Distributed Implementations of Timed Component-based Systems

Ahlem Triki

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Ahlem Triki

Thèse dirigée par **Saddek Bensalem**
et codirigée par **Jacques Combaz**

préparée au sein **VERIMAG, UMR5104**
et de **MSTII**

Distributed Implementations of Timed Component-based Systems.

Thèse soutenue publiquement le **9 juin 2015**,
devant le jury composé de :

Marie-Laure Potet
Professeur, Grenoble INP, Présidente
Eugene Asarin
Professeur, Université Paris Diderot - Paris 7, Rapporteur
Kamel Barkaoui
Professeur, Cnam Paris, Rapporteur
Félix Ingrand
Chargé de Recherche, LAAS, Examinateur
Jean-Pierre Talpin
Directeur de Recherche, INRIA, Examinateur
Saddek Bensalem
Professeur, Université Joseph Fourier, Examinateur
Jacques Combaz
Ingénieur de Recherche, CNRS, Examinateur

Acknowledgments

First of all, I would like to thank the jury members, Prof Eugene Asarin and Prof Kamel Barkaoui for the time and effort taken to review my manuscript and Prof Jean Pierre Talpin, Dr Félix Ingrand and Prof Marie Laure Potet for examining my work and for their valuable remarks.

I wish to express my greatest thank to my advisor Prof Saddek Bensalem for giving me the chance to do my PhD in Verimag. His encouragement and trust in the hardest moments helped me a lot to accomplish this work.

My thanks go also to my co-advisor Dr Jacques Combaz for his time and his availability. I thank him for the fruitful discussions we had and for his great help in both theoretical and technical aspects.

I would like to thank D. Marius Bozga for his precious remarks and help, Dr Jean Quilbeuf and Dr Borzoo Bonakdarpoor for their collaborations.

I want to thank all Verimag's members and DCS team for the great working environment. My special thanks go to my colleagues and friends Najah, Souha, Ayoub from Verimag and Molka, Wijdene, Yassine and Oussama from CEA for the special moments we had and for the time we were talking in the CTL's caffet.

Finally, I want to express my infinite gratitude and thank to my family, my parents and my brother for unconditionally providing their love, guidance and encouragement. A special thank to my husband Wael for his support and for being by my side all the time.

Abstract

Correct distributed implementation of real-time systems has always been a challenging task. The coordination of components executing on a distributed platform has to be ensured by complex communication protocols taking into account their timing constraints.

In this thesis, we propose rigorous design flow starting from a high-level model of an application software in BIP (Behavior, Interaction, Priority) and leading to a distributed implementation. The design flow involves the use of model transformations while preserving the functional properties of the original BIP models. A BIP model consists of a set of components synchronizing through multiparty interactions and priorities. Our method transforms high-level BIP models into Send/Receive models that operate using asynchronous message passing. The obtained models are directly implementable on a given platform.

We present three solutions for obtaining Send/Receive BIP models. In the first solution, we propose Send/Receive models with a centralized scheduler that implements interactions and priorities. Atomic components of the original models are transformed into Send/Receive components that communicate with the centralized scheduler via Send/Receive interactions. The centralized scheduler is required to schedule interactions under some conditions defined by partial state models. Those models represent high-level representation of parallel execution of BIP models.

In the second solution, we propose to decentralize the scheduler. The obtained Send/Receive models are structured in 3 layers: (1) Send/Receive atomic components, (2) a set of schedulers each one handling a subset of interactions, and (3) a set of components implementing a conflict resolution protocol.

With the above solutions, we assume that communication latencies are negligible so that there is no delay between the decision to execute an interaction in a scheduler and its execution in the participating components. In the third solution, we propose Send/Receive models that execute correctly even in presence of slow communications. This solution is based on the fact that schedulers plan interactions execution and notify components in advance. In order to plan correctly the interactions, we show that the schedulers are required to observe additional components, besides the ones participating in the interactions. We present also a method to optimize the number of observed components, based on the use of static analysis techniques.

From a given Send/Receive model, we generate a distributed implementation where Send/Receive interactions are implemented by TCP sockets. The experimental results on non trivial examples and case studies show the efficiency of our method.

Résumé

L'implémentation distribuée des systèmes temps-réel a été toujours une tâche non-triviale. La coordination des composants s'exécutant sur une plateforme distribuée doit être assurée par des protocoles de communication complexes en tenant compte de leurs contraintes de temps. Dans cette thèse, nous proposons un flot de conception rigoureux à partir d'un modèle de haut niveau d'un logiciel d'application décrit en BIP (Behavior, Interaction, Priority) et conduisant à une implémentation distribuée. Le flot de conception implique l'utilisation de transformations de modèles tout en conservant les propriétés fonctionnelles des modèles originaux de BIP. Un modèle BIP se compose d'un ensemble de composants qui se synchronisent à travers des interactions et des priorités. Notre méthode transforme les modèles BIP en un modèle Send/Receive qui fonctionnent en utilisant le passage de messages asynchrones. Les modèles obtenus sont directement implémentés sur une plate-forme donnée. Nous présentons trois solutions pour obtenir un modéle Send/Receive. Dans la première solution, nous proposons des modèles Send/Receive qui fonctionnent avec un ordonnanceur centralisé qui implémente les interactions et les priorités. Les composants atomiques des modèles originaux sont transformés en composants Send/Receive qui communiquent avec l'ordonnanceur centralisé via des interactions de type Send/Receive. L'ordonnanceur centralisé exécute les interactions sous certaines conditions définies par les modèles à états partiels. Ces modèles représentent une description haut niveau de l'exécution parallèle de modèles BIP. Dans la deuxième solution, nous proposons de décentraliser l'ordonnanceur. Les modèles Send/Receive obtenus sont structurées en trois couches: (1) les composants Send/Receive (2) un ensemble d'ordonnanceurs, chacun exécutant un sous-ensemble d'interactions, et (3) un ensemble de composants implémentant un protocole de résolution des conflits. Avec les solutions décrites ci-dessus, nous supposons que les latences de communication entre les composants sont négligeables. Ceci est du au fait que les modèles Send/Receive sont conçus de telle sorte qu'il n'y ait pas retard entre la décision d'exécuter une interaction dans un ordonnanceur et son exécution dans les composants participants. Dans la troisième solution, nous proposons des modéles Send/Receive qui exécutent correctement même en présence de latences de communication. Cette solution est basée sur le fait que les ordonnanceurs planifient l'exécution des interactions et notifient les composants à l'avance. Afin de planifier correctement les interactions, nous montrons que les ordonnanceurs sont tenus à observer des composants supplémentaires, en plus de ceux qui participent aux interactions. Nous présentons également une méthode pour optimiser le nombre de composants observés, en se basant sur l'utilisation de techniques d'analyse statique. A partir d'un modèle Send/Receive donné, nous générons une application distribuée où les interactions Send/Receive sont implémentées par les sockets TCP. Les résultats expérimentaux sur des exemples non triviaux et des études de cas montrent l'efficacité de notre méthode.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context

Nowadays, computer systems are becoming widely pervasive. They are touching different type of application domains such as automotive, avionics, air traffic control, smart home, telecommunications, etc. Despite of the existence of some techniques such as formal verification, simulation and testing, used to improve system performance, reliability, and correctness, systems meeting desired properties is still time-consuming and non trivial task due do their increasing complexity.

Component-based approach is established as the most fundamental technique to cope with complexity of systems. The principle of this approach is to build complex systems by assembling smaller components (building blocks) characterized by their interface, an abstraction that is adequate for composition and reuse. Composition of components is achieved with respect to a notion of "glue" operator. "Gluing" can be seen as an operation that takes input components and their constraints, and that outputs a complex system. Component-based systems provide clear descriptions of their behaviors which make them adequate for a correct-by-construction process. In addition, they allow incremental modification of components without requiring global changes, which may significantly simplify the verification process.

Real-time systems [1] are systems that are subject to "real-time constraints". The correctness of such systems depends not only on the logical results of the actions, but also on the physical instants at which these results are produced. Correctness of real-time system includes the satisfaction of its timing constraints, that is, the instants at which actions are produced have to meet specific conditions. Timing constraints are usually expressed by user-defined bounds in which the system has to react (i.e. executing actions), e.g. deadlines. There are two main classes of real-time systems: hard real-time systems where it is absolutely imperative that the system react within the specified bounds; and soft real-time where

response times are important but the system can still function if the constraints are occasionally violated. Meeting timing constraints depends on features of the execution platform such as its speed. For instance, when the platform is not speed enough, the timing constraints may be violated. Worst Case Execution Time (WCET) can be performed in order to verify whether the target platform is fast enough to respect the timing constraints.

The building process of real-time systems includes in general two essential phases, modeling and implementation.

1.1.1 Modeling

In the modeling phase, the real-time system specifications are transformed into a model. This phase is very beneficial as it deals with the complexity of real-time systems at a high-level layer where implementation details are omitted. The high-level model abstracts the system behavior and the interaction between components. For instance, abstractions assume instantaneous execution of actions, and zero-delay communication between components.

Modeling provides advantages such as:

- construction ease;
- integrating models of heterogeneous components;
- introducing nondeterminism behaviors, and
- analysis using formal methods.

In practice, high-level models are written in high-level programming languages, sometimes with formal semantics. Those languages provides high-level primitives used for components coordination. In this thesis, we consider high-level models consisting of sets of components, subject to timing constraints and coordinating their actions through multiparty interactions [2], that is strong synchronizations between possibly more than two components.

1.1.2 Implementation

The second phase towards building a real-time system is to derive the implementation from its high-level model. Implementation compromises the abstractions of the high-level model. For instance, in high-level models, actions are assumed to take zero time. Nonetheless, in the implementation actions may take arbitrary execution times. This may lead the system to not meet the timing constraints specified in the high-level model.

The implementation could be either centralized or distributed. This depends on the topology of the underlying architecture on which the real-time system will be deployed. In this thesis, we are interested in distributed implementation of real-time systems.

Distributed implementation assumes that components are located on networked computers and communicate by exchanging messages. One reason for considering distributed implementations could be the physical locations of components that constitute the considered systems. For instance, processing sensor data and controlling actuators requires dedicated computing units in specific locations. A distributed system [3] consists of a set of autonomous and independent components. At each step, a component executes one of three different types

of action: sending a message, performing a local computation and waiting for an incoming message. The decision of the next action to execute is taken locally by the components, depending on the messages received so far and the computation results. Distributed implementation requires the use of low-level primitives for communication between components. This may be a real issue for deriving distributed implementations from high-level models as high-level synchronization primitives could not be easily expressed by low-level communication primitives. This requires the use of protocols to ensure correct implementation of high-level primitives. As already stated, we consider multiparty interactions as high-level primitives for our models. Distributed implementation of multiparty interactions is not trivial. The difficulty resides in ensuring globally consistent decisions in the components while each one is running independently. In the literature, many protocols have been proposed for distributed implementation of multiparty interactions. In Section 1.2, we present some of these protocols.

Distributed implementation of systems is even more challenging and complex when systems are subject to timing constraints, that is, for real-time systems. This is due to the fact that one not only has to consider typical problems of distributed systems, but also should ensure that all subtle interleavings of the system meet the timing constraints. In addition, distributed implementation of real-time systems has to cope with specific issues such as clocks drift [4]. In fact, components are deployed on distributed machines having local clocks that generally cannot be perfectly synchronized. Thus, components may not have the same reference of time, which may discard system consistency, e.g. strict ordering of operations may be compromised. To overcome this problem, usual solutions rely on clock synchronization protocols [5, 6] that guarantee the synchronization of clocks up to given precision. In this thesis, we do not address the issue of clock drifts. We assume that distributed machines share a common clock, or equivalently that a clock synchronization protocol is already established.

Building correct and efficient distributed implementations of real-time systems is a real challenge. The implemented system must meet the specifications described by the high-level model such as timing constraints. However, it is still unclear how to transform a high-level model where atomicity is assumed and implementation details are omitted by the use of high-level primitives into a correct distributed implementation.

1.2 Existing Protocols for Distributed Implementation of Multiparty Interactions

Multiparty interactions provide a high-level description of a distributed system in terms of components and interactions. An action of the system is an interaction, which is a coordinated operation between an arbitrary number of components. Consider n components B_i, $i \in \{1 \dots n\}$ and a set of multiparty interactions γ. An interaction $a \in \gamma$ is a synchronization of a subset of components $\{B_i\}_{i \in I}$, $I \subset \{1 \dots n\}$. Two interactions a and b are conflicting if they share a common component. Correct distributed implementation of multiparty interactions ensure mutual exclusion of conflicting interactions. That is, if two conflicting interactions are possible from a given state of the system, only one of them can execute.

Distributed implementation of a multiparty interactions results in solving the committee

coordination problem [7], where a set of professors are organized in a set of committees and two committees can meet concurrently only if they have no professor in common; i.e., they are not conflicting. Conflict resolution is the main obstacle in distributed implementation of multiparty interactions.

The problem of distributed implementation of multiparty interactions has been studied extensively in [8–14]. These papers proposed protocols for multiparty interaction implementation and established their correctness. In this section we present an overview of some of these protocols.

1.2.1 α-core Protocol

The α-core protocol [15] provides a fully distributed solution where each multiparty interaction is handled by a separate coordinator. The α-core protocol specify two different behaviors for participants and coordinators. Communications between participants and coordinators is done by exchanging messages.

- Participants send to coordinators the following messages.

  PARTICIPATE This message indicates that the participant is interested in a single particular interaction (hence it can commit to it).

  OFFER This message indicates that the participant is interested in one out of several potentially available interactions (a non-deterministic choice).

  OK This message is a response to a **LOCK** message sent from a coordinator (described below) to indicate that the participant is willing to commit on the interaction.

  REFUSE This messages is a response to an **OFFER** message to indicate that the latter is not valid anymore or also a response to **LOCK** message from the coordinator.

- Coordinators send to participants the following messages.

  LOCK This message is response to an **OFFER** message from a participant, that requests the participant to commit to the interaction.

  UNLOCK This message is sent to a locked participant, indicating that the current interaction is canceled.

  START This message notifies a participant that it can start the interaction.

  ACKREF This message is an acknowledgment to a **REFUSE** message from a participant.

Initially, each participant computes its available interactions. If there is only one interaction available, then the participant sends a **PARTICIPATE** message to the corresponding coordinator. Otherwise, it sends an **OFFER** message to each coordinator handling available interactions and waits for **LOCK** messages. For the coordinator side, whenever all **OFFER** or **PARTICIPATE** messages have been received from the participants, the interaction is enabled. In the case where all participants have sent a **PARTICIPATE** message, a **START** message can be directly sent by the coordinator. Otherwise, the coordinator sends **LOCK**

messages to the participants that have sent **OFFER** to lock them, according to the global order on the participants. The **LOCK** message could be granted by the participant by sending an **OK** message to the coordinator or rejected by sending **REFUSE**. If all participants have been locked, the coordinator sends **START** messages to all participants. If the case where a **REFUSE** message has been received before the locking phase completes, the coordinator acknowledges receiving this message by sending **ACKREF** to the **REFUSE** message sender and sends **UNLOCK** messages to locked participants.

1.2.2 Kumar's Token

In [13], Kumar presented another solution for implementing multiparty interactions. The proposed solution does not require additional components, but rather it embeds a protocol in the original components. The protocol requires a token for each interaction. This token tries to progress from least to greatest component of the interaction, according to a fixed global order on the components.

Intuitively, a token can progress only if the visited process can commit to the corresponding interaction. If the token goes through all components participating in the interaction, then the interaction is safely executed. The last component that receives the token is responsible for notifying all other participating components to execute the interaction.

The component can propagate the token if:

- it can execute the interaction corresponding to the token, and
- it has not yet committed to another interaction.

When the component commits to an interaction, it propagates its corresponding token and waits until the latter succeeds of fails.

In the case where the component receives a token of an interaction that it cannot execute, the latter is canceled. Canceling an interaction corresponds to sending a cancel message to each component that already propagated the token. After receiving a cancel message, a component is not committed to the interaction anymore.

In the case where the component has already committed to an interaction and receives tokens for other interactions, it holds these tokens until the committed interaction succeeds or fails. If the committed interaction fails, one of the held tokens is propagated. Otherwise, that is, if the committed interaction succeeds, each interaction of the held tokens is canceled.

Note that even when the interaction is canceled, the token remains in the process it was visiting. When the process finishes its local computation and reaches a stable state, it sends backs each token corresponding to an enabled interaction to the first process of its path.

Intuitively, the correctness of this protocol comes from the following facts:

- Mutual exclusion is ensured by the fact that each component allows only one token to traverse at a time, then waits until the corresponding interaction succeeds or fails. In particular, conflicting interactions involve at least one common process and this process ensures their mutual exclusion.
- Synchronization comes from the fact that the start message is sent by the last process on the interaction path to all the other involved processes.

- If an interaction is enabled, then either the corresponding token can travel along the whole path and the interaction executes, or the token is blocked at the first process shared with a conflicting interaction. The global order ensures that the first conflicting component is the same for both interactions. Progress is ensured by the fact that either the conflicting interaction succeeds or a cancel message allows the token to go further along the path.

1.2.3 Bagrodia's protocols

Bagrodia proposes three protocols for distributed implementation of multiparty interactions [9, 16] namely, centralized implementation, token-ring based implementation and dining philosophers based implementation. These protocols are hand-shake-like protocols between components and several entities called managers. The first step of these protocols is initiated by components by sending offers containing available interactions to managers. Managers are required then to select one of the enabled interactions and send a message to participating components to notify them for the interaction execution.

In order to ensure mutual exclusion of conflicting interactions, managers rely on counters that count the number of offers sent by each component. Indeed, counters ensure that each offer cannot be consumed by more than one interaction.

In this thesis, we are using Bagrodia's protocols as conflict resolution protocols for our distributed implementation of multiparty interactions. We give details on these protocols in Subsection 5.3.3.

1.3 Existing Approaches for Modeling and the Implementation of Real-Time Systems

When developing real-time systems, it is important to make a clear distinction between physical and logical time. The notion of time serves two purposes. Firstly, it is used to specify the order of execution of individual actions of systems applications. Secondly, it can be used to specify durations. In one hand, the logical time can be used in both cases, especially in the design phase. In the other hand specifying the order of execution based on the physical time leads to the non-determinism of execution, since physical time is not known at the design stage. Existing rigorous implementation techniques use specific programming models. We present the time triggered and even-triggered approaches, and the synchounous paradigm. Finally, we present some component-based frameworks that allow the design and the implementation of real-time systems.

1.3.1 Time-Triggered Approach

The time-triggered approach [17] is dedicated for the implementation of safety-critical applications whose behavior needs to be certified. It addresses the fundamental issues of real-time systems by treating time as a first class quantity. It is based on a two-phased design methodology including an architecture design phase and a component design phase. The architecture

design phase specifies essentially the interactions between components as well as the components interfaces in the value and the temporal domains, which makes the system fully time deterministic. The component design phase takes these interface specifications as constraints for building components.

The time-triggered systems rely on a notion of Logical Execution Time (LET) [18–20] which corresponds to the difference between the release time and the due time of an action, defined in the program using an abstract notion of time. To cope with uncertainty of the underlying platform, a program behaves as if its actions consume exactly their LET: even if they start after their release time and complete before their due time, their effect is visible exactly at these times. This is achieved by reading input of actions exactly at their release time, and by producing output of actions exactly at their due time. The time-triggered approach consists in assigning to each action its desired execution time guaranteeing by construction the end-to-end constraints. The system is then implemented over an execution kernel running on the platform and ensuring the following.

1. An action must not start before the real time date of the corresponding logical instant, as this could compromise data coherence among components guided by the logical time.

2. An action being executing should not exceed the duration specified by its LET.

In order to prove correctness of safety critical systems, one approach could be an a-priori Worst Case Execution Time (WCET) analysis that verifies whether execution time requirements are respected by a given target platform. The non safety critical systems can occasionally violate the timing constraints. In case of violation there exists mechanisms for keeping the system in a consistent state (skip job execution, degraded mode, skip execution of less important tasks, etc.)

In the following we present three models based on the time-triggered approach, namely Oasis, Port-based Object and Giotto.

Oasis

Oasis [21–24] is a framework for safety-critical real-time systems, based on a time-triggered architecture. Oasis is a framework encompassing models, methodologies, and tools for the development of embedded critical software exhibiting completely deterministic temporal behavior. Oasis implementation comprises a programming language PsyC (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 set of communicating and interacting tasks called agents. Agents perform computation in parallel on private data, and communicate only using dedicated mechanisms namely temporal variables and message box. The processing of each task is synchronous at predefined time instants: at each end of the logical execution time, variables are made visible to other tasks.

Each task T in the system S is associated with a real-time clock H_T representing the set of physical instants at which input and output data are (or can be) observed. The system S defined a global and periodic real-time clock H_S which includes all observable instants of the system. It represents the smallest clock from which all clocks are derived, i.e. H_S exhibits a factor K_T and an offset δ_T such as $H_T = K_T * H_S + \delta_T$. Figure 1.1 shows the global clock

H_S and two other clocks H_{T_1} and H_{T_2} deriving from H_S as follows: $H_{T_1} = 2 * H_S + 1$ and $H_{T_2} = 3 * H_S$.

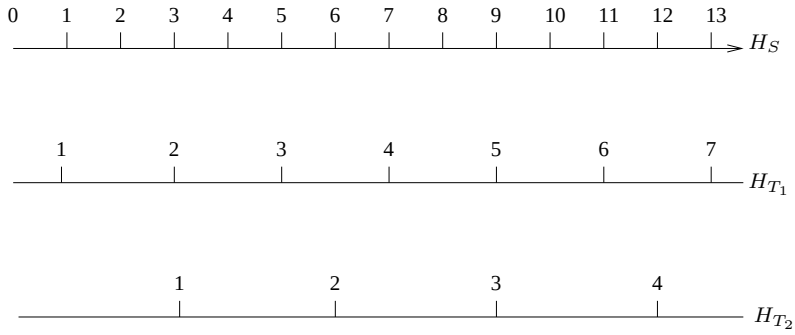


Figure 1.1: Example of clocks definition in Oasis.

In Oasis, tasks manipulate time through an instruction called "*advance*". An *advance* instruction splits the task code into two parts: the part of code (a processing) before and the part of code after the *advance* instruction. The instruction *advance*(k) sets both a deadline for the first part of code and a next activation instant for the second one. It is computed from the current instant of clock H_T plus $k * H_T$. Thus, agent's future instants are declared (i.e. activation instants) and both earliest start date and latest end date of each processing. Such specific temporal dates are called temporal synchronization points. Figure 1.2 shows elementary instructions in an Oasis application and the associated time intervals. *processing*1 has a deadline defined by *sta* defined by the instruction *advance*(k_1) which corresponds to the activation date of *processing*2. The deadline of *processing*2 is *end* defined by *advance*(k_2).

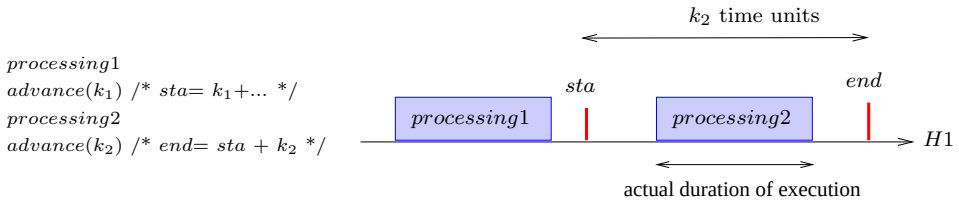


Figure 1.2: Elementary instruction and the associated time intervals.

There are two modes of communication between tasks. The first mode uses the shared variables, also called temporal variables, the second is by sending messages. Modifications of temporal variables are made visible at synchronization points only, while messages has explicit definition of visibility dates.

Each temporal variable defines a real-time data flow: its values, available to any agent that consults them, are stored and updated by a single writer, the owner agent, at a predetermined temporal rhythm. This rhythm is expressed in the source code as a regular time period parameter and allows computation of a periodic updating date. Based on the example given in Figure 1.2, at each physical instant between the two dates *sta* and *end*, the agent is logically considered at *sta* date. Assume that the agent has a temporal variable x whose value is modified by *processing*2, and that another agent "observes" the value of x, the last

activation date of observer agent is $t0$ between *sta* and *end*. At this date, the agent "observes" the past value of x corresponding to the date *sta*, i.e. $x(t0) = x(sta)$ (see Figure 1.3(a)).

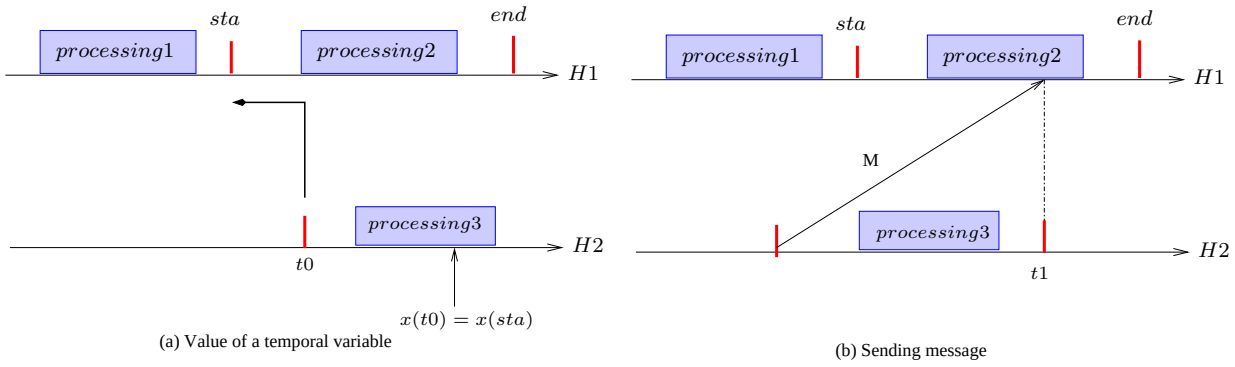


Figure 1.3: Communication mechanism in Oasis.

For the sending message mechanism, each message defines a visibility date specified by the sender. This date specifies the date beyond which a message can be observed by the recipient agent. The latter has queues for receiving messages that are sorted by their visibility. As an example, consider an agent sending a message M with visibility date $t1$ to the agent from Figure 1.2. The message cannot be observed before the *sta* date, since $t1 > sta$ (see Figure 22 (b)).

Oasis provides an off-line tool chain responsible for extracting the application's temporal behavior in order to generate a runtime. More precisely, it computes all the possible temporal behavior from which they can size communication buffers and analyze the timing constraints on the execution times. At runtime, Oasis relies on an Early Deadline First (EDF) algorithm [25], for dynamically scheduling elementary instructions of agents based on their temporal synchronization points and on monoprocessor architectures.

Oasis-D

Oasis-D [26–28] is an extension of the Oasis approach towards distributed architectures. It is based on TDMA-like protocol used to manage network accesses. TDMA (Time Division Multiple Access) [29] divides the global time into a sequence of time slots, each of them is assigned to an unique CPU. TDMA guarantees the absence of network interference in the slots. Such approach allows a safe and deterministic use of the network. Bounds on data transfers can be computed and taken into account as constraints to verify the schedulability of network accesses. In Oasis-D, network runtimes encapsulate the static TDMA schedule of network accesses.

The Oasis-D provides a tool chain that is responsible for:

1. extracting the network communication needs from the PsyC source code of a given application, and

2. computing the communication load of a given application over the network.

If this load is within the network capacity, it generates inside network runtimes:

1. the structure of each Oasis packet,
2. the required size for TDMA slots based on the network capacity, as well as the application communication needs, and
3. the schedule of network accesses.

The TDMA approach assumes the use of a global clock, so that each CPU can correctly use the network following the time-triggered paradigm. However, because each CPU has a local clock that can drift, a clock synchronization protocol between CPUs is required. Oasis-D currently uses a dedicated CPU which periodically sends correction values that each CPU of the distributed system needs to apply to its local clock.

Port-Based Object

The port-based object (PBO) [30] provides a software framework to program reconfigurable robots. A PBO model is composed of a set of tasks called PBO. Each PBO has input and output ports. PBOs are activated periodically and communicate through input and output ports via state variables that are stored in a global table. Indeed, each PBO stores in its own local table a subset of the data that is needed from the global table. Before executing a PBO, the state variables corresponding to its input ports stored in its local table are updated from the global table. After the execution completes, the state variables corresponding to its output ports are copied from its local table to the global table.

The access to the same state variable in the global table by two PBOs must be mutually exclusive. To ensure this, the PBO framework provides a mechanism based on spin-locks [31]. When a PBO needs to access the global table, it first locks the processor on which it is executing, and then waits to obtain the global lock for the global table. The global lock has the highest priority of all the PBOs. When holding the global lock, the PBO accesses the global table and exchanges data.

The amount of communication between a PBO and the global table varies from time to time depending on the size of exchanged data. As a result, if another PBO wants to read the output ports, it may or may not get the results obtained from the current cycle. For example, consider two PBOs, PBO1 and PBO2, where PBO2 reads the output of PBO1. Suppose that PBO1 and PBO2 are running on different CPUs and PBO1 is activated at every instant $t = 2k$ and PBO2 is activated at every intant $t = 4k + 2$, where $k = 0, 1, 2....$ Figure 1.4 depicts a possible execution trace of PBO1 and PBO2. Remark that the output of PBO1 has not been produced in the first period of PBO2. Therefore, PBO2 reads the last value of the state variable corresponding to the output of PBO1. However, in the second period of PBO1, PBO2 reads the fresh output of PBO1.

Giotto

Giotto [20] is a programming methodology for embedded control systems running on possibly distributed platforms. It provides a time-triggered programming model for periodic, hard

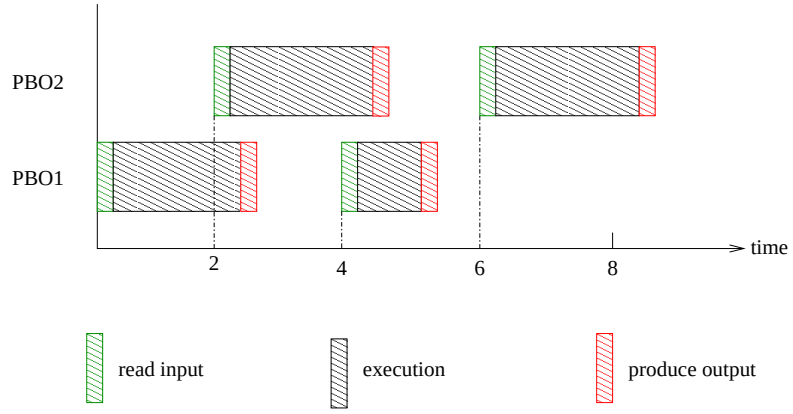


Figure 1.4: Example of PBO execution.

real-time systems. In this model, the execution of tasks is triggered by periodic clocks. 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. For example, consider a task that is activated every 2 ms as shown in Figure 1.5. The input data is consumed at the beginning of the 2 ms, the execution is performed within the period, and the output is made available at the end of the period. Thus no matter how quickly or slowly the controller executes, as long as it can finish in 2 ms, the outputs will be produced at the exact time instants.

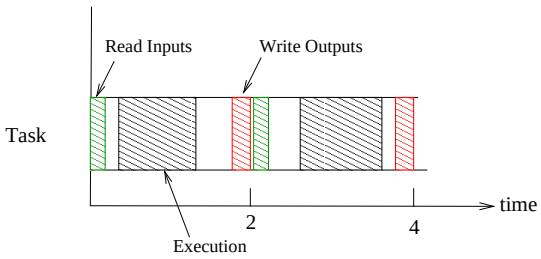


Figure 1.5: Example of Giotto execution.

In Giotto all data is communicated through ports. A port represents a typed variable with a unique location in a globally shared name space. There are three types of ports in a Giotto program: sensor ports, actuator ports, and task ports. The task ports are used to communicate data between concurrent tasks. 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 [20].

1.3.2 Event-Triggered Approach

The event-triggered approach [32] is adopted for the design of systems reacting to unpredictable events. In the following , we present two even-triggered models, namely Timed Multitasking and Ptides.

Timed Multitasking

The Timed Multitasking (TM) model assumes that a system is composed of a set of components called actors [33]. An actor has an interface that consists of input/output ports and parameters. An actor can specify its execution requirements, including priority, execution times, and deadlines, through its parameters.

There are two types of actors in TM which are:

- actors that handle external events called interrupt service routines (ISRs), and
- actors that are triggered internally by messages generated by other actors, called regular tasks.

An ISR actor converts external events from the environment into events that triggers other actors. It does not specify deadlines and trigger conditions in its interface. A regular task actor has a much richer interface (including priorities, execution times and deadlines) than an ISR actor. It is triggered by events at its input ports. Communications between regular actors is done through sending/receiving events through their ports. A regular actor is allowed to execute only if there is an input event that triggers it. Output events are produced only when the deadline is reached. By controlling when output events are produced, TM controls both the starting time and stopping time of each task, thus obtaining deterministic timing properties [34]. TM assumes sequential execution of actors as there is only a single computation resource shared by all the task actors. The execution of ISR actors is managed by an independent thread and is triggered chronologically based on the timestamp of the external events. The execution of regular task actors is handled by a priority-based event handler, which sorts events based on their priorities. If preemption is allowed, then the execution of a lower priority task can by preempted by higher priority tasks. For example, when task A is running, an ISR actor produces an event that triggers task B which has a higher priority than A, then A will be preempted by B. The runtime environment of TM tracks when an output event can be produced. If the task can finish its execution before its deadline, the output is made available to its downstream tasks when the deadline is reached. In case a task misses its deadline, TM uses overrun handlers to complete the current execution of the task quickly to preserve time determinism as much as possible.

Ptides

Ptides [35] is a concurrent event-triggered model for distributed real-time systems. It uses a discrete-event (DE) model [36] as the underlying formal semantics to achieve deterministic behavior in both time and value. DE models consists of a set of concurrent components interacting via events. Each event has an associated time value called timestamp. Correct execution of these models requires respecting the order of timestamped events.

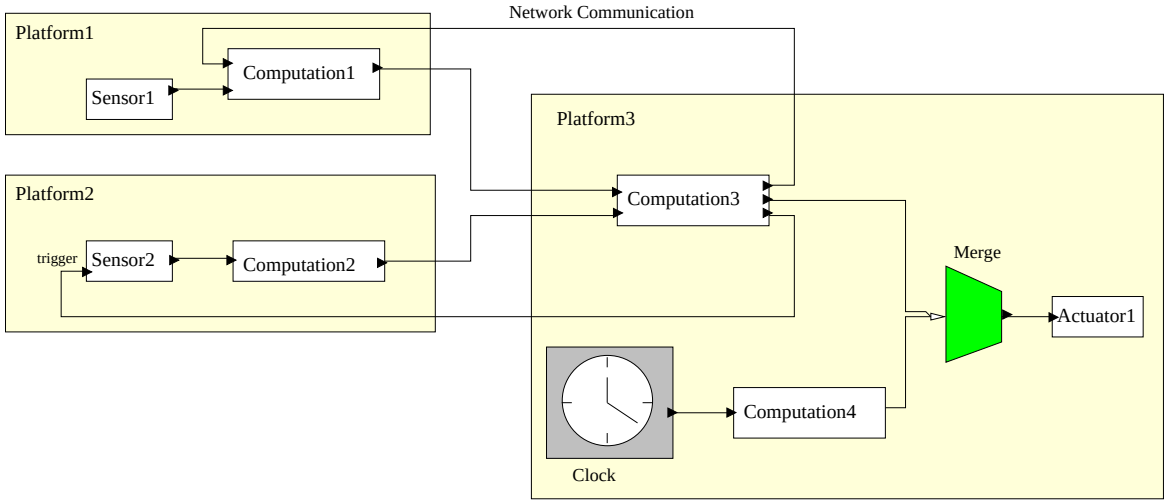


Figure 1.6: Distributed Embedded System.

Figure 1.6 (taken from [37]) depicts a model a simple embedded system distributed over 3 platforms. Each of its components produces and consumes events through event ports. These events cannot be produced/consumed at arbitrary times. For instance, an output event of Sensor1 component of Figure 1.6 is timestamped by the local time at which the sensor reading is taken. Thus, the real time at which the output event is produced is larger than its timestamp. Conversely, an input event of Actuator1 component is timestamped by the lastest real time at which the actuator is allowed to execute. Thus, the local real time at which the input event is produced is smaller than or equal to its timestamp. In fact, this timestamp can be considered as a deadline for the delivery of the event to the actuator.

For network interfaces, Ptides assumes that platforms use a local network to communicate timestamped events. Moreover, it assumes that the network delay is bounded and known in advance [37]. The timing constraint and the bounded delay guarantee an upper bound, relative to the timestamp, of the real time at which a timestamped event is received at its destination.

Ptides binds timestamps to real time only at sensors, actuators and network interfaces. For all other components, Ptides requires that input events are processed in timestamp order, but before or after real time reaches timestamps.

Ptides relies on static causality analysis techniques to check whether an event can be processed safely. It also enables independent events to be processed out of timestamp order. For dependent events, the technique requires local clocks to be synchronized with a bounded and known precision. To this end, Ptides relies on network time synchronization, where the computing nodes on the network share a common notion of time with a known precision. Thanks to time synchronization and real-time constraints, checking if an event is safe to process or not is done by simply letting time progress. For instance, an output event from component Computation4 of Figure 1.6 could be processed after the local clock reaches a predetermined real-time value.

The execution strategy of Ptides models is based on two layers: global coordination,

and local resource scheduling. The global coordination layer determines whether an arriving event from the network can be processed immediately or if it has to wait for other potentially preceding events. As soon as the current event can be safely processed according to DE semantics, it hands the event over to local resource scheduler, which may use existing real-time scheduling algorithms, such as earliest deadline first (EDF) to prioritize the processing of all pending events [38].

1.3.3 Synchronous Systems

Synchronous programming is a design method for modeling, specifying, validating and implementing safety critical applications [39–43]. The synchronous paradigm considers systems designed as the parallel composition of strongly synchronized components. It defines a set of primitives that allows a program to be considered as instantaneously reacting to external events [44]. Synchrony is achieved by considering that the system's reaction is fast enough with respect to the environment. Responsiveness and precision are limited by step duration. The interaction with the environment is performed in global computation steps. In each step, all the system components perform some quantum of computation. The components execution constitutes a sequence of non-interruptible steps that define a logical notion of time. One of the fundamental characteristics of synchronous languages resides in the use of clocks that specify the synchronization points between components. A clock is an infinite subset of the natural numbers. A simple example of a synchronous system consists of a number of components that share a single periodic clock. The operational semantics of such a system is defined as a sequence of cycles executed at each tick of the shared clock. A cycle consists of three phases: acquisition of inputs, computation and publication of outputs. The actual duration of computations is then irrelevant. The synchrony hypothesis assumes that computations duration is negligible compared to that of communication among components. Correctness (i.e. validity of the synchrony hypothesis) is evaluated by first computing the bounds or estimates of worst-case execution time (WCET) of each computations, then performing an end-to-end delay analysis for the entire system [45].

There exist many hardware description languages such as Lustre [40], Signal [39] and Esterel [46], adopting the synchronous paradigm. These languages are used, among others, in signal processing and automatic control applications. In the following, we present (only) Lustre to illustrate synchronous languages.

Lustre

Lustre [40, 47] is a data-flow synchronous language with formal semantics developed at the VERIMAG laboratory for the past 25 years. On the top of the language, there are a number of tools, like code generator, model checker, tool for simulation of the system on design, etc., that constitute the Lustre platform. Lustre has been industrialized by Esterel Technologies in the SCADE tool. SCADE has been used by several companies in the area of aeronautics and automotive. It has been recently used for the development of the latest project of Airbus, the A380 carrier airplane.

In Lustre, a program operates based on flows of values, which are infinite sequences

$(x_0, x_1, \ldots, x_n, \ldots)$ of values at logical time instants $(0, 1 \ldots, n, \ldots)$. A Lustre program consists of a set of nodes. Each node defines a set of inputs and outputs. The outputs flows are computed by the node from input flows. An abstract syntax of a Lustre program is presented below.

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

In (resp. Out) denotes input (resp. output) of a node. Symbols N, E, x, v, b denote respectively node names, expressions, flows, constant values and Boolean values. Each flow (and expression) is associated with a logical clock. Implicitly, there exist a unique and basic clock which defines the logical time instants (or basic clock cycle) of a synchronous program. This clock has the finest grain among the clocks defined in the program. A program clock is derived from the basic clock using a flow of Boolean values, that is, it corresponds to the sequence of instants at which this flow takes the value true. In Luste, there are few elementary basic types which are Boolean, integer and one type constructor which is tuple. Complex types can be imported from a host language. Usual operators such as arithmetic, boolean, relational and conditional are available. Functions can be imported from the host language. Lustre defines operators that apply either on single clock or on multiple clocks. The combinatorial operator **op** and the unit delay **pre** operator are defined as single-clock operators. The **when** and **current** operators are defined to operate on multi clocks.

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

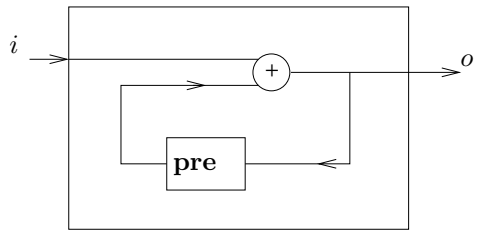


Figure 1.7: An integrator in Lustre.

Figure 1.7 shows a graphical representation of a discrete integrator. Its corresponding Lustre program is written as follows:

```
node Integrator(i: int)
      returns o: int;
```

let o = i + **pre**(o,0); **tel**

In this example, two single-clock operators are defined namely "+" and **pre**. An input flow i and an output flow o are also defined. The expression **pre**(o, 0) gives access to the value of the output flow o at the previous time instant which is initialized to zero. The output flow o is obtained by adding the input flow i to its previous value **pre**(o, 0). The equation of the integrator is the arithmetic operation "+" between a flow and the expression **pre**. Table 1.1 shows the instants of a possible execution for the first six clock cycles.

Table 1.1: Instants of a possible execution of the example of Figure1.7

basic clock	1	2	3	4	5	6
i	2	5	-1	3	1	2
pre	0	2	7	6	9	10
o	2	7	6	9	10	12

Lustre defines also multi-clocks operators which are the following.

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

Table 1.2 shows examples of using **when** and **current** operators. The basic clock defines six clock cycles. The Boolean flow b and the flow x operate on the basic clock. The flow b defines a slower clock operating at the cycles 3, 5 and 6 of the basic clock. These are the instants such that the value of b is true. The sampling operator **when** defines the flow y that operates on the slower clock b. The flow y is evaluated when b is defined, that is, at the clock cycles 3, 5 and 6. The interpolation operator **current** produces the flow z on the basic clock. For the first two instances, the value of z is undefined because y is evaluated for first time at the clock cycle 3. For any other instant, if b is true, the value of z is evaluated to y. Otherwise, it takes the value of y at the last instant when b was true. For instance, at the clock cycle 3, the slower clock b is defined and the value of z is evaluated to the current value of y, that is x_3. For clock cycle 4, the slower clock b is not defined and the value of z takes x_3, that is the value of z at the last instant b was true.

The Lustre compiler guarantees that the system under design is deterministic and conforms to the properties defined by the synchronous hypothesis [48]. It accomplishes this task thanks to static verification which is summarized in the following steps:

- *Definition checking*: every local and output variable should have one and only one definition;

Table 1.2: Examples of using **when** and **current** operators

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

- *Clock consistency*: every operator is applied to operands on suitable clocks;
- *Absence of cycles*: any cycle should use at least one **pre** operator.

For the mono-processor and mono-thread implementation, the compiler generates monolithic endless single loop C code. The body of this code implements the inputs to outputs transformations of one clock cycle. The generation of C code is done in two steps. First, it introduces variables for implementing the memory needed by the **pre** operators and then it stores the equations in order to respect data-dependencies.

1.3.4 Component-based Frameworks

Component-based design techniques are used to cope with the complexity of the systems. The idea is that complex systems can be obtained by assembling components. This is essential for the development of large-scale, evolvable systems in a timely and affordable manner. It offers flexibility in the construction phase of systems by supporting the addition, removal or modification of components without any or very little impact on other components. Components are usually characterized by abstractions that ignore implementation details and describe relevant properties to their composition, e.g. transfer functions, interfaces. The main feature of component-based design frameworks is allowing composition. This composition is used to build complex components from simpler ones. It can be formalized as an operation that takes, as input, a set of components and their integration constraints and provides, as output, the description of a new more complex component. This approach allows to cope with the complexity of systems by offering incrementality in the construction phase. In the following, we give an overview of two component-based frameworks, namely Marte and AADL, which allow building real-time systems.

Marte

Marte [49] extends the Unified Modeling Language (UML) [50] with concepts required to model real-time embedded systems. The Marte specification consists of the following main packages[51].

- Foundations: This package provides basic model constructs for non-functional properties, time and time-related concepts, allocation mechanisms and generic resources, including concurrent resources. These foundational concepts are then refined in two

other packages to respectively support modeling and analysis concerns of real-time embedded systems.

- Model-based design: This package provides high-level model constructs to depict real-time embedded features of applications, but also for enabling the description of detailed software and hardware execution platforms.

- Model-based analysis: This package provides a generic basis for quantitative analysis of sub-domains. This basis has been refined for both schedulability and performance analysis.

In Marte, the application model is defined by its repetitive structures, interaction ports and the communication between them. Tasks and their interaction are defined at data communication level [52].

The time model in Marte is heavily inspired by the Tagged Signal Model [53], synchronous languages [54], and the Globally Asynchronous and Locally Synchronous (GALS) architecture. Clocks can be defined independently and can be progressively composed with a family of possible time evolutions [55]. The time model in Marte is associated with the Clock Constraint Specification Language (CCSL)[56]. A CCSL specification consists of clock relations and clock expressions. For example, $alternatesWith$ and $isPeriodicOn$ are instances of clock relations, $delayedFor$ and $sampledOn$ are clock expressions.

