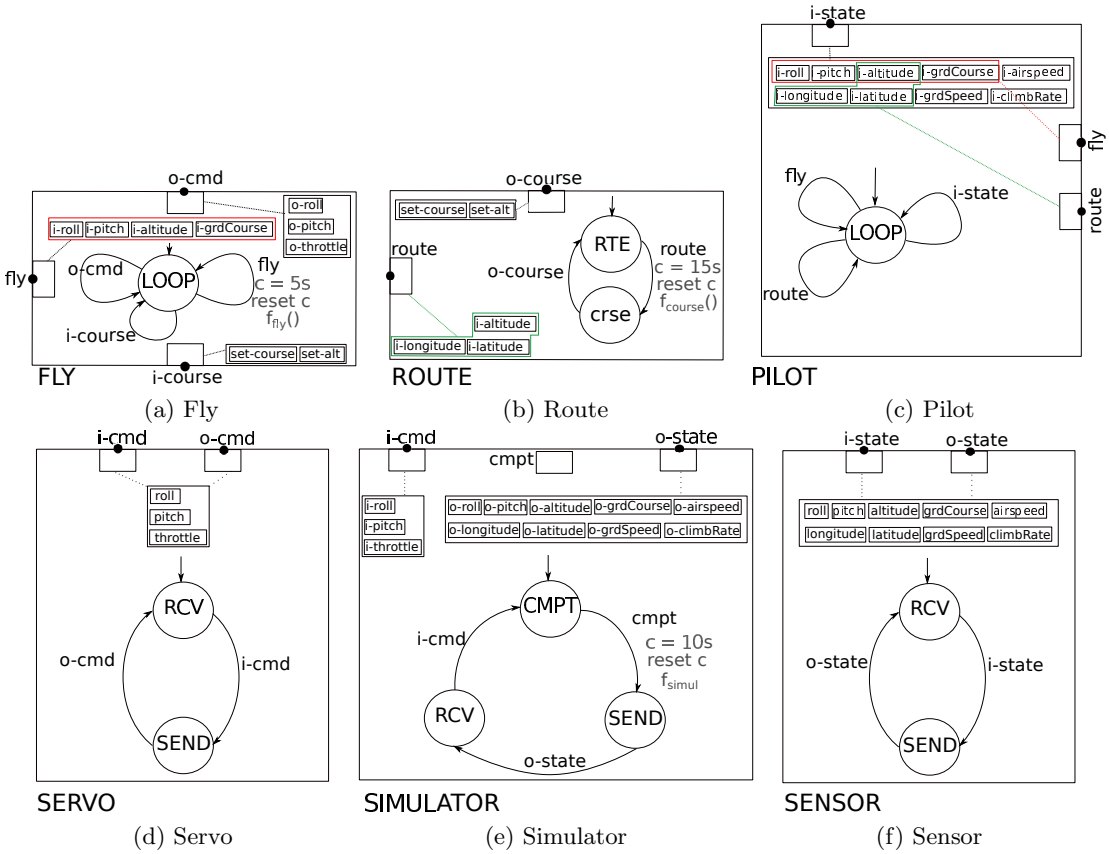
6.3.1.2 BIP to TT-BIP Transformation

We apply the transformation of the BIP2TT-BIP tool in order to derive the TT-BIP model following different task mapping strategies (cf. Table 6.1).

Figure 6.17, shows the obtained model for the task mapping $TM1$. For clarity reason, behaviours of TTCC and CRP components are not displayed. Nonetheless, since all TTCC components are connecting exactly two tasks, their automata are strictly similar to those of Figure 4.11 and Figure 4.12.

For this specific example, the obtained TT-BIP models for mappings $TM2$ and $TM3$ have each as many TTCC components as in the model of Figure 6.17. This unchanged number of TTCC components is due to the fact that interactions α_{fly}, α_{route} and α_{course} are conflicting either directly or indirectly with the intertask interactions $\alpha_{SensorPilot}$ and $\alpha_{FlyServo}$. In the TT-BIP model related to the mapping $TM4$, the interaction $\alpha_{SimSensor}$ is not handled through a dedicated TTCC component. The original connector is kept intact since it executes an internal interaction with respect to task T_3.

Figure 6.18 displays components TT-fly, TT-route, TT-pilot and TT-servo which are common for all task mapping strategies. In Figure 6.19, we display the TT-sim



(a) Fly (b) Route (c) Pilot

(d) Servo (e) Simulator (f) Sensor

Figure 6.16: Components of the Flight Sim BIP Model

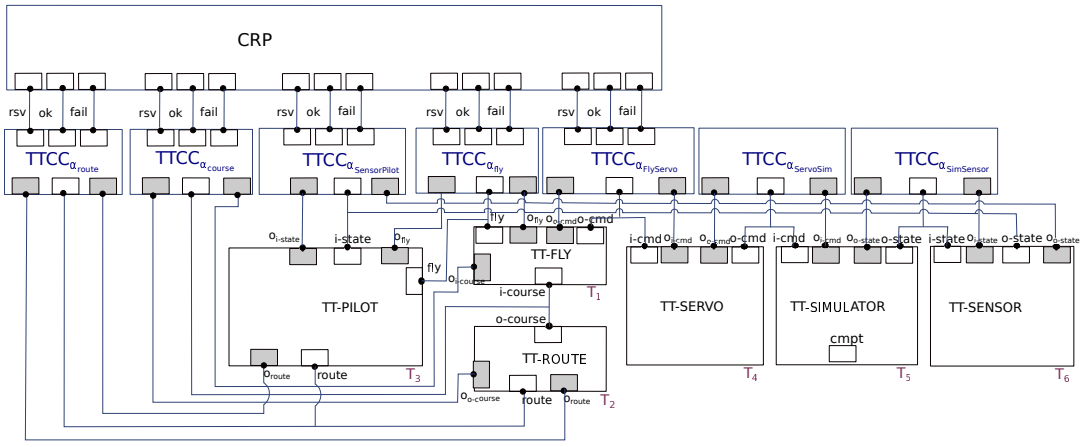


Figure 6.17: FS TT-BIP Model for the Task mapping TM1

Task Mapping Strategy	List of Tasks
TM1	$T_1 = \{FLY\}, T_2 = \{ROUTE\}, T_3 = \{PILOT\},$ $T_4 = \{SERVO\}, T_5 = \{SIMULATOR\}, T_6 = \{SENSOR\}.$
TM2	$T_1 = \{FLY, ROUTE\}, T_2 = \{PILOT\}, T_3 = \{SERVO\},$ $T_4 = \{SIMULATOR\}, T_5 = \{SENSOR\}.$
TM3	$T_1 = \{FLY, ROUTE, PILOT\}, T_2 = \{SERVO\},$ $T_3 = \{SIMULATOR\}, T_4 = \{SENSOR\}.$
TM4	$T_1 = \{FLY, ROUTE, PILOT\}, T_2 = \{SERVO\},$ $T_3 = \{SIMULATOR, SENSOR\}.$

Table 6.1: Different Task Mapping Strategies

component common for mappings $TM1$, $TM2$ and $TM3$ (Figure 6.19a) and the TT-sim component for mapping TM4 (Figure 6.19b). Similarly, different versions of TT-sensor component are shown in Figure 6.20.