AADL

AADL (Architecture Analysis and Design Language) [57, 58] is an architecture description language for model-based engineering of embedded real-time systems. It is used to design and analyze the software and hardware architectures of embedded real-time systems.

A system in AADL is described as a hierarchy of software and hardware components. There are three distinct sets of component categories:

- software components: thread, thread group, subprogram, data and process.

- hardware components: processor, memory, bus, device, virtual processor and virtual bus.

- composite system which represents composite sets of software and hardware. components.

An AADL component is defined by its type ans its implementation. The type specifies the component's visible characteristics (e.g. ports). The implementation specifies the internal structure of the component such as its subcomponents, interactions and the flows across a sequence of subcomponents.

AADL lacks a formal semantics. This lack is particularly important for real-time embedded systems as many of them are safety-critical systems which usually needs formal semantics to prove their correctness.

1.3.5 Discussion

We have presented some existing approaches for modeling and implementation of real-time systems. The time-triggered approach rely on a notion of logical execution time (LET) which corresponds to the difference between the release time and the due time of an action, defined in the program using an abstract notion of time. Time-safety is violated if an action takes more than its LET to execute. One drawback of time-triggered approach is the lack on flexibility and the restrictive design process. This approach considers fixed LET for actions, that is, time-deterministic abstract models. In addition, their models are action-deterministic, that is, only one action is enabled at a given state. Our work is more general as we consider more general models by considering more general timing constraints which allow the design of non-deterministic systems

Even-triggered approach is suitable for the design of systems where actions or events are unpredictable. The main advantage of event-triggered systems is their ability to react to asynchronous external events which are not known in advance.

Synchronous programs can be considered as a network of strongly synchronized components. Their execution is a sequence of non-interruptible steps that define a logical notion of time. In a step each component performs a quantum of computation. An implementation is correct if the worst-case execution times (WCET) for steps are less than the requested response time for the system. One of the difficulties in the synchronous paradigm is to meet the synchrony assumption, in particular when high responsiveness to the environment is required.

There exist many component-based frameworks without rigorous semantics. They use ad-hoc mechanisms for building systems from components and offer syntax level concepts only. In this case, using ad-hoc transformations, may easily lead to inconsistencies e.g. transformations may not be confluent. Generally, component-based frameworks can be divided in two categories. The first category provides high-level design and modeling, however it is still unclear how to derive correct and efficient implementation from the high-level models. In contrast, the second category provides efficient implementation, however the design process is either based on low-level primitives, or not expressive enough. In our work, we rely on BIP framework that provides rigorous semantics and allows building correct systems.

1.4 Rigorous Design Flow

In order to obtain correct distributed implementations of real-time systems, one has to consider a rigorous design flow that starts by an application software and leads to a distributed implementation. The design flow involves the use of model transformations for progressively deriving distributed implementation by making adequate design choices. Each transformation preserves the functional properties of the original model. The design flow should rely on a single semantics. That is, models are written in the same language with the same semantics.

Figure 1.8 shows a design flow that involves n model transformations. Intermediate models have different levels of abstraction. Each of these models could be dedicated to perform various tasks such as verification, performance analysis and code generation.

The last model before code generation represents exactly the software to be implemented. Code generation only implements the semantics of this model using a low-level programming

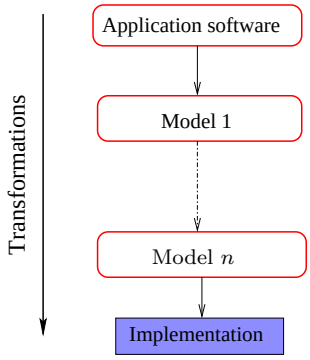


Figure 1.8: A design Flow.

language such as C++. The obtained code is correct-by-construction. This is due the fact that intermediate models are functionally equivalent to the application software.

The design flow is implemented through a set of tools automatically performing the different transformations. The designer provides a high-level model of the system from which the implementation is generated. In particular, the mechanisms to handle messages are generated automatically during the transformations. Human intervention is required to choose the parameters of the transformations.

1.5 Our Contributions

We present a method that allows deriving automatically distributed implementations from a high-level model of application software in BIP. BIP (Behavior, Interaction, Priority) is a component-based framework with formal semantics that rely on multiparty interactions for synchronizing components and priorities for scheduling between interactions. The key idea of our method is the use of transformations that preserve functional properties for progressively deriving distributed implementations.

Input models

Our input models are described in BIP. A BIP model consists of a set of components whose behavior is modeled using timed automata [59] extended with data. The components model takes into account only platform-independent timing constraints expressing user requirements. Transitions are labeled by ports and are assumed to be timeless. Components are composed using two operators, namely Interaction and Priority. The Interaction operator is parameterized by a set of interactions that are synchronizations of components's transitions. The Priority operator is parameterized by a partial order on the interactions. The global state semantics of such models is defined by a labeled transition systems LTS defining when the system can wait (i.e. when time may progress) and when it can execute the interactions. Time can progress from a global state if all components allow it from their local states. An interaction can execute from a global state (1) if all its participant components are ready to

execute that interaction, and (2) if there is no higher priority interaction enabled from that state.

Breaking Atomicity

The semantic model of BIP is based on the notion of global states. That is, the system states considered by the standard semantics are such that all the components are not executing and ready to synchronize on interactions. The execution of an interaction is defined by the atomic execution of corresponding transitions of the participating components, transforming the current global state into a new global state.

In real systems, transitions in components could not be instantaneous. They are necessarily subject to execution times when transitions correspond to computations, or also communication delays when the transitions correspond to sending messages. During the transition execution, the component state becomes unknown. In order to model these unknown states, we derive the *partial state* model from the BIP model where the components can be either in regular states when they are ready to interact, as in the global state semantics, or in busy states when they executes. Ensuring parallelism bowls down to executing interactions from states where some components are still busy. We identify conditions for which the partial state model is observationally equivalent to the original BIP model (global state model). Observational equivalence can be enforced by strengthening the premises of the operational semantics rules by a predicate called `safe` that ensure safe execution from partial states. The computation of this predicate requires to know the future state reached by components after execution. However, this cannot be computed in practice. Instead, we propose other alternative predicates $\texttt{safe}^*$ that can be operationally computed, in particular whose are based on approximations of reached states.

Building Send/Receive Models with Centralized Scheduler

Partial state models correspond to a high-level representation for parallel execution of BIP models. Those models cannot be directly implemented since, in general, distributed platforms do not offer direct support for multiparty interactions an priorities, but rather allow components to communicate using lower level primitives such as asynchronous message passing. In this thesis, we assume that (distributed) components can send messages, wait for messages or execute internal computation.

Our first solution is to transform BIP models into Send/Receive BIP models that involve only Send/Receive interactions. In Send/receive models, the distributed components exchange messages with a coordinator called scheduler (constructed as a BIP component) responsible for executing interactions and for notifying components to execute corresponding transitions. With this version of Send/Receive models, we assume that there is no delay between the decision to execute an interaction in the scheduler and the execution in participating components. Therefore, we assume that this version of Send/Receive models are implemented on platforms that provides fast communications (e.g. multi-process platforms) to meet perfect synchronization in components.

In order to evaluate the enabled interactions at a given state, the distributed components are required to send information about their current states to the scheduler component.

Regarding the scheduler, it is required to listen to the distributed components and decides the execution of interactions on-line. To maximize progress in the system, the scheduler may not have a global knowledge about system to decide the execution of an interaction. In this case, the scheduling decision is based on a partial knowledge of the system state, which corresponds precisely to an execution from a partial state in the partial state semantics. Safe executions from partial states are ensured using the predicate `safe`. Any execution decided by the scheduler should also agree with the predicate `safe` in order to produce correct executions. In practice we are using $\texttt{safe}^*$ based on approximations of reachable states.

Decentralizing the scheduler

Send/Receive models with centralized scheduler allow parallelism between components. However, parallelism between interactions is not maximal since the execution of interactions is decided by a single scheduler.

We propose a solution to obtain Send/Receive models with decentralized schedulers. Each decentralized scheduler is responsible for executing a subset of interactions. Thus, our decentralization method is parametrized by an interactions partition. Each partition class defines a set of interactions handled in a dedicated scheduler.

Adding multiple schedulers in a Send/Receive model introduces conflicts between them. A conflict between schedulers occurs when they try to execute interactions involving the same component, which may lead to violate the semantics of the centralized model.

We propose a solution to resolve dynamically conflicts between interactions. The solution is based on the incorporation of a conflict resolution protocol. Before executing a conflicting interaction, the scheduler has to ask the conflict resolution protocol. The latter grants the execution by sending an *ok* message if it does not break the semantics or rejects the execution by sending a $fail$ message otherwise. Our solution incorporates several implementations of the conflict resolution protocol. In this thesis we consider the three different implementations of conflict resolution protocol proposed by Bagrodia [60].

In this version of decentralized schedulers, we assume again that communication latencies are negligible so that there is no delay between the decision to execute an interaction in a scheduler and its execution in the participating components. We explain below how to handle non negligible communication latencies.

Taking Decision Earlier

The assumption of instantaneous communication restricts the applicability of the approach to platforms having communication means that are fast enough so that communication delays can be neglected.

We propose a method to obtain Send/Receive models that execute correctly even in presence of slow communications. To cope with communication delays, instead of notifying components at the very last moment, we propose a method where schedulers plan interactions execution and notify components in advance. With this approach, schedulers choose as soon as possible a date for executing a selected interaction and send it immediately to the participating components. Once received, the components wait for this date and then execute. The scheduling decisions are taken according to a scheduling policy which is considered as a

parameter of the approach. Any scheduling policy meeting conditions for correct execution can be considered here. The study of scheduling policies is beyond the scope of the present thesis.

In order to ensure the correct scheduling of an interaction based on this approach, we show that the schedulers need to observe additional components besides the components participating to the interactions. As in the previous version, conflicts between schedulers are referred to the conflict resolution protocol. However, with this approach, it may happen that the conflict resolution protocol confirms (through an *ok* message) the execution of an interaction while the scheduler cannot find a valid date to execute it. Such situation may introduce deadlocks in the system. To avoid this, we integrate a cancel execution mechanism between the schedulers and the conflict resolution protocol to cancel the execution of interactions. To do so, when such situation occurs, the schedulers send cancel requests to the conflict resolution protocol, which are acknowleged.

In order to correctly schedule an interaction, the scheduler has to know the state of observed components, and therefore to receive messages from them. We propose an optimization method that aims at minimizing the number of observed components when possible, which minimizes the number of messages exchange for executing an interactions. In fact, we define for each interaction a predicate on global states that characterizes components that could not be observed by the interaction. The satisfiability of this predicate is checked using static analysis techniques such as the technique presented in [61].

Implementation

Send/Receive BIP models with a centralized scheduler corresponds to an implementation of partial state models. In practice, this model is never built, but is used for sake of clarity. Instead, we developed a multi-thread real-time scheduler that implements interactions and priorities of partial state models. A code generator has been also developed to generate C++ code for atomic components that execute each one in separate thread that communicate with the multi-thred real-time scheduler. The communication between the multi-thread real-time scheduler and the atomic components is ensured by message passing through FIFO channels.

In order to generate distributed implementations for Send/Receive BIP models with multiple schedulers, we developed a code generator that takes a Send/Receive model as input and outputs a distributed implementation. We generate C++ code for each component in the Send/Receive BIP model where Send/Receive interactions are implemented by TCP sockets. The generated code scheme is the same for each component regardless its role (Send/Receive atomic component, scheduler, conflict resolution protocol component).

1.6 Organization of the Thesis

This thesis is structured as follows:

- In Chapter 2 we provide the abstract and concrete models of the BIP framework. The BIP framework provide a single semantics used consistently throughout this thesis.

- In Chapter 3, we preset partial state models that correspond to a high-level representation for parallel execution of BIP models. We identify conditions for which the partial state model is observationally equivalent to the global state model.
- In Chapter 4, we present our first solution for implementing multiparty interactions and priorities. The solution relies on a centralized scheduler for executing all interactions.
- In Chapter 5, we propose a solution for decentralizing the centralized scheduler. The solution consists in splitting the centralized scheduler into several distributed schedulers, each one handling a subset of interactions. The obtained models are augmented by a conflict resolution protocol in order to resolve conflicts between interactions. Distributed models of Chapter 4 and Chapter 5 are assumed to be implemented on platforms providing fast communications.
- In Chapter 6, we propose a solution to build distributed models that execute correctly even if communications are not instantaneous.
- In Chapter 7, we present an overview of the tools involved in the BIP framework. We focus on the tools related to this thesis. More precisely, we present the multi-threaded real-time scheduler that implements directly partial state models. Also, we present the method for generating distributed code from Send/Receive BIP models.
- In Chapter 8, we evaluate the performance of our tools. Finally, we conclude and outline some future work in Chapter 9.

2

High-Level BIP Models

Component-based design is a paradigm that allows construction of complex systems by assembling components. Components models are described by abstract platform-independent models where implementation details are completely ignored. Components are composed in order to build complex components.

BIP [62–64] is a framework for the construction of complex, heterogeneous embedded applications. BIP is highly expressive [65], component-based framework with formal semantics. A BIP model is described using the superposition of three layers: Behavior, Interaction, Priorities (see Figure 2.1).

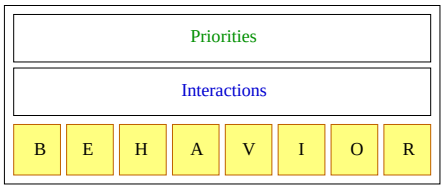


Figure 2.1: BIP model structure.

BIP allows the construction of complex, hierarchically structured models from atomic components characterized by their behavior and their interfaces (ports). An atomic component behavior is described using a transition system whose transitions are labeled by ports. Atomic components are composed using glue operators consisting of *Interactions* and *Priorities*. Interactions are used to specify multiparty synchronizations between atomic components. Priorities are used to filter amongst possible interactions and to steer system evolution so as to meet performance requirements ,e.g. to express scheduling policies. Priorities define partial orders between interactions that can change dynamically.

This chapter is structured as follows. The abstract model of BIP is described in Section 2.1. It gives an abstract formalization of the layers Behavior, Interaction, Priority. Section 2.2 describes the concrete model of BIP. In this model, atomic components are described using Perti Nets extended with data and clocks.

2.1 Abstract Models of BIP

We provide a formalization of BIP framework focusing on each layer of BIP models. In this section, we introduce the model corresponding to each layer.

2.1.1 Preliminary Definitions

We use *timed automata* [59, 66, 67] to model the behavior of real-time systems. A timed automaton is basically an automaton equipped with clocks. Its transitions are annotated with timing constraints and its states are annotated with time progress conditions. In the following, we give definitions of each of these elements.

Automaton

Definition 1 (Automaton)**.** *An automaton A is defined by a tuple (L, P, T), where:*

- *L is a finite set of control locations.*
- *P is a finite set of ports.*
- *$T \subseteq L \times P \times L$ is a set of finite transitions.*

We say that a port in *enabled* at location ℓ if there exists a transition labeled by p outgoing from location ℓ. Throughout this thesis, we consider deterministic automata which means that at a given location ℓ there is at most one outgoing transition from ℓ labeled by p.

Clocks

We consider discrete-time models, that is, time is represented using the set of non-negative integers denoted by $\mathbb{Z}_{\geq 0}$. We assume that time progress is measured by *clocks*. Clocks are non-negative integer variables increasing synchronously. A clock can be reset (i.e. set to 0) independently of other clocks. That is, clocks keep track of the time elapsed since their last reset.

We denote by $\mathcal{T}(\mathsf{C})$ the set of all possible values of clocks in C. As we consider discrete-time models, $\mathcal{T}(\mathsf{C})$ corresponds to $\mathbb{Z}^n_{\geq 0}$, where n is the number of clocks in C .

Definition 2 (clocks valuation)**.** *Given a set of clocks C, a* clocks valuation $t : \mathsf{C} \to \mathbb{Z}_{\geq 0}$ *is a function associating with each clock c its value $t(c)$. Given a subset of clocks $\mathsf{C}' \subseteq \mathsf{C}$ and a clock value $l \in \mathbb{Z}_{\geq 0}$, we denote by $t[\mathsf{C}' \mapsto l]$ the clocks valuation that coincides with t for all clocks $c \in \mathsf{C} \setminus \mathsf{C}'$, and that associates l with all clocks $c \in \mathsf{C}'$. It is defined by:*

$$t[\mathsf{C}' \mapsto l](c) = \begin{cases} l \text{ if } c \in \mathsf{C}' \\ t(c) \text{ otherwise.} \end{cases}$$

Timing Constraints

Timing constraints are used to specify when actions are enabled. Given a set of clocks C, we consider simple constraints on clocks which are atomic formulas of the form $c \sim ct$, where $c \in \mathsf{C}$ is a clock , $ct \in \mathbb{Z}_{\geq 0}$ is a constant, and $\sim$ is a comparison operator such that $\sim \in \{\leq, =, \geq\}$. They are used to build general timing constraints defined by the following grammar:

$$\mathsf{tc} := \texttt{true} \mid \texttt{false} \mid c \sim ct \mid \mathsf{tc} \wedge \mathsf{tc} \mid \mathsf{tc} \vee \mathsf{tc} \mid \neg \mathsf{tc}.$$

We simplify $\neg(c \leq ct)$ into $c \geq ct + 1$, and $\neg(c \geq ct)$ into $c \leq ct - 1$. This allows putting any timing constraint tc into the following disjunctive form:

$$\mathsf{tc} = \bigvee_{k=1}^{m} \mathsf{tc}_k$$

such that expressions tc_k are conjunctions of simple constraints and for each k_1 and k_2 $\mathsf{tc}_{k_1} \wedge \mathsf{tc}_{k_2} = \texttt{false}$. Any conjunction of simple constraints tc_k can be put into an interval conjunction:

$$\mathsf{tc}_k = \bigwedge_{c \in \mathsf{C}} l_c^k \leq c \leq u_c^k$$

such that for all $c \in \mathsf{C}$, $l_c^k \in \mathbb{Z}_{\geq 0}$ and $u_c^k \in \mathbb{Z}_{\geq 0} \cup \{+\infty\}$. Thus, any timing constraint tc can be put into the following form:

$$\mathsf{tc} = \bigvee_{k=1}^{m} \bigwedge_{c \in \mathsf{C}} l_c^k \leq c \leq u_c^k \tag{2.1}$$

We denote by $\mathsf{TC}(\mathsf{C})$ the set of timing constraints over a set of clocks C. The evaluation of a timing constraint tc for a valuation t is the Boolean value $\mathsf{tc}(t)$ obtained by replacing each clock c by its value $t(c)$.

Time Progress Conditions

Time progress conditions are used to specify whether time can progress at a given state of the system. They are a special case of timing constraints where $\sim$ is restricted to $\{\leq\}$ and operators $\neg$ and $\vee$ are disallowed. Formally, time progress conditions are defined by the following grammar:

$$\mathsf{tpc} := \texttt{true} \mid \texttt{false} \mid c \leq ct \mid \mathsf{tpc} \wedge \mathsf{tpc}.$$

where $c \in \mathsf{C}$, $ct \in \mathbb{Z}_{\geq 0}$. Note that any time progress condition tpc can be put into a conjunctive form:

$$\mathsf{tpc} = \bigwedge_{c \in \mathsf{C}} c \leq u_c \tag{2.2}$$

such that for all $c \in \mathsf{C}$, $u_c \in \mathbb{Z}_{\geq 0} \cup \{+\infty\}$. We denote by $\mathsf{TPC}(\mathsf{C})$ the set of time progress conditions over a set of clocks C.

2.1.2 Modeling Behavior

An atomic component is the most basic entity of BIP models that represents Behavior. A formal definition for an atomic BIP component is given below:

Definition 3. *An atomic component B is a timed automaton represented by the tuple (L, P, T, C,tpc), where:*

- *L is a finite set of locations,*
- *P is a finite set of communication ports,*
- *C is a finite set of clocks,*
- *$T \subseteq L \times (P \times \mathsf{TC}(\mathsf{C}) \times 2^{\mathsf{C}}) \times L$ is a finite set of labeled transitions. A transition τ is a tuple $(\ell, p, \mathsf{tc}, r, \ell')$ where p is a communication port, tc is a timing constraint over C and r is a subset of clocks that are reset by the transition τ.*
- *tpc is a time progress condition function assigning to each location $\ell \in L$ a time progress condition $\mathsf{tpc}_\ell \in \mathsf{TPC}(\mathsf{C})$, that is, tpc: $L \longrightarrow \mathsf{TPC}(\mathsf{C})$.*

The semantics of an atomic component B is a Timed Transition System (TTS) in which there are two types of steps: discrete steps and delay steps. A discrete step corresponds to firing a labeled transition of B. A delay step corresponds to letting time progress on a given state. An execution sequence of B is a sequence of such transitions with an alternance between discrete and delay steps. The definition of the semantics of B is given below.

Definition 4. *The semantics of an atomic component $B = (L, P, T, \mathsf{C}, \mathsf{tpc})$ is a transition system $S_B = (Q, \Sigma, \rightarrow)$ where:*

- *$Q = L \times \mathcal{T}(\mathsf{C})$ is the set of states,*
- *$\Sigma = P \cup \mathbb{Z}_{\geq 0}$ is the set of labels: ports or time values,*
- *$\rightarrow$ is the set of labeled transitions defined as follows:*
  - Discrete seps. *Let (ℓ, t) and (ℓ', t') be two state and $p \in P$. We have $(\ell, t) \xrightarrow{p} (\ell', t')$ where $t' = t[r \mapsto 0]$ if transition $\tau = (\ell, p, \mathsf{tc}, r, \ell')$ is in T and $\mathsf{tc}(t)$ evaluates to $\mathtt{true}$. Notice that the execution of a discrete step is timeless.*
  - Delay steps. *Let (ℓ, t) be a state and $\delta \in \mathbb{Z}_{\geq 0}$ be a delay. We have $(\ell, t) \xrightarrow{\delta} (\ell, t+\delta)$ if $\mathsf{tpc}_\ell(t + \delta)$ evaluates to $\mathtt{true}$, where $t + \delta$ is the usual notation for the valuation defined by $(t + \delta)(c) = t(c) + \delta$ for any $c \in \mathsf{C}$.*

A transition $\tau = (\ell, p, \mathsf{tc}, r, \ell')$ can be fired from state (ℓ, t) if its corresponding timing constraint tc evaluates to $\mathtt{true}$ with respect to the clocks valuation t, i.e. $\mathsf{tc}(t) = \mathtt{true}$.

Time can progress by δ from state (ℓ, t) only if it is allowed by the time progress condition of ℓ, i.e. tpc_ℓ evaluates to $\mathtt{true}$ for the valuation $t + \delta$.

The definition of an execution sequence of B is as follows:

Definition 5. *A finite (resp. infinite) execution sequence of* $B = (L, P, T, \mathsf{C}, \mathsf{tpc})$ *from an initial state* (ℓ_0, t_0) *is a sequence that alterns discrete and time steps:*

$$(\ell_i, t_i) \xrightarrow{\sigma_i} (\ell_{i+1}, t_{i+1})$$

where $\sigma_i \in P \cup \mathbb{Z}_{\geq 0}$ *and* $i \in \{0, 1, \ldots, n\}$.

Remark. The semantics of timed automata presented here is slightly different from the one found in [59], as we consider time progress conditions instead of invariants. In contrast to invariant, an atomic component B may reach a state (ℓ, t) violating the corresponding time progress condition tpc_ℓ. In this case B cannot wait and is forced to execute a transition from (ℓ, t). In this thesis, we consider systems that cannot reach states violating time progress conditions. A state (ℓ, t) from which B can neither execute a transition nor wait is a *timelock* [68]. In this thesis, we also consider systems that cannot reach timelocks.

Example 1. *Figure 2.2 depicts a graphical representation of an atomic component setter* $= (L, P, T, \mathsf{C}, \mathsf{tpc})$, *where* $L = \{\ell^1, \ell^2\}$, $P = \{sync, set\}$, $\mathsf{C} = \{c\}$, $T = \{$ τ_1=(ℓ^1, *sync,* `true`, $\{c\}$, ℓ^2), τ_2=(ℓ^2, *set,* $c \geq l$, $\emptyset$, ℓ^1)$\}$ *where* $l \in \mathbb{Z}_{\geq 0}$, *and* tpc *is defined as follows:*

- $\mathsf{tpc}(\ell^1) =$ `true`.
- $\mathsf{tpc}(\ell^2) = c \leq u$, $u \in \mathbb{Z}_{\geq 0}$ *such that* $l \leq u$.

Notice that when a time progress condition (resp. timing constraint) is not shown on the graphical representation of a location (resp. transition), we consider `true` *as default value.*

Consider an execution sequence from state $(\ell^1, 0)$*:*

- *Since the time progress condition of location* ℓ^1 *is* `true`, B *can wait infinitely at this location. Thus, any delay transition* $\delta_1 \in \mathbb{Z}_{\geq 0}$ *is possible from* $(\ell^1, 0)$ *,i.e.* $(\ell^1, 0) \xrightarrow{\delta_1} (\ell^1, \delta_1)$. *Since the only transition issued from* ℓ^1 *is* τ_1 *with label port sync and with timing constraint* `true`, *the discrete step sync is possible from state* (ℓ^1, δ_1), *i.e.* $(\ell^1, \delta_1) \xrightarrow{sync} (\ell^2, 0)$. *This discrete step is always possible from* (ℓ^1, δ_1), *and it is the only one possible from this state. However, any delay step is also possible from* (ℓ^1, δ_1).

- *From state* $(\ell^2, 0)$, *time cannot progress more than* u *time units due to the time progress condition of location* ℓ^2. *Thus, any delay step* $\delta_2 \leq u$ *is possible from* $(\ell^2, 0)$. *Since the only transition issued from* ℓ^2 *is* τ_2 *with labeled port set and a timing constraint* $c \geq l$, *the discrete transition set is possible if* δ_2 *satisfies* $l \leq \delta_2$.

To summarize the execution sequences of the abstract behavior B *from the initial state* $(\ell^1, 0)$ *can be written in following form:*

$$(\ell^1, 0) \xrightarrow{\delta_1} (\ell^1, \delta_1) \xrightarrow{sync} (\ell^2, 0) \xrightarrow{\delta_2} (\ell^2, \delta_2) \xrightarrow{set} (\ell^1, \delta_2)$$

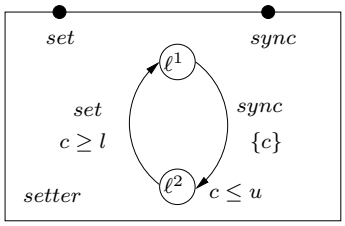


Figure 2.2: An example of Abstract Behavior.

2.1.3 Modeling Interactions

We compose a set of n components behaviors $\{B_i = (L_i, P_i, T_i, \mathsf{C}_i, \mathsf{tpc}_i)\}_{i=1}^n$ by using interactions. We assume that their sets of ports and clocks are disjoint, i.e. for all $i \neq j$ we have $P_i \cap P_j = \emptyset$ and $\mathsf{C}_i \cap \mathsf{C}_j = \emptyset$. We define the set $P = \bigcup_{i=1}^n P_i$ of all ports in the composition. Interactions are explicitly defined as subsets of ports.

Definition 6. *An interaction a is a non-empty subset $a \subseteq P$ of ports, such that $\forall i \in \{1, \ldots, n\}$, $|a \cap P_i| \leq 1$. We often denote $a = \{p_i\}_{i \in I}$, where $I \subseteq \{1, \ldots, n\}$ contains the indexes of components participating in a and p_i is the only port in $P_i \cap a$.*

We denote by $part(a)$ the set of components that have ports participating in a. The set $part(a)$ is formally defined as:

$$part(a) = \{B_i \mid P_i \cap a \neq \emptyset\}$$

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

2.1.4 Modeling Priorities

In a behavior, more than one interaction can be enabled at the same time, introducing a degree of non-determinism. This can be restricted with priorities by filtering the possible interactions enabled at a given state. The formal definition for a priority is given below.

Definition 7. *A priority is defined by a relation $\pi \subseteq \gamma \times \gamma$. where γ is a set of interactions. We write $a \pi a'$ for $(a, a') \in \pi$ to express the fact that a has weaker priority than a'.*

Note that Definition7 defines static priorities. This definition could be extended to dynamic priorities that depend on the system state as presented in [63]. In this thesis, we focus only on static priorities.

2.1.5 Composition of Atomic Components

Given a set of component $\{B_1, \ldots, B_n\}$, an interaction model γ and a priority model π, a composite component denoted by $GL(B_1, \ldots B_n)$ is obtained by applying a glue GL to components $B_1, \ldots, B_n$ where GL is limited to interactions (i.e. $GL = \gamma$), or can correspond to interactions subject to priorities (i.e. $GL = \gamma\pi$). For each of them we define the semantics below.

Definition 8. *The semantics of the composite component $\gamma(B_n, \ldots, B_n)$, where B_i=(L_i,P_i, T_i, C_i, tpc_i) are atomic components with semantics $S_{B_i} = (Q_i, \Sigma_i, \longrightarrow_i)$, is the transition system $S_\gamma = (Q, \Sigma, \longrightarrow_\gamma)$ where:*

- *$Q = L \times \mathcal{T}(\mathsf{C})$, where $L = L_1 \times \ldots \times L_n$ is the set of global locations and $\mathsf{C} = \bigcup_{i=1}^{n} \mathsf{C}_i$ is the global set of clocks. We write a global location as $\ell = (\ell_1, \ldots, \ell_n)$ where $\ell_i \in L_i$ for all i, and a global clocks valuation as $t = (t_1, \ldots, t_n)$ where t_i is a valuation of the clocks C_i for all i.*
- *$\Sigma = \gamma \cup \mathbb{Z}_{\geq 0}$.*
- *$\longrightarrow_\gamma$ is the set of labeled transitions defined by the following rules:*
  - Discrete steps:

$$\textsc{Interaction}\ \frac{a = \{p_i\}_{i \in I} \in \gamma \qquad \forall i \in I \quad (\ell_i, t_i) \xrightarrow{p_i}_i (\ell'_i, t'_i) \qquad \forall i \notin I \quad (\ell'_i, t'_i) = (\ell_i, t_i)}{(\ell, t) \xrightarrow{a}_\gamma (\ell', t')}$$

  - Delay steps:

$$\textsc{Delay}\ \frac{\delta \in \mathbb{Z}_{\geq 0} \qquad \forall i \in \{1, \ldots, n\}\ \mathsf{tpc}_{\ell_i}(t + \delta)}{(\ell, t) \xrightarrow{\delta}_\gamma (\ell, t + \delta)}$$

In Definition 8, discrete steps correspond to the execution of interactions. An interaction $a = \{p_i\}_{i \in I} \in \gamma$ can execute from a global state (ℓ, t), where $\ell = (\ell_1, \ldots, \ell_n)$ and $t = (t_1, ..., t_n)$, if for each $i \in I$ the corresponding component B_i can execute a transition labeled by p_i from its local state (ℓ_i, t_i). That is, from state (ℓ_i, t_i) there exists a transition $\tau_i = (\ell_i, p_i, \mathsf{tc}_i, r_i, \ell'_i) \in T_i$ where the timing constraint tc_i evaluates to `true` at the clocks valuation t_i.

From global state (ℓ, t), a delay transition corresponds to letting time progress by δ at location ℓ. Time is allowed to progress by δ if it is allowed by the time progress condition tpc_{ℓ_i} of each component B_i, $i \in \{1, \ldots, n\}$.

Definition 9. *Let $S_\gamma(Q, \Sigma, \longrightarrow_\gamma)$ be the semantics of $\gamma(B_n, \ldots, B_n)$. We define the semantics of $\pi\gamma(B_n, \ldots, B_n)$ as the labeled transition system S_π=$(Q, \Sigma, \longrightarrow_\pi)$ where $\longrightarrow_\pi$ restricts discrete steps defined by $\longrightarrow_\gamma$ as follows:*

$$\textsc{Prio}\ \frac{(\ell, t) \xrightarrow{a}_\gamma (\ell', t') \qquad \forall a' \in \gamma\ a\pi a' \Rightarrow (\ell, t) \not\xrightarrow{a'}_\pi}{(\ell, t) \xrightarrow{a}_\pi (\ell', t')}$$

From a state (ℓ, t), an interaction $a \in \gamma$ can execute if (1) a is enabled at (ℓ, t) (i.e.such that $(\ell, t) \xrightarrow{a}_{\gamma} (\ell', t')$) and (2) each interaction a' that has higher priority (i.e. $a \pi a'$) is not enabled at state (ℓ, t) (i.e. $(\ell, t) \not\xrightarrow{a'}_{\gamma}$).

Example 2. *Figure 2.3 presents a composite component* $\pi\gamma(setter_1, getter_1, setter_2)$ *where:*

- $setter_1$, $getter_1$, *and* $setter_2$ *are atomic components where* $setter_1$ *and* $setter_2$ *are instances of setter atomic component shown Figure 2.2 with* $l = 10$ *and* $u = 20$ *for* $setter_1$, *and* $l = u = 5$ *for* $setter_2$,
- *interactions* $\gamma = \{a_1, a_2, a_3\}$ *are defined by* $a_1 = \{sync_1, sync_2, sync_3\}$, $a_2 = \{set_1, get_1\}$ *and* $a_3 = \{set_2, get_2\}$, *and*
- *priority* π *is such that* $a_2 \pi a_3$.

From the initial location $((\ell_1^1, \ell^1, \ell_2^1), 0)$, *it can be easily shown that the execution sequences of the system have the following form:* $((\ell_1^1, \ell^1, \ell_2^1), 0) \xrightarrow{a_1}_{\pi} ((\ell_1^2, \ell^2, \ell_2^2), 0) \xrightarrow{5}_{\pi} ((\ell_1^2, \ell^2, \ell_2^2), 5) \xrightarrow{a_3}_{\pi} ((\ell_1^2, \ell^3, \ell_2^1), 5) \xrightarrow{\delta}_{\pi} ((\ell_1^2, \ell^3, \ell_2^1), 5 + \delta) \xrightarrow{a_2}_{\pi} ((\ell_1^1, \ell^1, \ell_2^1), 5 + \delta) \xrightarrow{a_1}_{\pi} ((\ell_1^2, \ell^2, \ell_2^2), 0)$, *where* $5 \leq \delta \leq 15$. *Notice that control location err cannot be reached in* $getter_1$ *due to the application of priority* $a_2 \pi a_3$.

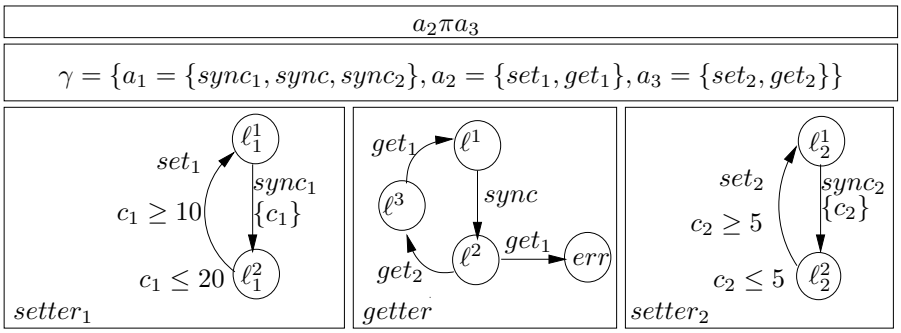


Figure 2.3: Example of abstract composition of 3 components

2.2 Concrete Models of BIP

Abstract models from the previous section focus on control. We now present how data is handled in BIP. Data allows representation of complex behavior using *Boolean guards* expressed over variables to prevent transitions and interactions. Data is modified through *update functions*. The mechanisms to handle data added on top of the abstract model constitute the concrete model of BIP.

2.2.1 Preliminary definitions

Petri Nets

In concrete models, atomic components are described using Perti Nets. The definition of Petni Nets is given below.

Definition 10. *A* Petri net *is defined by a triple $S = (L, P, T)$, where L is a finite set of* places*, P is a finite set of ports, and $T \subseteq 2^L \times P \times 2^L$ is a set of transitions. A transition τ is a triple $({}^\bullet\tau, p, \tau^\bullet)$, where ${}^\bullet\tau$ is the set of* input places *of τ and $\tau^\bullet$ is the set of* output places *of τ.*

A Petri net is often modeled as a directed bipartite graph $G = (L \cup T, E)$. Places are represented by circular vertices and transitions are represented by rectangular vertices (see Figure 2.4). The set of directed edges E is the union of the sets $\{(\ell, \tau) \in L \times T \mid \ell \in {}^\bullet\tau\}$ and $\{(\tau, \ell) \in T \times L \mid \ell \in \tau^\bullet\}$. We depict the *state* of a Petri net by marking its places with tokens [69]. We say that a place is *marked* if it contains a token. Formally, a *marking* is an application $m : L \to \mathbf{N}$ that indicates how many tokens are in each place. A transition τ is *enabled* at a state if all its input places ${}^\bullet\tau$ are marked. Formally, τ is enabled if wor each $\ell \in {}^\bullet\tau$ we have $m(\ell) > 0$. Upon the execution of τ at marking m, one token is removed in each place of τ and one token is added to each output place of τ. Formally, by executing τ at marking m, one reaches the marking m' characterized by:

$$\forall \ell \in L \quad m'(\ell) = m(\ell) - \tau^-(\ell) + \tau^+(\ell) \tag{2.3}$$

where

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

In this thesis, we focus on *1-Safe* Petri nets. Given an initial state m_0, a Petri net (L, P, T) is *1-Safe* (also commonly referred as 1-bounded or simply safe) if for any execution from m_0 output places of enabled transitions are never marked. The behavior of a 1-Safe Petri net (L, P, T) is defined as a finite labeled transition system $(2^L, P, \to)$, where 2^L is the set of states, P is the set of ports, and $\to \subseteq 2^L \times P \times 2^L$ is the set of transitions defined as follows. We have $(m, a, m') \in \to$, denoted by $m \xrightarrow{a} m'$, if there exists $\tau = ({}^\bullet\tau, a, \tau^\bullet) \in T$ such that ${}^\bullet\tau \subseteq m$ and $m' = (m \backslash {}^\bullet\tau) \cup \tau^\bullet$. In this case, we say that a is *enabled* at m. We say that the Petri net (L, P, T) is *deterministic* for initial state m_0 if for any execution from m_0 two transitions $\tau_1 \neq \tau_2$ labeled by same port p are not enabled at the state.

Example 3. *Figure 2.4 illustrates the execution of a transition on the marking in a Petri net which has five places $\{\ell_1, \ldots, \ell_5\}$ and three transitions $\{p_1, p_2, p_3\}$. The places containing a token are depicted with gray background. The marking on the right shows the resulting state after executing transition p_2 from the marking giving on the left.*

Variables

Given a variable x, the *domain* of x is the set $\mathcal{D}(x)$ of all values possibly taken by x. For example, if x is an integer variable then $\mathcal{D}(x) = \mathbb{Z}_{\geq 0}$. Given a set of variables X, a *valuation*

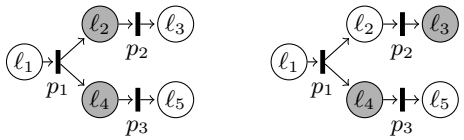


Figure 2.4: A simple Petri net

of X is a function $v : X \to \bigcup_{x \in X} \mathcal{D}(x)$ assigning a value to each variable of X, that is, such that for all variable x we have $v(x) \in \mathcal{D}(x)$. We denote by $\mathcal{V}(X)$ the set of all possible valuations of X. The restriction of $v \in \mathcal{V}(X)$ to a subset of variables $X' \subseteq X$ is the valuation $v_{|X'} \in \mathcal{V}(X')$ that coincides with v on X', that is, $v_{|X'}(x) = v(x)$ for all $x \in X'$. When it is not ambiguous, we write v also for $v_{|X'}$.

Given valuations $v \in \mathcal{V}(X)$ and $v' \in \mathcal{V}(X')$ of variables X and X' such that $X' \subseteq X$, we denote by $v[X' \leftarrow v']$ the variables valuation of X that coincides with v' for all variables of X', and with v for all variables of $X \setminus X'$. It is defined by:

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

Guard

A *guard* is a predicate on a set of variables X. Given a guard g on X and a valuation $v \in \mathcal{V}(X)$, we denote by $g(v) \in \{\texttt{false}, \texttt{true}\}$ the evaluation of g for v.

Update function

An *update function* $f : \mathcal{V}(X) \to \mathcal{V}(X)$ for variables X is used to assign new values $f(v)$ to variables in X from their current values v. It extends to any larger set of variables $X' \supseteq X$ considering that extra variables $X' \setminus X$ are unchanged, i.e., f transforms $v \in \mathcal{V}(X')$ into $v[X \leftarrow f(v)]$.

2.2.2 Atomic Components

In concrete BIP models, atomic components are Petri nets extended with a set of variables and clocks. Ports are associated with some variables and are used for communication among different components. Each transition of the Petri nets:

- is labeled by a port,
- is guarded by a predicate on variables and a timing constraint on clocks whose bounds may be arbitrary expressions[1] on variables in addition to constant values (for example, $c \geq f(v)$ is a timing constraint on clock c that depends of variable v),
- and triggers an update function that modifies a subset of variables and resets a subset of clocks.

[1]The only restriction for expressions involved in timing constraints is that they must evaluate to an integer value.

Each place of the Petri net defines a time progress condition on clocks whose bounds may also be arbitrary expressions on variables.

Definition 11. *An* atomic component B *is defined by* $B = (L,\ P,\ T,\ \mathsf{C},\ X,\ \{X_p\}_{p\in P},\ \{g_\tau\}_{\tau\in T},\ \{\mathsf{tc}_\tau\}_{\tau\in T},\ \{f_\tau\}_{\tau\in T},\ \{r_\tau\}_{\tau\in T},\ \{\mathsf{tpc}_\ell\}_{\ell\in L})$
where:

- (L, P, T) *is a* deterministic 1-Safe Petri net.
- C *is a finite set of clocks.*
- X *is a finite set of discrete variables.*
- *For each port* $p \in P$, $X_p \subseteq X$ *is the set of variables* exported *by* p (*i.e., variables visible from outside the component through port* p).
- *For each transition* $\tau \in T$, g_τ *is a guard on* X, tc_τ *is a timing constraint over* C, f_τ: $\mathcal{V}(X) \longrightarrow \mathcal{V}(X)$ *is a function that updates the set of variables* X *and* $r_\tau \subseteq \mathsf{C}$ *is a subset of clocks that are reset by the transition* τ.
- *For each place* $\ell \in L$, tpc_ℓ *is a time progress condition over* C.

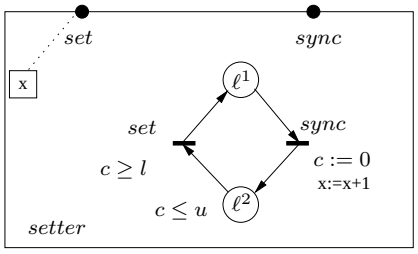


Figure 2.5: An example of atomic component

Example 4. *Figure 2.5 presents a concrete atomic component. Its 1-safe Petri net is actually an automaton namely the one shown in Figure 2.2. It has been extended with variable* x. *The variable* x *is associated to port set, that is, the variable* x *can be read and written when an interaction involving port set takes place. The update function associated to transition labeled by sync is a pseudo code that increments the value of* x *by 1.*

Definition 12. *The semantics of an atomic component* $B = (L,\ P,\ T,\ \mathsf{C},\ X,\ \{X_p\}_{p\in P},\ \{g_\tau\}_{\tau\in T},\ \{\mathsf{tc}_\tau\}_{\tau\in T},\ \{f_\tau\}_{\tau\in T},\ \{r_\tau\}_{\tau\in T},\ \{\mathsf{tpc}_\ell\}_{\ell\in L})$ *is defined as the labeled transition system* $S_B{=}(Q, \Sigma, \rightarrow)$, *where:*

- $Q = 2^L \times \mathcal{V}(X) \times \mathcal{T}(\mathsf{C})$ *is the set of states,*
- $\Sigma = P \cup \mathbb{Z}_{\geq 0}$ *is set of labels: ports or time values.*
- $\rightarrow$ *is the set of labeled transitions defined as follows.*

- Discrete steps: *Let* (m, v, t) *and* (m', v', t') *be two states and* $p \in P$ *be a port. We have* $(m, v, t) \xrightarrow{p(v'')} (m', v', t')$*, if* (1) *transition* $\tau = ({}^\bullet\tau, p, \tau^\bullet)$ *is enabled at* m *in the Petri net* (L, P, T)*,* (2) $g_\tau(v) \wedge \mathtt{tc}_\tau(t)$ *is* `true`*,* (3) $v' = f_\tau(v[X_p \leftarrow v''])$ *and* (4) $t' = t[r_\tau \mapsto 0]$*, where* r_τ *is the set of clocks reset by* τ*. In this case, we say that* p *is enabled from* (m, v, t)*.*
- Delay steps: *Let* (m, v, t) *be a state and* $\delta \in \mathbb{Z}_{\geq 0}$ *be a delay. We have* $(m, v, t) \xrightarrow{\delta} (m, v, t + \delta)$ *if* $\bigwedge_{\ell \in m} \mathtt{tpc}_\ell(t + \delta)$ *is* `true`*.*

An atomic component B can execute a transition $\tau = (m, p, m')$ from a state (m, v, t) if its Boolean guard is met by the valuation of variables v and its timing constraint is met by the valuation t. The execution of τ moves tokens of places ${}^\bullet\tau$ to $\tau^\bullet$, updates a subset of variables in X and resets clocks in r_τ. From state (m, v, t), time can progress by $\delta > 0$ if time progress conditions of places belonging to m stays `true` at $t + \delta$ i.e. $\bigwedge_{\ell \in m} \mathtt{tpc}_\ell(t + \delta) = \mathtt{true}$.