6.3.1.3 TT-BIP* to PharOS Implementation

After composing different composite task components of the obtained TT-BIP models, we apply the code generation of the TT-BIP2ΨC tool.

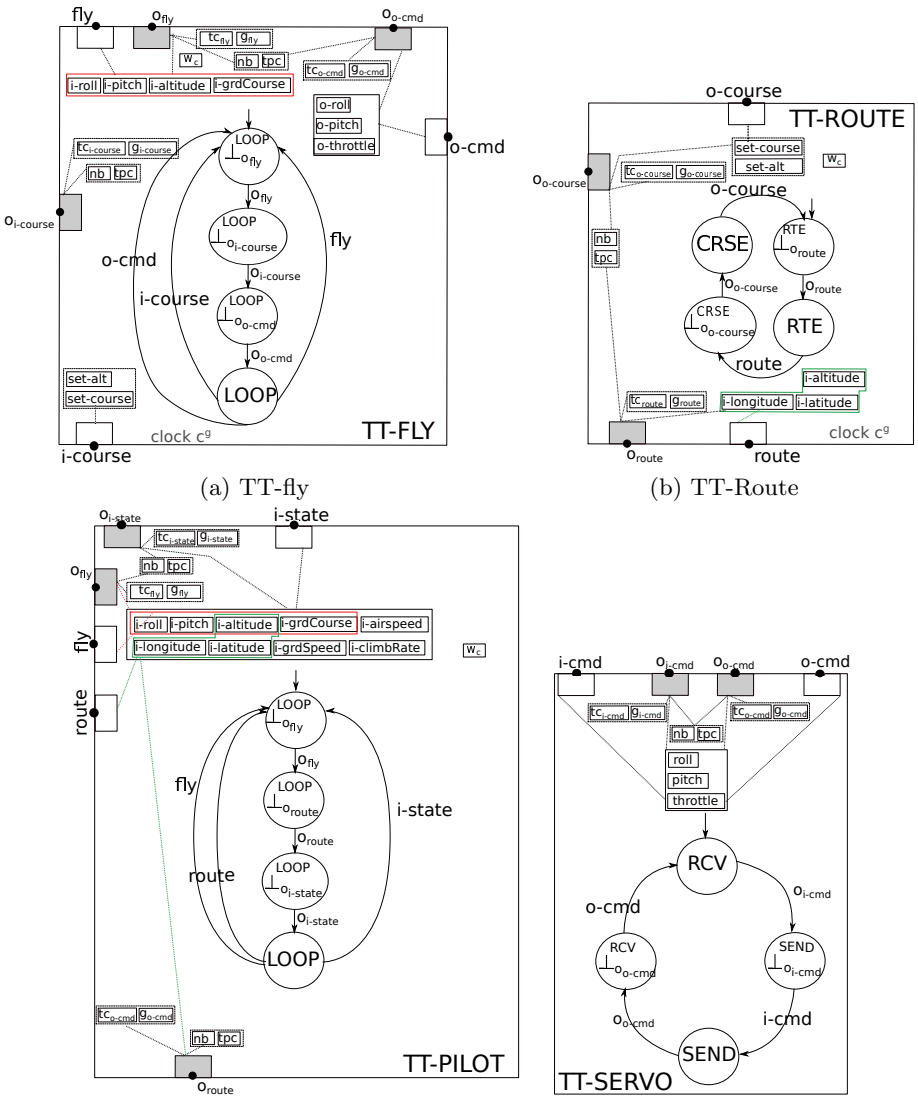
6.3.1.4 Evaluation

Functional Evaluation

In order to be able to compare the functionality of the original BIP model and the obtained TT-BIP* model with the generated ψC code—for all task mapping strategies, we use BIP simulator that generates C++ code from the original and the TT-BIP models. Simulation of both generated C++ codes allowed us to visualize and compare the output signals. A band shows the trajectories of left and right wingtips and illustrates the roll movement that precedes the change in course at each waypoint, while the plane progressively reaches its desired altitude. Figure 6.21 presents the simulation results of the BIP and the derived TT-BIP model, for the waypoints (300,0,300), (300,300,300), (0,300,300) and (0,0,300). Visual inspection reveals that the output of the transformed model is strictly similar to that of the original model. Simulations of the derived $\Psi Ccode$ for each of the task mapping strategies are still under construction.

Comparison between different task mappings

The overhead of communication can be estimated using the number of generated temporal variables. Intuitively, one temporal variable is generated for each send/receive cross-layer interaction in the TT-BIP model. Note that in Table 6.2, the number of generated temporal variable is the same for the first three task mapping strategies. This is explained by the fact that in the original model, the interactions α_{course}, α_{fly}, α_{route} and $\alpha_{FlyServo}$ are conflicting. In all task mapping strategies, the interaction $\alpha_{FlyServo}$ is