Petri nets are a well suited formalism for encoding parallel and concurrent execution. As we will see in the next chapters, we use 1-safe Petri nets in order to express schedulers introduced by our method. Any 1-safe Petri net (L, P, T) can be transformed into an equivalent automaton $(2^L, P, T)$ where 2^L is the set of locations [70]. In particular, an atomic component whose behavior is described by a Petri net could transformed into an equivalent atomic component whose behavior is described by a timed automaton. In fact, each transition $\tau = (m_1, p, m_2)$ in the atomic component with Petri net is transformed into a transition $\tau' = (\ell_1, p, \ell_2)$ in the equivalent atomic component with timed automaton where $\ell_1 = \bigotimes_{\ell \in m_1 | m_1(\ell) > 0)} \ell$ and $\ell_2 = \bigotimes_{\{\ell \in m_2 | m_2(\ell) > 0\}} \ell$. The time progress condition of locations ℓ_1 and ℓ_2 are defined by $\bigwedge_{\{\ell \in m_1 | m_1(\ell) > 0\}} \mathtt{tpc}_\ell$ and $\bigwedge_{\{\ell \in m_2 | m_2(\ell) > 0\}} \mathtt{tpc}_\ell$, respectively. Transition τ' has the same Boolean guard, timing constraint, clocks reset and update function as transition τ.

Example 5. *Figure 2.6 depicts a transformation of two Petri net transitions (a) into two automaton transitions (b). For instance, consider transition* $\tau_1 = (m_1, p_1, m_2)$ *in the Petri net, where* m_1 *and* m_2 *are defined such that:*

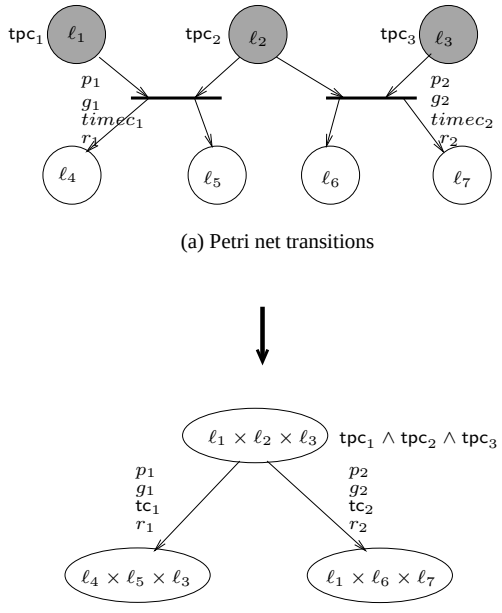
- $m_1(\ell_1) = m_1(\ell_2) = m_1(\ell_3) = 1$*,* $m_1(\ell_4) = m_1(\ell_5) = m_1(\ell_6) = 0$ *and*
- $m_1(\ell_4) = m_1(\ell_5) = m_1(\ell_3) = 1$*,* $m_1(\ell_1) = m_1(\ell_2) = m_1(\ell_6) = 0$*.*

The transformation of this transition in the automaton gives transition $\tau_1' = (\ell_1 \times \ell_2 \times \ell_3, p_1, \ell_4 \times \ell_5 \times \ell_3)$*.*

In the sequel, we consider atomic components involving simple automata instead of general Petri nets, that is, in which transitions of components have a single input place and a single output place. Petri nets will only be used for new components introduced by model transformations needed by our method and presented in Chapters 4, 5 and 6

2.2.3 Interactions and Connectors

Similarly to atomic components, interactions are also taking into account variables. An interaction consists of a set of ports, each one exporting a set of variables. The set of these



(a) Petri net transitions

(a) Equivalent automaton transitions

Figure 2.6: Transformation of Petri net transitions into automaton transitions.

variables can be accessed by the interaction. More precisely, a predicate on variables called guard, has to be true to enable the interaction. A data transfer function modifies the variables upon the execution of the interaction.

Consider a set of atomic component $\{B_1, \ldots, B_n\}$, where for each $i \in \{1, \ldots, n\}$ B_i is defined as $(L_i, P_i, T_i, \mathsf{C}_i, X_i, \{X_p\}_{p \in P_i}, \{g_\tau\}_{\tau \in T_i}, \{\mathsf{tc}_\tau\}_{\tau \in T_i}, \{r_\tau\}_{\tau \in T_i}, \{f_\tau\}_{\tau \in T_i}, \{\mathsf{tpc}_\ell\}_{\ell \in L_i})$. We require that the sets of locations, ports, clocks, and variables of these components are pairwise disjoint i.e., for any two $i \neq j$ from $\{1, \ldots, n\}$, we have $L_i \cap L_j = \emptyset$, $P_i \cap P_j = \emptyset$, $\mathsf{C}_i \cap \mathsf{C}_j = \emptyset$, and $X_i \cap X_j = \emptyset$. We denote $P = \bigcup_{i=1}^{n} P_i$ the set of all the ports in the composite component.

Definition 13. *An* interaction *a is a triple (P_a, G_a, F_a), where:*

- *P_a is a set of ports such that $\forall i \in \{1, \ldots, n\}$ $|P_a \cap P_i| \leq 1$. We denote $P_a = \{p_i\}_{i \in I}$, where $I \subseteq \{1, \ldots, n\}$ is the set of indices of participants in a and p_i is the only port in $P_a \cap P_i$,*
- *G_a is a predicate defined over the set $X_a = \bigcup_{i=1}^{n} X_{p_i}$ of variables involved in a,*
- *F_a is a data transfer function that modifies the variables X_a, that is, $F_a : \mathcal{V}(X_a) \longrightarrow \mathcal{V}(X_a)$.*

Connectors In BIP, *connectors* are used to represent sets of related interactions in a compact manner. A connector is defined over a set of ports P that forms its support and, defines a subset of interactions, which is, a subset of 2^P. A connector may export a port that is used for the construction of hierarchical connectors.

Definition 14. *A* connector γ *is a triple* (P_γ, A_γ, p)*, where:*

- P_γ *is the support set of* γ*, that is, the set of ports that* γ *synchronizes,*
- A_γ *is the set of interactions,*
- p *is the exported port by the connector* γ*.*

There are two types of ports (*trigger* or *synchron*) which define interactions of a connector. A trigger (denoted by a triangle) is an active port that can initiate an interaction without synchronizing with other ports. A synchron (denoted by a circle) is a passive port that needs synchronization with other ports.

Figure 2.7: Examples of connectors and theirs interactions in BIP.

In BIP, we distinguish between two types of models for synchronization on connectors.

- *Strong synchronization* or *Rendez-vous*: The only feasible interaction is the maximal one, i.e. it contains all the ports of γ. This corresponds to the case where all the ports of the support set are synchrons. Figure 2.7 (a) depicts an example of Rendez-vous connector. This connector defines only one interaction $\gamma = \{p_1p_2p_3\}$.
- *Weak synchronization* or *Broadcast*: All feasible interactions are those containing at least one trigger port. This port "initiates" the interaction even if all other ports are disabled. Figure 2.7 (b) depicts an example of Broadcast connector including one trigger port p_1. This connector contains the set of all interactions involving port p_1, that is, $\gamma = \{p_1, p_1p_2, p_1p_2p_3\}$.

Connectors provide mechanisms for handling data. For each connector, the data transfer function F_a of an interaction a is specified by an *up* and *down* actions. The action *up* updates local variables of the connector, from the values of variables exported by the ports. Conversely, the action *down* updates the variables exported by the ports, from the values of the connector variables. The guard G_a of the interaction a is a Boolean expression.

Example 6. *Figure 2.8 shows a rendez-vous connector. The connector involves three ports* p_1*,* p_2 *and* p_3 *exporting variables* x_1*,* x_2 *and* x_3*, respectively. The only feasible interaction is* $p_1p_2p_3$ *that takes place only if the guard* $x_1 > 0 \wedge x_2 > 0 \wedge x_13 > 0$ *evaluates to true. If the interaction is selected for execution, the action up computes the maximum of the variables associated to the ports and stores it in the connector variable* y*. The action down sets the variables associated to the ports to the maximum values that is stored in* y*.*

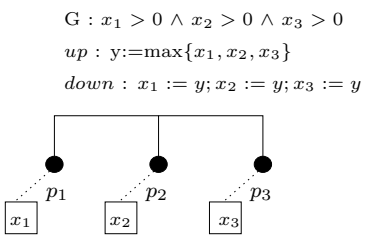


Figure 2.8: A example of connector computing the maximum of exported values.

Hierarchical connectors A *hierarchical connector* is obtained by combining simple connectors to form a structure of connectors acting as a single connector. This is achieved by including in the support (set of ports that forms the connector) of a top level connector the exported port of a low level connector. Hierarchical connectors are formalized using the algebra of connectors defined in [71]. In this thesis, we do not consider hierarchical connectors. Therefore, we do not formalize here the construction of hierarchical connectors. Interested readers may refer to [72] for more details. Instead, we give two simple examples as shown in Figure 2.9.

Figure 2.9: Examples of Hierarchical connectors and their interactions in BIP.

- The connector (a) is called *Atomic Broadcast* [71]. The bottom connector is a rendezvous synchronizing ports p_2 and p_3 and exporting the port u. This connector defines only interaction p_1p_2. The top connector is a broadcast synchronizing ports p_1 and u. Thus, it allows interactions p_1 and p_1u. Therefore, the feasible interactions defined by this connector are given by $\gamma = \{p_1, p_1p_2p_3\}$.

- The connector (b) is called *Causality Chain* [71]. The bottom connector is a broadcast synchronizing ports p_1 and p_2 and exporting the port u. This connector allows interactions p_2 and p_2p_3. The top connector is similar to the bottom connector, allowing interactions: p_1 and p_1u. Therefore, the feasible interactions defined by this connector are given by $\gamma = \{p_1, p_1p_2, p_1p_2p_3\}$.

Notice that hierarchical connectors can be transformed into "flat" connectors as defined in Definition 14 using the transformation method described in [73].

2.2.4 Priority rules

Priorities are defined in order to reduce non-determinism in the system, that is, they are used to filter interactions among enabled ones. As already mentioned, in this thesis, we use static priorities that does not depend on system states. Therefore, Definition 7 remains unchanged for concrete BIP models.

2.2.5 Composition of components

We compose the atomic components using interactions and priorities. Given a set of components $\{B_1, \ldots, B_n\}$ and a set of interactions γ, we denote by $\gamma(B_1, \ldots, B_n)$ the composite component resulting from the composition of these components using interactions γ. Similarly, given a priority π, we denote by $\pi\gamma(B_1, \ldots, B_n)$ the behavior obtained by applying both interactions γ and priority π. Below, we define the semantics of the two models.

Definition 15. *The semantics of the composite component $\gamma(B_n, \ldots, B_n)$, where $B_i = (L_i, P_i, T_i, \mathsf{C}_i, X_i, \{X_p\}_{p\in P_i}, \{g_\tau\}_{\tau\in T_i}, \{\mathsf{tc}_\tau\}_{\tau\in T_i}, \{r_\tau\}_{\tau\in T_i}, \{f_\tau\}_{\tau\in T_i}, \{\mathsf{tpc}_\ell\}_{\ell\in L_i})$ are atomic components, is the transition system $S_\gamma = (Q, \Sigma, \longrightarrow_\gamma)$ where:*

- *$Q = L \times \mathcal{V}(X) \times \mathcal{T}(\mathsf{C})$ is the set of global states, where $L = L_1 \times \ldots \times L_n$, $X = \bigcup_{i=1}^n X_i$ and $\mathsf{C} = \bigcup_{i=1}^n \mathsf{C}_i$. We write a state $q = (\ell, v, t) \in Q$ where $\ell = (\ell_1, \ldots, \ell_n) \in L$ is a global location, $v = (v_1, \ldots, v_n) \in \mathcal{V}(X)$ is a global variables valuation and $t = (t_1, \ldots, t_n) \in \mathcal{T}(C)$ is a global clocks valuation.*
- *$\Sigma = \gamma \cup \mathbb{Z}_{\geq 0}$ is the set of labels,*
- *$\longrightarrow_\gamma$ is the set of labeled transitions defined by the following rule:*
  - Discrete steps:

$$\textsc{Interaction}\ \frac{\begin{array}{c} a = (\{p_i\}_{i\in I}, G_a, F_a) \in \gamma \qquad G_a(\{v_i\}_{i\in I}) \qquad \{v_i''\}_{i\in I} = F_a(\{v_i\}_{i\in I}) \\ \forall i \in I\ (\ell_i, v_i, t_i) \xrightarrow{p_i(v''_i)}_i (\ell_i', v_i', t_i') \qquad \forall i \notin I\ (\ell_i', v_i', t_i') = (\ell_i, v_i, t_i) \end{array}}{(\ell, v, t) \xrightarrow{a}_\gamma (\ell', v', t')}$$

  - Delay steps:

$$\textsc{Delay}\ \frac{\delta \in \mathbb{Z}_{\geq 0} \qquad \forall i \in \{1, \ldots, n\}\ \mathsf{tpc}_{\ell_i}\delta(t_i + \delta)}{(\ell, v, t) \xrightarrow{\delta}_\gamma (\ell, v, t + \delta)}$$

An interaction $a = \{\{p_i\}_{i\in I}, G_a, F_a\}$ can execute from state (ℓ, v, t) if (1) for each port p_i, the corresponding atomic component B_i can execute a transition labeled by p_i from the projection (ℓ_i, v_i, t_i) of (ℓ, v, t) on B_i, and (2) the guard G_a of the interaction evaluates to `true` on the variables exported by the ports participating in interaction a. Execution of interaction a triggers the function F_a which modifies the variables of the components exported by ports p_i. The new values obtained, encoded in the valuations v_i'' are then processed by the components' transitions. The clocks valuation t' is obtained when components reset clocks

according to the set of clocks reset associated to their transitions. The states of components that do not participate in the interaction remain unchanged.

A delay transition δ can execute from a state (ℓ, v, t) if it is allowed by the time progress condition tpc_{ℓ_i} of each component B_i, $i \in \{1, \ldots, n\}$.

Definition 16. *Let $S_\gamma = (Q, \Sigma, \longrightarrow_\gamma)$ be the semantics of $\gamma(B_n, \ldots, B_n)$. The semantics of $\pi\gamma(B_n, \ldots, B_n)$, is the transition system $S_\pi = (Q, \Sigma, \longrightarrow_\pi)$ where $\longrightarrow_\pi$ restricts discrete transitions of $\longrightarrow_\gamma$ as follows:*

$$\textsc{Prio}\ \frac{(\ell, v, t) \xrightarrow{a}_\gamma (\ell', v', t') \qquad \forall a' \in \gamma\ a \pi a' \Rightarrow (\ell, v, t) \not\xrightarrow{a'}_\gamma}{(\ell, v, t) \xrightarrow{a}_\pi (\ell', v', t')}$$

The application of priority π filters out the interactions which are not maximal. An interaction $a = (\{p_i\}_{i \in I}, G_a, F_a)$ executes from state $q = (\ell, v, t)$ if any other interaction a' having higher priority is disabled from that state.

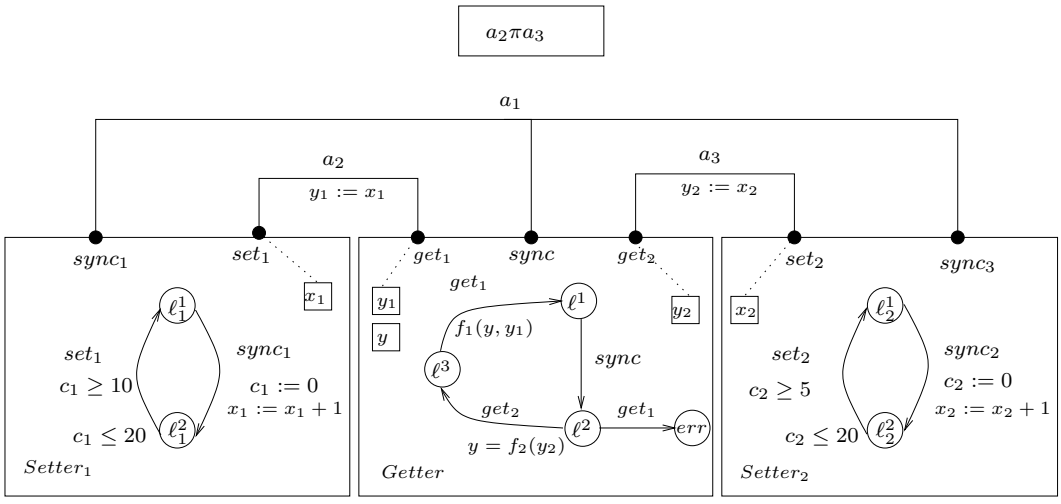


Figure 2.10: A composite component.

Example 7. *Figure 2.10 depicts a composition of three components $setter_1$, $setter_2$ and $getter_1$. It corresponds to an extension of the example from Figure 2.3 with variables. Components $setter_i$ have variables x_i associated to ports set_i. Whenever interaction a_1 occurs, transitions $sync_i$ of components $setter_i$ increments variable x_i by one. Interactions a_2 and a_3 transmits the value of variables x_i of components $setter_i$ to component $getter_1$ that stores them in variables y_i. Component $getter_1$ computes $f_2(y_2)$ whenever interaction a_3 occurs. The result is stored in variable y. Then, it computes $f_1(y, y_1)$ whenever interaction a_2 occurs. The timing constraints in $setter_1$ and $setter_2$ imposes that a_2 executes after a_3. Thus a correct execution of the model does not lead to location err in component $getter_1$.*

2.3 Conclusion

In this chapter, we presented the component framework BIP which is a formalism for modeling heterogeneous component-based system. The description of systems is allowed through the composition of atomic components characterized by their behavior and their interfaces (communication ports). BIP relies on two operators for the composition: Interaction and Priority. Interactions are used to specify multiparty synchronizations between components. Priorities are used to restrict the system behavior in order to reduce non-determinism.

The BIP semantics presented in this chapter assumes atomic execution of interactions. This semantics is called global state semantics. It assumes that the system moves from a global state to another global state. Implementing such semantics provides sequential execution of the system.

In the next chapter, we present partial state models which represent high-level representation for parallel execution of BIP models. We study conditions for partial state models to be observationally equivalent to original BIP models.

3 Partial State Models

The semantic model of BIP presented in the previous chapter is based on the notion of global states. That is, at each state of the system, a complete information about components states is available and the execution of transitions is atomic. These models are called *global state models*. In this chapter, we present *partial state models* in which the assumption of atomic execution of transitions is discarded. In fact, in real systems, transitions in components could not be instantaneous. They are necessarily subject to execution times when transitions correspond to computations, or also communication delays when the transitions correspond to sending messages. During the transition execution, the component state becomes unknown. In order to model these unknown states, we derive the partial state model from the atomic component where we distinguish between global states where the component is ready to interact and partial states where the component is busy by the execution.

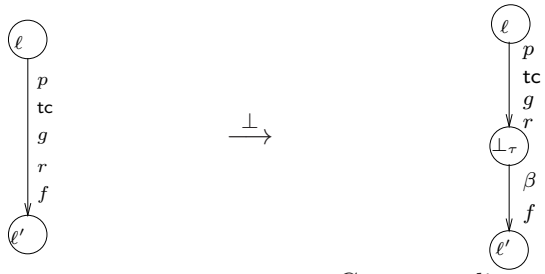
The main objective of this chapter is to identify conditions for which the partial state model is observationally equivalent to the global state model. Preservation can be achieved by strengthening the premises of the operational semantics rules by a predicate called `safe` that ensure safe execution from partial states. This predicate depends on the future state reached by components after execution.

3.1 Partial State Semantics

3.1.1 Atomic Components.

In order to obtain an atomic component with partial states $B^{\perp}$ from the corresponding atomic component B, we follow the approach presented in [74]. We split each transition $\tau = (\ell, p, \ell')$ of B into two transitions: (1) a visible transition labeled by p indicating the beginning of the execution of transition τ (2) and an internal transition labeled by β indicating the end of the execution of transition τ. These transitions are separated by a *busy* location added in

the atomic component, modeling the fact that the components is in unknown state when it executes (see Figure 3.1).



Transition $\tau = (\ell, p, \ell')$ in the atomic component.

Corresponding sequences of transitions in the partial state model.

Figure 3.1: Transformation of transitions of the atomic component.

Definition 17. *Given an atomic component* $B = (L,\ P,\ T,\ \mathsf{C},\ X, \{X_p\}_{p\in P},\ \{g_\tau\}_{\tau\in T},\ \{\mathsf{tc}_\tau\}_{\tau\in T},\ \{r_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T},\ \{\mathsf{tpc}_\ell\}_{\ell\in L})$. *the corresponding atomic component with partial state is defined by* $B^\perp = (L \cup L^\perp,\ P \cup \{\beta\},\ T^\perp,\ \mathsf{C},\ X,\ \{X_p\}_{p\in P},\ \{g_\tau\}_{\tau\in T^\perp},\ \{\mathsf{tc}_\tau\}_{\tau\in T^\perp}, \{r_\tau\}_{\tau\in T^\perp}, \{f_\tau\}_{\tau\in T^\perp},\ \{\mathsf{tpc}_\ell\}_{\ell\in L\cup L^\perp})$ *where:*

- $L^\perp = \{\perp_\tau \mid \tau \in T\}$ *is the set of busy locations.*
- β *is an additional port.*
- $T^\perp \subseteq (L \cup L^\perp) \times (P \cup \{\beta\}) \times (L \cup L^\perp)$ *where if* $\tau = (\ell, p, \ell')$ *is a transition in* T*, then the corresponding transitions in* $T^\perp$ *are* $\tau_p = (\ell, p, \perp_\tau)$ *and* $\tau_\beta = (\perp_\tau, \beta, \ell')$*. Transition* τ_p *has a Boolean guard* $g_{\tau_p} = g_\tau$*, a timing constraint* $\mathsf{tc}_{\tau_p} = \mathsf{tc}_\tau$*, reset clocks* $r_{\tau_p} = r_\tau$*, and no update function. Transition* τ_β *has an update function* $f_{\tau_p} = f_\tau$*.*
- *The time progress condition on locations* $L^\perp$ *are* `true`*.*

Note that $B^\perp$ is an atomic component whose semantics is given by Definition 12.

The execution of a transition $\tau = (\ell, p, \ell')$ of B corresponds to executing two transitions in the corresponding partial state model $B^\perp$. First, a visible (observable) transition $\tau_p = (\ell, p, \perp_\tau)$ marks the beginning of the execution of τ. This transition defines the same timing constraint, Boolean guard and clocks reset defined in τ. Second, an internal (unobservable) transition $\tau_\beta = (\perp_\tau, \beta, \ell')$ marks the end of the execution of τ. It defines the same update function of τ.

The time progress condition of the busy location $\perp_\tau$ is `true`. The latter models the fact that transition τ may take arbitrary time to execute.

In the partial state model $B^\perp$, we assume arbitrary execution times for transitions, ranging from 0 to $+\infty$, which is modeled by the time progress condition `true` for busy states. Notice that $B^\perp$ can be further constrained if bounds of the execution times of transitions are known. For instance, if we know an estimate of the worst-case execution time of a transition τ, denoted by $WCET(\tau)$, the associated time progress condition of location $\perp_\tau$ is replaced by

$x_\tau \leq WCET(\tau)$ instead of `true`, where x_τ is a clock that is reset whenever τ is started. This allows us to statically check the correctness of the application running on the platform, but this is beyond the scope of this thesis.

In a partial state model $B^\perp$, the execution of a transition $\tau = (\ell, p, \mathsf{tc}, r, \ell')$ is followed by a lapse of time $\delta(\tau) \in \mathbb{Z}_{\geq 0}$ at the partial state $\perp_\tau$, before a β-transition is executed:

$$(\ell, v, t) \xrightarrow{p} (\perp_\tau, t[r \mapsto 0]) \xrightarrow{\delta(\tau)} (\ell', v', t[r \mapsto 0] + \delta(\tau)). \tag{3.1}$$

This corresponds to the following execution sequence in the atomic component B, if such a sequence is feasible:

$$(\ell, v, t) \xrightarrow{p} (q', t[r \mapsto 0]) \overset{\delta(\tau)}{\rightsquigarrow} (\ell', t[r \mapsto 0] + \delta(\tau)). \tag{3.2}$$

Notice that the time step $\delta(\tau)$ of $B^\perp$ in (3.1) may not be a time step of B in (3.2) if $\delta(p)$ is not allowed by the time pogress condition of location ℓ', meaning that the partial state model violates time progress conditions defined in the corresponding atomic component. In this case, we say that the considered execution sequence is not *time-safe*.

A correct implementation must execute only time-safe sequences. Time-safety violations occur in a partial state model when the execution time of an transitions is larger than what is allowed by the time progress conditions of the corresponding atomic component. Correct implementations are obtained for platforms that are sufficiently fast for executing the application without violating time-safety. When this cannot be ensured for a given platform, we propose to detect time-safety violations at run-time and to stop the system in order to prevent the application from incorrect executions.

3.1.2 Composition using Interaction

Let $\gamma(B_1, \ldots, B_n)$ be a composition of n atomic components $\{B_1, \ldots, B_n\}$ with semantics $(Q^{\mathbf{g}}, \Sigma, \longrightarrow_\gamma)$. The corresponding partial state model is $\gamma(B_1^\perp, \ldots, B_n^\perp)$ where $B_i^\perp$ are the corresponding atomic components with partial states of B_i. The semantics of $\gamma(B_1^\perp, \ldots, B_n^\perp)$ is the transition system $(Q^{\mathbf{g}} \cup Q^{\mathbf{p}}, \Sigma \cup \{\beta\}, \rightsquigarrow_\gamma)$ where:

- $Q^{\mathbf{p}}$ is the set of partial states where at least one component is at a busy location.
- $\rightsquigarrow_\gamma$ is the set of labeled transitions defined as follows.
  - *Discrete steps:*

$$\frac{\begin{array}{c} a = \{\{p_i\}_{i\in I}, G_a, F_a\} \in \gamma \qquad G_a(\{v_i\}_{i\in I}) \qquad \{v_i''\}_{i\in I} = F_a(\{v_i\}_{i\in I}) \\ \forall i \in I (\ell_i, v_i, t_i) \xrightarrow{p_i(v''_i)}_i (\ell_i', v_i', t_i') \qquad \forall i \notin I (\ell_i', v_i', t_i') = (\ell_i, v_i, t_i) \end{array}}{(\ell, v, t) \overset{a}{\rightsquigarrow}_\gamma (\ell', v', t')}$$

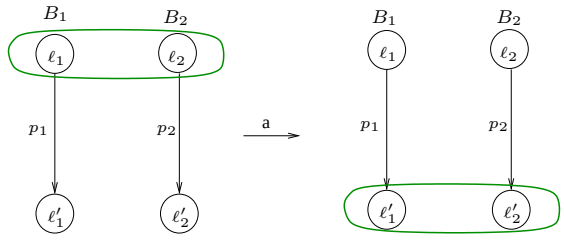
  - *Internal steps:*

$$\frac{\exists i \in \{1, \ldots, n\}\ (\ell_i, v_i, t_i) \xrightarrow{\beta} (\ell_i', v_i', t_i) \qquad \forall j \neq i\ (\ell_j', v_j', t_j) = (\ell_j, v_j, t_j)}{(\ell, v, t) \overset{\beta}{\rightsquigarrow}_\gamma (\ell', v', t)}$$

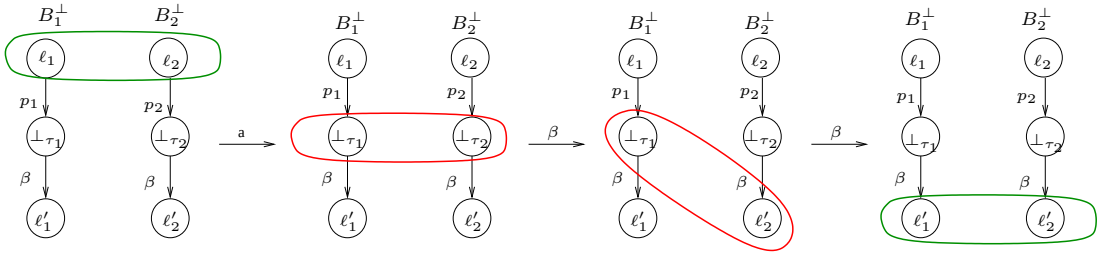
– *Delay steps:*

$$\frac{\delta \in \mathbb{Z}_{\geq 0} \qquad \forall i \in \{1,\ldots,n\} \;\; \mathsf{tpc}_{\ell_i(t_i+\delta)}}{(\ell, v, t) \overset{\delta}{\rightsquigarrow}_\gamma (\ell, v, t+\delta)}$$

Note that the first and the third inference rules (discrete and delay steps) are the same as for the composition rule in the global sate semantics (Definition 15). The second rule corresponds to the completion of a busy component.



(a) Execution of interaction a an abstract model



(b) Execution of interaction a in the corresponding partial state model

Figure 3.2: Execution of interaction a in the global state model and a corresponding execution in the partial state model.

In the partial state model $\gamma(B_1^\perp, \ldots, B_n^\perp)$, the execution of an interaction $a{=}\{\{p_i\}_{i\in I}, G_a, F_a\}$ of $\gamma(B_1, \ldots, B_n)$ can be decomposed as follows. First, the beginning of the execution of a represented in a single transition in $\gamma(B_1^\perp, \ldots, B_n^\perp)$. Second, each component $B_i^\perp$ completes by executing its internal β transition.

Figure 3.2 shows the execution of an interaction $a = \{\{p_1, p_2\}, -, -\}$ in the global state semantics (a) and its corresponding execution in the partial state semantics (b). In (a) the execution of the interaction a is atomic. That is, components B_1 and B_2 move instantaneously from the global state $((\ell_1, \ell_2), v, t)$ to the global state $((\ell'_1, \ell'_2), v', t')$. Whereas, firing interaction a in the partial state model moves the components from global state $((\ell_1, \ell_2), v, t)$ to the partial state $((\perp_{\tau_1}, \perp_{\tau_2}), v, t')$. Then, each component $B_1^\perp$ and $B_2^\perp$ completes by executing its β transition independently, leading to the global state $((\ell'_1, \ell'_2), v', t')$.

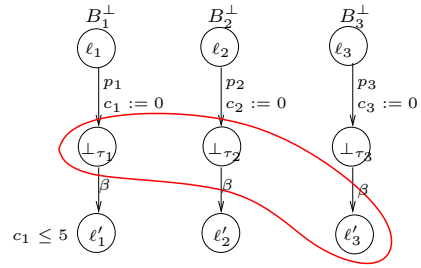
As explained in [74], the relation $\overset{\beta}{\rightsquigarrow}_\gamma$ is terminating and confluent. This property leads to the following definition.

Definition 18. *Let $\gamma(B_1^\perp, \ldots, B_n^\perp)$ be a partial state model. Given a partial state $q = (\ell, v, t) \in Q^{\mathtt{p}}$, we define $q^{\mathtt{g}} = (\ell^{\mathtt{g}}, v^{\mathtt{g}}, t) \in Q^{\mathtt{g}}$, the corresponding global state reached by executing β transitions only (in particular without delay).*

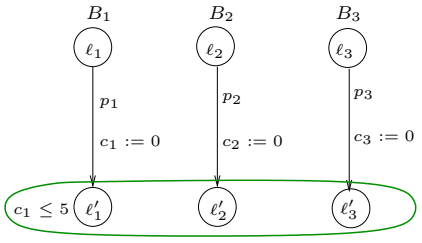
Note that $q^{\mathtt{g}}$ is unique. This comes from the fact that we consider only deterministic automata for components.

As already explained, the executions of the partial state model $B^\perp$ corresponding to the atomic component $B^\perp$ may be not time-safe, due to the execution of disallowed delay steps from busy states. When composing $B_i^\perp$, the execution of the obtained partial state model $\gamma(B_1^\perp, \ldots, B_n^\perp)$ may be also not time-safe. This is because for each component $B_i^\perp$ that is in a partial location, its time progress condition is `true`. Thus, from partial state q, time progress is constrained only by time progress conditions of components that are in a stable location.

Example 8. *Let $\gamma(B_1, B_2, B_3)$ be the composition of two components B_1, B_2 and B_3 and let $\gamma(B_1^\perp, B_2^\perp, B_3^\perp)$ be its corresponding partial state model. Suppose that $B_1^\perp, B_2^\perp$ and $B_3^\perp$ be at locations $\perp_{\tau_1}, \perp_{\tau_1}$ and ℓ_3', respectively (see Figure 3.3(a)). The delay allowed from the partial state $((\perp_{\tau_1}, \perp_{\tau_1}, \ell_3'), v, 0)$ is $\delta \in [0, \infty[$. This is because all time progress conditions of locations $\perp_{\tau_1}$, $\perp_{\tau_1}$ and ℓ_3 are `true`. However, for the corresponding global state $((\ell_1', \ell_2', \ell_3'), v, 0)$ the allowed delay is restricted only to $\delta \in [0, 5]$ due to the time progress condition of location ℓ_1' (see Figure 3.3(b)).*



(a) Delay allowed at the partial state $((\perp_{\tau_1}, \perp_{\tau_2}, \ell_3), v, 0)$ is $\delta \in [0, \infty[$.



(b) Delay allowed at the corresponding global state $((\ell_1, \ell_2, \ell_3), v, 0)$ is $\delta \in [0, 5]$.

Figure 3.3: Illustrative example for delay step execution from partial states.

3.1.3 Adding Priority

The behavior of the partial state model $\pi\gamma(B_1^\perp, \ldots, B_n^\perp)$ is a restriction of $\gamma(B_1^\perp, \ldots, B_n^\perp)$ with respect to the priority π. Its semantics is given by the labeled transition system $(Q^{\mathbf{g}} \cup Q^{\mathbf{p}}, \Sigma \cup \{\beta\}, \rightsquigarrow_\pi)$ where $\rightsquigarrow_\pi$ restricts the discrete steps of $\rightsquigarrow_\gamma$ as shown follows:

$$\textsc{Prio}\ \frac{(\ell, v, t) \overset{a}{\rightsquigarrow_\gamma} (\ell', v', t') \qquad \forall a' \in \gamma\ a\pi a' \Rightarrow (\ell, v, t) \overset{a'}{\not\rightsquigarrow_\gamma}}{(\ell, v, t) \overset{a}{\rightsquigarrow_\pi} (\ell', v', t')}$$

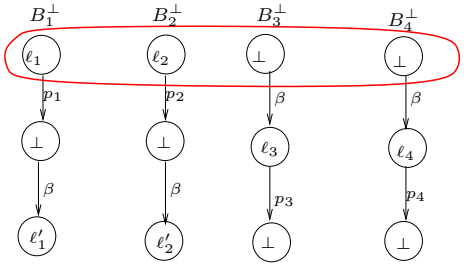
Let a and $a' \in \gamma$ be to interactions such that $a\pi a'$, and let $q \in Q^{\mathbf{p}}$ be a partial state of $\pi\gamma(B_1^\perp, \ldots, B_n^\perp)$. Clearly, if interaction a' is not enabled at q, interaction a can execute from the partial state q if it is enabled. However, it is possible that the interaction a' is enabled at the corresponding global state $q^{\mathbf{g}} \in Q^{\mathbf{g}}$, which disallows the execution of a from $q^{\mathbf{g}}$. Thus, the partial state model $\pi\gamma(B_1^\perp, \ldots, B_n^\perp)$ may execute interactions from partial states q that are not allowed by $\pi\gamma(B_1, \ldots, B_n)$ from the corresponding global states $q^{\mathbf{g}}$ according to the priority π.

Example 9. *Let $\pi\gamma(B_1, B_2, B_3, B_4)$ be a composition of four components. Let $a = \{\{p_1, p_2\}$,`true`,$-\}$ and a'=$\{\{p_3$,$p_4\}$,`true`,$-\}$ be two interactions in γ such that $a\pi a'$. Suppose that timing constraints of p_1, p_2, p_3 and p_4 are* `true` *Figure 3.4 (a) shows that interaction a is enabled from the partial state $q = ((\ell_1, \ell_2, \perp_3, \perp_4), v, t)$. This is because ports p_1 and p_2 are enabled from partial state q. Due to the disabledness of a' at partial state q, the priority $a\pi a'$ does not apply and thus, a may execute from q. However, Figure 3.4 (b) shows that a' in enabled in the corresponding global state $q^{\mathbf{g}} = ((\ell_1, \ell_2, \ell_3, \ell_4), v, t)$, since p_3 and p_4 are enabled at this state. This prevent a from executing at global state $q^{\mathbf{g}}$.*

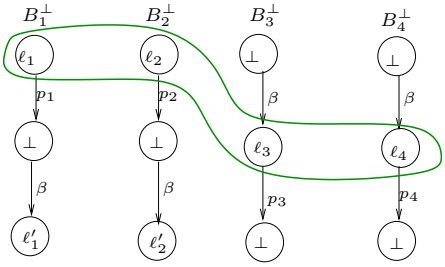
Example 10. *Consider the composite component $\pi\gamma(B_1, B_2, B_3)$ of Example 2.3. Figure 3.5 shows the corresponding partial state model $\pi\gamma(B_1^\perp, B_2^\perp, B_3^\perp)$. The execution of interaction a_1 requires that all components $B_1^\perp$, $B_2^\perp$ and $B_2^\perp$ be in global locations ℓ_1^1, ℓ_2^1 and ℓ_3^1 respectively. After executing a_1, it is possible, due to the integration of partial states, to reach a (partial) state where $B_1^\perp$ is at ℓ_2^1, $B_2^\perp$ is at ℓ_2^2 and $B_3^\perp$ is at $\perp_{\tau_3^1}$. At this partial state, time progress is constrained only by the time progress condition of ℓ_2^1, i.e. $c_1 \leq 20$, as time progress condition of ℓ_2^2 and $\perp_{\tau_3^1}$ are* `true`. *In addition, a_3 is disabled and the priority cannot apply to a_2. As a result, it is possible that interaction a_2 executes after a delay $\delta \in [10, 20]$ and leads to location err in B_2 which is not a reachable location in the composition $\pi\gamma(B_1, B_2, B_3)$. That is, the execution sequences of the partial state model $\pi\gamma(B_1^\perp, B_2^\perp, B_3^\perp)$ differ from the execution sequences of global state model $\pi\gamma(B_1, B_2, B_3)$.*

3.1.4 Notion of Correctness

Let $\pi\gamma(B_1 \ldots, B_n)$ be a composition, and let $\pi\gamma(B_1^\perp, \ldots, B_n^\perp)$ be its corresponding partial state model. Informally, we say that $\pi\gamma(B_1^\perp, \ldots, B_n^\perp)$ is correct if each execution (delay or interaction) from a given partial state q, is allowed from the corresponding global state $q^{\mathbf{g}}$. Below, we define correct execution of partial state models.



(a) Interaction $a = \{p_1, p_2\}$ may execute from the partial state $q = ((\ell_1, \ell_2, \perp, \perp), v, t)$ due to the disabledness of $a' = \{p_3, p_4\}$ at q.



(b) Interaction a is disallowed from the corresponding global state because a' is enabled.

Figure 3.4: Illustrative example for interaction execution from partial states.

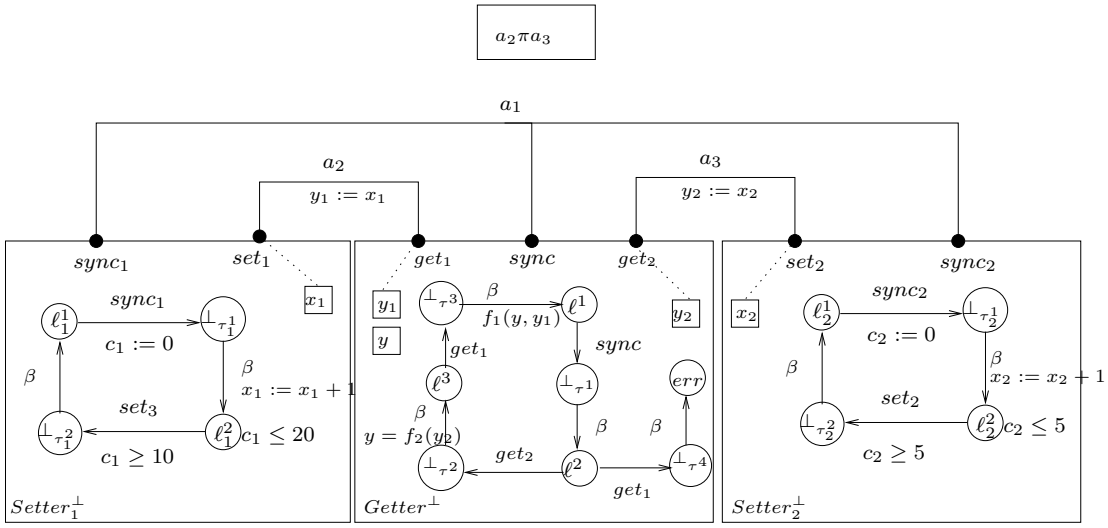


Figure 3.5: Partial state model of Example 2.3.

Definition 19. *Let $\pi\gamma(B_1 \ldots, B_n)$ be a composition, and let $\pi\gamma(B_1^{\perp}, \ldots, B_n^{\perp})$ be its corresponding partial state model. Let $(Q^{\boldsymbol{g}}, \Sigma, \longrightarrow_{\pi})$ and $(Q^{\boldsymbol{p}} \cup Q^{\boldsymbol{g}}, \Sigma \cup \{\beta\}, \rightsquigarrow_{\pi})$ be their semantics respectively. Let $q = (\ell, v, t) \in Q^{\boldsymbol{p}}$ be a partial state, and let $q^{\boldsymbol{g}} \in Q^{\boldsymbol{g}}$ be its corresponding global state. Let $\delta \in \mathbb{Z}_{\geq 0}$ be a delay and $a \in \gamma$ be an interaction.*

- *We say that the time step $q \overset{\delta}{\rightsquigarrow}_{\pi}$ is correct in the partial state model $\pi\gamma(B_1^{\perp}, \ldots, B_n^{\perp})$, if the time step $q^{\boldsymbol{g}} \overset{\delta}{\longrightarrow}_{\pi}$ is allowed in the composition $\pi\gamma(B_1, \ldots, B_n)$.*
- *We say that the discrete step $q \overset{a}{\rightsquigarrow}_{\pi}$ is correct in the partial state model $\pi\gamma(B_1^{\perp}, \ldots, B_n^{\perp})$, if the discrete step $q^{\boldsymbol{g}} \overset{a}{\longrightarrow}_{\pi}$ is allowed in the composition $\pi\gamma(B_1, \ldots, B_n)$.*

Our notion of correctness is derived from *observational equivalence* [75] between partial state models and global state models by considering that β transitions are not observable.

In general, observational equivalence of two transition systems $A = (Q_A, \Sigma_A \cup \{\beta\}, \rightarrow_A)$ and $B = (Q_B, \Sigma_B \cup \{\beta\}, \rightarrow_B)$ is based on the usual definition of weak bisimilarity [75], where β transitions are considered unobservable.

Definition 20. *A* weak simulation *from A to B, denoted $A \subset B$, is a relation $R \subseteq Q_A \times Q_B$, such that $\forall (q, r) \in R,\ a \in P :\ q \overset{a}{\rightarrow}_A q' \implies \exists r' :\ (q', r') \in R \wedge r \overset{\beta^* a \beta^*}{\rightarrow}_B r'$ and $\forall (q, r) \in R :\ q \overset{\beta}{\rightarrow}_A q' \implies \exists r' :\ (q', r') \in R \wedge r \overset{\beta^*}{\rightarrow}_B r'$*

A weak bisimulation over A and B is a relation R such that R and R^{-1} are both weak simulations. We say that A and B are *observationally equivalent* and we write $A \sim B$ if for each state of A there is a weakly bisimilar state of B and conversely.

We use observational equivalence to prove the correctness of the partial state models, as it has a direct effect on the correctness of execution sequences. In fact, when the partial state model and the global state model are observationally equivalent, both have the same execution sequences by considering only observable steps (interactions and delay steps).

As already explained through the examples, the partial state model does not provide the same execution sequences as the global state model. Thus, in general, the two models are not observationally equivalent. In the next section, we study sufficient conditions for the partial state model to be observationally equivalent to the global state model.

3.2 Enforcing Correctness

As it is shown in example 10, the partial state models may provide incorrect executions with respect to the global state models. In order to obtain correct execution, we require that partial state models be observationally equivalent to the global state models. In this section, we study conditions for the partial state models to be observationally equivalent to the global state models.

Let $\pi\gamma(B_1, \ldots, B_n)$ be a composition and let $\pi\gamma(B_1^{\perp}, \ldots, B_n^{\perp})$ be its corresponding partial state model. From global states, the transitions executed in $\pi\gamma(B_1, \ldots, B_n)$ and $\pi\gamma(B_1^{\perp}, \ldots, B_n^{\perp})$ are the same. Consider the execution of an interaction a in $\pi\gamma(B_1^{\perp}, \ldots, B_n^{\perp})$ from the partial state $q = (\ell, v, t)$, and the corresponding global state $q^{\mathbf{g}} = (\ell^{\mathbf{g}}, v^{\mathbf{g}}, t)$ in $\pi\gamma(B_1, \ldots, B_n)$. As explained in [74], if a is enabled at q, it is also enabled at $q^{\mathbf{g}}$. However, in order to respect

the semantics of the composite model $\pi\gamma(B_1, \ldots, B_n)$, a should be disabled due to priority π if there exists an interaction a' enabled at state $q^{\mathbf{g}}$ such that $(a, a') \in \pi$. Thus, a should be blocked if enabledness of interaction a' cannot be decided at q.

In a similar way, a delay transition δ enabled in $\pi\gamma(B_1^{\perp}, \ldots, B_n^{\perp})$ at partial state q can be disallowed in $\pi\gamma(B_1, \ldots, B_n)$ at the corresponding global state $q^{\mathbf{g}}$ if there exists at least one component B_i having as time progress condition $\mathtt{tpc}_{\ell_i^{\mathbf{g}}}$ that does not allow the time step δ, that is, such that $\mathtt{tpc}_{\ell_i^{\mathbf{g}}}(t + \delta)$ evaluates to `false`.

To prevent $\pi\gamma(B_1^{\perp}, \ldots, B_n^{\perp})$ from incorrect execution, we define a predicate $\mathtt{safe}(q, \sigma)$ on $Q^{\mathbf{g}} \cup Q^{\mathbf{p}} \times \gamma \cup \mathbb{Z}_{\geq 0}$ indicating whether the execution of an interaction $\sigma \in \gamma$ or the time step $\sigma \in \mathbb{Z}_{\geq 0}$ violates the global state semantics. Clearly, the behavior of $\pi\gamma(B_1^{\perp}, \ldots, B_n^{\perp})$ is already safe for global states. In the following, we define conditions that should be satisfied by any predicate `safe`.

- For an interaction a, a partial state $q = (\ell, v, t)$ and its corresponding global state $q^{\mathbf{g}} = (\ell^{\mathbf{g}}, v^{\mathbf{g}}, t)$, the predicate `safe` satisfies:

$$\mathtt{safe}((\ell, v, t), a) \;\Rightarrow \nexists b \in \gamma \;.\; a\pi b \;\wedge (\ell^{\mathbf{g}}, v^{\mathbf{g}}, t) \overset{b}{\rightsquigarrow}_{\gamma} \tag{3.3}$$

- For a time step δ, `safe` also satisfies:

$$\mathtt{safe}((\ell, v, t), \delta) \;\Rightarrow\; \forall i \in \{1, \ldots, n\} \mathtt{tpc}_{\ell_i^{\mathbf{g}}}(t + \delta) \tag{3.4}$$

The following definition defines the safe partial state model.

Definition 21. *Given a predicate* `safe` *on* $Q^{\mathbf{g}} \cup Q^{\mathbf{p}} \times \gamma \cup \mathbb{Z}_{\geq 0}$ *satisfying conditions(3.3) and (3.4), we define the safe partial state model denoted by* $\mathtt{safe}[\pi\gamma(B_1^{\perp}, \ldots, B_n^{\perp})]$. *Its semantics is defined by the labeled transition system* $(Q^{\mathbf{g}} \cup Q^{\mathbf{p}}, \Sigma \cup \{\beta\}, \xrightarrow[\mathtt{safe}]{})$ *where* $\Sigma = \gamma \cup \mathbb{Z}_{\geq 0}$ *and* $\xrightarrow[\mathtt{safe}]{}$ *is the set of labeled transitions defined as follows:*

- Discrete steps:

$$\frac{(\ell, v, t) \overset{a}{\rightsquigarrow} (\ell', v', t') \qquad \mathtt{safe}((\ell, v, t), a)}{(\ell, v, t) \xrightarrow[\mathtt{safe}]{a} (\ell', v', t')}$$

- Internal steps:

$$\frac{(\ell, v, t) \overset{\beta}{\rightsquigarrow} (\ell', v', t)}{(\ell, v, t) \xrightarrow[\mathtt{safe}]{\beta} (\ell', v', t)}$$

- Delay steps :

$$\frac{(\ell, v, t) \overset{\delta}{\rightsquigarrow} (\ell, v, t + \delta) \qquad \mathtt{safe}((\ell, v, t), \delta)}{(\ell, v, t) \xrightarrow[\mathtt{safe}]{\delta} (\ell, v, t + \delta)}$$

Theorem 1. $\texttt{safe}[\pi\gamma(B_1^\perp, \ldots, B_n^\perp)] \sim \pi\gamma(B_1, \ldots, B_n)$.

Proof. We denote by Q_U the set of states of $\pi\gamma(B_1, \ldots, B_n)$ and Q_V the set of states of $\texttt{safe}[\gamma^\perp(B_1^\perp, \ldots, B_n^\perp)]$. Let $R = \{(q^{\mathbf{g}}, q) \in Q_U \times Q_V \wedge q \xrightarrow[\texttt{safe}]{\beta^*} q^{\mathbf{g}}\}$. We show that it is a weak bisimulation. Since the BIP model $\pi\gamma(B_1, \ldots, B_n)$ does not have β transition, it is enough to prove that:

(i) For each $(q^{\mathbf{g}}, q) \in R$ such that $q \xrightarrow[\texttt{safe}]{\beta} q'$, $(q^{\mathbf{g}}, q') \in R$.

(ii) For all $(q^{\mathbf{g}}, q) \in R$ and $\sigma \in \gamma \cup \mathbb{Z}_{\geq 0}$ such that $q^{\mathbf{g}} \xrightarrow{\sigma} q'^{\mathbf{g}}$, there is q' such that $(q'^{\mathbf{g}}, q') \in R$ and $q \xrightarrow[\texttt{safe}]{\beta^*\sigma\beta^*} q'$.

(iii) For each $(q^{\mathbf{g}}, q) \in R$ and $\sigma \in \gamma \cup \mathbb{Z}_{\geq 0}$ such that $q \xrightarrow[\texttt{safe}]{\sigma} q'$, there is $q'^{\mathbf{g}}$ such that $(q'^{\mathbf{g}}, q') \in R$ and $q^{\mathbf{g}} \xrightarrow{\sigma} q'^{\mathbf{g}}$.

(i) Consequence of Definition 18.

(ii) If σ is an interaction $a \in \gamma$. Suppose that $(q^{\mathbf{g}}, q) \in R$ and $q^{\mathbf{g}} \xrightarrow{a} q'^{\mathbf{g}}$. We have $q \xrightarrow[\texttt{safe}]{\beta^*} q^{\mathbf{g}}$ by definition of R. Then, by definition of $\xrightarrow[\texttt{safe}]{}$, we have $q^{\mathbf{g}} \xrightarrow[\texttt{safe}]{a} q' \xrightarrow[\texttt{safe}]{\beta^*} q'^{\mathbf{g}}$. We conclude by remarking that $(q'^{\mathbf{g}}, q') \in R$. The same reasoning applies if σ is a time step $\delta \in \mathbb{Z}_{\geq 0}$.

(iii) If σ is an interaction $a \in \gamma$. Suppose that $(q^{\mathbf{g}}, q) \in R$ and $q \xrightarrow[\texttt{safe}]{a} q'$. According to the definition of $\xrightarrow[\texttt{safe}]{}$, $\texttt{safe}(q, a)$ holds. That is, there is no interaction b having higher priority that is enabled from state $q^{\mathbf{g}}$. That is, a can execute from $q^{\mathbf{g}}$ according to the global state semantics i.e., $q^{\mathbf{g}} \xrightarrow{a} q'^{\mathbf{g}}$. According to Definition 18, from q' there exists a unique state $q'^{\mathbf{g}}$ reachable by doing β transitions, that is $q' \xrightarrow[\texttt{safe}]{\beta^*} q'^{\mathbf{g}}$. Thus, we conclude that $(q'^{\mathbf{g}}, q') \in R$.

The same reasoning applies if we consider that σ is a time step and the fact that $\texttt{safe}(q, \delta)$ holds.

□

Ideally, `safe` should be obtained by using equivalence instead of implication in (3.3) and (3.4), corresponding to the less restrictive predicate allowing the maximal parallelism in the system. However, its computation requires the knowledge of the reachable global state $(\ell^{\mathbf{g}}, v^{\mathbf{g}}, t)$ from any partial state (ℓ, v, t). This requires to guess what will be the timing constraints, guards and time progress conditions after the completion of all busy components. In general, it is not possible to compute ideally the predicate `safe`, since it may depend on the future values of the variables of the busy components. Nonetheless, we can always over-approximate it [76]. To this end, we define $\texttt{safe}^*$ an over-approximation of `safe` such that:

$$\texttt{safe}^* \Rightarrow \texttt{safe}. \tag{3.5}$$

This predicate will be used for the implementation of real systems.

3.3 Conclusion

In this chapter, we defined partial state models that correspond to a high-level representation for parallel execution of BIP models. We defined conditions for which partial state models are observationally equivalent to global state models. However, partial state models do not provide details on how to implement multiparty interactions. In the next chapter we present a solution to implement multiparty interactions based on a centralized scheduler.

4

Parallel Real-Time Systems Design with Centralized Scheduler

In the previous chapter, we defined partial state models that correspond to high-level representation for parallel execution of BIP models. Those models cannot be directly implemented since they do not provide details on how to implement multiparty interactions and priorities in distributed settings.

In a distributed context, we assume that components communicate through asynchronous message-passing. Thus, each component becomes a distributed component that is able either to send a message, to wait for a message or to execute an internal computation. Our solution to implement multiparty interactions is based on two-way handshake protocol. Indeed, distributed components exchange messages with a coordinator called *scheduler* (built as a BIP component) responsible for triggering interactions. In order to evaluate the enabled interactions at a given state, the distributed components are required to send their current states (e.g. enabled ports, time progress condition, timing constraints, etc.) to the scheduler component. This is realized by splitting each component transition into two transitions: one transition to send its current state to the scheduler in a message called offer and another transition to receive notification from the scheduler triggering the port to be executed. Regarding the scheduler, it is required to accumulate offers from distributed components and decides the execution of interactions on-line. As offers are sent asynchronously, the scheduler may not have a global knowledge about system and it may decide to execute an interaction based on a partial knowledge. This corresponds to an execution from a partial state of the partial state models presented in the previous chapter. As explained in Subsection 3.2, safe interactions executions from partial states are ensured using the predicate `safe`. Thus, any execution decided by the scheduler should also agree with the predicate `safe`. As explained in 3.2, in practice we are using $\texttt{safe}^*$ based on over-approximations of reached states.

In this chapter,we describe in Section 4.1 the class of BIP models that allows the construction of our target Send/Receive models. In Section 4.2, we define formally the transformation from the centralized to Send/Receive BIP models. Finally, in Section 4.3, we prove the correctness of our obtained models.

4.1 Target Models

We target models that could be directly implementable using basic message-passing primitives. We consider three types of elementary actions: message sending, message receiving and internal computation. Thus, our target models include three types of ports: unary ports, send ports and receive ports. Unary ports are used when components have to execute independently from each others, which are formally expressed using unary interactions (i.e. an interaction consisting of a single port). Such unary interactions correspond to internal computation in components. Send and receive ports participate in message-passing interactions which are interactions between a single send port and one or more receive ports. We call those interactions Send/Receive interactions. When such an interaction executes, the variables exported by the send port are copied in the variables exported by the receive port. We require that send port cannot be blocked by corresponding receive ports. That is, each time a send port is enabled, its corresponding Send/Receive interaction has to be enabled too so that every Send/Receive interaction can take place whenever the send port is enabled.We denote this class of BIP models by *Send/Receive* models. The definition of such models is given below.

Definition 22. *We say that* $B^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR})$ *is a* Send/Receive *BIP model iff we can partition the set of ports of* B^{SR} *into three sets* P_s*,* P_r*, and* P_u *that are respectively the set of* send-ports*,* receive-ports*, and* unary ports*, such that:*

- *Each interaction* $a \in \gamma^{SR}$*, is either (1) a Send/Receive interaction with* $a = (s, r_1, r_2, \ldots, r_k)$*,* $s \in P_s$*,* $r_1, \ldots, r_k \in P_r$*,* $G_a = \texttt{true}$ *and* F_a *copies the variables exported by port* s *to the variables exported by ports* $r_1, r_2, \ldots, r_k$*, or, (2) a unary interaction* $a = \{p\}$ *with* $p \in P_u$*,* $G_a = \texttt{true}$*,* F_a *is the identity function.*
- *If* s *is a port in* P_s*, then there exists one and only one Send/Receive interaction* $a \in \gamma^{SR}$ *with* $a = (s, r_1, r_2, \ldots, r_k)$ *and all ports* $r_1, \ldots, r_k$ *are receive-ports. We say that* $r_1, r_2, \ldots, r_k$ *are the receive-ports associated with* s*.*
- *If* $a = (s, r_1, \ldots, r_k)$ *is a Send/Receive interaction in* γ^{SR} *and* s *is enabled at some global state of* B^{SR}*, then all its associated receive-ports* $r_1, \ldots, r_k$ *are also enabled at that state.*

Definition 22 defines a class of BIP models for distributed implementation based on asynchronous message passing. In such systems, communication is sender-triggered, where a message can be emitted by the sender regardless of the availability of receivers. The third property of the definition requires that all receivers are ready to receive whenever the sender may send a message. This ensures that the sender is never blocked and can trigger the

Send/Receive interaction. Intuitively, a model that meets properties of Definition 22 can be seen as a set of independent processes communicating through asynchronous message passing.

4.1.1 Send/Receive BIP models Architecture

Let $B = \pi\gamma(B_1, \ldots, B_n)$ be an input BIP model of the proposed transformation. The Send/Receive BIP model corresponding to B is based two layers.

- The *Atomic Component Layer* consists of a transformation of atomic components B_i into Send/Receive atomic component B_i^{SR}. Components B_i^{SR} send asynchronously offer messages to notify the scheduler about their current state.
- The *Scheduler Layer* deals with execution of interactions. Based on offers sent by atomic components, the scheduler may decide the execution of an interaction at a given time and send back notifications to participating components specifying which port has to be executed.

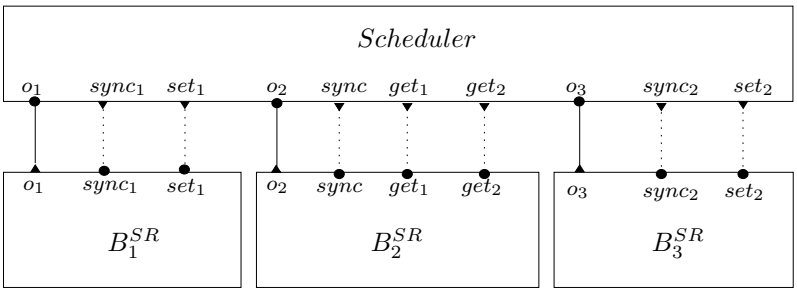


Figure 4.1: Send/Receive BIP model architecture of Figure 2.10.

Example 11. *Figure 4.1 presents the Send/Receive BIP model obtained from the model of Figure 2.10. Note that we assume that* B_1=*setter*$_1$, B_2=*getter*$_1$ *and* B_3=*setter*$_2$.

Communications between atomic components and the centralized scheduler are represented by Send/Receive interactions whose execution is instantaneous according to the semantics of BIP, which is not realistic for all kind of systems. In this chapter, we consider systems that provide fast communications. We will see that this assumption is an important requirement for the correctness of the implementation of our Send/Receive models. We will see in Chapter 6 how to deal with non instantaneous communications.

4.1.2 Expressing Timing Constraints and Time Progress Conditions Using a Global Clock

Every atomic component can define a set of local clocks. They can be reset at any time and are involved in timing constraints and time progress conditions. In this thesis, we make use of a global clock in Send/Receive models. In fact, a global clock allows a common time scale among components and the scheduler. It reduces the effort of the scheduler when keeping

track of the actual progress of time, since it needs to maintain (update) only one clock. Any component clock c is obtained by simply shifting the global clock g by an amount of time that is constant as soon as the clock c is not reset, which is very efficient. The clock g is initialized to 0 and is never reset, and measures the absolute time elapsed since the system started executing. It is used when atomic components inform the scheduler about their timing constraints and time progress conditions. Therefore, we follow the approach of [77]: for each clock c of an atomic component B_i we introduce a variable ρ_c that stores the absolute time of the last reset c. This variable is updated whenever the component executes a transition resetting clock c. In fact, when the scheduler executes an interaction a, its stores its execution date in a variable t_a^{ex}. The value of this variable is sent by the scheduler to participating components when notifying them. Each participating component executes then the corresponding transition according to the received notification, and updates each variable ρ_c to t_a^{ex} if the transition resets clock c in the original model. Notice that the value of c can be computed from the current value of g and ρ_c by using the equality $c = g - \rho_c$. This allows to entirely get rid of clocks of components B_i, keeping only the clock g and variables ρ_c, $c \in \mathsf{C}_i$. Using ρ_c, any timing constraint tc involved in a component B can be expressed using the clock g instead of clocks C. Using (2.1), we transform tc as follows:

$$\mathsf{tc} = \bigvee_{k=1}^{m} \bigwedge_{c \in \mathsf{C}} l_c^k \leq c \leq u_c^k = \bigvee_{k=1}^{m} \bigwedge_{c \in \mathsf{C}} l_c^k + \rho_c \leq g \leq u_c^k + \rho_c.$$

That is, tc is a disjunction of interval constraints on g of the form:

$$\mathsf{tc} = \bigvee_{k=1}^{m} \max\{l_c^k + \rho_c\}_{c \in \mathsf{C}} \leq g \leq \min\{u_c^k + \rho_c\}_{c \in \mathsf{C}}. \tag{4.1}$$

Similarly, any time progress condition tpc involved in component B is transformed using the clock g. Using (2.2), we transform tpc as follows:

$$\mathsf{tpc} = \bigwedge_{c \in \mathsf{C}} c \leq u_c = \bigwedge_{c \in \mathsf{C}} g \leq u_c + \rho_c.$$

That is, tpc is an interval constraint on g of the form:

$$\mathsf{tpc} = g \leq \min\{u_c + \rho_c\}_{c \in \mathsf{C}}. \tag{4.2}$$

4.2 Transformations

In this section, we present the formal definition of our transformation from BIP models into Send/Receive BIP models. First, we explain in Subsection 4.2.1 how to transform an atomic component of the original model into a Send/Receive atomic component. Then, we explain in Subsection 4.2.2 how to build the centralized scheduler component. Finally, we define the interactions between Send/Receive atomic components and the scheduler in Subsection 4.2.3.

4.2.1 Transformation of Atomic Components

The transformation of an atomic BIP component B into a Send/Receive atomic component B^{SR} is as follows. It relies on decomposing each atomic transition of B into a send and a receive transition. As already said, the idea is to create a protocol between atomic components and the scheduler. The protocol's first step is initiated by the atomic component B^{SR} by sending an offer to the scheduler through a dedicated send port. Then, the second step consists on waiting a notification from the scheduler arriving on a receive port. The offers contain necessary information to determine whether an interaction can be safely executed.

Since each offer sent by a component is followed by a notification from the scheduler, we split each location ℓ into two locations, namely ℓ itself and $\perp_\ell$ as shown in Figure 4.2.

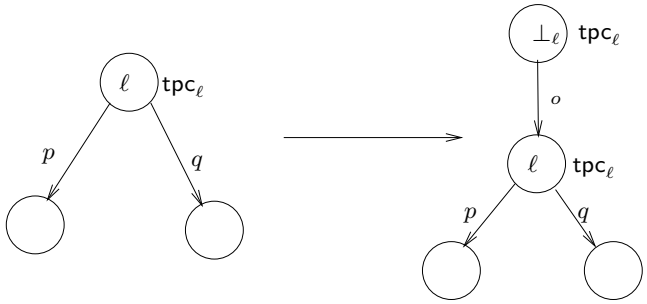


Figure 4.2: Atomic component transformation.

From location $\perp_\ell$, the Send/Receive atomic component is not at a stable location and is able to send an offer to the scheduler. By introducing $\perp_\ell$, atomicity of original transition is broken. Thus, this location could be seen as a busy location in the partial state model.

According to condition (3.4) of Subsection 3.2, time is allowed to progress from a partial state of a partial state model only if it is allowed by its corresponding next reached global state. This condition could be satisfied by enforcing each atomic component to respect, at each busy location, the time progress condition of each next stable location. To satisfy this condition in the Send/Receive atomic component, we put at location $\perp_\ell$ the time progress condition tpc_ℓ originally defined in the atomic component. That is, we enforce the Send/Receive atomic component to complete its internal computation and to send its offer to the scheduler from location $\perp_\ell$ before the time progress conditions tpc_ℓ becomes false.

Send/Receive atomic components sends offers to the scheduler to inform about their states. Based on the offers received, the scheduler decides the execution of interactions. As offers are sent asynchronously, the scheduler may not have a global knowledge about system and it may decide to execute an interaction based on a partial knowledge. This corresponds to an execution from a partial state of the partial state models. According to condition (3.3) of Subsection 3.2, an interaction a is allowed to execute from a partial state only if there is no enabled interaction b having higher priority than a and enabled from the corresponding global state. To compute the enabledness of a the scheduler needs information about the current states of components participating in a. Thus, we require that the offers includes information about their current states. In order to safely execute a, the scheduler is required to compute the enabledness of each interaction b, such that $a\pi b$ to ensure safe execution of a. As the

computation of interaction b requires waiting fresh offers from components participating in b, the scheduler may use information of the last received offers from these components to approximate the enabledness of b from the next reached state. To this end, we require that components includes also in their offers information containing approximations of their next reached states.

Offer Construction

An offer contains exact variables that encode the current state, and approximated variables that encode an approximation of the next reached state.

We include the following variables to inform about the current state of the component:

- an exact time progress condition variable tpc that stores the time progress condition of the current location of the component.
- for each port p, an exact timing constraint variable tc_p that is set to the timing constraint of the transition labeled by p and enabled at the current location if exists, or is set to `false` otherwise.
- for each port p, an exact Boolean guard variable g_p that is set to the evaluation of the Boolean guard of the transition labeled by p and enabled at the current location if exists, or is set to `false` otherwise.

We also include variables that stores the approximations of next reached states of the component. At this step, we dot not provide how to compute these approximations, but we define only variables needed to store approximations. The next paragraph discusses how these approximations are computed.

- For each port p, an approximated timing constraint variable tc^*_p that is set to the disjunction of the timing constraints approximations of transitions labeled by p and enabled from next locations if exist, or and is set to `false` otherwise.
- For each port p, we include an approximated Boolean guard variable g^*_p. This variable stores the disjunction of approximated evaluation of Boolean guard of transitions labeled by p and enabled from next locations if exists, or and is set `false` otherwise.

Note that approximations involve only timing constraints and Boolean guards of ports enabled from the next reached location of the component. These approximations are needed to approximate the enabledness (timing constraint and Boolean guard) of higher priority interactions when the scheduler wants to execute a lower priority interaction.

Example 12. *Consider an atomic component as depicted in Figure 4.3. It contains three ports namely p, p_1 and p_2. In the corresponding Send/Receive atomic component, an offer includes the following variables: the exact time progress condition variable* tpc*, the exact timing constraint variables* tc_{p_1}*,* tc_{p_2} *and* tc_p*, the exact Boolean guard variables g_{p_1} and g_{p_2}, the approximated timing constraint variables* $\mathsf{tc}^*_{p_1}$*,* $\mathsf{tc}^*_{p_2}$ *and* tc^*_p *and the approximated Boolean*

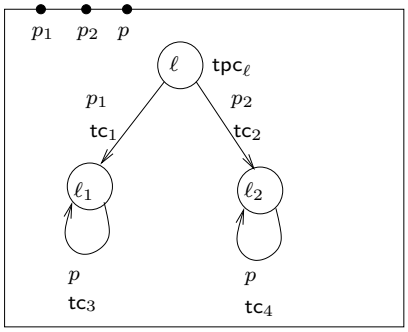


Figure 4.3: Illustrative Example.

guard variables $g^*_{p_1}$ and $g^*_{p_2}$. We denote by $\widetilde{x}$, the approximation value of x, where x could be either a timing constraint or a Boolean guard.

At location $\bot_\ell$ in the corresponding Send/Receive atomic component, the values of these variables are the following.

- tpc *is set to the time progress condition of location ℓ,i.e.* $\mathsf{tpc} := \mathsf{tpc}_\ell$
- *Since transitions labeled by p_1 and p_2 are enabled from location ℓ, the variables tc_{p_1} and tc_{p_2} are set to the timing constraint of these transitions i.e.* $\mathsf{tc}_{p_1} := \mathsf{tc}_1$ *and* $\mathsf{tc}_{p_2} := \mathsf{tc}_2$. tc_p *is set to* `false` *since there is no transition enabled from ℓ and labeled by p.*
- *Since the enabled transitions from ℓ_1 and ℓ_2 are labeled by p, the variable tc^*_p is set to the disjunction of approximated timing constraints of these transitions .i.e.* $\mathsf{tc}^*_p := \widetilde{\mathsf{tc}}_3 \vee \widetilde{\mathsf{tc}}_4$. *The approximated timing constraint variables $\mathsf{tc}^*_{p_1}$ and $\mathsf{tc}^*_{p_2}$ are set to* `false` *since there is no transition enabled from locations ℓ_1 and ℓ_2 and labeled by p_1 or p_2.*
- *Since there is no guard* (`true` *by default*)*, on all transitions labeled by p and enabled from ℓ_1 and ℓ_2, the variable g^*_p is set to* `true`. *$g^*_{p_1}$ and $g^*_{p_2}$ are set to* `false` *since there is no transition enabled from locations ℓ_1 and ℓ_2 and labeled by p_1 or p_2*

Remark . Note that the information about approximation of reached state could be more precise, in particular for timing constraints. In fact, the approximated timing constraint variable tc^*_p contains disjunction of approximated timing constraints of each transition enabled from a next location reached by executing a transition from current location and labeled by p. That is, we consider all possible timing constraints of transitions enabled from a next location regardless the transition (port execution) leading to that location. This can be improved by computing each possible timing constraint of p separately, depending on the port execution that leads to the timing constraint of p. To do that, we define for each pair of ports p, p', such that p' is enabled from a location reached by executing port p, the refined approximated timing constraint variable $\mathsf{tc}^*_p(p')$. This variable is set to the approximation of timing constraint of transition labeled by p and enabled from location reached from the current location by executing a transition labeled by port p'. Then, the scheduler selects the right approximation depending on what port it chooses for execution. This method for

computing approximations involves a lot of variables which may be in the detriment of clarity and simplicity. Therefore, we do not consider this method.

Computing approximations

Consider a location ℓ. Let tc and g be a timing constraint and a Boolean guard to be approximated from ℓ, respectively. We denote by $\widetilde{\mathsf{tc}}$ the approximation of tc, and $\widetilde{g}$ the approximation of g.

Computation of $\widetilde{\mathsf{tc}}$. As explained in Subsection 4.1.2, we transform the timing constraint tc into a timing constraint expressed on clock g of the form $\mathsf{tc} = \bigvee_{k=1}^{m} \max\{l_c^k + \rho_c\}_{c\in\mathsf{C}} \leq g \leq \min\{u_c^k + \rho_c\}_{c\in\mathsf{C}}$. In order to compute $\widetilde{\mathsf{tc}}$ we need to approximate the bounds $u_c^k + \rho_c$, $l_c^k + \rho_c$ of each clock c. Let $\widetilde{u}_c^k + \widetilde{\rho}_c^l$ and $\widetilde{l}_c^k + \widetilde{\rho}_c^u$ be the approximations of the bounds $u_c^k + \rho_c$ and $l_c^k + \rho_c$ respectively. As $\widetilde{\mathsf{tc}}$ is an-over approximation of tc, the values of $\widetilde{l}_c^k$ and $\widetilde{\rho}_c^l$ correspond to the minimal possible values of l_c^k and ρ_c respectively, and the values of $\widetilde{u}_c^k$ and $\widetilde{\rho}_c^u$ correspond to the maximal possible values of l_c^k and ρ_c, respectively. These values are computed as follows.

- If l_c^k (resp. u_c^k) involves expressions containing non-constant variables, then $\widetilde{l}_c^k$ is set to 0 (resp. $+\infty$), otherwise $\widetilde{l}_c^k$ ($\widetilde{u}_c^k$, resp.) is set to l_c^k (resp. u_c^k). That is, the bounds a timing constraint are over-approximated to the extreme values if they can not be evaluated statically.

- If clock c is not reset by any transition τ enabled from ℓ, then $\widetilde{\rho}_c^l$ and $\widetilde{\rho}_c^u$ are set to ρ_c. Otherwise, $\widetilde{\rho}_c^l$ is set to t^{ex} where t^{ex} stores the last execution of the component and $\widetilde{\rho}_c^u$ is set to $+\infty$. That is, the minimal possible value of ρ_c corresponds exactly to the last execution date of the component, which means that for the best case, the component resets c at the date of its last execution. For the maximal values, we put $+\infty$ as default value.

Computation of $\widetilde{g}$. $\widetilde{g}$ is set to g if variables involved in g are constant. Otherwise, $\widetilde{g}$ is set to `true`.

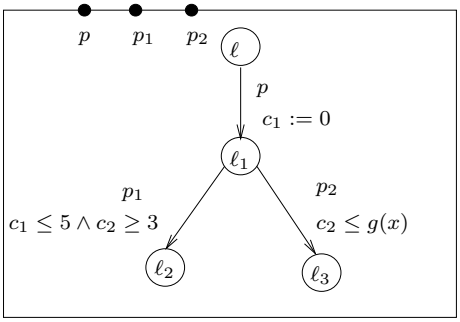


Figure 4.4: Illustrative Example.

Example 13. *Consider an atomic component as shown in Figure 4.4. The atomic component includes two clocks. Suppose that the component is at location ℓ, and that the clocks reset dates ρ_{c_1} and ρ_{c_2} at this location are set to 0. At this location, we need to approximate the timing constraints tc_{τ_1} and tc_{τ_2} of transitions $\tau_1 = (\ell_1, p_1, \ell_2)$ and $\tau_2 = (\ell_1, p_2, \ell_3)$, since ℓ_1 is the reached location from ℓ. Let $\widetilde{\mathsf{tc}}_{\tau_1}$ and $\widetilde{\mathsf{tc}}_{\tau_2}$ be the approximation values of tpc_{ℓ_1}, tc_{τ_1} and tc_{τ_2} respectively. These values are computed as follows.*

- *Since the timing constraint $\mathsf{tc}_{\tau_2} = c_2 \leq g(x)$ has only an upper bound that involves non-constant values, then its approximation $\widetilde{\mathsf{tc}}_2$ is set to $g \leq +\infty$ which is simplified to* `true`.
- *The timing constraint $\mathsf{tc}_{\tau_1} = c_1 \leq 5 \wedge c_2 \geq 3$ is expressed on two clocks c_1 and c_2 and involves constant values, which is expressed on clock g as follows: $\mathsf{tc}_{\tau_1} = g \leq 5 + \rho_{c_1} \wedge g \geq 3 + \rho_{c_2}$. In order to compute $\widetilde{\mathsf{tc}}_{\tau_1}$, we have to approximate reset dates ρ_{c_1} and ρ_{c_2}. First, as ρ_{c_1} is involved in the upper bound of the timing constraint $g \leq 5 + \rho_{c_1}$, then we have to compute its maximal value $\widetilde{\rho}^u_{c_1}$. As c_1 is reset by the transition leading to ℓ_1, $\widetilde{\rho}^u_{c_1}$ is set to $+\infty$. Second, as ρ_{c_2} is involved in the lower bound of timing constraint $g \geq 3 + \rho_{c_2}$, we have to compute its minimal possible value $\widetilde{\rho}^l_{c_2}$. As c_2 is not reset by the transition leading to ℓ_1, $\widetilde{\rho}^u_{c_1}$ is set to the last reset date of clock c_1 which is 0. To summarize, $\widetilde{\mathsf{tc}}_{\tau_1}$ is set to $g \geq 3$.*

Definition of Send/Receive Atomic Component

As already explained, Send/Receive components express timing constraint and times progress conditions using the global clock g. The translation of such constraints is done through the use of reset dates variables ρ_c of each clock c. These variables are updated whenever the scheduler sends notifications for execution to the component. These notifications bring the execution date of the component. We define in the Send/Receive atomic component the variable t^{ex} that stores the execution date of the component. The value of ρ_c is set to the execution date t^{ex}, whenever the component of the original model resets clock c.

We are now ready to define the transformation of B into B^{SR}.

Definition 23. *Let $B = (L, P, T, \mathsf{C}, X, \{X_p\}_{p\in P}, \{g_\tau\}_{\tau\in T}, \{\mathsf{tc}_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T}, \{r_\tau\}_{\tau\in T}, \{\mathsf{tpc}_\ell\}_{\ell\in L})$ be an atomic component. The corresponding Send/Receive component is $B^{SR} = (L^{SR}, P^{SR}, T^{SR}, C^{SR}, X^{SR}, \{X^{SR}_p\}_{p\in P^{SR}}, \{g_\tau\}_{\tau\in T^{SR}}, \{\mathsf{tc}_\tau\}_{\tau\in T^{SR}}, \{f_\tau\}_{\tau\in T^{SR}}, \{r_\tau\}_{\tau\in T^{SR}}, \{\mathsf{tpc}_\ell\}_{\ell\in L^{SR}})$ such that:*

- *$L^{SR} = L \cup \{\bot_\ell \mid \ell \in L\}$. The time progress conditions of locations $\bot_\ell$ and ℓ are equal to tpc_ℓ expressed on clock g.*
- *$X^{SR} = X_{p\in P} \cup \{\mathsf{tc}_p\}_{p\in P} \cup \{g_p\}_{p\in P} \cup \{\mathsf{tpc}\} \cup \{\widetilde{\mathsf{tc}}_p\}_{p\in P} \cup \{\widetilde{g}_p\}_{p\in P} \cup \{\rho_c\}_{c\in C} \cup \{t^{ex}\}$ where tc_p and $\widetilde{\mathsf{tc}}_p$ are exact and approximated timing constraint variables, g_p and $\widetilde{g}_p$ are exact and approximated Boolean variables, tpc is the exact time progress condition variable, ρ_c are clock reset date variables, and t^{ex} is the execution date variable.*
- *$\mathsf{C}^{SR} = \{g\}$*

- $P^{SR} = P \cup \{o\}$ *where* o *is a* send-port *and* $p \in P$ *is a* receive-port. *The port* o *is associated with the set of variables* $X_o^{SR} = X \cup \{\mathsf{tc}_p\}_{p\in P} \cup \{g_p\}_{p\in P} \cup \{\mathsf{tpc}\} \cup \{\widetilde{\mathsf{tc}}_p\}_{p\in P} \cup \{\widetilde{g}_p\}_{p\in P}$, *that is, variables needed to compute the current state, as well as the approximations of the next reached states. For all other ports* $p \in P$, *we have* $X_p^{SR} = X_p \cup \{t^{ex}\}$.

- *For each place* $\ell \in L$, *we include an intermediate place* $\perp_\ell$ *and an* offer *transition* $\tau_\ell = (\perp_\ell, o, \ell)$ *in* T^{SR}. *The guard* g_{τ_ℓ} *and the timing constraint* tc_{τ_ℓ} *are* `true`, *and the update function* f_{τ_ℓ} *is the identity function.*

- *For each transition* $\tau = (\ell, p, \ell') \in T$, *we include in* T^{SR} *a* notification *transition* $\tau_p = (\perp_\ell, p, \ell')$. *The guard* g_{τ_p} *is* `true`. *The function* f_{τ_p} *applies first the original update function* f_τ, *then updates the variables needed to compute the current state as follows:*

  - $\forall c \in r_\tau \quad \rho_c := t^{ex}$,
  - $\mathsf{tpc} := \mathsf{tpc}_{\ell'}$,
  - $\forall p' \in P \quad \mathsf{tc}_{p'} := \begin{cases} \mathsf{tc}_{\tau'} & \text{if } \tau' = (\ell', p', \ell'') \in T \\ \mathtt{false} & \text{otherwise.} \end{cases}$
  - $\forall p' \in P \quad g_{p'} := \begin{cases} g_{\tau'} & \text{if } \tau' = (\ell', p', \ell'') \in T \\ \mathtt{false} & \text{otherwise.} \end{cases}$

  and updates variables needed to approximate next reached state as follows.

  Let $T' = \{\tau' \mid \tau' = (\ell', p', \ell'') \in T\}$ *be the set of transitions enabled from location* ℓ'. *Let* $L'' = \{\ell'' \mid \tau' = (\ell', p', \ell'') \in T\}$ *be the set of locations reachable after executing a transition from the location* ℓ'. *For each port* $p'' \in P$, *let* $T_{p''} = \{\tau'' = (\ell'', p'', \ell''') \mid \ell'' \in L'' \wedge \tau'' \in T\}$ *be the set of possible transitions at location* $\ell'' \in L''$ *and labeled by port* p''. *For each port* $p'' \in P$, *the approximated timing constraints variables* $\mathsf{tc}^*_{p''}$ *and the approximated Boolean guards variables* $\widetilde{g}^*_{p''}$ *are updated as follows:*

  - $\forall p'' \in P \; \mathsf{tc}^*_{p''} := \begin{cases} \bigvee_{\tau'' \in T_{p''}} \widetilde{\mathsf{tc}}_{\tau''} & \text{if } T_{p''} \neq \emptyset \\ \mathtt{false} & \text{otherwise.} \end{cases}$
  - $\forall p'' \in P \; g^*_{p''} := \begin{cases} \bigvee_{\tau'' \in T_{p''}} \widetilde{g}_{\tau''} & \text{if } T_{p''} \neq \emptyset \\ \mathtt{false} & \text{otherwise.} \end{cases}$

In the above definition, the execution of a transition $\tau = (\ell, p, \ell')$ of a component B corresponds to the following two execution steps in B^{SR}. Firstly, an offer transition $\tau_\ell = (\perp_\ell, o, \ell)$ is used to transmit information about the current state of component B which is of two types::

- information needed by the scheduler to compute enabled interactions involving B, that is:

  - for each port $p' \in P$, the values of its variables $X_{p'}$. Note that the values of these variables are also needed for implementing transfer data functions.
  - for each port $p' \in P$, the exact timing constraint variable $\mathsf{tc}_{p'}$ corresponding to the enabledness of p' at ℓ with respect to time.
  - for each port $p' \in P$, the exact Boolean guard variable $g_{p'}$ corresponding to the enabledness of p' at ℓ with respect to the actual valuation of variables.
  - the exact time progress condition variable tpc corresponding to the current time progress condition at ℓ.

- information needed by the scheduler to restrict enabled interactions into safe ones, that is:
  - for port $p'' \in P$, the approximated timing constraint variable $\mathsf{tc}^*_{p''}$ corresponding to the potential enabledness of p'' reached after the execution of a single transition from ℓ. The timing constraint $\mathsf{tc}^*_{p''}$ corresponds to the disjunction of approximated timing constraints of transitions enabled from ℓ' and labeled by p''.
  - for port $p'' \in P$, the approximated Boolean guard variable $g^*_{p''}$ corresponding to the enabledness of p'' after the execution of a transition enabled from ℓ. The Boolean guard $g^*_{p''}$ corresponds to the disjunction of approximated Boolean guards of transitions enabled from ℓ' and labeled by p''.

Secondly, a response transition $\tau_p = (\ell, p, \perp_{\ell'})$ is executed once the scheduler decides to execute an interaction involving port p. Similar to τ in B, τ_p updates values of variables X according to f_τ. It also updates variables tpc, $\mathsf{tc}_{p'}$, $g_{p'}$, $\mathsf{tc}^*_{p''}$ and $g^*_{p'}$ to set up-to-date values for the next offer (i.e. starting from ℓ').

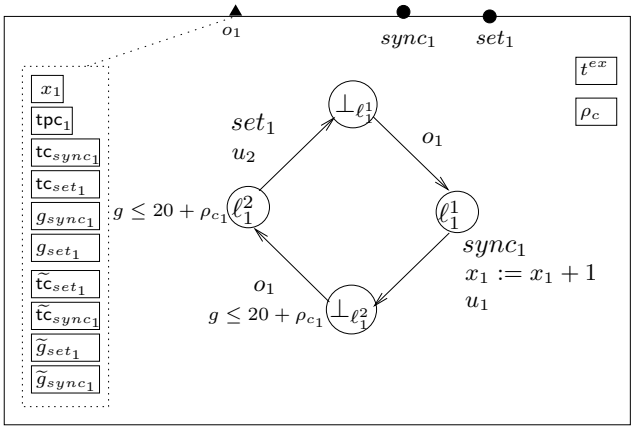


Figure 4.5: Send/Receive version of the $Setter_1$ atomic component from Figure 2.10.

Example 14. *Figure 4.5 shows the transformed version of the atomic component* $Setter_1$ *from Figure 2.10. For each original location* ℓ*, a busy control location* $\perp_\ell$ *has been added,*

with a transition labeled by the offer port o_1 from $\perp_\ell$ to ℓ. Before executing these transitions the update functions u_1 or u_2 are called. They are defined as follows:

$$u_1 = \begin{cases} \rho_{c_1} = t^{ex} \\ \mathtt{tpc}_1 := g \leq 20 + \rho_{c_1} \\ \mathtt{tc}_{sync_1} := \mathtt{false} \\ \mathtt{tc}_{set_1} := g \geq 10 + \rho_{c_1} \\ g_{sync_1} := \mathtt{false} \\ g_{set_1} := \mathtt{true} \\ \mathtt{tc}^*_{sync_1} := \mathtt{true} \\ \mathtt{tc}^*_{set_1} := \mathtt{false} \\ g^*_{sync_1} := \mathtt{true} \\ g^*_{set_1} := \mathtt{false} \end{cases}$$

$$u_2 = \begin{cases} \mathtt{tpc}_1 := \mathtt{true} \\ \mathtt{tc}_{sync_1} := \mathtt{true} \\ \mathtt{tc}_{set_1} := \mathtt{false} \\ g_{sync_1} := \mathtt{true} \\ g_{set_1} := \mathtt{false} \\ \mathtt{tc}^*_{sync_1} := \mathtt{false} \\ \mathtt{tc}^*_{set_1} := g \geq 10 + t^{ex} \\ g^*_{sync_1} := \mathtt{false} \\ g^*_{set_1} := \mathtt{true} \end{cases}$$

For instance, when reaching $\perp_{\ell_1^1}$, the variable $\mathtt{tc}_{sync_1}$ is set to the timing constraint of the transition enabled from ℓ_1^1 and labeled by $sync_1$. The variable $\mathtt{tc}_{set_1}$ is set to `false` *since there is no enabled transition labeled by port set_1 from location ℓ_1^1. Moreover, the variable $\mathtt{tpc}_1$ is set to the time progress condition of ℓ_1^1. Regarding approximations, we look at location ℓ_1^2 which is the only reached location from ℓ_1^1 in the atomic component $Setter_1$. We approximate the timing constraint $g \geq 10 + \rho_c$ of transition labeled by set_1. Since the clock c_1 is reset by the transition labeled by $sync_1$ and is involved in the lower bound of the timing constraint $g \geq 10 + \rho_c$, then, the variable $\mathtt{tc}^*_{set_1}$ is set to $g \geq 10 + t^{ex}$. The variable $\mathtt{tc}^*_{sync_1}$ is set to* `false` *since there is no enabled transition labeled by $sync_1$ from location ℓ_1^2.*

4.2.2 Building the Scheduler in BIP

We present now how to implement the scheduler component in BIP. The scheduler works with a partial view of the global state. Information about components states is given to the scheduler through offers. To maintain a safe approximation of system state at any moment, the scheduler uses the approximated values delivered by offers for components that are executing. Indeed, when the scheduler receives an offer from a Send/Receive component, its uses the variables corresponding to its current state until it executes. Once the scheduler notifies a component for executing on a selected port, it uses the approximated values of its last offer, until a new offer is received from the component informing about its new current state. Based on received offers, the scheduler computes enabled interactions and takes decision by either executing one interaction or waiting for receiving more offers.

The behavior of the scheduler is described as a Petri net in which there is a token for each component circulating between three different types of places as shown in Figure 4.6:

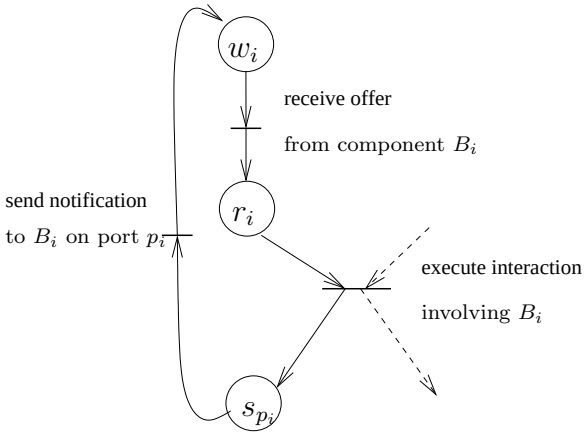


Figure 4.6: Circuit of a single token corresponding to component B.

- *waiting place*: there is one waiting place for each component. In Figure 4.6, the place labeled by w_i is a waiting place corresponding to component B_i. At this place the scheduler waits for a fresh offer from the corresponding component.

- *received place*: there is one received place for each component. In Figure 4.6, the place labeled by r_i is a received place corresponding to component B_i. Such place is marked once the scheduler has received a fresh offer from the corresponding component until an interaction involving that component is selected for execution.

- *sending place*: there is one sending place for each port of each component. In Figure 4.6, the place labeled by s_{p_i} is a sending place corresponding to port p_i of component B_i. At this place, the scheduler has selected an interaction and is willing to send a notification to one of the participating component in order to execute on the selected port.

Initially, tokens are in waiting places. Once an offer is received the corresponding token moves from the waiting place to the received place. The scheduler copies then the values of variables contained in the offer in its local variables. When all offers from components involved in an interaction have been received, the scheduler computes its timing constraint and its Boolean guard. If the guard evaluates to `true` and when the timing constraint becomes `true` with respect to current valuation of g, the scheduler executes the interaction and moves the tokens from received to sending places corresponding to the ports involved in the interaction. Finally, from the sending places, the scheduler sends notifications to the involved components and puts back tokens to waiting places.