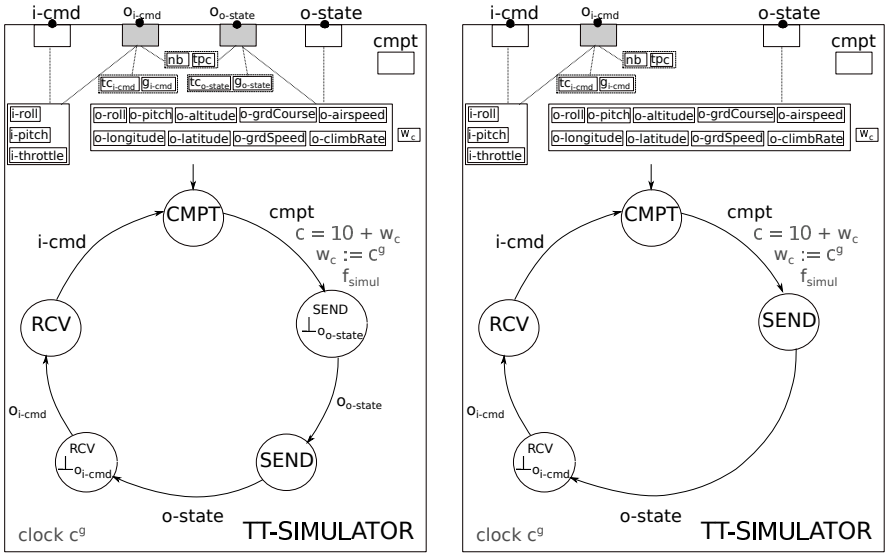


(a) TT-fly (b) TT-Route

(c) TT-Pilot (d) TT-Servo

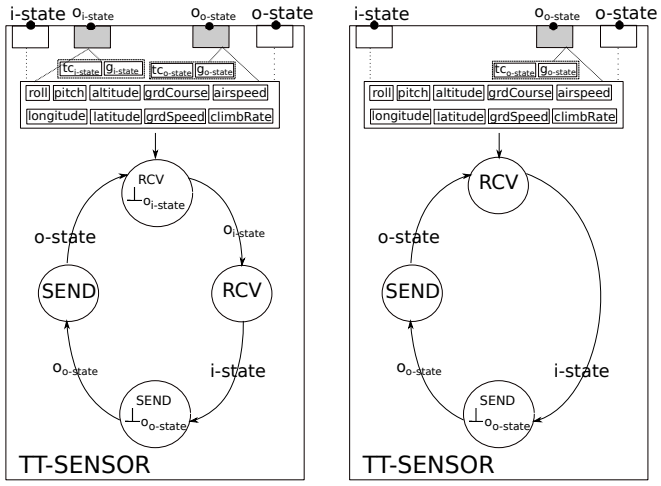
Figure 6.18: Components of the FlightSim TT-BIP Models for all task mapping strategies

an inter-task interaction. Therefore all other interactions are replaced by TTCC components and considered as external interactions. Which explains why the number of generated temporal variables remains intact for these three task mapping strategies. For the task mapping $TM4$, the number of temporal variables is slightly lower since the interaction $\alpha_{SimSensor}$ is an intra-task interaction, and no TTCC component is generated for this interaction in the corresponding TT-BIP component.



(a) Task Mappings TM1-TM3 (b) Task Mapping TM4

Figure 6.19: Components TT-sim



(a) Task Mappings TM1-TM3 (b) Task Mapping TM4

Figure 6.20: Components TT-sensor

6.3.2 The Medium Voltage Protection Relay Application (MVPR)

The second case study is the medium voltage protection relay application of [48].

A protection relay is a device designed to detect and isolate faults in an electrical network. A sensor measures the current that flows on the network and transmits this information to the relay. The relay receives this information, applies signal processing

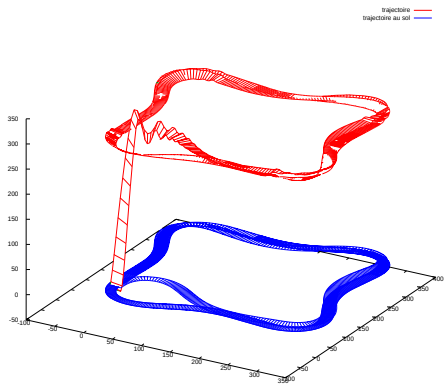


Figure 6.21: Trajectories of left and right wingtips of the BIP and the TT-BIP models

Task Mapping Strategy	Number of Agents	Number of temporal variables
TM1	13	36
TM2	12	36
TM3	11	36
TM4	9	33

Table 6.2: Number of generated agents and temporal variable in each task mapping strategy

algorithms and protection algorithms and takes control decisions. The original version of this case study —presented in [48] —is written manually in ΨC code. The protection relay software consists of three stages: acquisition, measurement and protection stages. Tasks within each stage are periodic tasks. The software architecture of the protection relay application is shown in Figure 6.22 which is taken from [48].

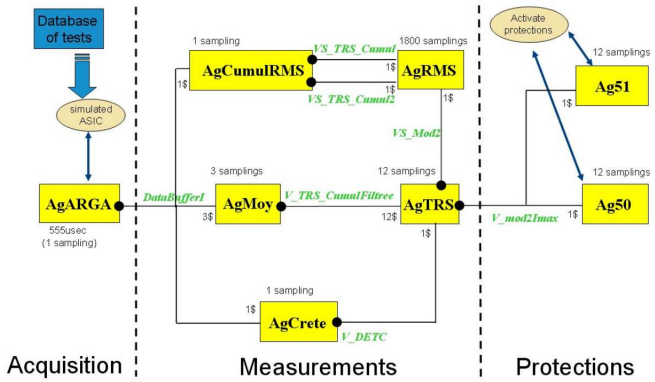


Figure 6.22: Software Architecture of the Protection Relay Application

The acquisition stage. Contains only the task `AgARGA`. This latter collects data and

makes them available to the other components of the system. Data are periodically collected every 555 μs (the sampling rate).

The measurement stage. It computes —using different algorithms—different values that will be used in the protection stage in order to detect potential faults and decide whether safety-function of the protection relay should be activated. In this case, the measurement stage consists of a computation of an average, a computation of the magnitude of the fundamental (50 Hz) and some harmonics (100 Hz, 150 Hz, etc.), a computation of a crest value and a computation of a root mean square. More details about these signal processing algorithms are provided in [78]. The average value is computed by the `AgMoy` task and consists in the computation of the average of the last three values acquired by the `Acquisition` stage. This task produces value every time the `Acquisition` stage acquires three new data items, (i.e. every 1.665 ms). The crest value is computed by the `AgCrest` task and consists in the computation of the crest value of every value acquired by the `Acquisition` stage. This value is computed for every data acquired by the `Acquisition` (i.e. every 555 μs). The computation of the magnitude of the fundamental and some harmonics is made by the `TRS` task. This latter uses the last 12 values computed by the `AgMoy` task and the last value of the `Crest` task. New values are computed every 12 new data items of `AgMoy` task (i.e. every 6.660 ms). The RMS value is computed by tasks `AgCumulRMS` and `AgRMS`.