Computing timing constraints and Boolean guards of interactions

Let $a = (\{p_i\}_{i\in I}, G_a, F_a)$ be an interaction. The scheduler computes the Boolean guard of a denoted by g_a as follows:

$$g_a = G_a \bigwedge_{p\in a} g_{p_i} \tag{4.3}$$

Where g_{p_i} are exact Boolean guard variables of ports p_i. That is, the Boolean guard of a corresponds to the conjunction of G_a with the Boolean guards of each port p_i participating in a.

It computes the timing constraint of a denoted by tc_a as follows:

$$\mathsf{tc}_a = \bigwedge_{p\in a} \mathsf{tc}_{p_i} \tag{4.4}$$

Where tc_{p_i} are exact timing constraint variables of port p_i. That is, the timing constraint of a corresponds to the conjunction of timing constraints of each port p_i participating in a.

In order to enable only safe execution of a, the scheduler removes from tc_a each date at which there is an interaction a' that has higher priority than a and is potentially enabled.

Let a' be an interaction such that $a\pi a'$. Interaction a' is enabled if $\mathsf{tc}_{a'} \wedge g_{a'}$, where $\mathsf{tc}_b = \bigwedge_{p_j\in b} \mathsf{tc}_{p_j}$ and $g_b = G_b \bigwedge_{p_j\in a} g_{p_j}$, evaluates to true. That is, if its timing constraint $\mathsf{tc}_{a'}$ and its Boolean guard $g_{a'}$ are true. The expression $\mathsf{tc}_{a'} \wedge g_{a'}$ could be seen as a timing constraint as the evaluation of $g_{a'}$ could be either `true` or `false` (these values are included in the grammar of timing constraints). The interaction a' is then disabled if the timing constraint $\neg(\mathsf{tc}_{a'} \wedge g_{a'})$ is true.

We define K_a the timing constraint at which a is allowed to execute according to the priority. Informally, K_a is a timing constraint containing dates at which each interaction a' having higher priority than a is disabled. Formally, K_a is defined as follows:

$$K_a = \bigwedge_{a\pi a'} \neg(\mathsf{tc}_{a'} \wedge g_{a'}) \tag{4.5}$$

The computation of K_a requires receiving fresh offers from each component participating in each interaction b such that $a\pi b$. As offers contains approximations of reached states, the scheduler may over-approximate K_a when the offers of one or more components are not fresh. We denote by K_a^* the over-approximation of K_a. For offers of components that are not fresh, the scheduler uses the approximated values when computing K_a^*. To keep track of the freshness of offers, we include in the scheduler a Boolean variable rcv_i associated with components B_i indicating whether offers corresponding to components are fresh. The variable rcv_i is useful for the scheduler to decide whether it uses the approximated or the exact values received in the last offer of component B_i. More precisely if rcv_i is evaluated to true, then the scheduler uses the exact values received in the last offer. Otherwise, it uses approximations. The variable rcv_i is updated as follows: it is set to `true` whenever the scheduler receives an offer from the component B_i and set to `false` whenever the scheduler "consumes" the offer by executing an interaction involving component B_i.

K_a^* is obtained from K_a by replacing each occurrence of a timing constraint tc_{p_j} or a Boolean guard g_{p_j} by either its exact value or its approximation depending on the freshness

of the corresponding offer . We denote by $\mathsf{tc}_{p_j}^+$ the occurrence of the timing constraint tc_{p_j}, $g_{p_j}^+$ the occurrence of Boolean guard g_{p_j} in K_a. The value of $\mathsf{tc}_{p_j}^+$ and $g_{p_j}^+$ are computed as follows:

$$\mathsf{tc}_{p_j}^+ = \begin{cases} \mathsf{tc}_{p_j} & \text{if } \ rcv_j = \texttt{true} \\ \\ \mathsf{tc}_{p_j}^* & \text{otherwise.} \end{cases}$$

$$g_{p_j}^+ = \begin{cases} g_{p_j} & \text{if } \ rcv_j = \texttt{true} \\ \\ g_{p_j}^* & \text{otherwise.} \end{cases}$$

For the Boolean guard $G_{a'}$, it can be computed only if all offers from components participating in a' are fresh. We approximate it to `true` if one of these offers are not fresh. We denote by $G_{a'}^+$ the occurrence of $G_{a'}$ in K_a. It is computed as follows:

$$G_{a'}^+ = \begin{cases} G_{a'} & \text{if } \ \forall \ \ p_j \in a' \ \ rcv_j = \texttt{true} \\ \\ \texttt{true} & \text{otherwise.} \end{cases}$$

To sum up, K_a^* has the following expression:

$$K_a^* = \bigwedge_{a\pi a'} \neg(\bigwedge_{p_j \in a'} \mathsf{tc}_{p_j}^+ \wedge G_{a'}^+ \bigwedge_{p_j \in a} g_{p_j}^+) \tag{4.6}$$

To enable only safe execution of interaction a, the scheduler restricts tc_a to $\mathsf{tc}_a \wedge K_a^*$. The timing constraint $\mathsf{tc}_a \wedge K_a^*$ contains dates at which a is enabled and safe to execute.

We are now ready to define the centralized scheduler.

Definition 24. *Let $\pi\gamma(B_1, \ldots, B_n)$ be a BIP model. The centralized scheduler for this model is defined as the Send/Receive BIP component $S = (L^S, P^S, T^S, C^S, X^S, \{X_p^S\}_{p\in P^S}, \{g_\tau\}_{\tau\in T^S}, \{\mathsf{tc}_\tau\}_{\tau\in T^S}, \{f_\tau\}_{\tau\in T^S}, \{r_\tau\}_{\tau\in T^S}, \{\mathsf{tpc}_\ell\}_{\ell\in L^S})$ where:*

- *The set of variables X^S contains a copy of each variable exported by an offer port. They consist of the following.*
  - *For each component B_i, we include an exact time progress condition variable tpc_i.*
  - *For each port p, we include an exact and approximated timing constraint variables tc_p and tc_p^**
  - *For each port p, we include an exact and approximated Boolean variables g_p and g_p^**
  - *For each port p we include a copy of each variable x_p exported by p.*

  X^S includes also the following additional variables.
  - *For each component B_i, we include a Boolean variable rcv_i that indicates whether r_i place is marked or not (that is, the offer from component B_i is fresh or not).*

- *For each interaction $a \in \gamma$, we include an execution date variable t_a^{ex} that stores the execution date of the last occurrence of interaction a.*
- *For each component B_i, we include an execution date variable t_i^{ex} that stores the last execution date of component B_i.*

- *The set of places L^S consists of the following.*
  - *For each component B_i, we include* waiting *and* received *places w_i and r_i, respectively. WThe place w_i does not have time progress condition (i.e. it defaults to true). The time progress condition of place r_i is tpc_i. That is, after receiving an offer from component B_i, we require that the scheduler executes an interaction involving component B_i before its current time progress condition becomes false.*
  - *For each port $p \in P$ where P is the set of all ports exported by $B_1, \ldots, B_n$, we include a* sending *place s_p. The notification to an offer of a component B_i^{SR} is sent from this place to port p of B_i^{SR}. The time progress condition of s_p is* `false`*. That is, when an interaction is selected for execution, we require that the scheduler notifies the participating components immediately (without any delay).*
- $\mathsf{C}^S = \{g\}$.
- *The set of ports P^S is the following.*
  - *For each component B_i, we include an offer port o_i to receive offers from B_i. Each port o_i is associated with variables X_{o_i}.*
  - *For each port p of a component B_i, we include a send-port p. The variables associated with port p are $X_p^S = \{X_p\} \cup \{t_i^{ex}\}$ where i is the index of component B_i that includes port p. The port p is used to notify component B_i^{SR} to execute on port p.*
  - *For each interaction $a \in \gamma$, we include a unary port. Unary ports do not have any associated variables.*

  The set of transitions consists of the following.
  - *In order to receive offers from a component B_i, we include an offer transition (w_i, o_i, r_i). This transition has no guard nor timing constraint (i.e. they default to* `true`*). The update function of this transition sets the variable rcv_i to* `true`*.*
  - *For each port $p \in P$ of component B_i, we include a transition (s_p, p, w_i). This transition notifies the corresponding component to execute the transition labeled by p. This transition has no guard, no timing constraint and no update function.*
  - *For each interaction $a = (P_a, G_a, F_a) \in \gamma$, we include the transition $\tau_a = (\{r_i \mid B_i \in part(a)\}, a, \{s_{p_i} \mid p_i \in a\})$. This transition has a Boolean guard $g_{\tau_a} = g_a$ where g_a is computed as in (4.3). The timing constraint of τ_a is:*

$$\mathsf{tc}_{\tau_a} = \mathsf{tc}_a \wedge K_a^*$$

  where tc_a is computed as in (4.4) and K_a^* is computed as in (4.6).The associated update function f_{τ_a} applies first the following updates:

$*$ $t_a^{ex} := t(g)$,
$*$ $\forall$ $B_i \in part(a)$, $t_i^{ex} := t_a^{ex}$,
$*$ $\forall$ $B_i \in part(a)$, $rcv_i := \texttt{false}$,

and then applies the the data transfer function F_a. The execution of τ_a moves the tokens from receiving to sending places.

Note that when an interaction a is executed, the current valuation of g is assigned to variable t_a^{ex}, i.e. the scheduler performs $t_a^{ex} := t(g)$. This value is sent by the scheduler to the components involved in a along with notifications for execution. By construction, the component involved in a executes at the global date t_a^{ex} and the notification triggers the execution in the components at that date which ensures synchronization of components. In the implementation, we require that these models are implemented on platforms that provide fast communications (e.g. multi-process) to ensure perfect synchronization of components when the scheduler notifies them for execution.

Example 15. *Figure 4.7 presents the Petri net of the scheduler component of Figure 4.1. The execution of transition labeled o_i, $i \in \{1,2,3\}$ indicates that an offer from component B_i has been received. This transition updates the Boolean variable rcv_i to* `true`.

Consider the interaction $a_1 = (\{sync_1, sync_2, sync_3\}, \texttt{true}, -)$. In Figure 4.1, the execution of the transition labeled a_1 indicates the execution of interaction a_1. The execution of this transition requires that offers from components B_1, B_2 and B_3 have been received. The timing constraint of this transition is:

$$\mathsf{tc}_{\tau_{a_1}} = \mathsf{tc}_{sync_1} \wedge \mathsf{tc}_{sync_2} \wedge \mathsf{tc}_{sync_3}.$$

The interaction a_2 has originally less priority than a_3. Therefore, the timing constraint of transition a_2 is restricted by disabling a_2 when a_3 is potentially enabled (i.e. when both ports set_2 and get_2 are potentially enabled). That is,

$$\mathsf{tc}_{\tau_{a_2}} = \mathsf{tc}_{set_1} \wedge \mathsf{tc}_{get_1} \wedge \neg(\mathsf{tc}^+_{set_2} \wedge \mathsf{tc}^+_{get_2}).$$

4.2.3 Connections Between Send/Receive Atomic Components and the Scheduler

In this section, we define the Send/Receive interactions between the transformed atomic components and the scheduler. Given a component B and a port p, we denote by $B.p$ the port p involved in component B.

Definition 25. *Given a BIP model $\pi\gamma(B_1, ..., B_n)$, we define a Send/Receive model $\gamma^{SR}(B_1^{SR},...,B_n^{SR}, S)$ where B_i^{SR}, $i \in \{1, \ldots, n\}$ are Send/Receive atomic components, S is the scheduler and γ^{SR} is the set of Send/Receive interactions defined as follows:*

- *For each Send/Receive component B_i^{SR}, we include in γ^{SR} the Send/Receive interaction $(B_i^{SR}.o_i, S.o_i)$ where $B_i^{SR}.o_i$ is a send port and $S.o_i$ is a receive port.*

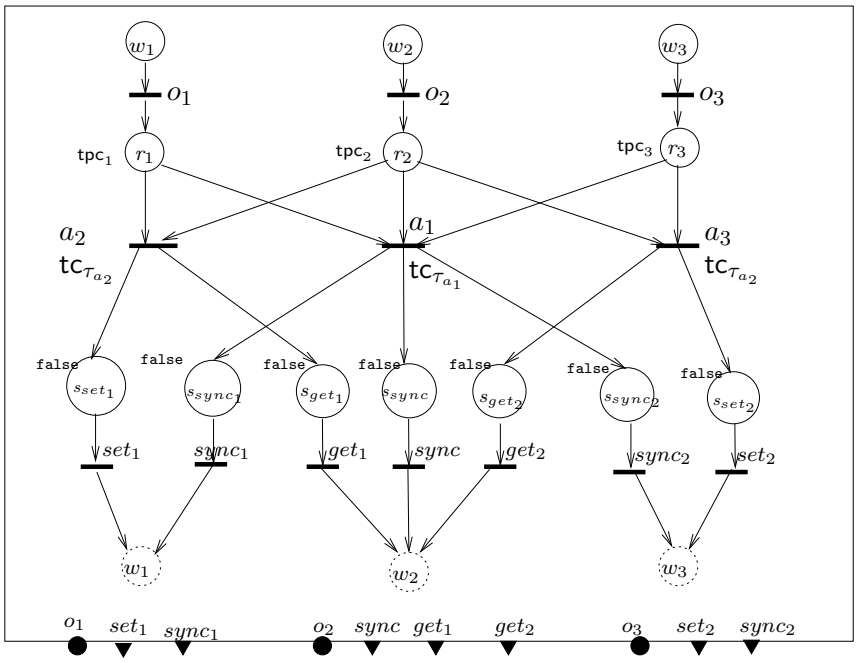


Figure 4.7: Petri net of the scheduler component of Figure 4.1.

- *For each port $p \in P$ of a component B_i^{SR}, we include in γ^{SR} the Send/Receive interaction $(S.p, B_i^{SR}.p)$ where $S.p$ is a send port and $B_i^{SR}.p$ is a receive port.*
- *For each interaction $a \in \gamma$, we add in γ^{SR} the unary interaction $(S.a)$ where $S.a$ is a unary port.*

Example 16. *The Send/Receive interactions of the Send/Receive model of Figure 4.1 link each send port with its corresponding receive port.*

4.3 Correctness

In this section we show that the obtained Send/Receive model is observationally equivalent to the original model. First, we show that the transformed model meet the properties of Definition 22. Then we prove observational equivalence of the original and the transformed model.

Validity of the Transformed Model

We need to show that when a receive-port of $\gamma^{SR}(B_1^{SR}, ..., B_n^{SR}, S)$ is enabled, the corresponding send-port is also enabled. This holds since communications between atomic components and the scheduler follow a request/acknowledgement pattern. Whenever an atomic component sends an offer, it enables the receive-port to receive a response and no new offer is sent until the first one is acknowledged.

Lemma 1. *Given a BIP model $\pi\gamma(B_1,...,B_n)$, the model $\gamma^{SR}(B_1^{SR},...,B_n^{SR},S)$ obtained by transformation of Section 4.2 meets the properties of Definition 22.*

Proof. The first two constraints of Definition 22 are trivially met by construction. This is because (1) each interaction has only one send-port and multiple receive-ports, and (2) each send-port is associated with one and only Send/Receive interaction.

We now prove that the third constraint also holds; i.e, whenever a send-port is enabled, all its associated receive-ports are enabled as well. Between components and the scheduler, for all interactions involving a component B_i, we distinguish between 3 classes of states:

- The first class contains all states where all the places w_i in the scheduler component contain a token, and B_i^{SR} is in a busy location $\perp_\ell$. This class contains the initial state. From that class, the only enabled send-port involved in an interaction with B_i^{SR} is the port o_i. By definition of the class, all associated receive-ports are also enabled, and the Send/Receive interaction can take place to reach a state of the second class.

- In the second class, the component B_i^{SR} is in a place ℓ that is not a busy location, and in the scheduler component, the r_i place contains a token. From that configuration, there is no enabled send-port involved in an interaction with B_i^{SR}. The next class of states is reached when the scheduler executes a transition corresponding to an interaction involving B_i.

- In the remaining one, there is a token in a place s_p where p is a port of B_i. The port p of this scheduler is enabled. By construction, the component B_i^{SR} sends an offer from $\perp_\ell$ for port p only if the receive port p is enabled from place ℓ in B_i^{SR}. Thus the Send/Receive interaction can take place to reach back the first class of states.

□

The proof of Lemma 1 ensures that any component ready to perform a transition labeled by a send-port will not be blocked by waiting for the corresponding receive-ports. In other terms, it proves that any Send/Receive interaction is initiated by the sender.

Observational Equivalence between Original and Transformed BIP Models

In this subsection, our goal is to show that $\gamma(B_1,...,B_n)$ and $\gamma^{SR}(B_1^{SR},...,B_n^{SR},S)$ are observationally equivalent. We consider the correspondence between actions of $\gamma(B_1,...,B_n)$ and $\gamma^{SR}(B_1^{SR},...,B_n^{SR},S)$ as follows. To each discrete step $a \in \gamma$ of $\pi\gamma(B_1,...,B_n)$, we associate the unary interaction a of $\gamma^{SR}(B_1^{SR},...,B_n^{SR},S)$. To each delay transition $\delta \in \mathbb{Z}_{\geq 0}$ of $\pi\gamma(B_1,...,B_n)$, we associate the same delay transition δ of $\gamma^{SR}(B_1^{SR},...,B_n^{SR},S)$. All other interactions of $\gamma^{SR}(B_1^{SR},...,B_n^{SR},S)$ (i.e., offer and notification) are unobservable and denoted by β.

We proceed as follows to complete the proof of observational equivalence. We denote by q^{SR} a state of $\gamma^{SR}(B_1^{SR},...,B_n^{SR},S)$ and q a state of $\pi\gamma(B_1,...,B_n)$. A state of $\gamma^{SR}(B_1^{SR},...,B_n^{SR},S)$ from where no β transition is possible is called a *stable state*, in the sense that any β transition from this state does not change the state of the atomic components layer.

Lemma 2. *From any state q^{SR}, there exists a unique stable state $[q^{SR}]$ such that $q^{SR} \xrightarrow{\beta^*} [q^{SR}]$.*

Proof. The state $[q^{SR}]$ exists since each Send/Receive component B_i^{SR} can do at most two β transitions: receive a response and send an offer. Since two β transitions involving two different components are independent (i.e. do not change the same variable or the same place), the same final state is reached independently of the order of execution of β actions. Thus $[q^{SR}]$ is unique. □

The above lemma proves the existence of a well-defined stable state for any of the transient states reachable by the Send/Receive model $\gamma^{SR}(B_1^{SR}, ..., B_n^{SR}, S)$. This stable state will be used later to define our observational equivalence. Furthermore, combining this lemma with Lemma 1, we obtain the following property.

Lemma 3. *At a stable state $[q^{SR}]$, the Send/Receive model verifies the following properties.*

- *All atomic components are in a non busy place ℓ.*
- *All tokens in the scheduler component are in receive places r_i.*
- *The clock g and all variables in the atomic components have the same value than their copies in the scheduler component.*

Proof. The two first points comes from the Lemma 1 that guarantees possible execution of a Send/Receive interaction if its send-port is enabled. Therefore no place s_p in the scheduler component (respectively $\perp_\ell$ in atomic components) can be active at $[q^{SR}]$, otherwise the answer p (respectively the offer from $\perp_\ell$) could occur. Furthermore, since all offers have been sent, no token can be in a w_i place.

The last point can be proven as follows. From each variable x of the scheduler, the last modifying transition is the offer transition from the corresponding atomic component B_i. Thus, the value of x in the scheduler is the same in the atomic components that sent the offer modifying x. The clock g has the same value in the atomic components and the scheduler as it corresponds to a global clock which is never reset.

We are now ready to state and prove our central result.

Theorem 2. $\gamma^{SR}(B_1^{SR}, ..., B_n^{SR}, S) \sim \pi\gamma(B_1, ..., B_n)$.

Proof. We define a relation between the set of states Q^{SR} of $\gamma^{SR}(B_1^{SR}, ..., B_n^{SR}, S)$ and the set of states Q of $\pi\gamma(B_1, ..., B_n)$ as follows. For each state $q^{SR} \in Q^{SR}$, we build an equivalent state $equ(q^{SR})$ by

1. considering the unique stable state $[q^{SR}]$ reachable by doing β transitions,
2. taking the control location ℓ of B_i^{SR} as control location for B_i in $equ(q^{SR})$ (Lemma 3 ensures that it is a valid control state for B_i),
3. restricting the valuation of variables of B_i^{SR} to a valuation of variables in B_i, and
4. talking the valuation of original clock c_i in B_i to the valuation of $g - \rho_{c_i}$.

We then define the equivalence relation R by taking:

$$R = \{(q^{SR}, q) \in Q^{SR} \times Q \mid q = equ(q^{SR})\}$$

The three next assertions prove that R is a weak bisimulation:

(i) If $(q^{SR}, q) \in R$ and $q^{SR} \xrightarrow{\beta}_{\gamma^{SR}} r^{SR}$ then $(r^{SR}, q) \in R$.

(ii) If $(q^{SR}, q) \in R$ and $q^{SR} \xrightarrow{\sigma}_{\gamma^{SR}} r^{SR}$ then $\exists r \in Q : q \xrightarrow{a} r$ and $(r^{SR}, r) \in R$.

(iii) If $(q^{SR}, q) \in R$ and $q \xrightarrow{\sigma}_{\pi} r$ then $\exists r^{SR} \in Q^{SR} : q^{SR} \xrightarrow{\beta^*\sigma} r^{SR}$ and $(r^{SR}, r) \in R$.

(i) If $q^{SR} \xrightarrow{\beta}_{\gamma^{SR}} r^{SR}$, then $[q^{SR}] = [r^{SR}]$, and we have by definition $equ(q^{SR}) = equ(r^{SR})$.

(ii) The action σ in $\gamma^{SR}(B_1^{SR}, ..., B_n^{SR}, S)$ is either an interaction a or a delay step δ.

- If a is an interaction, it corresponds to executing a transition labeled by a unary port a in the scheduler. By construction of the scheduler, this transition has the Boolean guard $g_a = G_a \bigwedge_{p \in a} g_p$ where g_p are Boolean guards sent by the atomic components for each port $p \in a$. By Lemma 3, the values involved in g_a are the same in atomic components, and by extension in $q = equ(q^{SR})$. Thus, the Boolean guard of G_a and all the Boolean guard of transitions labeled by $p \in a$ from state q are true. The transition labeled by a is the scheduler has the timing constraint $\mathsf{tc}_a \wedge K_a^*$.
  - The timing constraint tc_a is defined by $\mathsf{tc}_a = \bigwedge_{p \in a} \mathsf{tc}_p$ where tc_p are timing constraints sent by the atomic components for each port $p \in a$. By Lemma 3, the values involved in tc_a are the same in atomic components, and by extension in $q = equ(q^{SR})$. By construction of the atomic components, the timing constraints sent from $\perp_\ell$ in B_i^{SR} are timing constraints at state ℓ in B_i. Therefore, the timing constraints tc_a is met at state q.
  - The timing constraint $K_a^* = \bigwedge_{a \pi a'} \neg(\mathsf{tc}_{a'}^* \wedge g_{a'}^*)$ where the evaluation of $\mathsf{tc}_{a'}^*$ and $g_{a'}^*$ at state q^{SR} are an over approximated evaluation of $\mathsf{tc}_{a'}$ and $g_{a'}$ at state $[q^{SR}]$. That is, if $\neg(\mathsf{tc}_{a'}^* \wedge g_{a'}^*)$ evaluates to true at state q^{SR}, then $\neg(\mathsf{tc}_{a'} \wedge g_{a'})$ evaluates to true at state $[q^{SR}]$. Recall that, the timing constraint $\neg(\mathsf{tc}_{a'} \wedge g_{a'})$ characterizes dates where interaction a' is disabled at $[q^{SR}]$. By Lemma 3, the values involved in $\neg(\mathsf{tc}_{a'} \wedge g_{a'})$ are the same in atomic components, and by extension in $q = equ(q^{SR})$.

  The last two points state that $equ(q^{SR}) \xrightarrow{a}_{\pi} r$. Finally, executing a in B^{SR} triggers the execution of the data transfer function F_a, followed by the computation in atomic components upon reception of the response. Thus at $[r^{SR}]$ the values in atomic components are the same as in r, which yields $(r^{SR}, r) \in R$.
- If a is a delay step, it corresponds to letting time progress by either in busy locations $\perp_{\ell_i}$ or in ℓ_i of atomic components B_i^{SR}. Location $\perp_{\ell_i}$ and ℓ_i has the time progress condition tpc_{ℓ_i}. All these time progress conditions are not false, otherwise the δ delay step would not be allowed. Those time pogress conditions

are expressed on clock g which could be expressed equiventely on clocks c_i of component B_i Therefore $q \xrightarrow{\delta} r$. Executing δ from state q^{SR} increases clock g by δ which has the same effect on the clocks of the original model, therefore $(r^{SR}, r) \in R$.

(iii) If σ can be executed in $\pi\gamma(B_1, \ldots, B_n)$ at state q, then from an equivalent state q^{SR}, one can reach the state $[q^{SR}]$ where the state, the valuation of clocks, and data of send/receive atomic atomic components coincide with those of q. Recall clocks c of the original model could be deduced from clock g as follows $c = g - \rho_c$ By Lemma 3, clocks, ports, timing constraints and data values are the same in atomic components and the scheduler. Furthermore, all r_i places are active. As previously, we distinguish the cases where σ is an interaciton a or a delay step δ.

- If σ is a an interaction a from q, then (1) the timing constraint and guard of a are true and (2) for each interaction a' such that $a\pi a'$ is not enabled from q thus $K_a = \bigwedge_{a\pi a'} \neg(\mathsf{tc}_{a'} \wedge g_{a'})$ is evaluated to true. As in the previous case, Lemma 3 ensures that if the interaction a is possible in the original model at state $q = equ(q^{SR})$, then the transition a is also possible in the scheduler at state $[q^{SR}]$. Therefore we have $q^{SR} \xrightarrow{\beta^*} [q^{SR}] \xrightarrow{a} r^{SR}$. As previously, the execution of a in both models leads to equivalent states. Thus we have $q^{SR} \xrightarrow{\beta^*} [q^{SR}] \xrightarrow{a} r^{SR}$ with $(r^{SR}, r) \in R$.
- If σ is a delay step δ, then from state $[q^{SR}]$, time can progress also by δ as at this state all places r_i are marked and have the time progress condition tpc_i sent by B_i^{SR} which is the same as the time progress condition of B_i at q. Also, at state $[q^{SR}]$, each atomic component B_i^{SR} is in location $\perp_{\ell_i}$ which has the time pogress condition tpc_{ℓ_i}. Thus we have $q^{SR} \xrightarrow{\beta^*} [q^{SR}] \xrightarrow{\delta} r^{SR}$ with $(r^{SR}, r) \in R$.

□

4.4 Conclusion

In this chapter, we proposed a method for building Send/Receive models from given BIP models using model-to-model transformations. The obtained models consists of transformed atomic components of original models with a centralized scheduler. The scheduler component is modeled such that the decision for scheduling an interaction is taken at the date of its execution. In other words, it implicitly assumes that there is no delay between the decision to schedule an interaction and its execution in the corresponding components. Such an assumption is only valid if the communication delays between the scheduler and the components are small enough to be neglected. Therefore, we assume that those models are implemented on platforms providing fast communications (e.g multi process platforms).

The solution based on a centralized scheduler allows parallelism between components. However, it restricts the parallelism between interactions. In the next chapter, we present a method to decentralize the scheduler. The idea is to split the scheduler into a set of

schedulers, each one handling a subset of interactions. We show that such decentralization creates conflicts, that need a conflict resolution protocol to be resolved.

5

Decentralizing the Scheduler

In the previous chapter, we proposed Send/Receive models relying on centralized scheduler for interactions execution. Those models allow parallelism between components. However, parallelism between interactions is restricted as the scheduler is defined as a single entity (component).

In this chapter, we propose a method for decentralizing the scheduler into a set of decentralized schedulers. Each scheduler is responsible of executing a subset of interactions. Thus, our decentralization method is parametrized by an interactions partition. Each partition class defines a set of interactions handled in a dedicated scheduler.

Introducing concurrency between interactions by adding schedulers introduces conflict between schedulers. A conflict between schedulers occurs when they try to execute multiple interactions involving the same component. A solution to avoid such conflict is to provide a particular partition of interactions consisting in grouping all conflicting interactions together in the same class as explained in Section 5.1. The obtained schedulers are then conflict-free by construction. For the more general case of partition, we propose in Section 5.3 a solution to resolve dynamically conflicts between interactions. The solution is based on the incorporation of a conflict resolution protocol. More precisely, we refer to the three different conflict resolution protocols proposed by Bagrodia [60].

The Send/Receive models are dedicated to be implemented on platforms that provide fast communications (e.g. multi-process platforms) as we assume that communication latencies are negligible so that there is no delay between the decision to execute an interaction in a scheduler and its execution in the participating components. Such assumption was considered also in the previous chapter. In the next Chapter, we will see how to build Send/Receive models that deals with communications delays.

Model Restrictions

We consider input models that meet the restrictions of the previous chapter. In addition, we also consider models that have no priorities. In this chapter, we consider the example from Figure 5.1 as running example. It corresponds to the same example of Figure 2.10 where

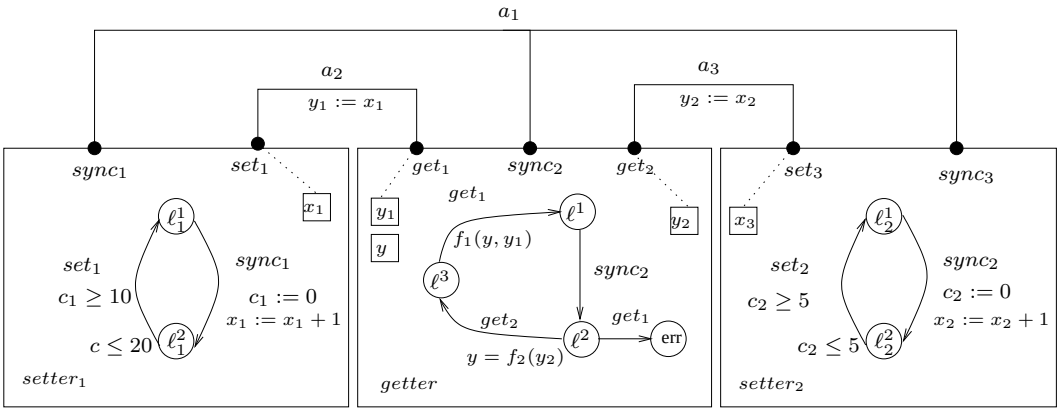


Figure 5.1: Example of BIP model meeting all restrictions.

the priority rule $a_2 \pi a_3$ is not considered. Note that both examples from Figure 2.10 and Figure 5.1 have the same execution sequences.

5.1 Conflicting Interactions

Intuitively, 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 discards the enabledness of the other). In systems without priorities, two interactions may conflict only if they involve a shared component. Figure 5.2 depicts examples of two conflicting interactions a and b. In Figure 5.2 (a), the conflict comes from the fact that a and b involve two ports p and q of the same component labeling two transitions enabled from the same location ℓ. When reaching location ℓ, the component can execute either transition labeled by p or the one labeled by q but not both. This implies that when a and b are enabled, only one of them should execute. Figure 5.2 (b) depicts a special case of conflict where interactions a and b share a common port p.

Below, we define formally conflicting interactions. Recall that we denote by $part(a)$ the set of components participating in interaction a.

Definition 26. *Let $\gamma(B_1, \ldots, B_n)$ be a BIP model. We say that two interactions a and b of γ are* conflicting *denoted by $a\#b$, iff , there exists an atomic component $B_i \in part(a) \cap part(b)$ that has two transitions $\tau_1 = (\ell, p, \ell'_1)$ and $\tau_2 = (\ell, q, \ell'_2)$ from the same control location ℓ such that $p \in a$ and $q \in b$, if a and b are not conflicting we say that they are conflict-free.*

Note that conflicts as defined in Definition 26 are an over approximation of conflicts since some conflicts may not be reachable due to system dynamics.

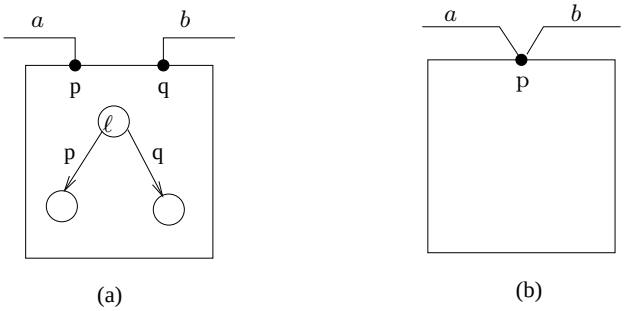


Figure 5.2: Conflicting interactions.

Consider the example from Figure 5.1. Interaction a_1 is conflicting with neither a_2 nor a_3 because a_1 is enabled from a state where a_2 ad a_3 are disabled. However, a_2 and a_3 are conflicting because port get_1 involved in a_2 and port get_2 involved in a_3 are both enabled from place ℓ_2^2 of component *getter*.

5.2 Interactions Partitioning

As said before, the decentralization is parametrized by a partition of interactions. Below, we define formally partition of interactions.

Definition 27. *Given a BIP model $\gamma(B_1, \ldots, B_n)$, a partition of γ is a set of subsets $\{\gamma_k\}_{k=1}^m$ such that :*

- $\gamma = \gamma_1 \cup \ldots \cup \gamma_m$ *and*
- $\forall i, j \in \{1, \ldots, m\}$ *such that* $i \neq j$ *we have* $\gamma_i \cap \gamma_j = \emptyset$.

The partition $\{\gamma_k\}_{k=1}^m$ is conflict-free if each for any $i \neq j$, no interaction of γ_i is conflicting with an interaction of γ_j.

Given a partition of interactions $\{\gamma_j\}_{j=1}^m$, and two conflicting interactions a and b such that $a \in \gamma_j$ and $b \in \gamma_k$, we distinguish between two types of conflict for a and b, according to the partition $\{\gamma_j\}_{j=1}^m$.

- A conflict is *internal* if a and b belong to the same class of the partition, i.e. $j = k$. In this case, we say that a and b are internally conflicting .
- A conflict is *external* if a and b belong to different classes of the partition, i.e. $j \neq k$. In this case, we say that a and b are externally conflicting.

Consider a BIP model $\gamma(B_1 \cdots B_n)$ and a partition of interactions $\{\gamma_j\}_{j=1}^m$. Each class of interactions γ_j is handled by a single scheduler S_j. Introducing concurrency by adding multiple schedulers could violate the BIP semantics due to conflicting interactions as explained as follows. Consider two different schedulers for handling interactions a and b of the model from Figure 5.2 (a), S_1 being responsible for interaction a and S_2 being responsible for interaction

b (see Figure 5.3). In the obtained Send/Receive model, the shared component is required to send offers to both S_1 and S_2. In order to respect the BIP semantics, these schedulers should ensure that only a or b is executed from a state enabling both a and b.. That is, at most one of the two schedulers may respond to an offer from the shared component.

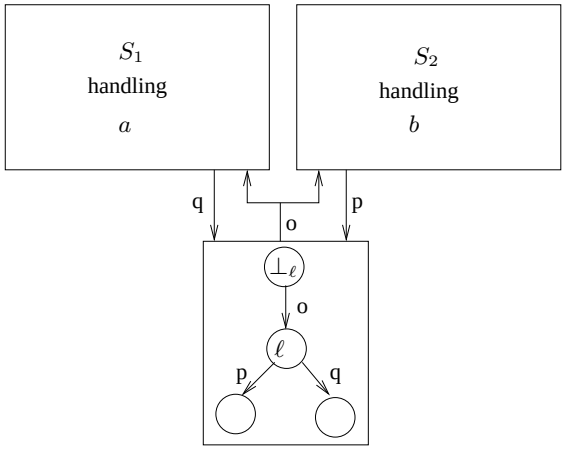


Figure 5.3: Issue of conflicting interactions when handled in different schedulers.

A solution to avoid the problem of conflicting interactions is to impose them to be internally conflicting. That is, two conflicting interactions are never handled by two different schedulers. The obtained model is called conflict-free. In this case, conflicts are resolved locally by schedulers as in the centralized scheduler described in the previous chapter. Indeed, when two conflicting interactions belong to the same scheduler, the corresponding transitions of its Petri net share a common input place (corresponding to received place of shared component). Thanks to the Petri net semantics, when one interaction transition executes, it consumes the shared input place and disables the other interaction transition. This ensures that, from a given state, at most one interaction is selected amongst a set of conflicting interactions. Obtaining conflict-free Send/Receive model depends on the input partition of interactions. A version of this solution has been presented in [78].

Consider again the example from Figure 5.1. In order to obtain a conflict-free partition, we group a_2 and a_3 in the same class of interactions as these interactions are conflicting. Interaction a_1 can be separated in an other class of interactions as a_1 is conflict-free with a_2 and a_3. Thus, we obtain the following conflict-free partition: $\gamma_1 = \{a_1\}$ and $\gamma_2 = \{a_2, a_3\}$.

We do not detail here the construction of Send/Receive BIP models with conflict-free schedulers. However, we give in Figure 5.4 an example of Send/Receive BIP model of the example from Figure 5.1 obtained from the conflict-free partition $\gamma_1 = \{a_1\}$, $\gamma_2 = \{a_2, a_3\}$. This version contains two schedulers: S_1 for handling interaction in γ_1 and S_2 for handling interactions in γ_2. The three components send offers to both schedulers since each one participates in one interaction handled by each scheduler.

Generally, conflict-free partition is not the optimal choice for decentralizing BIP models. Indeed, one may end up with a centralized scheduler if the BIP model has a chain of conflicting interactions. Therefore, a general method should incorporate a conflict resolution protocol.

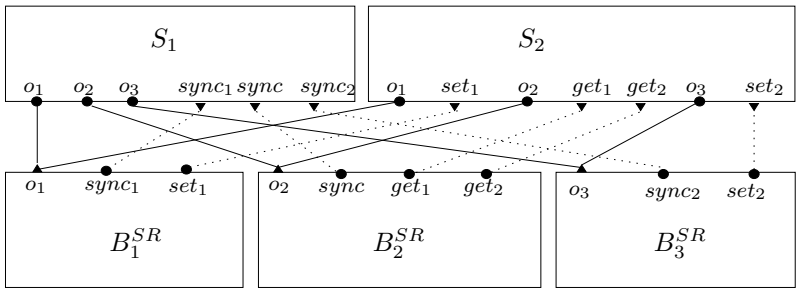


Figure 5.4: Example of the architecture of Send/Receive BIP model obtained from a conflict-free partition for the example from Figure 5.1.

In the following, we present our general method for decentralizing BIP models.

5.3 3-Layer Send/Receive Models

Our target Send/Receive model is based on three layers. The first layer contains Send/Receive atomic components which are obtained by transforming original atomic components defined in the BIP model. The second layer contains schedulers, each one handling a class of interactions defined by the partition of interactions. The third layer contains conflict resolution protocol components responsible for conflict resolution.

Similarly to what we proposed in the previous chapter, Send/Receive atomic components communicate with schedulers through offers. The schedulers notify the components when they are part of an interaction selected for execution. Conflicts between two interactions of the same scheduler (i.e. of the same class in the considered partition of interactions) are resolved locally in the scheduler as in the case of centralized scheduler. Conflicts between interactions handled by two different schedulers are arbitrated outside the schedulers in the conflict resolution protocol to decide their execution. That is, before executing interactions involved in such conflicts, schedulers send requests to the conflict resolution protocol. The latter decides to grant or to reject the execution of the requested interaction depending on whether conflicting interactions have already been executed or not.

The conflict resolution protocol is based on the idea of message-count technique presented in[60]. This technique is based on counting the number of times that a component participates in an interaction. Conflicts are then resolved by ensuring that each participation number is used only once. In Send/Receive models, components send new offers to schedulers each time they participate in an interaction. Counting the number of times a component participates in interactions bowls down to counting its offers. By numbering offers of the component, conflict resolution is simply achieved by comparing the offer numbers of participating components with numbers of their last execution.

From the implementation point of view, we use an integer variable n_i called participation number to count offers in each Send/Receive component B_i^{SR}. The participation number of a component B_i^{SR} is included in its offers, and is incremented whenever B_i^{SR} receives notification for execution from a scheduler, that is, whenever B_i^{SR} participates in an interaction.

Participation numbers are included in requests send by schedulers so that reservation

components can arbiter conflicts. As already said, reservation requests are sent by a scheduler to the conflict resolution protocol only for interactions conflicting with other interactions not handled by the same scheduler. The conflict resolution protocol has variables N_i that store the last value of the participation number of each component B_i^{SR}. A reservation request for an interaction a contains the participation numbers n_i of each component B_i^{SR} participating in a. If for each component B_i^{SR} the participation number n_i is greater than N_i, then the conflict resolution protocol grant the execution and updates N_i to the values of n_i. On the contrary, if there exists a component B_i whose participation number n_i is less or equal to what the conflict resolution protocol has recorded, then the conflict resolution component replies a failure since B_i^{SR} has already participated in an interaction with the participation number n_i.

Now, we are ready to detail the construction of the three layers of our Send/Receive model.

5.3.1 Send/Receive Atomic Components

In this subsection, we define the transformed Send/Receive atomic component B^{SR} that is capable of communicating with the scheduler component(s). This definition differs from Definition 28 in the fact that Send/Receive components do not compute approximations as we do not consider priorities. In addition, Send/Receive components defined here include the participation number needed for conflict resolution.

Definition 28. *Let* $B = (L, P, T, \mathsf{C}, X, \{X_p\}_{p\in P}, \{g_\tau\}_{\tau\in T}, \{\mathsf{tc}_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T}, \{r_\tau\}_{\tau\in T}, \{\mathsf{tpc}_\ell\}_{\ell\in L})$ *be an atomic component. The corresponding Send/Receive atomic component is* $B^{SR} = (L^{SR}, P^{SR}, T^{SR}, \mathsf{C}^{SR}, X^{SR}, \{X_p^{SR}\}_{p\in P}, \{g_\tau\}_{\tau\in T^{SR}}, \{\mathsf{tc}_\tau\}_{\tau\in T^{SR}}, \{f_\tau\}_{\tau\in T^{SR}}, \{r_\tau\}_{\tau\in T^{SR}}, \{\mathsf{tpc}_\ell\}_{\ell\in L^{SR}})$*, such that:*

- $L^{SR} = L \cup \{\perp_\ell |\ \ell \in L\}$*. The time progress conditions of locations* $\perp_\ell$ *and* ℓ *are equal to* tpc_ℓ *expressed on clock* g*.*
- $X^{SR} = X \cup \{\mathsf{tc}_p\}_{p\in P} \cup \{g_p\}_{p\in P} \cup \{\mathsf{tpc}\} \cup \{\rho_c\}_{c\in C} \cup \{t^{ex}\} \cup \{n\}$ *where* tc_p *are timing constraint variables,* g_p *are Boolean guard variables,* tpc *is time progress condition variable,* ρ_c *are reset date variables,* t^{ex} *is an execution date variable and* n *is a participation number variable.*
- $P^{SR} = P \cup \{o\}$*, where the* offer *port* o *exports the variables* $X_o^{SR} = \{n, \mathsf{tpc}\} \cup \bigcup_{p\in P}(X_p \cup \{\mathsf{tc}_p\} \cup g_p)$*, that is, the participation number variable , the time progress condition variable, the timing constraints variables and Boolean guards variables and variables originally exported by each port. For all other ports* $p \in P$*, we define* $X_p^{SR} = X_p \cup \{t^{ex}\}$*.*
- *For each place* $\ell \in L$*, we include an* offer *transition* $\tau_\ell = (\perp_\ell, o, \ell)$ *in* T^{SR} *with no Boolean guard, no timing constraint and no update function.*
- *For each transition* $\tau = (\ell, p, \ell') \in T$*, we include a* notification *transition* $\tau_p = (\ell, p, \perp_{\ell'})$ *in* T^{SR} *with no guard, no timing constraint. The update function* f_{τ_p} *first applies the original update function* f_τ *of* τ*, and then updates, clock reset date variables, time*

progress condition variables, timing constraint variables, Boolean guard variables and participation number as follows:

- $\forall c \in r_\tau \quad \rho_c = t^{ex}$,
- $\mathtt{tpc} := \mathtt{tpc}_{\ell'}$,
- $\forall p' \in P \quad \mathtt{tc}_{p'} := \begin{cases} \mathtt{tc}_{\tau'} & \text{if } \tau' = (\ell', p', \ell'') \in T \\ \mathtt{false} & \text{otherwise.} \end{cases}$
- $\forall p' \in P \quad g_{p'} := \begin{cases} g_{\tau'} & \text{if } \tau' = (\ell', p', \ell'') \in T \\ \mathtt{false} & \text{otherwise.} \end{cases}$
- $n := n + 1$

We recall that ρ_c variables are used to transform the timing constraints and the time progress conditions to be expressed on clock g using the transformations explained in Subsection 4.1.2.

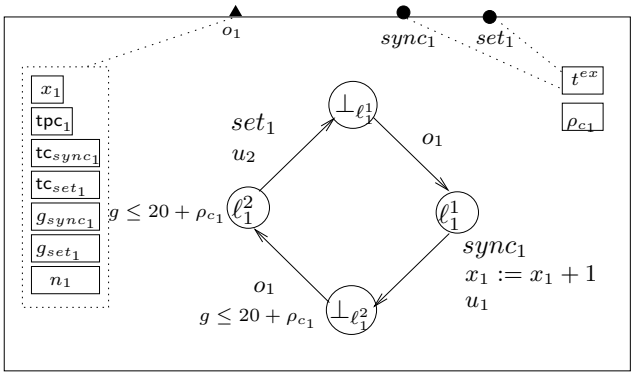


Figure 5.5: Send/Receive version of the $setter_1$ component from Figure 5.1.

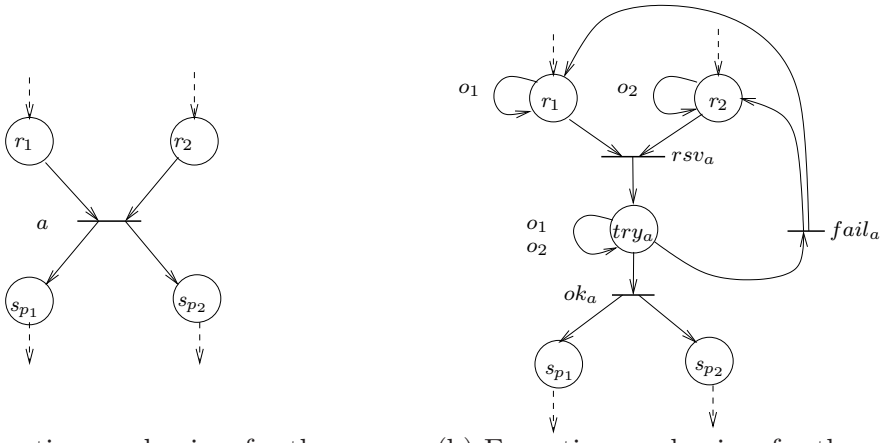
Example 17. *Figure 5.5 depicts the Send/Receive version of the $Setter_1$ component from Figure 5.1. The update functions u_1 and u_2 are defined as follows:*

$$u_1 = \begin{cases} \rho_{c_1} = t^{ex} \\ \mathtt{tpc}_1 = g \leq 20 + \rho_c \\ \mathtt{tc}_{sync_1} = \mathtt{false} \\ \mathtt{tc}_{set_1} = g \geq 10 + \rho_{c_1} \\ g_{sync_1} = \mathtt{false} \\ \mathtt{tc}_{set_1} = \mathtt{true} \end{cases}$$

$$u_2 = \begin{cases} \mathtt{tpc}_1 = \mathtt{true} \\ \mathtt{tc}_{sync_1} = \mathtt{true} \\ \mathtt{tc}_{set_1} = \mathtt{false} \\ g_{sync_1} = \mathtt{true} \\ g_{set_1} = \mathtt{false} \end{cases}$$

5.3.2 Building Schedulers in BIP

Consider a BIP model $\gamma(B_1 \cdots B_n)$ and a partition of interactions $\{\gamma_j\}_{j=1}^m$. Each class of interactions γ_j is handled by a single scheduler component S_j. As explained in Subsection 5.2, two conflicting interactions a and b could be either internally or externally conflicting according to the partition $\{\gamma_j\}_{j=1}^m$.



(a) Execution mechanism for the case where a is internally or not conflicting.

(b) Execution mechanism for the case where a is externally conflicting.

Figure 5.6: Mechanisms for Interactions Execution.

In the case where a and b are internally conflicting, their execution is done without requesting the conflict resolution protocol as conflicts are resolved locally. Similarly to the centralized scheduler presented in Chapter 4, the arbitration between two interactions that are internally conflicting is simply achieved by the Petri net of the corresponding scheduler: offers correspond to tokens which are "consumed" when one of the two conflicting interactions is selected. Figure 5.6 (a) shows such mechanism for the execution of interaction a involving ports p_1 and p_2. Note that such mechanism is also applied for not conflicting interactions.

In the case where a and b are externally conflicting, their execution is done through a reservation mechanism. Figure 5.6 (b) shows such mechanism for the execution of interaction $a = (\{p_1, p_2\}, G_a, F_a)$. Before executing interaction a, the scheduler sends a request to the conflict resolution protocol to reserve an interaction by executing transition rsv_a. This transition exectes only if interaction a is enabled, that is, received places r_1 and r_2 are marked (offers from participating components B_1 and B_2 are received). The execution of this transition removes tokens from received places and moves them to try_a place. Indeed, removing tokens from received places disables any other internally conflicting interaction with a to execute thanks to the Petri net semantics. Upon execution of transition rsv_a, a reservation request is sent to the conflict resolution protocol. The reservation request contains the participation numbers of the participating components. The conflict resolution protocol responds either positively on port ok_a if the reservation succeed or negatively on port $fail_a$ if the reservation cannot be granted. In the first case, the scheduler executes transition ok_a by moving token from try_a place to sending places s_{p_1} and s_{p_2} from which it sends notification

to the components. In the second case, transition labeled $fail_a$ executes and token moves from try_a places to received places r_1 and r_2.

When components take part in an external interactions, they send new offers to the schedulers. Thus, the schedulers may receive successive offers from components. In order to take into account these offers, loop transitions are added in received and trying places. These loops are added to match Definition 22 of Send/Receive interactions: each time a send (of an offer) is enabled, corresponding receives must be enabled too. Figure 5.6 (b) shows this kind of loop transitions. For instance, suppose that the scheduler of Figure 5.6 (b) receive an offer from component B_1. Token moves then from waiting place w_1 to received place r_1. As this offer is sent to other schedulers, an external interaction can execute and consume that offer. The component is then required to send new offer to the schedulers. The scheduler of Figure 5.6 (b) is then able to receive this offer due to the loop transition from received place r_1.

We are now ready to define the scheduler component S_j handling a subset of interactions $\gamma_j \subset \gamma$.

Definition 29. *Let $\gamma(B_1, \ldots, B_n)$ be a BIP model and $\gamma_j \subset \gamma$ be a subset of interactions. The corresponding scheduler S_j handling interactions of γ_j is defined as the Send/Receive BIP component $S_j = (L^{S_j}, P^{S_j}, T^{S_j}, \mathsf{C}^{S_j}, X^{S_j}, \{X_p\}_{p \in P^{S_j}}, \{g_\tau\}_{\tau \in T^{S_j}}, \{\mathsf{tc}_\tau\}_{\tau \in T^{S_j}}, \{f_\tau\}_{\tau \in T^{S_j}}, \{r_\tau\}_{\tau \in T^{S_j}}, \{\mathsf{tpc}_\ell\}_{\ell \in L^{S_j}})$ where:*

- *The set of variables X^{S_j} contains the following variables.*
  - *Variables updated whenever an offer from component B_i participating in an interaction belonging to γ_j is received. They consist of the following: timing constraint variable tc_p, Boolean variable g_p and a local copy of the variables X_p for each port p involved in an interaction belonging to γ_j, a time progress condition variable tpc_i and a participation number variable n_i for each component B_i participating in an interaction belonging to γ_j.*
  - *Variables updated whenever interaction a is scheduled. They consist of the following: execution date variable t_a^{ex} for each interaction $a \in \gamma_j$, that stores the last execution date of interaction a and, an execution date variable t_i^{ex} for each component B_i participating in an interaction belonging to γ_j.*
- $\mathsf{C}^{S_j} = \{g\}$.
- *L^{S_j} includes two types of places.*
  - *For each component B_i involved in interactions of γ_j, we include* waiting *and* received *places w_i and r_i, respectively. Place r_i has a time progress condition defined by the variable tpc_i.*
  - *For each port p involved in interactions of γ_j, we include a* sending *place s_p. The time progress condition of s_p is* `false`.
  - *For each interaction $a \in \gamma_j$ that is in external conflict with another interaction, we include a* trying *place try_a.*

- *The set of ports P^{S_j} consists of the following.*
  - *For each component B_i involved in an interaction of γ_j, P^{S_j} includes a receive-port o_i, to receive offers. Each port o_i is associated with the variables tc_p, g_p,and X_p for each port p of B_i as well as the variable tpc_i and n_i of B_i.*
  - *For each port p involved in interactions γ_j, P^{S_j} includes a send-port p, which exports the set of variables $X_p \cup \{t_i^{ex}\}$ where i is the index of component B_i that exports port p.*
  - *For each interaction $a \in \gamma_j$ that is externally conflicting, P^{S_j} includes ports rsv_a (reserve a, send-port), ok_a (success in reserving a, receive-port), and $fail_a$ (failure to reserve a, receive-port). The port rsv_a exports the variables $\{n_i\}_{B_i \in part(a)}$.*
  - *For each interaction $a \in \gamma_j$ that is internally conflicting, P^{S_j} include a unary port a.*
- *The set T^{S_j} of transitions consists of the following.*
  - *In order to receive offers from a component B_i, T^{S_j} includes transition (w_i, o_i, r_i). T^{S_j} also includes transitions (r_i, o_i, r_i) and $\{(try_a, o_i, try_a), B_i \in part(a)\}$ to receive new offers when B_i takes part in an external interaction. These transitions have no Boolean guards, no timing constraints and no update functions.*
  - *For each port p involved in interactions γ_j, T^{S_j} includes a transition (s_p, p, w_i) where i corresponds to the index of component exporting port p. This transition has no Boolean guard, no timing constraint and no update function. This transition is used to notify the corresponding component to execute the transition labeled by p.*
  - *For each interaction $a = (\{p_i\}_{i \in I}, G_a, F_a) \in \gamma_j$, that is internally conflicting with an other interaction or is not conflicting with any interaction, T^{S_j} includes the transition $\tau_a = (\{r_i\}_{B_i \in part(a)}, a, \{s_{p_i}\}_{p_i \in a})$, guarded by the Boolean guard $g_{\tau_a} = G_a \bigwedge_{p_i \in a} g_{p_i}$ and having the timing constraint $\mathsf{tc}_{\tau_a} = \bigwedge_{p_i \in a} \mathsf{tc}_{p_i}$. This transition updates the following variables.*
    - *$t_a^{ex} := t(g)$*
    - *$\forall\ B_i \in part(a)$, $t_i^{ex} := t_a$*

    then triggers the data transfer function F_a.
  - *For each interaction $a = (\{p_i\}_{i \in I}, G_a, F_a) \in \gamma_j$, that is externally conflicting with another interaction, T^{S_j} includes the following three transitions:*
    - *$\tau_{rsv_a} = (\{r_i\}_{B_i \in part(a)}, rsv_a, try_a)$ guarded by the Boolean guard $g_{\tau_{rsv_a}} = G_a \bigwedge_{p_i \in a} g_{p_i}$ and the timing constraint $\mathsf{tc}_{\tau_{rsv_a}} = \bigwedge_{p_i \in a} \mathsf{tc}_{p_i}$.*
    - *$\tau_{ok_a} = (try_a, ok_a, \{s_{p_i}\}_{p_i \in a})$ with no guard nor timing constraint. This transition updates the following variables:*
      - *$t_a^{ex} := t(g)$,*
      - *$\forall\ B_i \in part(a)$, $t_i^{ex} := t_a^{ex}$*

      then triggers the data transfer function F_a

* $\tau_{fail_a} = (try_a, fail_a, \{r_i\}_{B_i \in part(a)})$ *with no guard, no timing constraint and nor update function.*

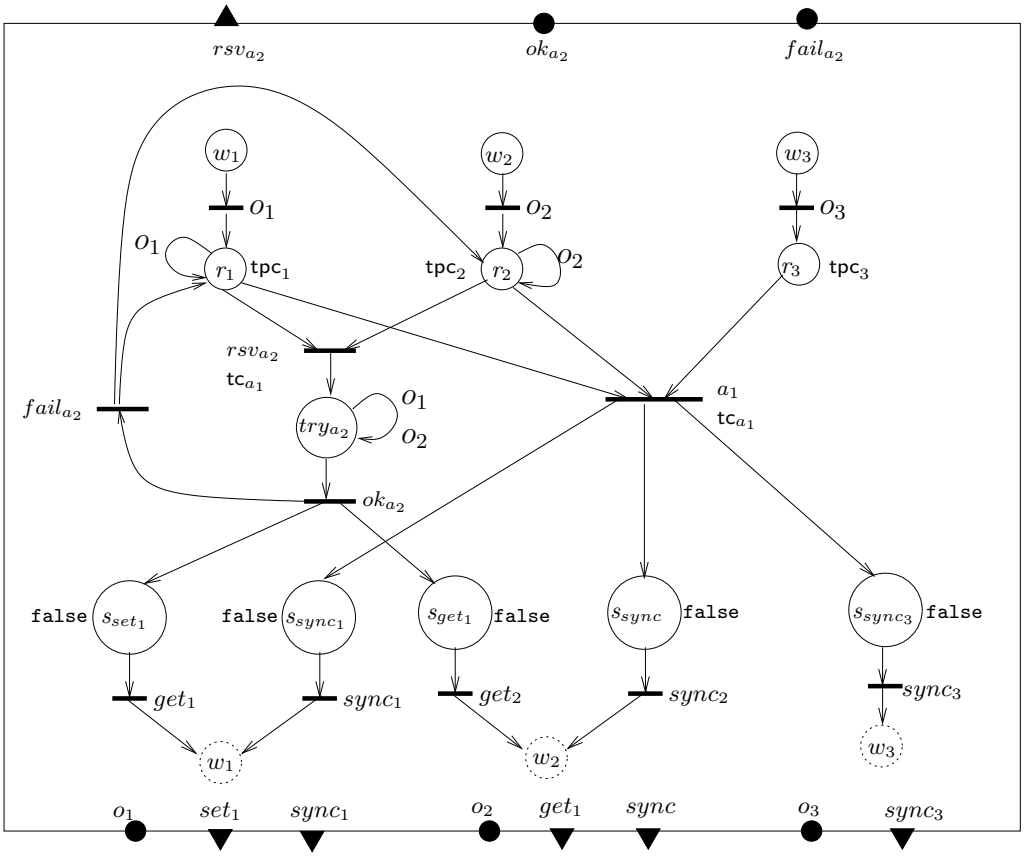


Figure 5.7: Distributed scheduler handling the class of interactions $\{a_1, a_2\}$.

Example 18. *Consider again the example from Figure 5.1. We consider the following partition of interactions* $\gamma_1 = \{a_1, a_2\}$ *and* $\gamma_2 = \{a_3\}$*. Figure 5.7 shows the scheduler handling partition* γ_1*. Since interaction* a_1 *in not conflicting with any interaction, its execution is done through the unary interaction* a_1*. Since interaction* a_2 *is conflicting with the external interaction* a_3*, its execution requires reservation from conflict resolution protocol. The reservation can be performed only at time instants meeting the timing constraint* tc_{a_3} *(expressed on clock g), that is, for values of the global clock for which* tc_{a_3} *evaluates to true. Note that the interaction guard of* a_3 *is true and thus it is not shown on the figure. The reservation request sends the participation numbers* n_1 *and* n_2*. These values are used by the conflict resolution protocol to check the validity of the offers sent by* B_1 *and* B_2*. The execution of interaction* a_2 *is triggered when the conflict resolution protocol responds positively on port* ok_{a_3}*.*

5.3.3 Conflict Resolution Protocol

In this subsection, we present three implementations of the conflict resolution protocol inspired from [9]. Intuitively, the main role of the conflict resolution protocol is to check the freshness of offers received for an interaction, that is, to check that no externally conflicting interactions has been already executed using the same offers. This ensures that two conflicting interactions cannot execute with the same offers, that is, with the same participation numbers. Mutual exclusion is ensured using participation numbers associated to offers. Indeed, the conflict resolution protocol keeps the last participation number of each component and compares it with the participation number provided along with the reservation request from the schedulers. If each participation number from the request is greater than the one recorded by the conflict resolution protocol, the interaction is then granted to execute. Otherwise, the interaction execution is disallowed.

Centralized Protocol

We present here the first version of the proposed Bagrodia's implementations in BIP. It corresponds to a centralized Send/Receive component denoted by CP.

Definition 30. *Let $\gamma(B_1, \ldots, B_n)$ be a BIP model and $\{\gamma_j\}_{j=1}^{m}$ be a partition of interactions. The corresponding centralized conflict resolution protocol component CP is defined by the Send/Receive BIP component $CP = (L^{CP}, P^{CP}, T^{CP}, \mathsf{C}^{CP}, X^{CP}, \{X_p\}_{p \in P^{CP}}, \{g_\tau\}_{\tau \in T^{CP}}, \{\mathsf{tc}_\tau\}_{\tau \in T^{CP}}, \{f_\tau\}_{\tau \in T^{CP}}, \{r_\tau\}_{\tau \in T^{CP}}, \{\mathsf{tpc}_\ell\}_{\ell \in L^{CP}})$*

- *L^{CP} contains for each externally conflicting interaction a the waiting place w_a and receive place r_a. Place w_a has no time progress condition. Place r_a has a time progress condition $\mathsf{tpc}_{r_a} = \mathtt{false}$.*
- *X^{CP} contains for each component B_i the last participation number N_i. In addition X^{CP} contains for each externally conflicting interaction a and for each component $B_i \in part(a)$, the participation number n_i^a.*
- *$\mathsf{C}^{CP} = \{g\}$.*
- *P^{CP} contains for each externally conflicting interaction a the reservation ports rsv_a, ok_a and $fail_a$. The port rsv_a exports variables $X_{rsv_a} = \{n_i^a \mid B_i \in part(a)\}$. The ports ok_a and $fail_a$ do not have exported variables.*
- *T^{CP} contains for each externally conflicting interaction a the following transitions.*
  - *$\tau_{rsv_a} = (w_a, rsv_a, r_a)$ with no guard, no timing constraint and no update function.*
  - *$\tau_{ok_a} = (r_a, ok_a, w_a)$ guarded by the guard $G_{\tau_{ok_a}} = \bigwedge_{B_i \in part(a)} n_i^a > N_i$ and having no timing constraint. The update function of this transition sets the variables N_i of each component $B_i \in part(a)$ to the value of n_i^a. That is, for all $B_i \in part(a)$ it performs $N_i := n_i^a$.*
  - *$\tau_{fail_a} = (r_a, fail_a, w_a)$ with no guard, no timing constraint and no update function.*

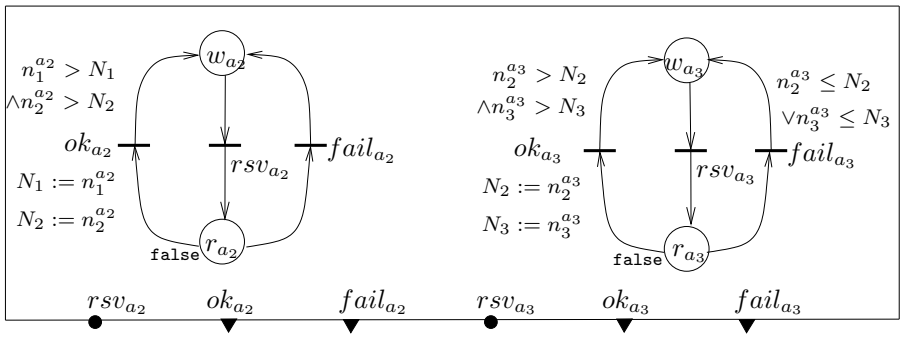


Figure 5.8: The centralized version of conflict resolution protocol for the BIP model of Figure 5.1 considering the partition $\gamma_1 = \{a_1, a_2\}$ and $\gamma_2 = \{a_3\}$.

In the centralized conflict resolution protocol CP, the time progress conditions of received places ensure that each request is treated immediately (no delay is allowed between the receiption of a request from a scheduler and the corresponding response of CP).

Notice that two conflicting interactions a and b with the same participation numbers for shared components cannot be both granted by the conflict resolution protocol CP, that is, CP correctly implements conflict resolution, which is explained as follows. Assume that CP has received requests for both a and b enabling both transitions labeled by ok_a and ok_b in CP. Due to atomic execution of transitions in CP, the execution ok_a (resp. ok_b) updates (in particular) the values of participation numbers N_i components shared between a and b which disables transition ok_b (resp. ok_a) leaving only the $fail_b$ transition.

Example 19. *Figure 5.8 presents CP component corresponding to the centralized version of conflict resolution protocol for the BIP model of Figure 5.1 considering the partition of interactions $\gamma_1 = \{a_1, a_2\}$ and $\gamma_2 = \{a_3\}$. It handles only interactions a_2 and a_3 as these are externally conflicting interactions with respect to the partition of interactions. Initially, tokens of CP component remain at waiting places until a reservation request of a_2 or a_3 is received. For instance, when a reservation request for interaction a_2 arrives, the token moves to receive place r_{a_2}. The request provides the participation numbers of components B_1 and B_2 participating in a_2, stored in $n_1^{a_2}$ and $n_1^{a_2}$ respectively. If for each component B_i, $i = 1, 2$, the variable $n_i^{a_2}$ is greater than variable N_i, then the transition labeled ok_{a_2} takes place. The transition labeled $fail_{a_2}$ takes place regardless the value of these variables.*

In the centralized protocol, atomic access to N_i variables is ensured through the Petri net semantics. The two remaining protocols are actually protocols to ensure atomicity for accessing the N_i variables.

Token Ring Protocol

The second implementation of the conflict resolution protocol is inspired from the token-based algorithm provided by Bagrodia [9]. This version presents a more distributed protocol as it includes one component for each externally conflicting interaction. A token carrying variables N_i circulates between all these components. Conflict resolution is ensured by the

fact that only the component holding the token can decide to execute an interaction, based on participation numbers as in the centralized conflict resolution protocol.

We denote by TR_a the token ring component handling reservation of the externally conflicting interaction a.

Definition 31. *Let $\gamma(B_1, \ldots, B_n)$ be a BIP model and $\{\gamma_j\}_{j=1}^{m}$ be a partition of interactions. Given an externally conflicting interaction a, the corresponding conflict resolution protocol component TR_a is defined as the Send/Receive BIP component $TR_a = (L^{TR_a}, P^{TR_a}, T^{TR_a}, \mathsf{C}^{TR_a}, X^{TR_a}, \{X_p\}_{p \in P^{TR_a}}, \{g_\tau\}_{\tau \in T^{TR_a}}, \{\mathtt{tc}_\tau\}_{\tau \in T^{TR_a}}, \{f_\tau\}_{\tau \in T^{TR_a}}, \{r_\tau\}_{\tau \in T^{TR_a}}, \{\mathtt{tpc}_\ell\}_{\ell \in L^{TR_a}})$ where:*

- *L^{TR_a} includes the waiting place w_a, the receive place r_a, the receiving token place rt_a and the sending token place st_a. The time progress conditions of places r_a and st_a are* `false`*. The other places do not have time progress conditions.*
- *X^{TR_a} includes for each component B_i the last participation number N_i. In addition, it includes, for each component $B_i \in part(a)$, the participation number n_i^a.*
- *P^{TR_a} contains the reservation ports rsv_a, ok_a and $fail_a$ as well as the sending and receiving token ports RT_a, ST_a.*
- *T^{TR_a} includes the following transitions:*
  - *transitions for interaction reservation:*
    - *$\tau_{rsv_a} = (w_a, rsv_a, r_a)$, with no guard, no timing constraint and no update function.*
    - *$\tau_{ok_a} = (\{r_a, rt_a\}, ok_a, w_a)$ guarded by $G_{\tau_{ok_a}} = \bigwedge_{B_i \in part(a)} n_i^a > N_i$, having no timing constraint, and updating variables N_i of each component $B_i \in part(a)$ as follows: for all $B_i \in part(a)$, it performs $N_i := n_i^a$.*
    - *$\tau_{fail_a} = (r_a, fail_a, w_a)$ guarded by $G_{\tau_{fail_a}} = \bigvee_{B_i \in part(a)} n_i^a \leq N_i$ and having no timing constraint and no update function.*
  - *transitions for sending and receiving token:*
    - *$\tau_{ST_a} = (\{st_a, w_a\}, ST_a, rt_a)$ with Boolean guard, no timing constraint and no update function.*
    - *$\tau_{RT_a} = (w_a, RT_a, st_a)$ with no Boolean guard, no timing constraint and no update function.*

As in the case of centralized conflict resolution protocol, the time progress conditions of receive request place r_a is `false`. This is to ensure that the response to a request is sent immediately without any delay. When the component has a reservation request, it can send a fail message even if it does not hold the token. However, it has to wait to receive the token to send an *ok* message. To enforce immediate responses to requests, we also need that the token circulates instantaneously between conflict resolution protocol components. To do that, we put `false` as time progress condition of place st_a, which imposes the component holding the token to send it immediately without any delay.

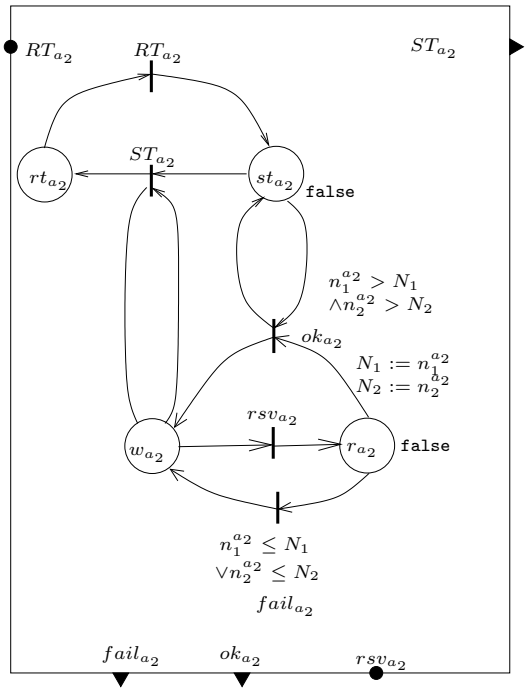


Figure 5.9: Conflict resolution component in the token ring protocol handling reservation of interaction a_2 of the example from Figure 5.1 considering the partition $\gamma_1 = \{a_1, a_2\}$ and $\gamma_2 = \{a_3\}$.

Example 20. *Figure 5.9 presents the reservation component TR_{a_2} that handles interaction a_2 in the conflict resolution protocol. Initially, the component is at states w_{a_2} and st_{a_2}. From those states it can either receive the token via port RT_{a_2} or receive a reservation request via port rsv_{a_2}. If a reservation request is received, a $fail$ message is sent if the values of variables N_i discard the request. Otherwise, an ok message is sent only if the component holds the token and if the participation number are fresh.*

The component sends back the token immediately if no reservation request is received, that is if the component is in location w_{a_2}. Otherwise, the token is sent back after responding to the request.

Send/Receive interactions between components of token ring protocol. Given a set of externally conflicting interactions $\{a_1, \ldots, a_m\}$, the token ring protocol is obtained by building one component TR_{a_j} for each externally conflicting interaction. These components communicate through Send/Receive interactions given by the set γ^{TR}. This set includes Send/Receive interactions that connect each component TR_{a_j} to its neighbor TR_{a_j+1}. That is, γ^{TR} includes for two sucessive externally conflicting interactions a_j, a_{j+1}, the Send/Receive interaction $(TR_{a_j}.ST_{a_j}, TR_{a_{j+1}}.RT_{a_{j+1}})$.

Figure 5.10 shows the connections between components in the token ring protocol layer for the example from Figure 5.1.

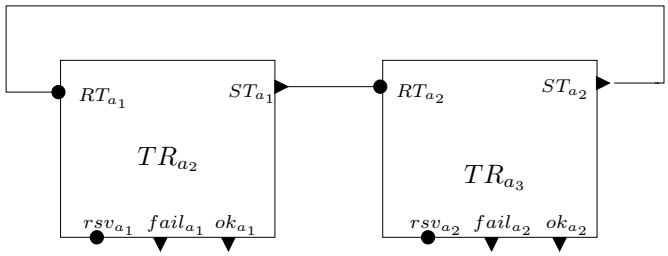


Figure 5.10: Global view of the token ring conflict resolution protocol for the example from Figure 5.1

Dining Philosophers Protocol

The third implementation of conflict resolution protocol corresponds to an adaption of the solution to the dining philosophers problem [10]. As in the token-ring version, there is one conflict resolution protocol component for each externally conflicting interaction. We denote by DP_a the dining philosophers component corresponding to an interaction a. If two interactions a and b are externally conflicting, the corresponding conflict resolution protocol components DP_a and DP_b share a fork carrying N_i variables corresponding to the latest participation numbers of components involved in both interactions. In order to respond to a reserve request sent by a scheduler, each component must obtain all forks shared with other components. The fork brings the latest participation numbers N_i recorded by its last owner. Once obtained, the component compares the participation numbers received from the reservation request sent by the scheduler and from the forks, and responds accordingly. Whenever a component responds to a reservation request, the fork becomes "dirty". In such case, the component may send the fork when it is requested to do so.

Given two interactions a and b, we denote by $conf(a,b) = part(a) \cap part(b)$ the set of components participating in both interactions. We denote also by $extconf(a)$ the set of interactions that are externally conflicting with a.

Definition 32. *Let $\gamma(B_1,\ldots,B_n)$ be a BIP model and $\{\gamma_j\}_{j=1}^m$ be a partition of interactions. Given an externally conflicting interaction a, the corresponding conflict resolution protocol component DP_a is defined as the Send/Receive BIP component $DP_a = (L^{DP_a}, P^{DP_a}, T^{DP_a}, \mathsf{C}^{DP_a}, X^{DP_a}, \{X_p\}_{p\in P^{DP_a}}, \{g_\tau\}_{\tau\in T^{DP_a}}, \{\mathsf{tc}_\tau\}_{\tau\in T^{DP_a}}, \{f_\tau\}_{\tau\in T^{DP_a}}, \{r_\tau\}_{\tau\in T^{DP_a}}, \{\mathsf{tpc}_\ell\}_{\ell\in L^{DP_a}})$, where:*

- *L^{DP_a} contains waiting place w_a, and the received place r_a. Moreover, for each interaction $b \in extconf(a)$, L^{DP_a} includes the sending and received request places sr_b and wr_b as well as sending, waiting and received fork places sf_b, wf_b and rf_b. The time progress condition of places r_a, sr_b, rf_b and sf_b are* `false`*. The other places do not have time progress conditions.*

- *X^{DP_a} includes the variables $\{N_i \mid B_i \in part(a)\}$ and the variables $\{n_i^a \mid B_i \in part(a)\}$. Moreover, for each interaction $b \in extconf(a)$, X^{DP_a} includes the additional variables $\{N_i^b \mid B_i \in conf(a,b)\}$, and the Boolean variables $fork_b$ and $dirty_b$.*

- P^{DP_a} *includes the ports* rsv_a, ok_a, $fail_a$, *and the unary port* u_a. *Port* rsv_a *exports variables* $\{n_i^a \mid B_i \in part(a)\}$. *Ports* ok_a *and* $fail_a$ *do not have exported variables. Moreover, for each interaction* $b \in extconf(a)$, P^{DP_a} *includes the sending and receiving request ports* SR_b^a *and* RR_b^a *as well as sending and receiving fork ports* SF_b^a *and* RF_b^a. *Ports* SF_b^a *and* RF_b^a *exports variables* $\{N_i^b \mid B_i \in conf(a,b)\}$. *Ports* SR_b^a *and* RR_b^a *do not have exported variables.*

- T^{DP_a} *contains the following transitions:*

  - $\tau_{rsv_a} = (w_a, rsv_a, r_a)$ *with no guard, no timing constraint and no update function.*
  - $\tau_{ok_a} = (\{rf_b | b \in extconf(a)\}, ok_a, w_a)$ *guarded by* $G_{\tau_{ok_a}} = \bigwedge_{b \in extconf(a)} fork_b \bigwedge_{B_i \in part(a)} n_i^a > N_i$, *having no timing constraint and updating the following variables:*
    - $\forall\ B_i \in part(a)\ \ N_i := n_i^a$
    - $\forall\ b \in extconf(a)\ \ dirty_b := \texttt{true}$
      $\forall\ B_i \in conf(a,b)$
      $N_i^b := N_i$
  - $\tau_{check_a} = (r_a, u_a, \{sr_b \mid b \in extconf(a)\})$ *guarded by* $G_{\tau_{check_a}} = \bigwedge_{B_i \in part(a)} n_i^a > N_i$, *having no timing constraint and no update function.*
  - $\tau^1_{fail_a} = (r_a, fail_a, w_a)$ *and* $\tau^2_{fail_a} = (\{rf_b | b \in extconf(a)\}, fail_a, w_a)$ *guarded by* $\bigvee_{B_i \in part(a)} n_i^a \leq N_i$ *and having no timing constraint. Transition* $\tau^1_{fail_a}$ *has no update function. Transition* $\tau^2_{fail_a}$ *has the following update function:* $\forall\ b \in extconf(a)\ \ dirty_b := \texttt{true}$.
  - *For each interaction* $b \in extconf(a)$, T^{DP_a} *includes also the following transitions.*
    - $\tau_{SR_b^a} = (sr_b, SR_b^a, wf_b)$, *with Boolean guard* $G_{\tau_{SR_b^a}} = \neg fork_b$, *no timing constraint and no update function.*
    - $\tau_{RR_b^a} = (wr_b, RR_b^a, sf_b)$ *with no guard, no timing constraint and no update function.*
    - $\tau_{SF_b^a} = (sf_b, SF_b^a, wr_a)$ *with Boolean guard* $G_{\tau_{SF_b^a}} = dirty_b$, *no timing constraint and* $fork_b := \texttt{false}$ *as update function.*
    - $\tau_{RF_b^a} = (wf_b, RF_b^a, rf_b)$, *with no guard and no timing constraint and updating the following variables:*
      - $dirty_b := \texttt{false}$
      - $fork_b := \texttt{true}$
      - $\forall\ B_i \in conf(a,b)\ \ N_i := max(N_i, N_i^b)$
      - $\forall\ c \in extconf(a)\ \ \forall\ B_i \in conf(a,c)\ \ N_i^c := N_i$
    - $\tau_{hasfork_b} = (sr_b, u_a, rf_b)$ *guarded by* $G_{\tau_{hasfork_b}} = fork_b$ *and having no timing constraint and no update function.*
    - $\tau_{hasnotfork_b} = (rf, u_a, sr_b)$ *guarded by* $G_{\tau_{hasnotfork_b}} = \neg fork_b$ *and having no timing constraint and no update function.*

Note that there is no delay allowed between the reception of the reservation request and the response. The reservation component DP_a should react immediately and responds to the reservation request. This is ensured by the time progress conditions of places separating the reception of request reception and the response.Although waiting fork places wf_b do not have time progress condition, time is not allowed to progress at these places. This is because each reservation component DP_b is required to send immediately the fork due to the time progress condition of place sf_a.

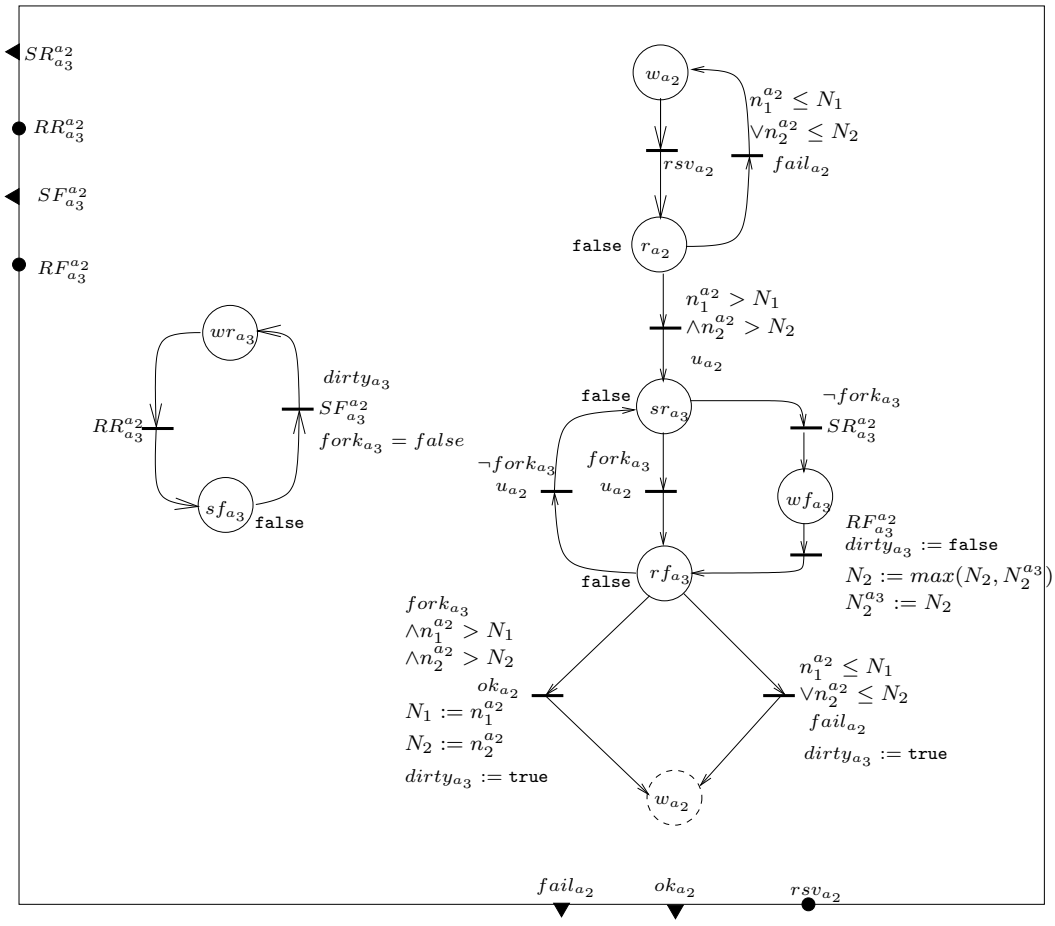


Figure 5.11: Components in the dining philosophers protocol handling reservation of interaction a_2 of the example from Figure 5.1 considering the partition $\gamma_1 = \{a_1, a_2\}$ and $\gamma_2 = \{a_3\}$.

Example 21. *Figure 5.11 presents the reservation component DP_{a_2} that handles interaction a_2 in the conflict resolution protocol. Since interaction a_2 is externally conflicting with a_3, there is a fork shared between components DP_{a_2} and DP_{a_3} where DP_{a_3} is the component handling interaction a_3. Thus, DP_{a_2} has to obtain the fork before deciding the execution of a_2.*

Initially, the DP_{a_2} component is at places w_{a_2} and wr_{a_3}. From place w_{a_2}, it can receive a reservation request rsv_{a_2} indicating that interaction a_2 wants to execute. Upon reception

of this request, the token moves to r_{a_3} place. From this place, a $fail$ message is sent if the variables N_i discard the request. Otherwise the token moves to place sr_{a_3} to start negotiating the fork with component DP_{a_3}. If component DP_{a_2} already holds the fork, token move directly to place rf_{a_3}. Otherwise, it sends a fork request to DP_{a_3} component in order to receive the fork. Upon receiving the fork, DP_{a_2} updates variables N_i as the fork brings the latest values. A newly received fork is always considered as clean. Receiving the fork moves the token to rf_{a_2} place. From this place, DP_{a_2} responds by either ok or $fail$ according to the values of N_i and $n_i^{a_2}$. Responding to a reservation request makes the fork dirty. DP_{a_2} may reuse the dirty fork as long as no fork request from DP_{a_3} is received. From place wr_{a_3}, DP_{a_2} waits for receiving fork request from component DP_{a_3}. Upon reception of this request, DP_{a_2} sends the fork if it is dirty.

Send/Receive interactions between components of the dining philosophers protocol. Given a set of externally conflicting interactions $\{a_1, \ldots, a_m\}$. The dining philosophers protocol is obtained by building one component DP_{a_j} for each externally conflicting interaction. These components communicate through Send/Receive interactions given by the set γ^{DP}. This set includes Send/Receive interactions to send and receive forks and requests between conflict resolution protocol components. More precisely, for any two externally conflicting interactions a and b, γ^{DP} contains:

- $(DP_b.SF_a^b,\ DP_a.RF_b^a)$ and,
- $(DP_b.SR_a^b, DP_a.RR_b^a)$.

Figure 5.12 shows a global view of the dining philosophers conflict resolution protocol for the example from Figure 5.1.

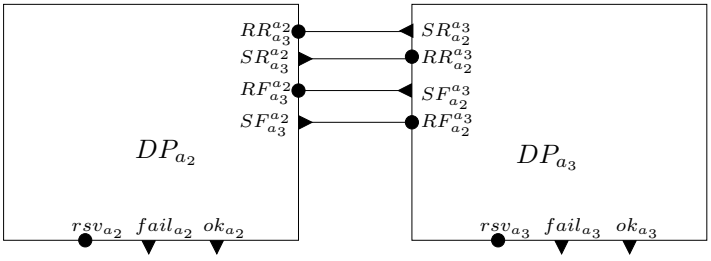


Figure 5.12: Global view of the dining philosophers conflict resolution protocol for the example from Figure 5.1.

5.3.4 Send/Receive Interactions

To complete the description of the 3-layer BIP models, we have to define the interactions between the distributed components. Between the components layer and the schedulers , offers and notifications are exchanged. Communication between the schedulers and the conflict resolution protocol involves rsv, ok and $fail$ messages transmission.

Definition 33. *Let $B = \gamma(B_1 \cdots B_n)$ be a BIP model and $\{\gamma_j\}_{j=1}^{k}$ be a partition of interactions. We denote by $\{a_1, \ldots, a_m\}$ the set of externally conflicting interactions. We define for each conflict resolution protocol the following Send/Receive model:*

- B_{CP}^{SR}=$\gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR}, S_1, \ldots, S_k, CP)$, *implementing the centralized conflict resolution protocol,*
- B_{TR}^{SR}=$\gamma^{SR} \cup \gamma^{TP}(B_1^{SR}, \ldots, B_n^{SR}, S_1, \ldots, S_k, TR_{a_1}, \ldots, TR_{a_m})$, *implementing the token ring protocol,*
- B_{DP}^{SR}=$\gamma^{SR} \cup \gamma^{TP}(B_1^{SR}, \ldots, B_n^{SR}, S_1, \ldots, S_k, DP_{a_1}, \ldots, DP_{a_m})$, *implementing the dining philosophers protocol.*

The Send/Receive interactions γ^{SR} are defined as follows.

- *For each component B_i^{SR}, let $S_{j_1}, \ldots, S_{j_l}$ be the scheduler components handling interactions involving B_i^{SR}. γ^{SR} includes the offer interaction $(B_i^{SR}.o, S_{j_1}.o_i, \ldots, S_{j_l}.o_i)$.*
- *For each port p in component B_i^{SR} and for each scheduler component S_j handling an interaction involving p, γ^{SR} includes the notification interaction $(S_j.p, B_i^{SR}.p)$.*
- *For each internally conflicting interaction $a \in \gamma$, γ^{SR} includes the unary interaction $(S_j.a)$, where S_j is the scheduler component handling the interaction a.*
- *For each externally conflicting interaction a, γ^{SR} includes the following interactions:*
  - $(rsv_a.S_j, rsv_a.C_{CRP})$, *and*
  - $(ok_a.S_j, ok_a.C_{CRP})$, *and*
  - $(fail_a.S_j, fail_a.C_{CRP})$,

  where C_{CRP} is either CP, or TR_a, or DP_a depending on the implemented conflict resolution protocol.

5.4 Correctness of the Proposed Model

In this section, we first show that the 3-layer model B_{CP}^{SR} is indeed a Send/Receive model. We then prove that the initial BIP model B is observationally equivalent to B_{CP}^{SR}. Finally, we prove weak simulation between B_{TR}^{SR}(resp. B_{DP}^{SR}) and B.

Compliance with Send/Receive Model

We need to show that receive ports of B_{CP}^{SR} will unconditionally become enabled whenever one of the corresponding send ports is enabled. Intuitively, this holds since communications between two successive layers follow a request/acknowledgement pattern. Whenever a layer sends a request, it enables the receive port to receive acknowledgement and no new request is sent until the first one is acknowledged.

Lemma 4. *Given a BIP model $B = \gamma(B_1, \ldots, B_n)$ and a partition of interactions $\{\gamma_j\}_{j=1}^{k}$, the model $B_{CP}^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR}, S_1, \ldots, S_k, CP)$ obtained by transformation of Section 5.3 meets the properties of Definition 22.*

Proof. The first two constraints of Definition 22 are trivially met by construction. We now prove that the third constraint also holds; i.e, whenever a send-port is enabled, all its associated receive-ports are enabled as well.

- Between a Send/Receive component B_i^{SR} and a scheduler S_j, we consider the places w_i, r_i and s_p for each port p exported by B_i. If there is no token in s_p place, then, it is easy to see that from this configuration, only interactions o_i is enabled. If there is a token at place s_p, it results from the execution of transition a or ok_a in the scheduler. In the first case, a is internally or not conflicting interaction. This case is similar to the centralized scheduler. In the second case, the interaction a is externally conflicting and is refereed to the conflict resolution protocol. The latter uses the current participation number of B_i for the execution of a and no other interaction is granted using the same participation number. Thus, s_p is the only active place, from which a notification could be sent.

- Between the scheduler S_j and the conflict resolution protocol, we consider the places try_a in the scheduler S_j, w_a and r_a in the conflict resolution protocol. If try_a is empty, and w_a is active, only reservation request through port rsv_a is enabled. When the rsv_a request is sent, the place try_a and r_a become active. From this configuration, only send ports ok_a and $fail_a$ are enabled in the conflict resolution protocol, and the associated ports are also enabled in the scheduler.

□

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

Observational Equivalence between Original and Transformed Models

We show that B and B_{CP}^{SR} are observationally equivalent. We consider the correspondence between actions of B and B_{CP}^{SR} as follows. To each interaction $a \in \gamma$ of B, we associate either the binary interaction ok_a or the unary interaction a of B_{CP}^{SR}, depending on whether a is externally conflicting. All other interactions of B_{CP}^{SR} (offer, notification, reserve, fail) are unobservable and denoted by β.

We proceed as follows to complete the proof of observational equivalence. Among unobservable actions β, we distinguish between β_1 actions, that are interactions between the atomic components and the schedulers (namely offer and notification), and β_2 actions that are interactions between the schedulers and the conflict resolution protocol (namely reserve and fail). We denote by q^{SR} a state of B_{CP}^{SR} and q a state of B. A state of B_{CP}^{SR} from where no β_1 action is possible is called a *stable state*, in the sense that any β action from this state does not change the state of the atomic components.