The protection stage. It detects failure by using different algorithms. In this model, two protection algorithms are considered: an instantaneous over-current protection called Protection 50 (performed by the task `Ag50`) and an inverse time over-current protection called Protection 51 (performed by the task `Ag51`). These two tasks check if the safety function of the protection relay must be activated whenever they receive data from the `TRS` task (i.e. every 6.660 ms).

In order to be able to apply our work to this application, we start by modelling it using the BIP framework. Then we apply the transformation of Chapter 4 (using the BIP2TT-BIP tool) in order to obtain its corresponding TT-BIP model. And finally we apply the transformation described in Chapter 5 (using the TT-BIP2PsyC tool) in order to generate its corresponding ΨC code.

The fact that the original version of the case study is written manually in ΨC code will serve as a point of comparison with the automatically generated code. This comparison concerns traces and some other features like communication and memory overheads.

6.3.2.1 BIP Modelling

Notice that tasks of the *The protection stage* have as input values from the `AgTRS` task and they are not related to the `AgRMS` task. Therefore, we chose not to present tasks `AgComulRMS` and `AgRMS` in the BIP model. A model of the application written in BIP—where the *The measurement stage* is composed only of three components—is shown by Figure 6.23.

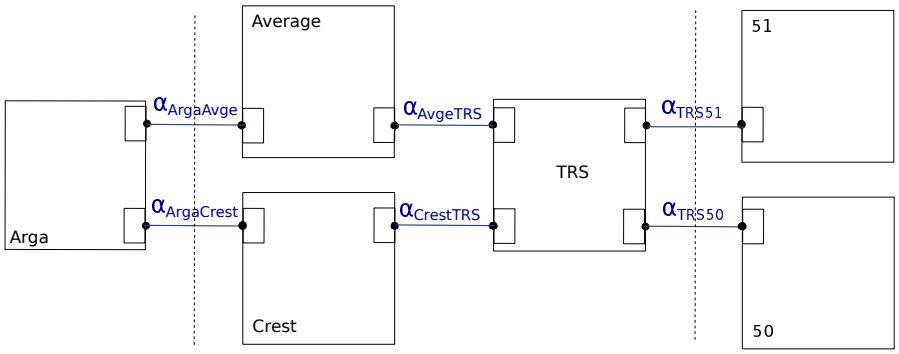


Figure 6.23: BIP Model of the protection relay application

6.3.2.2 BIP to TT-BIP Transformation

We have applied the automatic transformation of Chapter 4 in order to obtain the TT-BIP model from the BIP model of the case study. We chose to gather the two protection components in the same task. The rest of components are considered as independent tasks. The resulting TT-BIP model is shown in Figure 6.24. We have also observed identical values of the output flows generated by simulations —in BIP environment —of both BIP and TT-BIP models.

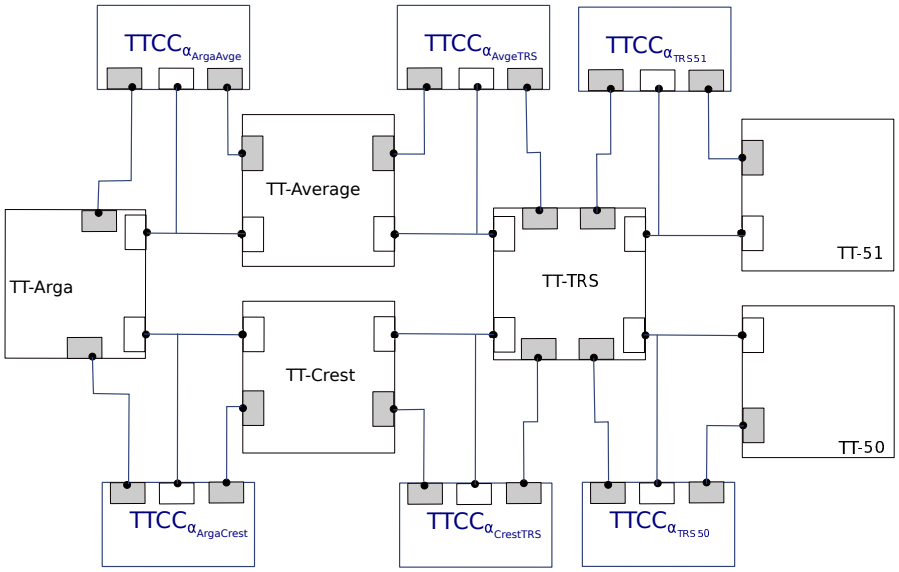


Figure 6.24: TT-BIP Model of the protection relay application

6.3.2.3 TT-BIP* to PharOS Implementation

After composing compsite task components of obtained TT-BIP model, we have applied the implemented transformation described in Chapter 5 to the TT-BIP* model of the case-study described above. For each component in the TT-BIP model, we generate an agent in PharOS. Communication between different components is performed using `advance` statements.

Preservation of the functional behaviour of each generated agent (compared to its corresponding component in the TT-BIP model), has been tested as well. We have also observed identical values of the output flows generated by simulations in both environments.

6.3.3 Evaluation

Functional Evaluation

A comparison of the temporal evolution of computed variables in both versions is also of interest. In Figure 6.25, we display the evolution of the variables *arga* and *crest* in both versions. Values of variable *arga* are transmitted by the sensor to the `acquisition` component, standing for the measures of the input current. *crest* values are computed by the `crest` component. In Figure 6.25, solid lines are reserved for the automatically generated application while dotted lines are reserved for the manually written one.

Visual inspection of different values of both variables in both versions, reveals that the output of the automatically generated model is strictly similar to that of the manual model.