Lemma 5. *From any state q^{SR}, there exists a unique stable state $[q^{SR}]$ such that $q^{SR} \xrightarrow{\beta_1^*}_{\gamma^{SR}} [q^{SR}]$.*

Proof. The state $[q^{SR}]$ exists since each Send/Receive component B_i^{SR} can do at most two β_1 transitions: receive a notification and send an offer. Since two β_1 transitions involving two different components are independent (i.e. modify distinct variables and places), the same final state is reached independently of the order of execution of β_1 actions. Thus $[q^{SR}]$ is unique. □

The above lemma proves the existence of a well-defined stable state for any of the transient states reachable by the Send/Receive model B_{CP}^{SR}. The state $[q^{SR}]$ verifies the property $q^{SR} \xrightarrow{\beta_1^*}_{\gamma^{SR}} [q^{SR}]$ and $[q^{SR}] \overset{\beta_1}{\not\to}_{\gamma^{SR}}$. Furthermore, Lemma 5 asserts that, at state $[q^{SR}]$, atomic components are in a stable state, and variables and clock g in the scheduler have the same value as in atomic components.

Lemma 6. *At a stable state $[q^{SR}]$, the Send/Receive model verifies the following properties.*

- *All atomic components are in a non busy place ℓ.*
- *All tokens in the scheduler components are in receive places r_i.*
- *The clock g and all variables in the atomic components have the same value than their copies in the scheduler components.*

Proof. The two first points come from Lemma 4 that guarantees possible execution of a Send/Receive interaction if its send-port is enabled. Therefore no place s_p in the scheduler components (respectively $\perp_\ell$ in atomic components) can be active at $[q^{SR}]$, otherwise the answer p (respectively the offer from $\perp_\ell$) could occur. Furthermore, since all offers have been sent, no token can be in a w_i place.

In each scheduler S_j, when all r_i are marked, the last transition executed is either an offer or a $fail$ message reception. The latter does not modify variables in the scheduler S_j. For each variable x in the scheduler S_j, the last modifying transition is the offer from the corresponding atomic component B_i, which ensures that each variable in the scheduler has the same value as the corresponding atomic component. The clock g has the same value in the atomic components and the schedulers as it is never reset. □

Lemma 7. *When B_{CP}^{SR} is in a stable state, for each $i \in \{1, \ldots, n\}$, we have $n_i > N_i$.*

Proof. Initially, for each component B_i, $N_i = 0$ and $n_i = 1$. By letting all components sending offers to all schedulers, we reach the first stable state where the property holds, since β_1 actions do not modify the N_i variables.

The variables N_i in the conflict resolution protocol are updated upon execution of an ok_a transition, using values provided by the scheduler, that are values from components according to Lemma 6. Thus, in the unstable state reached immediately after an ok_a transition, we have $n_i = N_i$ for each component B_i^{SR} participant in a. Then, the notification transition increments participation numbers in components so that in the next stable state $n_i > N_i$. For components $B_{i'}$ not participating in a, by induction on the number of ok interactions, we have $n_{i'} > N_{i'}$. □

Lemma 7 shows that the participation numbers propagate in a correct manner. In particular, at any stable state the conflict resolution protocol has only previously used values and schedulers have the freshest values, that is the same as in the atomic components.

We define a relation between the set of states Q^{SR} of B_{CP}^{SR} and the set of states Q of B as follows. For each state $q^{SR} \in Q^{SR}$, we build an equivalent state $equ(q^{SR})$ by:

1. considering the unique stable state $[q^{SR}]$ reachable by doing β transitions,
2. taking the control location ℓ of B_i^{SR} as control location for B_i in $equ(q^{SR})$, Lemma 6 ensures that it is a valid control state for B_i,
3. restricting the valuation of variables of B_i^{SR} to a valuation of variables in B_i, and
4. taking the valuation of original clock c_i in B_i as the valuation of $g - \rho_{c_i}$.

Theorem 3. $B \sim B_{CP}^{SR}$.

Proof. We define the equivalence R by taking:

$$R = \{(q^{SR}, q) \in Q^{SR} \times Q \mid q = equ(q^{SR})\}$$

The three next assertions prove that R is a weak bisimulation:

(i) If $(q^{SR}, q) \in R$ and $q^{SR} \xrightarrow{\beta}_{\gamma^{SR}} r^{SR}$ then $(r^{SR}, q) \in R$.

(ii) If $(q^{SR}, q) \in R$ and $q^{SR} \xrightarrow{\sigma}_{\gamma^{SR}} r^{SR}$ then $\exists r \in Q : q \xrightarrow{\sigma} r$ and $(r^{SR}, r) \in R$.

(iii) If $(q^{SR}, q) \in R$ and $q \xrightarrow{\sigma}_{\gamma} r$ then $\exists r^{SR} \in Q^{SR} : q^{SR} \xrightarrow{\beta^*\sigma} r^{SR}$ and $(r^{SR}, r) \in R$.

(i) If $q^{SR} \xrightarrow{\beta}_{\gamma^{SR}} r^{SR}$, then β is either β_1 action, thus by definition $[q^{SR}] = [r^{SR}]$, or β is β_2 action which does not change the state of the atomic components, thus $[q^{SR}] = [r^{SR}]$. Thus, we have $equ(q^{SR}) = equ(r^{SR})$.

(ii) The action σ in B_{CP}^{SR} is either an interaction a or a delay step δ.

- If a is an interaction, it corresponds to executing a transition labeled by a unary port a in the scheduler handing a or a transition labeled by ok_a in CP. These transitions are enabled according to valuations of variables and clock g in the scheduler handling a.

  If a is not externally conflicting, by construction of the scheduler, the transition labeled by a has the conjunction of timing constraints tc_p sent by the atomic components for each port $p \in a$. The Boolean guard of this transition is $G_a \bigwedge_{p \in a} g_p$, where g_p are boolean guards sent by the atomic components for each port $p \in a$. By Lemma 6, these values are the same in atomic components, and by extension in $q = equ(q^{SR})$. Thus the guard of a evaluates to true at $q = equ(q^{SR})$. By construction of the atomic components, the timing constraints tc_p are sent from $\perp_\ell$ in B_i^{SR} are the timing constraints at state ℓ in B_i. These timing constraints

are expressed on clock g which could be equivalently expressed on original clocks c involved in the original timing constraints of B_i. The valuation of each clock c involved in the timing constraint of a is computed from the valuation of $g - \rho_c$ where ρ_c is the last clock reset date of c. Therefore, at state q the timing constraint of a expressed on its original clocks are also met.

If a is externally conflicting, the transition labeled ok_a in CP is possible only if the transition rsv_a executes in the scheduler. This transition has the guard $G_a \bigwedge_{p \in a} g_p$ and timing constraint $\bigwedge_{p \in a} \mathsf{tc}_p$. Thus, if this transition is possible in the scheduler, then the timing constraint and the Boolean guard of a are met at q. Moreover, if transition labeled ok_a is enabled, this means that for each component B_i involved in a, $n_i > N_i$. In particular, for each involved component B_i, the offer corresponding to the number n_i has not been consumed yet.

We conclude that in both cases, we have $q \xrightarrow{a} r$. Finally, executing a in B^{SR} triggers the execution of the data transfer function F_a, followed by the computation in atomic component upon reception of the response. Thus at $[r^{SR}]$ the values in atomic components are the same as in r, which yields $(r^{SR}, r) \in R$.

– If a is a delay step, it corresponds by letting time progress by δ in either busy locations $\bot_{\ell_i}$ or in places r_i of the schedulers, corresponding to atomic components B_i. Location $\bot_{\ell_i}$ has the time progress condition tpc_{ℓ_i} and r_i has the time progress condition tpc_i sent from the atomic components and corresponding to time progress condition of location ℓ_i. Thus, all these time progress conditions are not false, otherwise the δ delay step would not be allowed. By Lemma 6, the values involved in time progress condition and sent from $\bot_\ell$ in B_i^{SR} are the values of time progress condition at state ℓ in B_i. These time progress conditions are expressed on clock g which could be equivalently expressed on original clocks c involved in the original time progress conditions B_i. The valuation of each clock c involved in the time progress condition of B_i is computed from the valuation of $g - \rho_c$ where ρ_c is the last clock reset date of c. If the time progress condition tpc_i expressed on clock g allows the time step δ in the scheduler, thus, δ is also allowed by the time progress condition tpc_{ℓ_i} expressed on original clocks of B_i. Therefore $q \xrightarrow{\delta} r$. Executing δ has the same effect on clocks in both models, therefore $(r^{SR}, r) \in R$.

(iii) If σ can be executed in B at state q, then from an equivalent state q^{SR}, one can reach the state $[q^{SR}]$ where the state, the clock g, and data of send/receive atomic atomic components coincide with those of q. Recall clocks c of the original model could be deduced from clock g as follows $c = g - \rho_c$ By Lemma 6, clock g, ports, timing constraints and data values are the same in atomic components and the schedulers. As previously, we distinguish the cases where σ is an interaction a or a delay step δ.

– If σ is a an interaction a from q, then the timing constraint and guard of a are true. By executing all possible $fail$ interactions (which correspond to β_2 actions), all tokens return back in r_i places in schedulers. Then, if a is not externally conflicting, it can be executed directly. Otherwise, the sequence $rsv_a ok_a$ executes, since Lemma 7 ensures that ok_a is enabled. In both cases, we have $q^{SR} \xrightarrow{\beta_1^*}$

$[q^{SR}] \xrightarrow{\beta_2^*} \xrightarrow{a} r^{SR}$. As previously, the execution of a in both models leads to equivalent states, thus $(r^{SR}, r) \in R$.

– If σ is a delay step δ, then from state $[q^{SR}]$, time can progress also by δ as at this state all places r_i are marked and having the time progress condition tpc_i sent by B_i^{SR} which is the same as the time progress condition of B_i at q. $q^{SR} \xrightarrow{\beta_1^*} [q^{SR}] \xrightarrow{\delta} r'^{SR}$. From state r'^{SR}, some fail interactions are possible (which correspond to β_2 actions) with zero delay, due to the time progress conditions that are equal to false in r_a places int the conflict resolution protocol. That is, even if some β_2 actions is possible from state r'^{SR}, no delay is allowed at this state. Thus we have $q^{SR} \xrightarrow{\beta_1^*} [q^{SR}] \xrightarrow{\delta} r'^{SR} \xrightarrow{\beta_2} r^{SR}$ with $(r^{SR}, r) \in R$.

□

Interoperability of Conflict Resolution Protocol

The centralized implementation CP of the conflict resolution protocol can be seen as a specification. Two other implementations, namely token ring and dining philosophers, can be used as conflict resolution protocol. However, these implementations are not observationally equivalent with the centralized implementation. The centralized version defines the most liberal implementation: if two reservation requests rsv_a and rsv_b are received, the protocol may acknowledge them in any order. This general behavior is not implemented neither by the token ring nor by the dining philosophers implementations, which are focused on ensuring progress. In the case of token ring, the response may depend on the order the token travels through the components. In the case of dining philosophers, the order may depend on places and the current status of forks.

Proposition 1. *$B_{TR}^{SR} \subset B$ and $B_{DP}^{SR} \subset B$.*

Proof. As B is observationnally equivalent to B_{CP}^{SR}, it is equivalent to prove that $B_{TR}^{SR} \subset B_{CP}^{SR}$ and $B_{DP}^{SR} \subset B_{CP}^{SR}$. As the behaviors B_{DP}^{SR} and B_{TR}^{SR} differ from B_{CP}^{SR} only in the conflict resolution protocol components, thus it is sufficient to prove that $TR \subset CP$ and $DP \subset CP$.

The observable actions are requests rsv_a, ok_a and $fail_a$ messages or a δ step. The unobservable actions are token passings for TR and forks exchange for DP. It is clear that whenever an action (either ok_a, $fail_a$ or rsv_a) is possible from a given state of TR (resp. DP) by moving the token (resp. the forks), then we can always reach a state of CP where this action is possible as well. δ steps are not allowed in the three implementations of the conflict resolution protocol due to the time progress conditions on sending places. That is, a request, token or forks are required to be sent without any delay. □

5.5 Conclusion

In this chapter, we proposed a method to decentralize the scheduler. The obtained models consist of (1) transformed atomic components of the original model, (2) a set of schedulers, each one handling a subset of interactions and (3) a conflict resolution protocol component(s)

responsible for resolving conflict between schedulers. The obtained models are dedicated to be implemented on platforms that provide fast communications, as in the schedulers, interactions scheduling dates correspond to execution dates in the components.

In the next chapter, we propose a method to obtain Send/Receive models that are dedicated to be implemented on platforms that provide slow communications. The idea in based on taking earlier the decision for executing an interaction. Indeed, the scheduler chooses earlier (as soon as possible) a date for executing an interaction and sends it to participating components so as they execute later at this date. However, deciding earlier the execution of an interaction may block components participating conflicting interaction. We will see in the next chapter how to avoid such blocking.

6

Taking Decision Earlier

In the previous chapter, we proposed a method for the generation of Send/Receive models with decentralized scheduling. Such models implicitly make the assumption that communications between components are instantaneous. This restricts the applicability of the approach to platforms having communication means that are fast enough so that communication delays can be neglected. In this chapter, we propose a method to obtain Send/Receive models that execute correctly even if communications are not instantaneous. To cope with communication delays, instead of notifying components at the very last moment (scheduing date), the schedulers have to plan interactions execution and to notify components in advance. In the proposed model, the schedulers choose as soon as possible a date for executing a selected interaction and send it to the participating components. Once received, the components wait for this date and then execute.

In order to ensure correct scheduling of interactions based on this approach, we show that the schedulers need to observe additional components not participating in the scheduled interactions. These components correspond to the ones participating in conflicting interactions (see Figure 6.1).

This chapter is structured as follows. In Section 6.1, we start by describing the restrictions that we impose on the original model. Then, we explain in Section 6.2 scheduling issue when considering earlier decision making approach. In Section 6.3, we define formally the transformation from the centralized BIP models into Send/Receive BIP models. As we will explain later in the chapter, scheduling interactions in advance may require to observe time progress conditions of additional components, in addition to the participating components. This is needed for checking that scheduling decisions will not block components if the scheduling date is not compatible with their time progress conditions. We will explain also how to minimize the number of observed components by applying static analysis techniques.

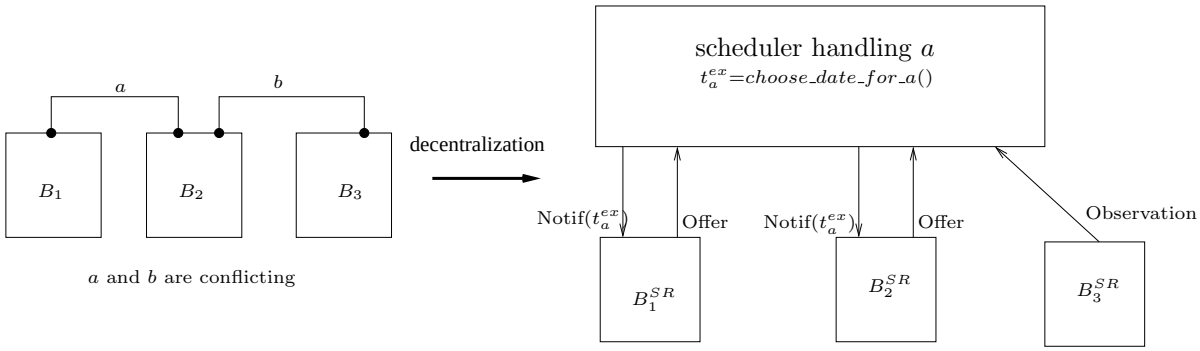


Figure 6.1: Earlier Decision Making Approach

6.1 Model Restrictions

We consider the same model restrictions of the previous chapter. In addition, we consider the following restrictions.

- We consider the following restricted grammar for timing constraints tc:

$$\mathsf{tc} := \texttt{true} \mid \texttt{false} \mid c \leq ct \mid c \geq ct \mid \mathsf{tc} \wedge \mathsf{tc}.$$

  where $c \in \mathsf{C}$ is a clock, $ct \in \mathbb{Z}_{\geq 0}$ is a constant. Any timing constraint tc can then be put into a conjunction of intervals as follows:

$$\mathsf{tc} = \bigwedge_{c \in \mathsf{C}} l_c \leq c \leq u_c \tag{6.1}$$

  such that for all $c \in \mathsf{C}$, $l_c \in \mathbb{Z}_{\geq 0}$ and $u_c \in \mathbb{Z}_{\geq 0} \cup \{+\infty\}$.

- We consider also atomic components that satisfy the following non-decreasing deadlines property, which is defined below.

  Definition 34. *Let $B = (L, P, T, \mathsf{C}, X, \{X_p\}_{p \in P}, \{g_\tau\}_{\tau \in T}, \{\mathsf{tc}_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T}, \{r_\tau\}_{\tau \in T}, \{\mathsf{tpc}_\ell\}_{\ell \in L})$ be an atomic component with a semantics $(Q, \Sigma, \rightarrow)$. We say that B has non-decreasing deadlines if for any state (ℓ, v, t) such that port p is enabled from state (ℓ, v, t), we have:*

$$\forall \delta \in \mathbb{Z}_{\geq 0} \; . \; (\ell, v, t) \xrightarrow{\delta} (\ell, v, t + \delta) \Rightarrow (\ell, v, t) \xrightarrow{p} (\ell', v', t') \xrightarrow{\delta} (\ell', v', t' + \delta)$$

Intuitively, an atomic component satisfies the non-decreasing deadlines property if the following condition is satisfied: if time can progress by δ from a given state (ℓ, v, t) it can also progress from state (ℓ', v', t') reached by executing port p from (ℓ, v, t).

Sufficient conditions for satisfaction of non-decreasing deadlines property. Given a timing constraint tc defined as in (6.1), we denote by $L[\mathsf{tc}](c) = l_c$ and $U[\mathsf{tc}](c) = u_c$ the lower bound and the upper bound of tc over clock c respectively.

Example 22. *Consider the timing constraint* $\mathsf{tc} = 3 \leq c_1 \leq 4 \wedge 5 \leq c_2 \leq 7$. *We have* $L[\mathsf{tc}](c_1) = 3$, $L[\mathsf{tc}](c_2) = 5$, $U[\mathsf{tc}](c_1) = 4$ *and* $U[\mathsf{tc}](c_2) = 7$.

The following proposition gives sufficient conditions on time progress conditions and timing constraints that ensure satisfaction of the non-decreasing deadlines property. They are simple syntactic conditions that can be checked statically on transitions of atomic components.

Proposition 2. *An atomic component* $B = (L, P, T, \mathsf{C}, X, \{X_p\}_{p\in P}, \{g_\tau\}_{\tau\in T}, \{\mathsf{tc}_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T}, \{r_\tau\}_{\tau\in T}, \{\mathsf{tpc}_\ell\}_{\ell\in L})$ *such that all its transitions* $\tau = (\ell, p, \ell') \in T$ *satisfy the following syntactic conditions*

$$\begin{cases} U[\mathsf{tpc}_\ell](c) \leq U[\mathsf{tpc}_{\ell'}](c) & \text{if } c \notin r_\tau \\ U[\mathsf{tpc}_\ell](c) \leq U[\mathsf{tpc}_{\ell'}](c) + L[\mathsf{tc}_\tau](c) & \text{if } c \in r_\tau \end{cases}$$

has non-decreasing deadlines.

Proof. Given a transition $\tau = (\ell, p, \ell')$ enabled from state (ℓ, v, t), we have to prove that if a time step δ is enabled from (ℓ, v, t) then it is also enabled from state (ℓ', v', t') reached after executing transition τ.

First, as we suppose that transition τ is enabled from state (ℓ, v, t), this means that $\mathsf{tc}_\tau(t)$ must be true. Thus, the valuation of each clock c is greater than or equal to the lower bound specified by the timing constraint tc_τ over c. Formally we have:

$$\forall c \in \mathsf{C},\ t(c) \geq L[\mathsf{tc}_\tau](c) \tag{6.2}$$

Second, as we suppose that δ is allowed from state (ℓ, v, t), then for each clock c the valuation $t(c) + \delta$ must be less than or equal to the upper bound specified by tpc_ℓ for c. Formally, we have:

$$\forall c \in \mathsf{C},\ t(c) + \delta \leq U[\mathsf{tpc}_\ell](c) \tag{6.3}$$

Inequations (6.2) and (6.3) gives the following inequation:

$$\forall c \in \mathsf{C},\ \delta \leq U[\mathsf{tpc}_\ell](c) - L[\mathsf{tc}_\tau](c). \tag{6.4}$$

Now, we prove that δ is enabled from state (ℓ', v', t') by considering the syntactic conditions of Proposition 2. This boils down to prove that for each clock c the valuation $t'(c) + \delta$ is less than or equel to the upper bound of the time progress condition $\mathsf{tpc}_{\ell'}$, That is:

$$\forall c \in \mathsf{C},\ t'(c) + \delta \leq U[\mathsf{tpc}_{\ell'}](c).$$

To prove that, we consider the following two cases.

- If c is not reset by transition τ, i.e. $c \notin r_\tau$, then at state (ℓ', v', t'), the valuation of c remains the same as at state (ℓ, v, t), i.e. $t(c) = t('c)$. According to the first syntactic condition of Proposition 2 and (6.3), we prove that $t'(c) + \delta \leq U[\mathsf{tpc}_{\ell'}](c)$.

- If c is reset by transition τ, i.e. $c \in r_\tau$, then at state (ℓ', v', t') the valuation of c is 0, i.e. $t'(c) = 0$. Thus, we need to prove that $\delta \leq U[\mathsf{tpc}_{\ell'}](c)$. According to the second syntactic conditions of Proposition 2, we have $[\mathsf{tpc}_\ell](c) - L[\mathsf{tc}_\tau](c) \leq U[\mathsf{tpc}_{\ell'}](c)$. By using (6.4), we prove that $\delta \leq U[\mathsf{tpc}_{\ell'}](c)$.

□

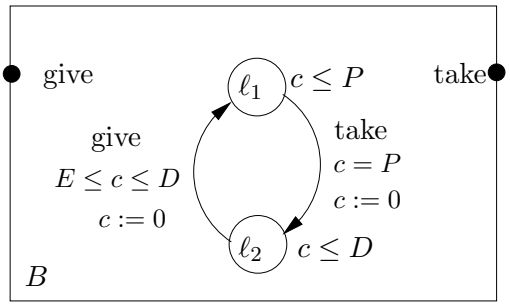


Figure 6.2: Example of atomic component having non-decreasing deadlines if $D \leq E + P$.

Example 23. *Figure 6.2 shows an atomic component B corresponding to a task that is processing items cyclically. It takes a new item to process via port* take*, and give it back after processing via port* give*. Transition $\tau_1 = (\ell_1, \text{take}, \ell_2)$ resets the clock c so as to measure the time it takes to give back an item after taking it. The time progress condition $c \leq P$ of ℓ_1 and the timing constraint $c = P$ of τ_1 ensures that τ_1 executes when c reaches P. We assume that B takes exactly E time units to process an item once it is taken. We also assume that once an item is processed, it can be kept by B for at most K time units before giving it back. For instance, B is a machine that is processing items at room temperature and they need to be kept cold or hot. This is represented by timing constraint $E \leq c \leq D$ for transition $\tau_2 = (\ell_2, \text{give}, \ell_1)$, where $D = E + K$. To enforce the execution of τ_2 before c reaches D, we also consider the time progress condition $\mathsf{tpc}_{\ell_2} = c \leq D$ for ℓ_2. According to conditions given in Proposition 2, component B satisfies the non-decreasing property only if $D \leq E + P$.*

Running Example

Figure 6.3 illustrates a BIP model $\gamma(B_1, B_2, B_3, B_4, B_5)$, which is the running example used throughout this chapter. Components B_2 and B_4 are instances of the component B of Figure 6.2, with E=3, D=P=5 for B_2 and E=6, D=P=8 for B_4. The set of interactions γ is $\{t_1, t_2, g_1, g_2\}$, where $t_1 = \{new_1, take_1\}$, $t_2 = \{new_2, take_2\}$, $g_1 = \{give_1, free\}$ and $g_2 = \{give_2, free\}$. Note that all components $B_1, \ldots, B_5$ have non-decreasing deadlines since they satisfy the syntactic conditions given in Proposition 2.

Initially, the system is at state $(\ell, 0, 0)$ where $\ell = (\ell_1^1, \ell_1^2, \ell_1^3, \ell_1^4, \ell_1^5)$. At this state, the time progress condition and the timing constraint in B_1 impose that the interaction t_1 executes after a delay of 5 time units. Then due to the time progress condition and the timing constraint in B_2, t_2 executes after a delay of 3 time units. The interaction g_1 executes then after a delay $\delta_1 \in [0, 2]$ then g_2 after a delay δ_2 such that $\delta_1 + \delta_2 \in [6, 8]$.

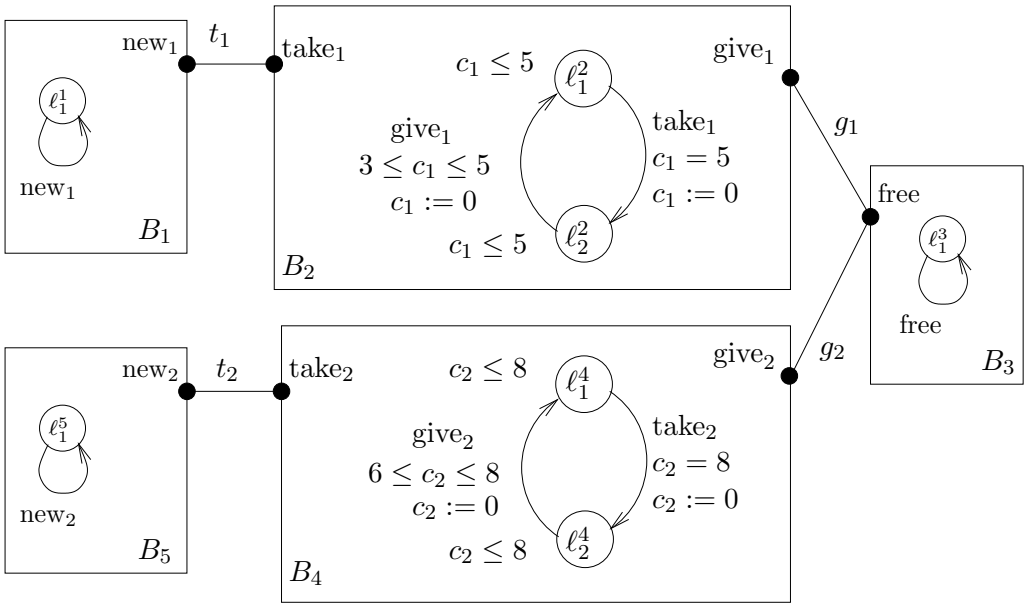


Figure 6.3: Example of BIP model that meets all restrictions.

6.2 Scheduling Issue of Earlier Decision Making Approach

Let $a = \{\{p_i \mid i \in I\}, G_a, F_a\}$ and $b = \{\{p'_j \mid j \in J\}, G_a, F_a\}$ be two conflicting interactions (i.e. such that $a\#b$) where $part(a) = \{B_i\}_{i\in I}$ and $part(b) = \{B_j\}_{j\in J}$. When interaction a is planned, the scheduling of interaction b needs to be blocked until a executes. If a is planed in δ_a time units, any component $B_i \in part(a)$ can only participate to a and is forced to wait for δ_a time units, which needs to be allowed by its corresponding time progress condition. Moreover, any component $B_j \in part(b)$ may participate to any interaction other than b. However, when the only enabled interaction is b, B_j cannot execute before the execution of a, that is, it is forced to wait for δ_a. In this case and if the time progress condition of B_j does not allow to wait for δ_a time units, scheduling a in δ_a introduces a timelock in B_j. That is, B_j cannot execute during δ_a time units but at the same time cannot wait for δ_a time units.

Note that such blocking situations cannot be reached when considering the approach of the last chapter. This is because the decision of executing a is taken at the very last moment i.e., its execution date. Before the decision for executing a is taken, there is no component "reserved" for executing a, in particular the components participating in conflicting interactions.

Example 24. *Figure 6.4 shows a BIP model composed of three components B_1, B_2 and B_3 with two conflicting interactions a and b. Suppose that initially the system is at state $((\ell_1, \ell_2, \ell_3), 0, 0)$. From this state, the schedulers may plan the execution of either interaction a or b, but not both, since interactions a and b are conflicting and both are enabled. Suppose that a is planed to execute in 5 time units. This means that components B_1 and B_2 are "reserved" and cannot participate in any other interaction before the execution of a. In particular, B_2 can not participate to interaction b for 5 time units. As a result, component B_3 is blocked*

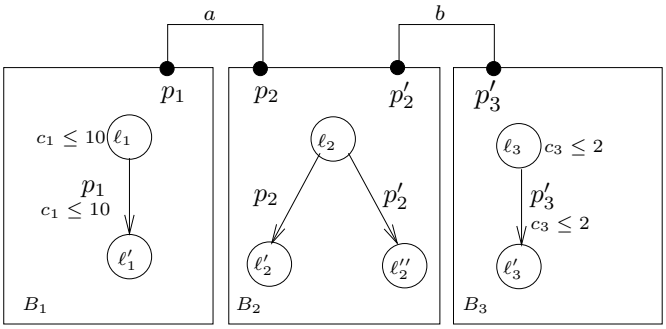


Figure 6.4: Scheduling Issue

for 5 *time units too as it can progress only by participating in interaction b through port* p'_3*. However, the time progress condition* $c_3 \leq 2$ *of component* B_3 *does not allow to wait for* 5 *time units for the initial state. Thus, scheduling a in* 5 *time units introduces a timelock in* B_3 *after 2 time units.*

6.2.1 Problem Formulation

We denote by $\mathcal{L}_{p_i}$ the set of locations of component B_i enabling port p_i, defined as follows: $\mathcal{L}_{p_i} = \{\ell_i \mid \wedge \tau_i = (\ell_i, p_i, \ell'_i) \in T_i\}$. Let $a = \{p_i\}_{i \in I}$ be an interaction. We denote by $\mathcal{L}_a$ the set of locations configurations enabling interaction a defined as follows: $\mathcal{L}_a = \bigotimes_{p_i \in a} \mathcal{L}_{p_i}$.

When scheduling a in δ_a time units from a location configuration $\ell_a \in \mathcal{L}_a$, it should be allowed by the timing constraint of transitions $\tau_i = (\ell_i, p_i, \ell'_i)$ of $B_i \in part(a)$ where $\ell_i \in \ell_a$ as well as the time progress conditions of locations ℓ_i. We define the predicate $\mathsf{sched\text{-}tc}_a[\ell_a]$ characterizing clocks valuations for which a could be scheduled from the configuration $\ell_a \in \mathcal{L}_a$ as follows:

$$\mathsf{sched\text{-}tc}_a[\ell_a] = \bigwedge_{\{\ell_i \in \ell_a \mid \tau_i = (\ell_i, p_i, \ell'_i) \in T_i\}} \mathsf{tc}_{\tau_i} \wedge \mathsf{tpc}_{\ell_i}.$$

In order to plan a from the configuration ℓ_a in δ_a time units, we have to verify that $\mathsf{sched\text{-}tc}_a[\ell_a](t + \delta_a)$ is true. Also, as explained in the previous section, we have to verify that each component B_j participating in a conflicting interaction b is allowed to wait for δ_a. As we consider components that satisfy the non-decreasing deadline property, we need only to check the time progress condition of the current state of B_j: if B_j can wait for δ_a from the current state, it will be able to do so even if it changes its state by executing other interactions.

We call *observed* components of an interaction a, denoted by $obs(a)$, the components that are not participating in a but that need to be observed in order to plan a. It is defined as follows:

$$obs(a) = \bigcup_{a \# b} part(b) \ \setminus \ part(a).$$

We denote by $\mathsf{safe\text{-}tc}_a[\ell_a]$ the predicate characterizing the valuations of clocks for which

a could be safely scheduled from the configuration ℓ_a. It is defined as follows:

$$\mathsf{safe\text{-}tc}_a[\ell_a] = \mathsf{sched\text{-}tc}_a[\ell_a] \bigwedge_{B_j \in obs(a)} \mathsf{tpc}_{\ell_j}.$$

In order to plan safely interaction a in δ_a from a state (ℓ, v, t) where the location ℓ enables a from configuration ℓ_a, we have to check that $\mathsf{safe\text{-}tc}_a[\ell_a](t + \delta_a)$ evaluates to true.

6.2.2 Proposed Solution For Send/Receive BIP Models

Let $a = (\{p_i\}_{i \in I}, G_a, F_a)$ be an interaction and let S_a be the scheduler handling a. In the Send/Receive BIP model proposed in the previous chapter, when scheduling an interaction a, S_a needs to receive offers only from components participating in a. However, this is not sufficient when considering the earlier decision making approach as S_a needs to observe components in $obs(a)$ to safely plan interaction a. In the Send/Receive BIP model presented in this chapter, we propose that the scheduler S_a receives not only offers from components participating in a but also observed components in $obs(a)$, that is components participating in conflicting interactions. The offers of observed components are used to observe their time progress conditions of components. Thanks to the non-decreasing deadlines assumption, the scheduler can safely use outdated offers of observed components to check their time progress conditions.

From offers received from participating components $B_i \in part(a)$, the scheduler S_a computes the scheduling timing constraint $\mathsf{sched\text{-}tc}_a$ of a as follows:

$$\mathsf{sched\text{-}tc}_a = \bigwedge_{p_i \in a} \mathsf{tc}_{p_i} \wedge \mathsf{tpc}_i$$

where tc_{p_i} and tpc_i are timing constraints and time progress conditions variables received in the offers of participating components B_i.

From offers received from observed components $B_j \in obs(a)$, the scheduler S_a restricts the scheduling timing constraint $\mathsf{sched\text{-}tc}_a$ into a safe scheduling timing constraint $\mathsf{safe\text{-}tc}_a$ as follows:

$$\mathsf{safe\text{-}tc}_a = \mathsf{sched\text{-}tc}_a \bigwedge_{B_j \in obs(a)} \mathsf{tpc}_j$$

where tpc_j are time progress conditions variables received in the offers of observed components B_j.

Consider again the example from Figure 6.3. Since g_1 and g_2 are conflicting, they observe the components of each others ,i.e. $obs(g_1) = \{B_4\}$ and $obs(g_2) = \{B_2\}$. Figure 6.5 shows the overall architecture of the Send/Receive distributed model of the example from Figure 6.3 considering the partition of interactions $\gamma_1 = \{t_1, g_1\}$ and $\gamma_2 = \{t_2, g_2\}$. As the component B_4 needs to be observed for interaction g_1, the corresponding Send/Receive atomic component B_4^{SR} sends offers to scheduler S_1 handling g_1. Similarly, Send/Receive atomic component B_2^{SR} sends offers to Scheduler S_2 as component B_4 needs to be observed for interaction g_2.

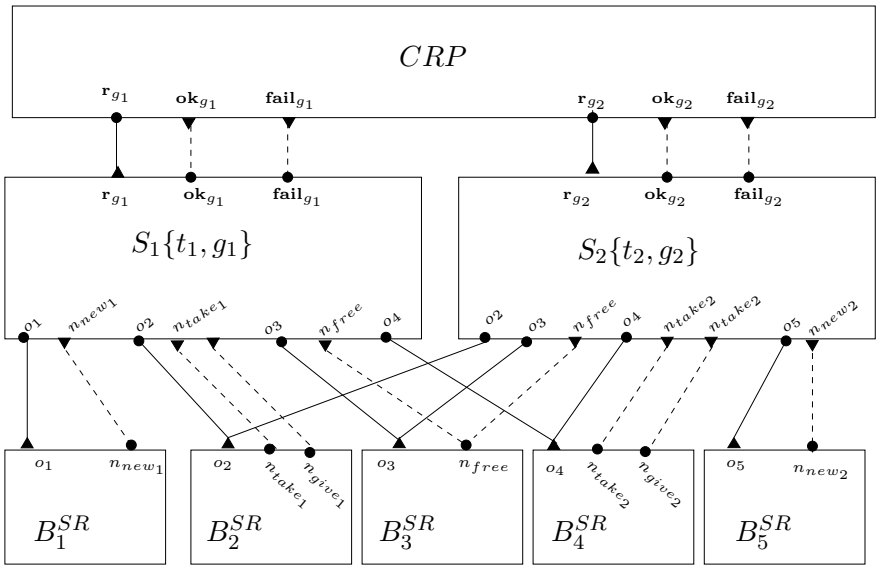


Figure 6.5: 3-layer distributed model of the example from Figure 6.3.

6.3 Transformations

6.3.1 Transformation of Atomic components

The transformation proposed here differs from the one of Definition 28 in the fact that it decouples notifications of components and their actual executions. Each notification brings the execution date chosen for the port participating in the interaction planed for execution by the scheduler. Once notified, a component wait for the corresponding date and then execute on the corresponding port.

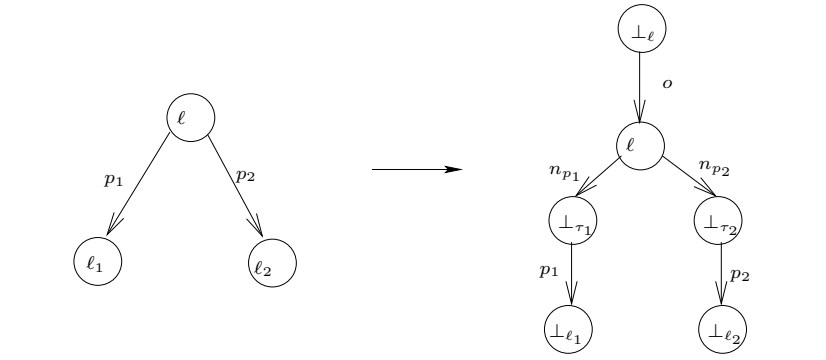


Figure 6.6: Example of transitions transformation.

Figure 6.6 shows the transformation of transitions labeled p_1 and p_2 enabled from location ℓ. In the transformed Send/Receive atomic component, we add the busy location $\perp_\ell$ from

which transition offer executes. Also, we add for transitions $\tau_1 = (\ell, p_1, \ell_1)$ and $\tau_2 = (\ell, p_2, \ell_2)$, busy locations $\perp_{\tau_1}$ and $\perp_{\tau_2}$ respectively. Transitions $(\ell, n_{p_1}, \perp_{\tau_1})$ and $(\ell, n_{p_2}, \perp_{\tau_2})$ correspond to the reception of notifications from the schedulers, regarding the execution of port p_1 and p_2 respectively. Notifications contain the date chosen for the execution of the selected port. Transitions $(\perp_{\tau_1}, p_1, \perp_{\ell_1})$ and $(\perp_{\tau_2}, p_2, \perp_{\ell_2})$ executes ports p_1 and p_2 respectively at the date received with the notification. In the following, we define formally the transformation of an atomic component into a Send/Receive atomic component.

Definition 35. *Let* $B = (L, P, T, \mathsf{C}, X, \{X_p\}_{p\in P}, \{g_\tau\}_{\tau\in T}, \{\mathsf{tc}_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T}, \{r_\tau\}_{\tau\in T}, \{\mathsf{tpc}_\ell\}_{\ell\in L})$ *be an atomic component. The corresponding Send/Receive atomic component is* $B^{SR} = (L^{SR}, P^{SR}, T^{SR}, \mathsf{C}^{SR}, X^{SR}, \{X_p^{SR}\}_{p\in P}, \{g_\tau\}_{\tau\in T^{SR}}, \{\mathsf{tc}_\tau\}_{\tau\in T^{SR}}, \{f_\tau\}_{\tau\in T^{SR}}, \{r_\tau\}_{\tau\in T^{SR}}, \{\mathsf{tpc}_\ell\}_{\ell\in L^{SR}})$ *such that:*

- $L^{SR} = L \cup \{\perp_\ell \mid \ell \in L\} \cup \{\perp_\tau \mid \tau \in T\}$. *The time progress condition of location* $\perp_\ell$ *and* ℓ *are equal to* tpc_ℓ *expressed on clock* g *and the time progress condition of location* $\perp_\tau$ *is* $g \leq t^{ex}$.
- $X^{SR} = X \cup \{\mathsf{tc}_p\}_{p\in P} \cup \{g_p\}_{p\in P} \cup \{\mathsf{tpc}\} \cup \{\rho_c\}_{c\in C} \cup \{t^{ex}\} \cup \{n\}$.
- $\mathsf{C}^{SR} = \{g\}$.
- $P^{SR} = P \cup \{o\} \cup \{n_p\}_{p\in P}$ *where* o *is a an offer port,* n_p *is a notification port and* p *is a unary port. The offer port* o *exports variables* $X_o^{SR} = \{n, \mathsf{tpc}\} \cup \bigcup_{p\in P}(X_p \cup \{\mathsf{tc}_p\} \cup \{g_p\})$*, that is, the participation number variable, the time progress condition variable, the timing constraints, guard variables, and variables associated with each port. Each notification port* n_p *exports the variables* $X_{n_p}^{SR} = X_p \cup \{t^{ex}\}$.
- *For each location* $\ell \in L$, T^{SR} *includes an offer transition* $\tau_\ell = (\perp_\ell, o, \ell)$. *This transition has no Boolean guard, no timing constraints and no update function.*
- *For each transition* $\tau \in T$, T^{SR} *includes a notification transition* $\tau_{n_p} = (\ell, n_p, \perp_\tau)$ *having no Boolean guard, no timing constraint and no update function, and an execution transition* $\tau_p = (\perp_\tau, p, \perp_{\ell'})$. *having no Boolean guard and a timing constraint* $g = t^{ex}$. *The update function* f_{τ_p} *first applies the original update function* f_τ *of* τ, *and updates the following variables:*
  - $\forall c \in r_\tau \quad \rho_c = t^{ex}$,
  - $\mathsf{tpc} := \mathsf{tpc}_{\ell'}$,
  - $\forall p' \in P \quad \mathsf{tc}_{p'} := \begin{cases} \mathsf{tc}_{\tau'} & \text{if } \tau' = (\ell', p', \ell'') \in T \\ \texttt{false} & \text{otherwise.} \end{cases}$
  - $\forall p' \in P \quad g_{p'} := \begin{cases} g_{\tau'} & \text{if } \tau' = (\ell', p', \ell'') \in T \\ \texttt{false} & \text{otherwise.} \end{cases}$
  - $n := n + 1$

Note that the time progress condition of location $\perp_\tau$ and the timing constraint of τ_p ensures the execution of port p at the chosen date t^{ex}. We recall that $\mathsf{tc}_{p'}$ and tpc are expressed using clock g and are computed using ρ_c as explained in Section 4.1.2.

In the above definition, the execution of a transition $\tau = (\ell, p, \ell')$ of a component B corresponds to the following three steps in B^{SR}. Firstly, an offer transition τ_ℓ transmits necessary information used by the schedulers for computing enabled interactions involving B^{SR}. Secondly, a notification transition τ_{n_p} is executed once the scheduler plans the execution of an interaction involving port p. The notification contains the date t^{ex} chosen by the scheduler for its execution. Finally, the execution transition τ_p executes the port p at the chosen date t^{ex}.

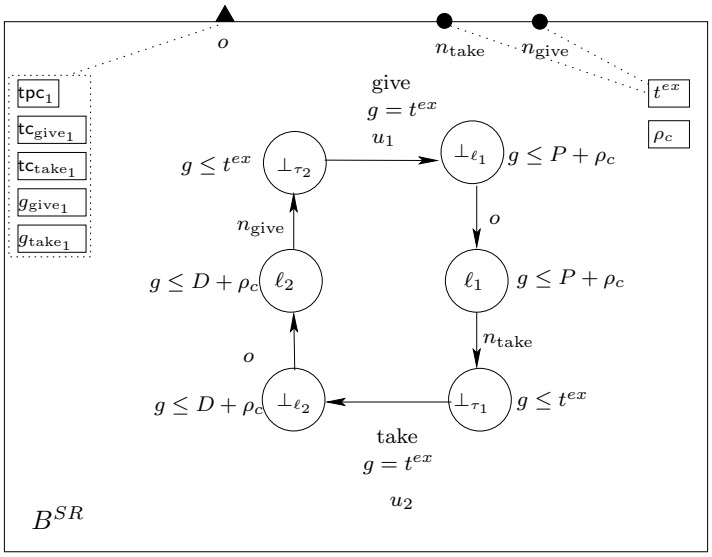


Figure 6.7: Transformation of component of Figure 6.2.