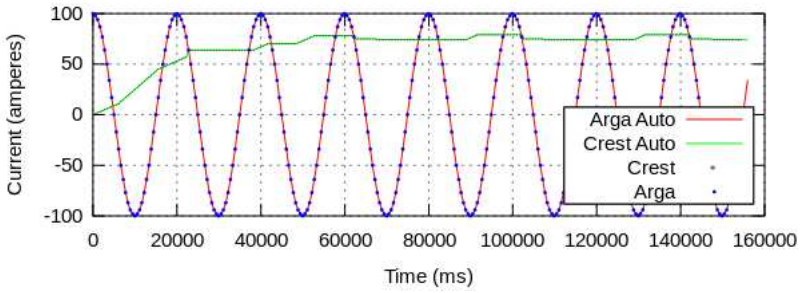


Figure 6.25: Execution trace

Evaluation of the Performances of the generated PharOS application

In this subsection, we compare the automatically generated code with a manually written one [48] for the same case study (cf. Table 6.3).Notice that with the implemented code generation tool we gain in terms of development time, even if in the present state, we need to adapt the generated code manually since some features are still not included in the implemented tool (e.g. optimisations). In the generated code we introduce almost

	Manually written code	Generated code
Development time	2-3 months	1 week (RT-BIP model writing and validation) + 2 days (code adaptation)
Text section size	41.7 kB	71.2 kB
Application text section size (w/o kernel)	13.9 kB	37.1 kB
Data section size	22.1 kB	31.1 kB
Number of *Temporal variables*	7	18

Table 6.3: Comparison between the generated and the manually-written source codes of the case study

two and a half times more *temporal variables* compared to the initial model, this is due to the communication atomicity breaking brought by the transformation from RT-BIP model into a TT-BIP model. These added *temporal variables* lead to a larger memory footprint. When comparing *text* and *data* segments sizes with the manually written version, we find out that segments of the automatically generated code have almost two times bigger size. This ratio is rather reasonable and very encouraging as we are not (yet) interested in optimizing the output model in terms of the number of agents and communications.

The evaluation of the generated code in terms of CPU overhead compared to the manually written code, is subject of ongoing work.

6.4 Discussion and Conclusion

In this chapter, we have presented the existing BIP tool-chain. We also provided an overview of the tool-flow implementing different transformations presented in Chapter 4 and Chapter 5 and allowing progressively to derive a TT implementation from high-level BIP model. The tool-flow is mainly composed of two transformation tools: BIP2TT-BIP tool and merge tool and a generation tool: the TT-BIP2 ΨC tool. The composition tool was not implemented during the thesis, which entailed some manual transformations (components composition) in the testing process of the developed tools.

We illustrate the applicability of the proposed tool-flow on two case study examples; the flight Simulator (FS) application and the medium voltage protection relay application. In both applications, we aim at comparing functionalities of original (BIP), intermediate(TT-BIP) and final (ΨC) models in order to confirm the correctness of the transformation. Simulations of the generated code in the FS application are still under construction. For the first application (i.e. the FS application), we study the impact of the task mapping on the generated code. And for the second application, we study the impact of the transformation on some performance aspects compared to a manually written version. All presented results were measured on simulated models.

Unfortunately, execution of the generated ΨC code on real PharOS machine was not possible due to the non-availability of an adapted platform in CEA. Even if accurate performance measures were not possible, we believe that the provided results are more than interesting. Since they prove that from a high-level model,a code with the same behaviour and satisfying the same properties can be automatically generated.

Conclusion and Perspectives

Achievements

In this thesis, we show that it is possible to propose an automatic and cost effective method for developing TT implementations by combining advantages of component-based rigorous design and time-triggered RTOS-based implementations. For this purpose, the applied method is based on the use of:

A high-level component-based modelling platform; timed BIP

This platform is based on well-defined operational semantics and is prone for expressing structured coordination between components. The behavior of each of the atomic components of a BIP model is described by using timed automata. Composite components are described as the composition of atomic components by using connectors and priorities. Verification and analysis of component-based BIP models are possible by using tools such as RTD-Finder [8] for compositional verification.

A safety-oriented Real-Time Operating System (RTOS); PharOS [9] implementation

This framework provides a language to describe a TT application as a set of communicating TT tasks (called agents). It provides low-level primitives allowing to specify timing constraints of different computations and communication actions of TT tasks. PharOS ensures, by principle, some important safety properties as the coherence of the data and determinism of real-time behavior [36].

Semantics-preserving transformation process

It allows to generate automatically correct-by-construction PharOS implementation from a high-level BIP model. Thus, all properties that are satisfied by the original model, are satisfied by-construction by the obtained implementation. A posteriori verification of these properties is thus unnecessary. And the determinism of the application is guaranteed by the PharOS platform. This process is defined in two steps:

- Step 1: *A model-to-model transformation.* It transforms an original BIP model into a restricted one (TT-BIP model) with respect to a user-defined task mapping. We assume that the source model of the transformation consists only of flat connectors and atomic components. This assumption can not be considered as a restriction, since an arbitrary BIP model with hierarchical connectors and composite components can

be transformed into a flat model where all connectors are flat and components are atomic as shown in [47]. Although BIP provides a rich set of interactions, we only considered rendezvous interactions, as it is possible to transform trigger interactions into rendezvous. The aim of the step1 transformation is to obtain a model which is closer to any TT implementation. That is, to obtain a model where all intertask interactions are executed by a dedicated components and all interactions between these communication components and task components are send/receive interactions. These latter provide, on top of the synchronization, a unidirectional data transfer. Another essential criterion for building the transformation rules is the respect of the equivalence to the original model where interactions' conflicts are resolved by the BIP engine. In order to satisfy this criterion, the obtained model contains a component dedicated to conflict resolution and implementing the fully centralized committee coordination algorithm presented in [10].

- Step 2: *A model-to-code transformation.* It generates automatically TT implementation from the intermediate model specified in the Step 1. The generated code is a ΨC code (the programming language of PharOS applications). The input model of this transformation is first adapted into a model where all task components are flattened, i.e. all atomic components of the same task are composed. the adapted model is called TT-BIP* model.

  In order to be able to provide formal correctness proofs of the transformation from TT-BIP* to ΨC, we provided a formal model of the target implementation and defined its operational semantics. This model is called the TCA model, which is in the same abstraction level as the ΨC language. In this model, a task is an automaton, where nodes present states and transitions allow to model actions. These latter are labelled by triplet-labels specifying release, deadline and/or synchronization dates. The transformation rules aim at transforming each transition of a the original component automaton, into a set of successive transitions in TCA model. Time progress conditions and timing constraints are mapped using deadlines and/or release dates in TCA model, while communicating transition (i.e. transitions labelled by send or receive ports), are transformed into a set of transitions, among labels of which we find a synchronization constraint.

  Since the semantics of the proposed TCA model are defined as LTSs, the correctness proof of the transformation is based on the notion of the bi-simulation between single LTSs of TT-BIP* components and their corresponding TCA tasks. The equivalence between the obtained application and the initial TT-BIP* model, follows from the equivalence between single components and tasks, since communication (i.e. data transfer) in both models is guaranteed by construction to happen in the same instant over the original clock.

Implementation of transformation tools

For the step 1 of the transformation, we have developed an automatic transformation tool that generates a 3-layer model called TT-BIP model depending on a user-defined task mapping. The step 2 of the transformation consists in merging atomic components of a composite task and then generating ΨC code. The code generator has been developed while the merge tool is still under construction. Note that a similar merging tool has been developed for untimed BIP models (cf. [47]). We believe that the adaptation of this tool for the timed models is straightforward. Regarding experiments, we considered two applications: the flight Simulator (FS) application and the medium voltage protection relay application. For the first application, we studied the impact of the task mapping on the generated code. And for the second application, we studied the impact of the transformation on some performance aspects compared to a manually written version.

Contribution of the thesis from the point of view of hard real-time software engineering

One of the major contributions of our transformational method, from software engineering point of view, is the cost reduction in the development of safety-critical applications. That is, it spares re-writing effort of the implementation code in case of re-designing, adding a newly defined component or modifying an existing component or connector in the original model. Being automatically implemented, the generation of the TT implementation code corresponding to the newly modified model does not incur additional development costs.

Another major asset of this approach, is the offered correspondence between the original model components and tasks of the generated source code. This allows the engineer to trace back faulty runs of the implementation code to the ill-behaving component of the BIP model. Note also that the choice of merging composite components only in the second step of the transformation enhances this backward association mechanism.

Future Work

For future work, we are considering several research directions:

About generalising the proposed approach

Here, we present some extensions to explore. The first two points are related to the extension of the input model of the transformational approach. While, the other points focus on different extension options of the target implementation, paradigm or execution model.

- **To a random input BIP model**. An important future direction is to consider BIP models with priorities. We agree that priorities complicate the problem. Unlike

conflict-resolution, priorities must be applied globally which requires approaches allowing to compute the global state of the system or at least to approximate the next reachable state. For TT implementation, this leads to an extremely considerable communication overhead. Since task agents need to communicate their states to TTCC agents in order to compute priorities and decide whether an interaction with a higher priority is enabled at that date.

- **To other high-level modelling languages.** We believe that one of the main contributions of our work is the correct-by-construction transformational approach, which is based on the well defined semantics of BIP models. This makes this latter essential to this approach. Nevertheless, BIP has been shown to be very expressive. In particular, there exist transformations of various existing languages (Lustre, Simulink, AADL,...) into untimed BIP models. Adapting these transformations for timed models would clearly extend the applicability of our transformation to any of these high-level modelling language.

- **To other RTOS implementations based on the TT paradigm**. We believe that, our generation approach can be adapted for any RTOS that is based on the TT paradigm, since it only relies on a primitive mechanism for communicating data and on timing constraints for implementing BIP synchronisation. The first step of the transformation is to be reused as it is. While the second step is to be adapted for the new programming language and its associated primitives.

- **To other paradigms: Event-Triggered (ET) paradigm for example**. In our approach, the transformation from BIP models to TT-BIP models (i.e. the first step) is motivated by the communication model of the TT-paradigm which is based essentially on the temporal variables. When the target paradigm is the event-triggered one, the same logic can be followed for the new communication model. Similarly, the code generation step for the new paradigm can follow the same principles while respecting the new primitive mechanisms.

  The most difficult part in considering the ET paradigm in this approach is to handle the external events. We believe that this can be resolved in BIP model level. Two options would be possible. The first one would consist in dedicating a component for handling the external events in the original BIP model. The idea of the second option is to extend the BIP language by introducing a mechanism that models the external events (e.g. a new port type, clock etc.).

- **To other execution models.** In all execution models tasks are modelled using automata. Therefore, to be able to extend the proposed approach, we need to present the semantics of the target task model in terms of LTS (as presented for the TCA model in this thesis). Therefore the transformation from BIP automata to the new task model and its formal correctness can follow the same principles as the proposed approach.

About the optimization of the generation phase

Another important line of research is to optimize the generation tool in order to generate an optimized code. In the proposed approach, we introduce in the original BIP models all the application components (sensors, actuators, simulators, controllers, etc.) and then we generate their corresponding source code for simulation. We believe that there is room for optimization of the generation process, especially when aiming at embedding the generated code. For example, some components (e.g. sensors, actuators etc.) can be mapped to their corresponding drivers. Therefore only source code for components of the control application can be generated by the generation tool.

In another hand, in TT-BIP models, TTCC components handle interactions that are originally modelled by BIP connectors and relating two tasks of the original application. Therefore These TTCC components —as well as the CRP component—are only instantiated for communication purpose. We strongly believe that in these components (i.e. TTCC and CRP components), we can identify exactly the same behavioural pattern of one or more of the RTOS services. Code generation could take this into account and only transform into TCA automata the part of the component which can not be mapped into an OS service. The identified pattern (corresponding to the RTOS service) could be, then, just mapped to a system call.

Bibliography

[1] Flexray communications system - electrical physical layer specification, v3.0.1, flexray consortium, october 2010. 41

[2] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-based implementation of real-time applications. pages 229–238, May 2010. 7, 75

[3] Rajeev Alur. Timed automata. In *Computer Aided Verification*, pages 8–22. Springer, 1999. 15

[4] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *Automata, languages and programming*, pages 322–335. Springer, 1990. 15

[5] Rajeev Alur and David Dill. The theory of timed automata. In *Real-Time: Theory in Practice*, pages 45–73. Springer, 1991. 7, 15

[6] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994. 15

[7] Rajeev Alur and A Thomas. Real-time system= discrete system+ clock variables. *International Journal on Software Tools for Technology Transfer*, 1(1-2):86–109, 1997. 15

[8] Lacramioara Aştefănoaei, Souha Ben Rayana, Saddek Bensalem, Marius Bozga, and Jacques Combaz. Compositional invariant generation for timed systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 263–278. Springer, 2014. 140, 150, 177

[9] Christophe Aussaguès, Damien Chabrol, Vincent David, Didier Roux, Natalia Willey, Arnaud Tournadre, and Marc Graniou. Pharos, a multicore os ready for safety-related automotive systems: results and future prospects. *Proc. of The Embedded Real-Time Software and Systems (ERTS2)*, 2010. 7, 44, 140, 177

[10] Rajive Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *Software Engineering, IEEE Transactions on*, 15(9):1053–1065, 1989. 9, 59, 60, 68, 83, 140, 178

[11] Massimo Baleani, Alberto Ferrari, Leonardo Mangeruca, Alberto L Sangiovanni-Vincentelli, Ulrich Freund, Erhard Schlenker, and H-J Wolff. Correct-by-construction transformations across design environments for model-based embedded software development. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 1044–1049. IEEE, 2005. 57

[12] Ananda Basu. *Component-based modeling of heterogeneous real-time systems in BIP*. PhD thesis, Université Joseph-Fourier-Grenoble I, 2008. 13

[13] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Formal Techniques for Networked and Distributed Systems–FORTE 2008*, pages 116–133. Springer, 2008. 154

[14] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 3–12. Ieee, 2006. 13, 20

[15] Ananda Basu, Laurent Mounier, Marc Poulhies, 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)*, pages 257–260. IEEE, 2007. 149

[16] Belgacem Ben Hedia and Etienne Hamelin. Projet openprod rapport r4.28 : Model to embedded real-time transformation. Technical report, 2012. 164

[17] Saddek Bensalem, Marius Bozga, T-H Nguyen, and Joseph Sifakis. Compositional verification for component-based systems and application. *IET software*, 4(3):181–193, 2010. 150

[18] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *International Conference on Computer Aided Verification*, pages 614–619. Springer, 2009. 150

[19] Simon Bliudze and Joseph Sifakis. The algebra of connectorsâĂŤstructuring interaction in bip. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008. 27

[20] Simon Bliudze and Joseph Sifakis. Causal semantics for the algebra of connectors. *Formal methods in system design*, 36(2):167–194, 2010. 27

[21] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 209–218. ACM, 2010. 57, 58, 60, 151

[22] Borzoo Bonakdarpour, Marius Bozga, and Jean Quilbeuf. Model-based implementation of distributed systems with priorities. *Design Automation for Embedded Systems*, 17(2):251–276, 2013. 151

[23] Etienne Borde, Smail Rahmoun, Fabien Cadoret, Laurent Pautet, Frank Singhoff, and Pierre Dissaux. Architecture models refinement for fine grain timing analysis of embedded systems. In *Rapid System Prototyping (RSP), 2014 25th IEEE International Symposium on*, pages 44–50. IEEE, 2014. 56

[24] Robert Bosch. CAN specification version 2.0. *Rober Bousch GmbH, Postfach*, 300240, 1991. 41

[25] Paraskevas Bourgos. *Rigorous Design Flow for Programming Manycore Platforms*. PhD thesis, Grenoble, 2013. 57

[26] 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 *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 11–20. IEEE, 2011. 149

[27] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-source architecture transformation for performance optimization in bip. *Industrial Informatics, IEEE Transactions on*, 6(4):708–718, 2010. 29, 71, 151

[28] Marius Bozga, Vassiliki Sfyrla, and Joseph Sifakis. Modeling synchronous systems in BIP. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 77–86. ACM, 2009. 149

[29] Fabien Cadoret, Etienne Borde, Sebastien Gardoll, and Laurent Pautet. Design patterns for rule-based refinement of safety critical embedded systems models. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*, pages 67–76. IEEE, 2012. 56

[30] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *ACM Sigplan Notices*, volume 38, pages 153–162. ACM, 2003. 56

[31] Damien Chabrol, Vincent David, Christophe Aussagues, Stéphane Louise, and Frédéric Daumas. Deterministic distributed safety-critical real-time systems within the oasis approach. In *IASTED PDCS*, pages 260–268, 2005. 7, 44

[32] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984. 60, 83

[33] K Mani Chandy and Jayadev Misra. Parallel program design: A foundation addison-wesley. *Reading, MA*, 1988. 59

[34] M Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. Translating aadl into bip-application to the verification of real-time systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 5–19. Springer, 2008. 149

[35] Flexray Consortium et al. Flexray communications system protocol specification version 2.1, 2005. *Available at http{ www. flexray. com}*. 41

[36] Vincent David, Jean Delcoigne, Evelyne Leret, Alain Ourghanlian, Philippe Hilsenkopf, and Philippe Paris. Safety properties ensured by the oasis model for safety critical real-time systems. In *International Conference on Computer Safety, Reliability, and Security*, pages 45–59. Springer, 1998. 7, 44, 140, 177

[37] Hartmut Ehrig and Claudia Ermel. Semantical correctness and completeness of model transformations using graph and rule transformation. In *International Conference on Graph Transformation*, pages 194–210. Springer, 2008. 58

[38] Wilfried Elmenreich. Time-triggered fieldbus networks state of the art and future applications. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 436–442. IEEE, 2008. 38, 39

[39] Wilfried Elmenreich, Gunther Bauer, and Hermann Kopetz. The time-triggered paradigm. In *Proceedings of the Workshop on Time-Triggered and Real-Time Communication, Manno, Switzerland*, 2003. vii, 37

[40] Hilding Elmqvist and Sven Erik Mattsson. An introduction to the physical modeling language modelica. In *Proceedings of the 9th European Simulation Symposium, ESS*, volume 97, pages 19–23. Citeseer, 1997. 164

[41] Shanna-Shaye Forbes. Real-time C code generation in Ptolemy ii for the giotto model of Computation. Technical report, DTIC Document, 2009. 57

[42] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. 56

[43] 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. 42

[44] Thomas A Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *Embedded Software*, pages 166–184. Springer, 2001. 42

[45] Thomas A Henzinger, Christoph M Kirsch, Marco AA Sanvido, and Wolfgang Pree. From control models to real-time code using giotto. *IEEE control systems*, 23(1):50–64, 2003. 57

[46] Jerome Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4):42, 2008. 56

[47] Mohamad Jaber. *Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP*. Theses, Université Joseph-Fourier - Grenoble I, October 2010. 29, 60, 67, 70, 71, 83, 105, 140, 151, 178, 179

[48] Mathieu Jan, Vincent David, Jimmy Lalande, and Maurice Pitel. Usage of the safety-oriented real-time OASIS approach to build deterministic protection relays. In 5^{th} *Intl. Symp. on Industrial Embedded Systems (SIES 2010)*, pages 128–135, Univ. of Trento, 2010. 46, 169, 170, 173

[49] Gabor Karsai and Anantha Narayanan. On the correctness of model transformations in the development of embedded systems. In *Monterey Workshop*, pages 1–18. Springer, 2006. 58

[50] Christoph M Kirsch, Marco AA Sanvido, Thomas A Henzinger, and Wolfgang Pree. A giotto-based helicopter control system. In *International Workshop on Embedded Software*, pages 46–60. Springer, 2002. 57

[51] Christoph M Kirsch and Ana Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012. 42

[52] Hermann Kopetz. The time-triggered approach to real-time system design. *Predictably Dependable Computing Systems. Springer*, 1995. 6, 37

[53] Hermann Kopetz. The time-triggered model of computation. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 168–177. IEEE, 1998. 37

[54] Hermann Kopetz. A comparison of ttp/c and flexray. *Inst. for Computer Eng., Vienna*, 2001. 38

[55] Hermann Kopetz. Time-triggered real-time computing. *Annual Reviews in Control*, 27(1):3–13, 2003. 37

[56] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer, 2011. 5, 35

[57] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered ethernet (tte) design. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 22–33. IEEE, 2005. 39, 40

[58] Hermann Kopetz and Gunther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003. 38, 56

[59] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: The mars approach. *IEEE Micro*, 9(1):25–40, 1989. 38

[60] Hermann Kopetz and Günter Grunsteidl. Ttp-a time-triggered protocol for fault-tolerant real-time systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 524–533. IEEE, 1993. 39

[61] Hermann Kopetz and KH kim. Temporal uncertainties in interactions among real-timebjects. In *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*, pages 165–174. IEEE, 1990. 37

[62] Hermann Kopetz and Roman Nossal. Temporal firewalls in large distributed real-time systems. In *Distributed Computing Systems, 1997., Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of*, pages 310–315. IEEE, 1997. 37

[63] Jean-Claude Laprie. Dependability: Basic concepts and terminology. In *Dependability: Basic Concepts and Terminology*, pages 3–245. Springer, 1992. 38

[64] Gabriel Leen and Donal Heffernan. Ttcan: a new time-triggered controller area network. *Microprocessors and Microsystems*, 26(2):77–94, 2002. 41

[65] Matthieu Lemerre, Vincent David, Christophe Aussaguès, and Guy Vidal-Naquet. An introduction to time-constrained automata. *arXiv preprint arXiv:1010.5571*, 2010. 50, 106

[66] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973. 47

[67] Stéphane Louise, Vincent David, Jean Delcoigne, and Christophe Aussagues. Oasis project: deterministic real-time for safety critical embedded systems. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 223–226. ACM, 2002. 7, 44

[68] Stéphane Louise, Matthieu Lemerre, Christophe Aussagues, and Vincent David. The oasis kernel: A framework for high dependability real-time systems. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pages 95–103. IEEE, 2011. 7, 44

[69] Robin Milner. *Communication and Concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995. 89, 126

[70] Lory D Molesky, Chia Shen, and Goran Zlokapa. Predictable synchronization mechanisms for multiprocessor real-time systems. *Real-Time Systems*, 2(3):163–180, 1990. 42

[71] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud, and Freddy Allilaire. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, 2, 2006. 157

[72] Kathy Dang Nguyen, PS Thiagarajan, and Weng-Fai Wong. A uml-based design framework for time-triggered applications. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 39–48. IEEE, 2007. 56

[73] Ayoub Nouri, Saddek Bensalem, Marius Bozga, Benoit Delahaye, Cyrille Jegourel, and Axel Legay. Statistical model checking qos properties of systems with sbip. *International Journal on Software Tools for Technology Transfer*, 17(2):171–185, 2015. 150

[74] Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. Timing analysis of the flexray communication protocol. *Real-time systems*, 39(1-3):205–235, 2008. 56

[75] Wolfgang Pree, Gerald Stieglbauer, and Josef Templ. Simulink integration of giotto/tdl. In *Automotive Software Workshop*, pages 137–154. Springer, 2004. 57

[76] Wolfgang Pree and Josef Templ. Modeling with the timing definition language (tdl). In *Automotive Software Workshop*, pages 133–144. Springer, 2006. 44, 57

[77] Jean Quilbeuf. *Distributed Implementations of Component-based Systems with Prioritized Multiparty Interactions. Application to the BIP Framework.* PhD thesis, Université de Grenoble, 2013. 20, 67, 70, 71, 83, 151

[78] Khaled Rahmouni, Patrice Gerin, Sebastien Chabanet, Paul Pianu, and Frédéric Pétrot. Modelling and architecture exploration of a medium voltage protection device. In *2009 IEEE International Symposium on Industrial Embedded Systems*, pages 46–49. IEEE, 2009. 171

[79] PD Stefan Resmerita and W Pree. Timing definition language (tdl) modeling in ptolemy ii. *Department of Computer Science, University of Salzburg, Tech. Rep*, 2008. 57

[80] Vassiliki Sfyrla. *Modeling Synchronous Systems in BIP.* PhD thesis, PhD thesis, Université de Grenoble, 2011. 149

[81] Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga, and Joseph Sifakis. Compositional translation of simulink models into synchronous bip. In *International Symposium on Industrial Embedded System (SIES)*, pages 217–220. IEEE, 2010. 149

[82] Shehryar Shaheen, Donal Heffernan, and Gabriel Leen. A comparison of emerging time-triggered protocols for automotive x-by-wire control networks. *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, 217(1):13–22, 2003. 38

[83] Joseph Sifakis. Component-based construction of real-time systems in bip. In *CAV*, pages 33–34, 2009. 13

[84] Till Steinbach, Franz Korf, and Thomas C Schmidt. Comparing time-triggered ethernet with flexray: An evaluation of competing approaches to real-time for in-vehicle networks. In *Factory Communication Systems (WFCS), 2010 8th IEEE International Workshop on*, pages 199–202. IEEE, 2010. 38

[85] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on software engineering*, 23(12):759–776, 1997. 41

[86] Ahlem Triki. *Distributed Implementations of Timed Component-based Systems*. PhD thesis, Grenoble Alpes, 2015. 57, 58, 60, 67, 70, 83, 152

[87] Jos B Warmer and Anneke G Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003. 158