Figure 6.7 illustrates the transformation of the component B of Figure 6.2 into its corresponding Send/Receive component B^{SR}. Consider an execution in B^{SR} starting from location $\perp_{\ell_1}$. First, B^{SR} sends offers to the schedulers, containing the time progress condition of location ℓ_1, the timing constraints of ports give and take as well as their Boolean guard variables, and the participation number variable n_1. It receives then a notification from the schedulers specifying that the component should execute port take at date t^{ex}. Note that the time progress condition of location ℓ_1 forces B^{SR} to be notified and to leave ℓ_1 before the value of clock g exceeds $P + \rho_c$, or equivalently clock c exceeds P. After receiving notification on port n_{take}, the transition labeled by take executes at the chosen date t^{ex} and executes the update

function u_2 defined as follows.

$$u_2 = \begin{cases} \rho_c := t^{ex} \\ \mathsf{tc}_{\text{take}} := \texttt{false} \\ \mathsf{tc}_{\text{give}} := E + \rho_c \leq g \leq D + \rho_c \\ g_{\text{take}} := \texttt{false} \\ g_{\text{give}} := \texttt{true} \\ \mathsf{tpc} =: g \leq D + \rho_c \\ n++ \\ t^{ex} := +\infty \end{cases}$$

The update function u_1 of transition labeled by give is defined as follows.

$$u_1 := \begin{cases} \rho_c := t^{ex} \\ \mathsf{tc}_{\text{take}} := g = P + \rho_c \\ \mathsf{tc}_{\text{give}} := \texttt{false} \\ \mathsf{tpc} := g \leq P + \rho_c \\ g_{\text{take}} := \texttt{true} \\ g_{\text{give}} := \texttt{false} \\ \mathsf{tpc} := g \leq P + \rho_c \\ n++ \\ t^{ex} := +\infty \end{cases}$$

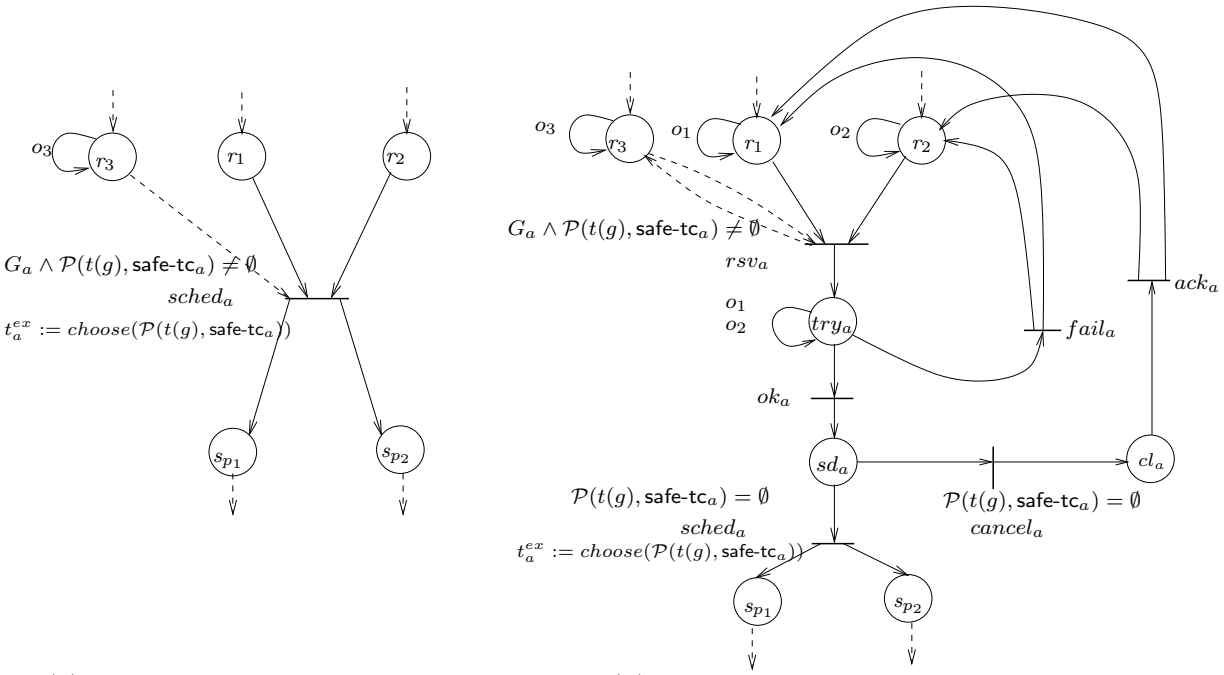
6.3.2 Building Distributed Schedulers

Given a BIP model $\gamma(B_1, \ldots, B_n)$ and a partition of interactions $\{\gamma_j\}_{j=1}^m$, we explain in this section how to build for each class γ_j the scheduler component S_j.

Let $a = \{\{p_i \mid i \in I\}, G_a, F_a\}$ be an interaction belonging to γ_j. Scheduling a needs receiving offers from components in $part(a)$ and $obs(a)$. Only offers from the participants need to be acknowledged by a response whenever the interaction a is planned. Regarding offers from observed components, they are used to compute a safe date for scheduling a. Note that only time progress conditions are used from offers received from observed components. In order to choose a safe date, S_j computes the safe timing constraint $\mathsf{safe\text{-}tc}_a$ that corresponds to the conjunction of timing constraints and time progress conditions of components participating in a with the time progress conditions of components observed by a. Given a scheduling policy $\mathcal{P}$, the safe timing constraint $\mathsf{safe\text{-}tc}_a$ and the actual valuation of clock g, S_j computes the set of safe dates $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a)$. If this set is not empty, S_j chooses a safe date t_a for scheduling a. Otherwise, a could not be scheduled since there is no safe date for its execution. We distinguish between the case where interaction a is internally or externally conflicting.

Figure 6.8 shows scheduling mechanisms of interaction a involving components B_1 and B_2 and observing component B_3.

- In the case where interaction a is internally or not conflicting, the scheduler executes transition $sched_a$ if the set $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a)$ is not empty. In this case, the scheduler chooses one safe date t_a^{ex} from the set $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a)$, such that $\mathsf{safe\text{-}tc}_a(t^e x_a)$ is true, and sends it to participating components B_1 and B_2.
- In the case where interaction a is externally conflict, the scheduler starts the reservation mechanism if the set $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a)$ is not empty. In the case where the conflict resolution protocol responds by $fail$ the scheduler returns back to received places. In the case where the conflict resolution protocol responds by ok, the scheduler checks again the set $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a)$. This check is necessary to ensure finding a valid date for scheduling interaction a. Indeed, if ok message from the conflict resolution protocol takes a long time to be received, it would be possible that the set $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a)$ becomes empty (e.g. $\mathsf{safe\text{-}tc}_a$ evaluates to false) which forbids the scheduler to find a valid date for scheduling a. In the case where the set $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a)$ is not empty, the scheduler proceeds to schedule interaction a by executing transition $sched_a$. In the other case, the scheduler sends a cancel request to the conflict resolution protocol on port $cancel_a$. This request is acknowledged by a message sent by the conflict resolution protocol on port ack_a.



(a) Scheduling mechanism in the case where a is internally or not conflicting.

(b) Scheduling mechanism for the case where a is externally conflicting.

Figure 6.8: Mechanisms for Interactions Scheduling.

Definition 36. *Let $\gamma(B_1, \ldots, B_n)$ be a BIP model and $\gamma_j \subset \gamma$ be a subset of interactions. The corresponding scheduler S_j handling interactions of γ_j is defined as the Send/Receive BIP*

component $S_j = (L^{S_j}, P^{S_j}, T^{S_j}, \mathsf{C}^{S_j}, X^{S_j}, \{X_p\}_{p\in P^{S_j}}, \{g_\tau\}_{\tau\in T^{S_j}}, \{\mathsf{tc}_\tau\}_{\tau\in T^{S_j}}, \{f_\tau\}_{\tau\in T^{S_j}}, \{r_\tau\}_{\tau\in T^{S_j}}, \{\mathsf{tpc}_\ell\}_{\ell\in L^{S_j}})$ *where:*

- *The set of variables* X^{S_j} *consists of the following.*
  - *Variables updated whenever an offer from component* B_i *that is participating in or observed by an interaction in* γ_j *is received. They consist of the following:*
    - *a timing constraint variable* tc_p*, a guard variable* g_p *and a local copy of the variables* X_p *for each port* p *of* B_i*.*
    - *a time progress condition variable* tpc_i *and a participation number variable* n_i *for each component* B_i*.*

    Note that for observed components, tc_p*,* g_p*,* X_p *and* n_i *are not used by* S_j*.*
  - *Variables updated whenever interaction* a *is scheduled. They consist of the following:*
    - *an execution date variable* t_a^{ex} *for each interaction* $a \in \gamma_j$*, that stores the last execution date of interaction* a*,*
    - *an execution date variable* t_i^{ex} *for each component* B_i *participating in an interaction belonging to* γ_j*.*
- $\mathsf{C}^{S_j} = \{g\}$*.*
- *The set of* L^{S_j} *contains the following places.*
  - *For each component* B_i *involved in interactions of* γ_j*,* L^{S_j} *includes* waiting *and* received *places* w_i *and* r_i*, respectively. Place* r_i *has a time progress condition defined by the variable* tpc_i*.*
  - *For each port* p *of component* B_i *participating in interactions of* γ_j*,* L^{S_j} *includes a* sending *place* s_p*. The time progress condition of this place is* $g \leq t_i^{ex}$*.*
  - *For each interaction* $a \in \gamma_j$ *that is externally conflicting,* L^{S_j} *includes a trying place* try_a*, scheduling place* sd_a *and a cancel place* cl_a*. The time progress condition of* try_a *and* sd_a *is defined by* $\bigwedge_{B_i\in part(a)} \mathsf{tpc}_i$*. The other places have no time progress condition.*
- *The set of ports* P^{S_j} *consists of the following:*
  - *For each component* B_i*,* P^{S_j} *includes an offer port* o_i*. Each port* o_i *is associated with the variables* tc_p*,* g_p *and* X_p *for each port* p *of* B_i *as well as the variable* tpc_i *and* n_i *of* B_i*.*
  - *For each port* p *of component* B_i *participating in interactions* γ_j*,* P^{S_j} *includes a send-port* n_p*, which exports the set of variables* $X_p \cup \{t_i^{ex}\}$*.*
  - *We include a unary port* $sched_a$ *for each interaction* $a \in \gamma_j$*.*
  - *For each interaction* $a \in \gamma_j$ *that is externally conflicting,* P^{S_j} *includes ports* rsv_a*,* ok_a*,* $fail_a$*,* $cancel_a$ *and* ack_a*. The port* rsv_a *exports the variables* $\{n_i\}_{B_i\in part(a)}$*. The other ports do not export any variable.*

- *The set of transitions T^{S_j} consists of the following:*
  - *In order to receive offers from a component B_i participating in or observed by an interaction of γ_j, T^{S_j} includes an offer transition (w_i, o_i, r_i). It includes also transitions (r_i, o_i, r_i) and $\{(try_a, o_i, try_a) \mid B_i \in part(a) \cup obs(a)\}$ to receive new offers when B_i takes part in other conflicting interaction not handled by S_j. These transitions have no Boolean guards, no timing constraints and no update functions.*
  - *In order to send notification, T^{S_j} includes for each port p of component B_i participating in an interaction belonging to γ_j, a notification transition (s_p, n_p, w_i) with no Boolean guard, no timing constraint and no update function.*
  - *For each interaction $a = \{\{p_i \mid i \in I\}, G_a, F_a\} \in \gamma_j$ that is internally or not conflicting, T^{S_j} includes transition $\tau_{sched_a} = (\{r_i \mid B_i \in part(a) \cup obs(a)\}, sched_a, \{s_{p_i} \mid B_i \in part(a)\} \cup \{r_i \mid B_i \in obs(a)\})$ guarded by $G_a \wedge \mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a) \neq \emptyset$ and having no timing constraint. The update function $f_{\tau_{sched_a}}$ updates the following variables.*
    - *$t_a^{ex} := choose(\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a))$, such that $\mathsf{safe\text{-}tc}_a(t^e x_a)$*
    - *$\forall\ B_i \in part(a)\ \ t_i^{ex} := t_a^{ex}$ and applies the data tansfer function F_a.*
  - *For each interaction $a = \{\{p_i \mid i \in I\}, G_a, F_a\} \in \gamma_j$ that is externally conflicting, T^{S_j} includes the following transitions*
    - *$\tau_{rsv_a} = (\{r_i \mid B_i \in part(a) \cup obs(a)\}, rsv_a, \{try_a\} \cup \{r_i \mid B_i \in obs(a)\})$ guarded by $G_a \wedge \mathcal{P}(\mathsf{safe\text{-}tc}_a, t(g)) \neq \emptyset$ and having no timing constraint and no update function.*
    - *$\tau_{ok_a} = (try_a, ok_a, sd_a)$ with no Boolean guard, no timing constraint and no update function.*
    - *$\tau_{fail_a} = (try_a, fail_a, \{r_i \mid B_i \in part(a)\})$ with no guard, no timing constraint and no update function.*
    - *$\tau_{cancel_a} = (sd_a, cancel_a, cl_a)$ guarded by $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a) = \emptyset$ and having no timing constraint and no update function.*
    - *$\tau_{ack_a} = (sd_a, ack_a, \{r_i \mid B_i \in part(a)\})$ with no Boolean guard, no timing constraint and no update function.*
    - *$\tau_{sched_a} = (sd_a, sched_a, \{s_{p_i} \mid B_i \in part(a)\})$ guarded by $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a) \neq \emptyset$, having no timing constraint. The update function $f_{\tau_{sched_a}}$ updates the following variables*
      - *$t_a^{ex} := choose(\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a))$, such that $\mathsf{safe\text{-}tc}_a(t^e x_a)$*
      - *$\forall\ B_i \in part(a)\ \ t_i^{ex} := t_a^{ex}$ and applies the data tansfer function F_a.*

Note that the time progress conditions of place try_a ($\bigwedge_{B_i \in part(a)} \mathsf{tpc}_i$) ensures that the response from the conflict resolution protocol is received before that one of the time progress conditions of components participating in a becomes. Also, the time progress conditions of place sd_a ($\bigwedge_{B_i \in part(a)} \mathsf{tpc}_i$) ensures that scheduling a is done before that one of the time progress conditions of components participating in a becomes false. The time progress condition of place s_p ($g \leq t_i^{ex}$), where p is a port of component B_i, ensures that the scheduler sends the notification to component B_i before the execution date t_i^{ex} expiration.

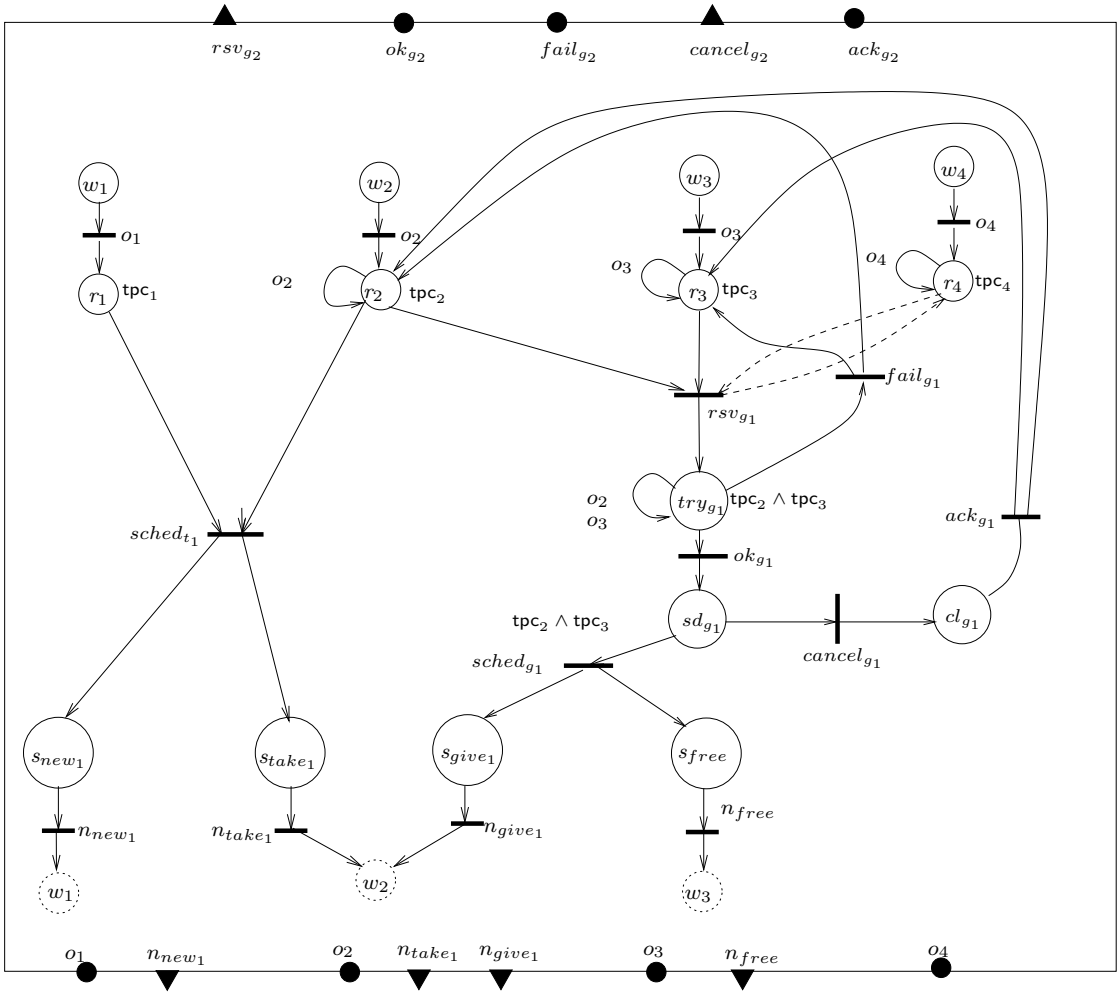


Figure 6.9: Decentralized Scheduler S_2 of Figure 6.5

Figure 6.9 shows the scheduler component S_1 of Figure 6.5. S_1 handles interactions t_1 and g_1. For sake of readability, the guards and update functions of transitions are hidden. As t_1 is not conflicting with any other interaction, it is handled locally in S_1. Interaction g_1 is in external conflict with g_2. To schedule interaction g_1, S_1 has to receive offers from participating components B_2 and B_3 as well as offers from observed component B_4. Its scheduling requires requesting reservation from the conflict resolution protocol through port rsv_{g_1}. If the reservation of interaction g_1 succeeds (transition labeled by *ok* executes) and if there is no safe date for its scheduling, S_1 cancel the interaction by sending a cancel request to the conflict resolution protocol. An acknowledgment from the conflict resolution protocol is sent on port *ack* to confirm the cancel. In the case where S_1 succeed to find a date for scheduling g_1, it executes the transition labeled by $sched_a$ and then notifies components B_2 and B_3.

6.3.3 Conflict Resolution Protocol

As explained in Subsection 5.3.3, the main role of the conflict resolution protocol is to check the validity of offers received for the execution of interaction a. This is done by comparing for each component B_i the last participation numbers N_i that is recorded by the conflict resolution protocol and the participation number n_i arriving along with the reservation request. In addition to that, the conflict resolution protocol has to acknowledge cancel requests sent by the schedulers. The cancel request returns back the value of variables N_i of each component B_i participating in a to the last value before sending an ok_a message. This allows each other conflicting interaction b that fails because of interaction a, to succeed when the scheduler handling b sends an other reservation request. To do this, we define new variable N_i^{cl} that stores the last values of variables N_i when an ok_a message is sent. When a cancel request for interaction a is received, the conflict resolution protocol puts backs the values of N_i to the values of N_i^{cl}.

In the following we show how to adapt the different implementations of the conflict resolution protocol presented in Subsection 5.3.3 in order to integrate the cancel mechanism.

Centralized Protocol

We adapt Definition 37 in order to integrate cancel mechanism in CP component.

Definition 37. *Let $\gamma(B_1, \ldots, B_n)$ be a BIP model and $\gamma_j \subset \gamma$ be a subset of partition of interactions. The corresponding centralized conflict resolution protocol CP is defined as the Send/Receive BIP component $CP = (L^{CP}, P^{CP}, T^{CP}, \mathsf{C}^{CP}, X^{CP}, \{X_p\}_{p \in P^{CP}}, \{g_\tau\}_{\tau \in T^{CP}}, \{\mathsf{tc}_\tau\}_{\tau \in T^{CP}}, \{f_\tau\}_{\tau \in T^{CP}}, \{r_\tau\}_{\tau \in T^{CP}}, \{\mathsf{tpc}_\ell\}_{\ell \in L^{CP}})$*

- *L^{CP} contains for each externally interactions the waiting place w_a, receive place r_a and cancel place cl_a.*
- *X^{CP} contains for each component B_i the variable N_i. In addition X^{CP} contains for each externally conflicting interaction a and for each component $B_i \in part(a)$, the variable n_i^a and the variables N_i^{cl} where N_i^{cl} are cancel variables needed to cancel the reservation of interaction a.*
- *$C^{CP} = \{g\}$.*
- *P^{CP} contains for each externally conflicting interaction a the reservation ports rsv_a, ok_a and $fail_a$. The port rsv_a exports variables $X_{rsv_a} = \{n_i^a \mid B_i \in part(a)\}$. The ports ok_a and $fail_a$ do not have exported variables. In addition, P^{CP} contains the cancel reservation ports $cancel_a$ and ack_a. These ports do no export variables.*
- *T^{CP} contains for each externally conflicting interaction a the following transitions:*
  - *$\tau_{rsv_a} = (w_a, rsv_a, r_a)$ with no Boolean guard, no timing constraint and no update function.*
  - *$\tau_{ok_a} = (r_a, ok_a, w_a)$ guarded by $g_{\tau_{ok_a}} = \bigwedge_{B_i \in part(a)} n_i^a > N_i$ and having no timing constraint. The update function of this transition updates the variables N_i^{cl} of each*

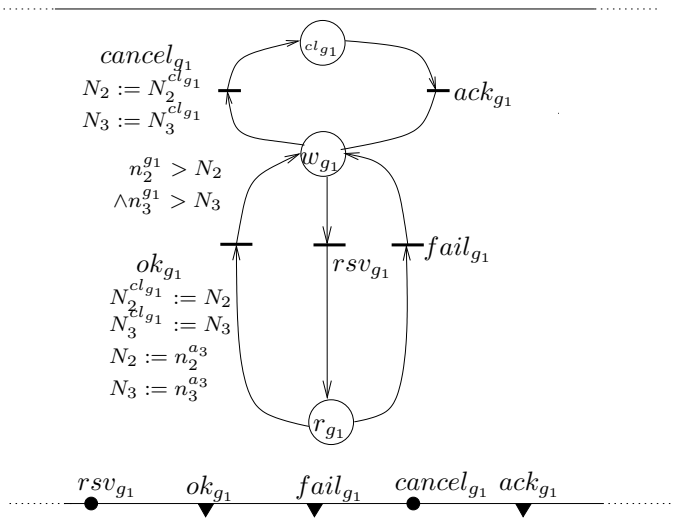


Figure 6.10: Fragment of the centralized version of conflict resolution protocol for handling interaction g_1.

component $B_i \in part(a)$ to N_i, and then updates variables N_i of each component $B_i \in part(a)$ to the value of n_i^a.

- *$\tau_{fail_a} = (r_a, fail_a, w_a)$ having no guard, no timing constraint and no update function.*
- *$\tau_{cancel_a} = (w_a, cancel_a, cl_a)$ with no Boolean guard and no timing constraint. This transition updates variables N_i to N_i^{cl} of each component $B_i \in part(a)$.*
- *$\tau_{ack_a} = (cl_a, ack_a, w_a)$ with no guard, no timing constraint and no update function.*

Note that in contrast to Definition 37, we do not put `false` as time progress condition in place r_a. This is to model the fact that the centralized protocol component may take arbitrary time to send a response to the scheduler. For the implementation point of view, this means that the communication delay between the centralized protocol component and the scheduler could be non-null.

Example 25. *Figure 6.10 presents a fragment of the centralized version of conflict resolution protocol for handling interaction g_1 of the model from Figure 6.3. Initially, there is a token in waiting place w_{g_1}. Upon receiving a reservation request, the token moves to r_{g_1} place. The transition labeled by ok_{g_1} can take place only if the freshness offers condition is true ($n_2^{g_1} > N_2 \wedge n_3^{g_1} > N_3$). This transition updates the variables N_2^{cl} and N_3^{cl} to the actual values of N_2 and N_3 respectively, and then updates variables N_2 and N_3 to the value of variables $n_2^{g_1}$ and $n_3^{g_1}$. When one of these transition (ok or fail) executes, the token returns back to w_{g_1} place. From this place, the transition labeled by $cancel_{g_1}$ takes place when receiving a cancel request from the scheduler. This transition updates variables N_2 and N_3 to the values of variables N_2^{cl} and N_3^{cl} respectively. That is, the variables N_2 and N_3 returns back to their values before sending an ok_a message. Upon execution of this transition, the token moves to cl_{g_1} place. As a cancel request has to be acknowledged, there is only a transition labeled by*

ack_{g_1} enabled from cl_{g_1} place. The execution of this transition returns back the token to w_{g_1} place.

Token Ring Protocol

We adapt Definition 38 in order to integrate cancel mechanism in the components of the token ring protocol.

Let TR_a be a component of the token ring protocol handling interaction a. As in the case of centralized protocol component CP, TR_a component stores the values of variables N_i in N_i^{cl} variables when sending an ok_a message. When a cancel request is received, TR_a component puts back the values of variables N_i to the values of variables N_i^{cl}. The cancel request is acknowledged by an *ack* message.

Definition 38. *Let $\gamma(B_1, \ldots, B_n)$ be a BIP model and $\gamma_j \subset \gamma$ be a subset of partition of interactions. Given an externally conflicting interaction a, the corresponding conflict resolution protocol component TR_a is defined as the Send/Receive BIP component $TR_a = (L^{TR_a}, P^{TR_a}, T^{TR_a}, \mathsf{C}^{TR_a}, X^{TR_a}, \{X_p\}_{p \in P^{TR_a}}, \{g_\tau\}_{\tau \in T^{TR_a}}, \{\mathsf{tc}_\tau\}_{\tau \in T^{TR_a}}, \{\tau\}_{\tau \in T^{TR_a}}, \{r_\tau\}_{\tau \in T^{TR_a}}, \{\mathsf{tpc}_\ell\}_{\ell \in L^{TR_a}})$ where:*

- *L^{TR_a} includes the waiting place w_a, the receiving place r_a, the receiving token place rt_a, the sending token place st_a and cancel place cl_a. All places have not time progress conditions.*
- *X^{TR_a} includes the variables $\{N_i\}_{i=1}^n$, the variables $\{n_i^a \mid B_i \in part(a)\}$ and the variables $\{N_i^{cl} \mid B_i \in part(a)\}$.*
- *P^{TR_a} contains the reservation ports rsv_a, ok_a, $fail_a$ and $cancel_a$ as well as the sending and receiving token ports RT_a, ST_a.*
- *T^{TR_a} includes the following transitions:*
  - *transitions for reserving and cancelling interaction a:*
    - *$\tau_{rsv_a} = (w_a, rsv_a, r_a)$, with no guard, no timing constraint and no update function.*
    - *$\tau_{ok_a} = (\{r_a, rt_a\}, ok_a, w_a)$ guarded by $\bigwedge_{B_i \in part(a)} n_i^a > N_i$ and having no timing constraint. The update function of this transition sets the variables N_i^{cl} of each component $B_i \in part(a)$ to N_i and, then updates variables N_i of each component $B_i \in part(a)$ to the value of n_i^a.*
    - *$\tau_{fail_a} = (r_a, fail_a, w_a)$ guarded by $\bigvee_{B_i \in part(a)} n_i^a \leq N_i$, having no timing constraint and no update function.*
    - *$\tau_{cancel_a} = (w_a, cancel_a, clcl_a)$ with no guard and no timing constraint. This transition updates variables N_i to N_i^{cl} of each component $B_i \in part(a)$.*
    - *$\tau_{ack_a} = (cl_a, ack_a, w_a)$ with no guard, no timing constraint and no update function.*
  - *transitions for sending and receiving token:*

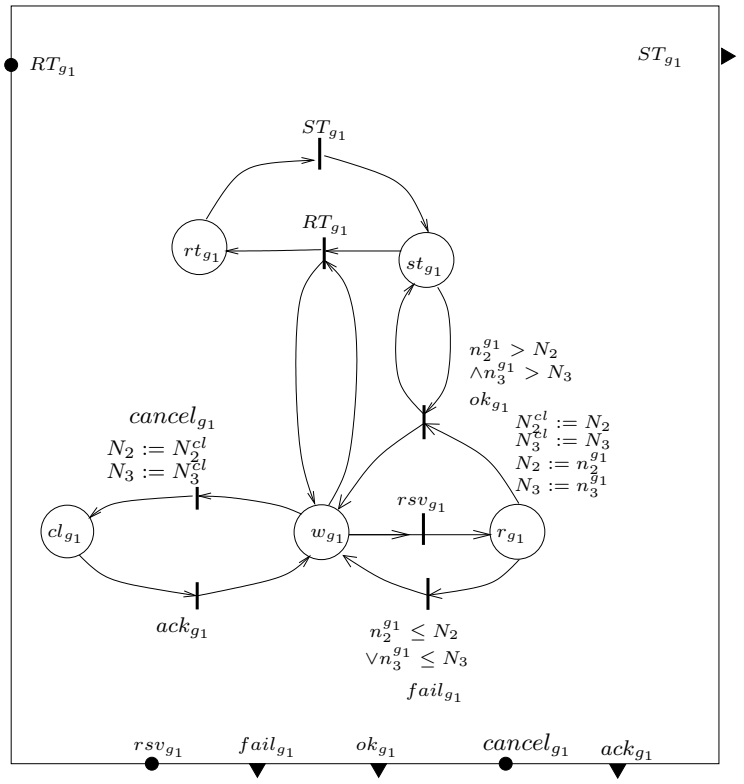


Figure 6.11: Conflict resolution protocol component of interaction g_1 in the token ring protocol.

* $\tau_{ST_a} = (\{st_a, w_a\}, ST_a, rt_a)$ *with no guard, no timing constraint and no update function.*
* $\tau_{RT_a} = (w_a, RT_a, st_a)$ *having no guard, no timing constraint and no update function.*

In order to model the fact that communications may take arbitrarily delays, we do not put a time progress condition on places r_a and st_a. That is, the component TR_a may take arbitrary time to either send are response to the scheduler or to send the token.

Example 26. *Figure 6.11 depicts the conflict resolution protocol component handling interaction g_1. The component is initially at waiting and receiving token places, i.e. w_{g_1} and st_{g_1}. Upon receiving a reservation request, the token moves to r_{g_1} place. The transition labeled by ok_{g_1} is executed only if the offers validity condition is true and the component holds the token. This transition updates variables N_2^{cl} and N_2^{cl} to the current values of N_2 and N_3 and then updates variables N_2 and N_3 to $n_2^{g_1}$ and $n_3^{g_1}$ respectively. The transition labeled by $fail_{g_1}$ and executes if one of the offers is not fresh. From place w_{g_1}, the transition labeled $cancel_{g_1}$ can execute if the component receives a cancel request from the scheduler. This transition returns back the values of N_2 and N_3 to the values of N_2^{cl} and N_3^{cl}. The execution of this transition enables the execution of transition labeled by ack_{g_1} so as the component may acknowledge the cancel request.*

Note that the interactions γ^{TR} between components of the token ring protocol is the same as in the version described in Subsection 5.3.3.

Dining philosophers Protocol

We adapt Definition 32 in order to integrate cancel mechanism in the components of dining philosophers protocol. Let DP_a be the conflict resolution protocol component handling interaction a. Component DP_a defines variables N_i^{cl} and updates them in the same way as in the centralized and the token ring protocol. When receiving a cancel request for interaction a its puts back the values of variables N_i to the values of variables N_i^{cl} s.t. $B_i \in part(a)$. Then, it collects all forks shared with all components DP_b such that b is an externally conflicting interaction with a in order to update their associated variables N_i^b s.t. $B_i \in conf(a,b)$. Recall that $conf(a,b) = part(a) \cap part(b)$ and $extconf(a)$ is the set of interactions externally conflicting with a.

Definition 39. *Let $\gamma(B_1,\ldots,B_n)$ be a BIP model and $\{\gamma_j\}_{j=1}^m$ be a partition of interactions. Given an externally conflicting interaction a, the corresponding conflict resolution component DP_a is defined as the Send/Receive BIP component $DP_a = (L^{DP_a}, P^{DP_a}, T^{DP_a}, \mathsf{C}^{DP_a}, X^{DP_a}, \{X_p\}_{p\in P^{DP_a}}, \{g_\tau\}_{\tau\in T^{DP_a}}, \{\mathsf{tc}_\tau\}_{\tau\in T^{DP_a}}, \{f_\tau\}_{\tau\in T^{DP_a}}, \{r_\tau\}_{\tau\in T^{DP_a}}, \{\mathsf{tpc}_\ell\}_{\ell\in L^{DP_a}})$, where:*

- *L^{DP_a} contains waiting place w_a, and the received place r_a. Moreover, for each interaction b in $extconf(a)$, L^{DP_a} includes places for fork negotiation: sr_b, wr_b, sf_b, wf_b, and rf_b. For canceling interaction a, L^{DP_a} includes, for each interaction b in $extconf(a)$, places sr_b^{cl}, wf_b^{cl}, rf_b^{cl}. All places do not have time progress conditions.*
- *X^{DP_a} includes the variables $\{N_i \mid B_i \in part(a)\}$, the variables $\{N_i^{cl} \mid B_i \in part(a)\}$, the variables $\{n_i^a \mid B_i \in part(a)\}$. Moreover, for each interaction $b \in extconf(a)$, X^{DP_a} includes the additional variables $\{N_i^b \mid B_i \in conf(a,b)\}$, and the Boolean variables $fork_b$, $dirty_b$, cl_b.*
- *P^{DP_a} includes the reservation ports rsv_a, ok_a, $fail_a$ and the unary port u_a. Moreover, P^{DP_a} includes, for each interaction $b \in extconf(a)$, the ports SR_b^a, RR_b^a, SF_b^a and RF_b^a. Port rsv_a exports variables $\{n_i^a \mid B_i \in part(a)\}$. Port SF_b^a exports variables $\{cl_b\} \cup \{N_i \mid B_i \in conf(a,b)\}$ and port RF_b^a exports variables $\{cl_b\} \cup \{N_i \mid B_i \in conf(a,b)\}$. The other ports do not have exported variables.*
- *T^{DP_a} contains the following transitions:*
  - *transitions for scheduling interaction a.*
    - *$\tau_{rsv_a} = (w_a, rsv_a, r_a)$ with no guard, no timing constraint and no update function.*
    - *$\tau_{ok_a} = (\{rf_b | b \in extconf(a)\}, ok_a, w_a)$ guarded by $\bigwedge_{b\in extconfa} fork_b \bigwedge_{B_i\in part(a)} n_i^a > N_i$, having no timing constraint and updating the following variables:*
      $\forall\ B_i \in part(a)\quad N_i := n_i^a$
      $\forall\ B_i \in conf(a,b)\quad N_i^b := N_i$
      $\forall\ b \in extconf(a)\quad dirty_b :=$ `true`

* $\tau_{check_a} = (r_a, u_a, \{sr_b \mid b \in extconf(a)\})$ *guarded by* $\bigwedge_{B_i \in part(a)} n_i^a > N_i$ *and having no timing constraint and no update function*
* $\tau_{fail_a^1} = (r_a, fail_a, w_a)$ *and* $\tau^2_{fail_a} = (\{rf_b | b \in extconf(a)\}, fail_a, w_a)$ *both guarded by* $\bigvee_{B_i \in part(a)} n_i^a \leq N_i$ *and having no timing constraint. Transition* $\tau_{fail_a^1}$ *has no update function. Transition* $\tau^2_{fail_a}$ *has the following update function:* $\forall\ b \in extconf(a)\ \ dirty_b := \texttt{true}$.
* *For each interaction* $b \in extconf(a)$,
  - $\tau_{SR_b^a} = (sr_b, SR_b^a, wf_b)$, *with Boolean guard* $\neg fork_b$, *no timing constraint and* $cl_b := \texttt{false}$ *as update function.*
  - $\tau_{RR_b^a} = (wr_b, RR_b^a, sf_b)$ *with no guard, no timing constraint and no update function.*
  - $\tau_{SF_b^a} = (sf_b, SF_b^a, wr_a)$ *with Boolean guard* $dirty_b$, *no timing constraint and having the following update function:*
    $fork_b := \texttt{false}$
    if (cl_b) **then** $cl_b := \texttt{false}$.
  - $\tau_{RF_b^a} = (wf_b, RF_b^a, rf_b)$ *with no guard, no timing constraint and updating the following variables:*
    $dirty_b := \texttt{false}$
    $fork_b := \texttt{true}$
    if$(\bigwedge_{c \in extconf(a)} \neg cl_c)$ **then** $\forall\ B_i \in conf(a,b)\ \ N_i := max(N_i, N_i^b)$
    if(cl_b) **then** $\forall\ B_i \in conf(a,b)\ \ N_i := N_i^b$

    $\forall\ c \in extconf(a)\ \ \forall\ B_i \in conf(a,c)\ \ N_i^c := N_i$
  - $\tau_{hasfork_b} = (sr_b, u_a, rf_b)$ *guarded by* $fork_b$ *and having no timing constraint and no update function.*
  - $\tau_{hasnotfork_b} = (rf_b, u_a, sr_b)$ *guarded by* $\neg fork_b$ *and having no timing constraint and no update function.*

– *Transitions for canceling interaction a:*

* $\tau_{cancel_a} = (w_a, cancel_a, sr_b^a)$ *having no guard, no timing constraint and having the following update function:*
  $\forall\ B_i \in part(a)\ \ N_i := N_i^{cl}$
  $\forall\ b \in extconf(a)\ \ dirty_b := \texttt{false}\ \ cl_b := \texttt{true}$.
* *For each interaction* $b \in extconf(a)$:
  - $\tau_{SR_b^a} = (sr_b^a, SR_b^a, wf_b^a)$ *guarded by* $\neg fork_b$, *having no timing constraint and no update function.*
  - $\tau_{RF_b^a} = (wf_b^a, RF_b^a, rf_b^a)$ *having no guard, no timing constraint and having the following update function:*
    $fork_b := \texttt{true}$
    $\forall\ c \in extconf(a)\ \ \forall\ B_i \in conf(a,c)\ \ N_i^c := N_i$.

- $\tau_{hasfork_b^{cl}}$=$(sr_b^{cl}, u_a, rf_b^{cl})$ *guarded by* $fork_b$*, and having no timing constraint and* $dirty_b :=$ `false` *as update function.*
- $\tau_{ack_a} = (\{rf_b^{cl} | b \in extconf(a)\}, ack_a, w_a)$ *with no guard, no timing constraint and having the following update function:*
  $\forall\ b \in extconf(a)$ $dirty_b :=$ `true`

Example 27. *Similarly to the case of token ring protocol, we model the fact that communications may take arbitrarily delays, we do not put a time progress condition on places* r_a *and* st_a*. That is, the component* DP_a *may take arbitrary time to either send are response to the scheduler or to send the forks.*

Figure 6.12 shows the conflict resolution component DP_{g_1} *handling interaction* g_1*. Initially, there are tokens at places* w_{g_1} *and* wr_{g_2}*. When receiving a reservation request, the component* DP_{g_1} *requests the fork from* DP_{g_2} *if it does not hold it. The fork brings the last value of* $N_3^{g_2}$ *and the Boolean variable* cl_{g_2}*. Note that only variable* $N_3^{g_2}$ *is associated with the fork as the only shared component between* g_1 *and* g_2 *is* B_3*. The Boolean variable* cl_{g_2} *is used to inform* DP_{g_1} *whether interaction* g_2 *was canceled or not. If* g_2 *was not canceled (*cl_{g_2} *is false), then* DP_{g_1} *updates its local variables* N_3 *to the maximum value between* N_3 *and* $N_3^{g_2}$*. In the other case,* DP_{g_1} *updated* N_3 *to* $N_3^{g_2}$*.*

If the freshness condition is true (that is, $N_2 \leq n_2^{g_1} \wedge N_3 \leq n_3^{g_1}$*), the transition labeled by* ok_{g_1} *executes. By executing this transition,* DP_{g_1} *stores the last values of* N_2 *and* N_3 *in variables* N_2^{cl} *and* N_3^{cl} *respectively. The latters are used to put back the values of* N_i *when a cancel request is received from the scheduler. Upon receiving a cancel request, the transition labeled by* $cancel_{g_1}$ *executes and the variables* N_2 *and* N_3 *are set to the values of variables* N_2^{cl} *and* N_3^{cl} *respectively. In order to propagate these new values, the component* DP_{g_1} *requests the fork from* DP_{g_2}*. This transition executes only if* DP_{g_1} *does not hold the fork. When receiving the fork, the variable* $N_3^{g_2}$*, associated with the fork, is updated to the new value of* N_3*.*

The set of interactions γ^{DP} between components of the dining philosophers protocol includes the same interactions as in the version described in Subsection 5.3.3.

6.3.4 Connections Between Layers

The set of Send/Receive interactions γ^{SR} of the obtained Send/Receive model includes the same interactions between the layers as described in Subsection 5.3.4 expect for notification interactions between schedulers $S_1, \ldots, S_m$ and Send/Receive atomic components B_i^{SR} that are defined as follows:

- For each port p of B_i and for each scheduler component S_j handling an interaction involving p, γ^{SR} includes the notification interaction $(S_j.n_p, B_i^{SR}.n_p)$.

The set γ^{SR} includes also for each externally conflicting interaction a, the interactions between schedulers $S_1, \ldots, S_m$ and the conflict resolution protocol CRP defined as follows:

- $(cancel_a.S_j, cancel_a.C_{CRP})$, and

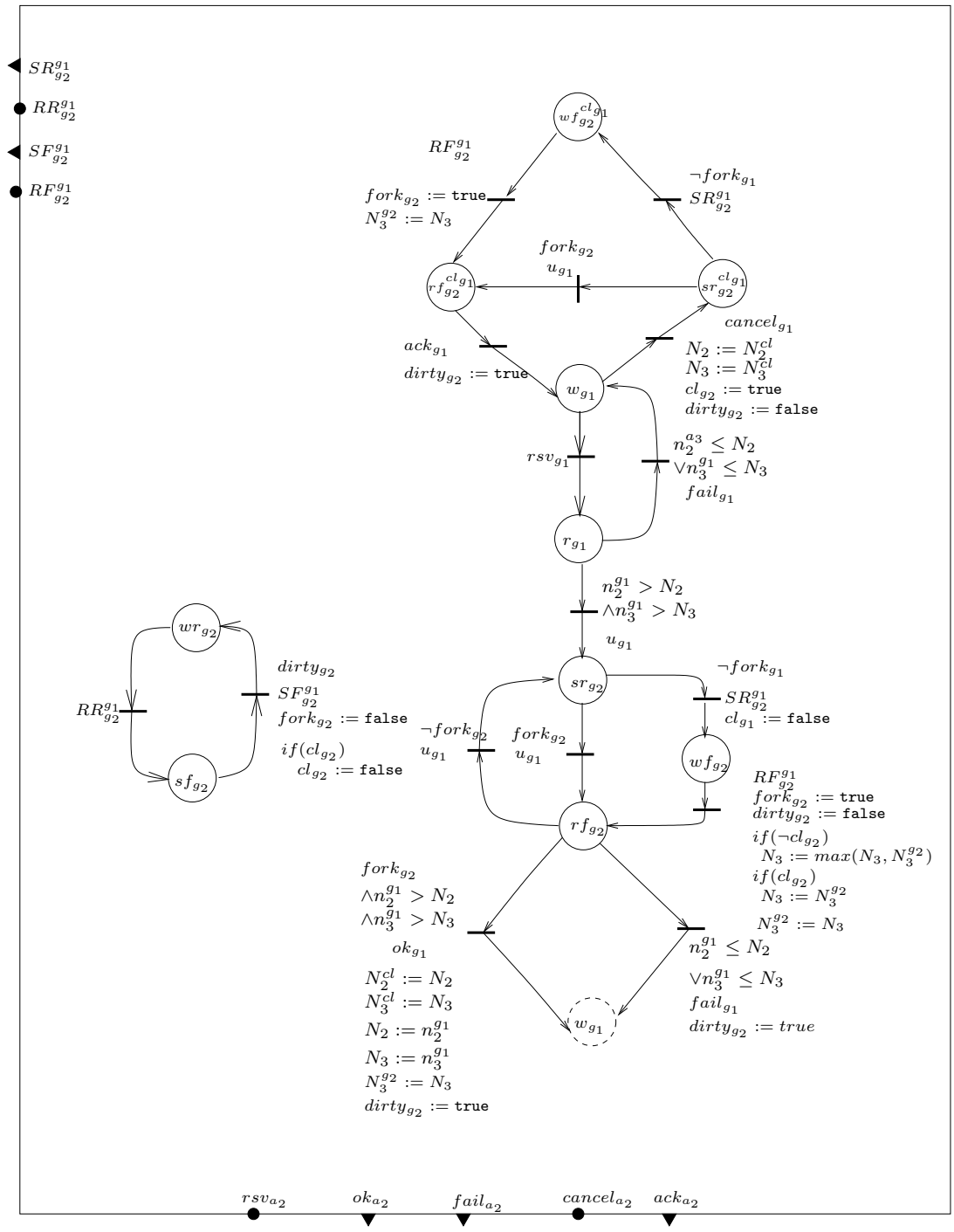


Figure 6.12: Component of the dining philosophers protocol handling interaction g_1 of Figure.

- $(ack_a.S_j, ack_a.C_{CRP})$. where C_{CRP} is either CP, or TR_a, or DP_a depending on the implemented conflict resolution protocol.

6.4 Correctness

In this section we prove the correctness of our Send/Receive model. First, we show that the obtained model is indeed a Send/receive model. Second, we show that the original model weakly simulates a new versions of our obtained Send/Receive models.

Compliance with Send/Receive Model

We need to show that receive ports of B_{CP}^{SR} will unconditionally become enabled whenever one of the corresponding send ports is enabled. Intuitively, this holds since communications between two successive layers follow a request/acknowledgment pattern. Whenever a layer sends a request, it enables the receive port to receive acknowledgment and no new request is sent until the first one is acknowledged.

Lemma 8. *Given a BIP model $B = \gamma(B_1, \ldots, B_n)$ and a partition of interactions $\{\gamma_j\}_{j=1}^{k}$, the model $B_{CP}^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR}, S_1, \ldots, S_k, CP)$ obtained by transformation of Section 6.3 meets the properties of Definition 22.*

Proof. The first two constraints of Definition 22 are trivially met by construction. We now prove that the third constraint also holds, i.e, whenever a send-port is enabled, all its associated receive-ports are enabled as well.

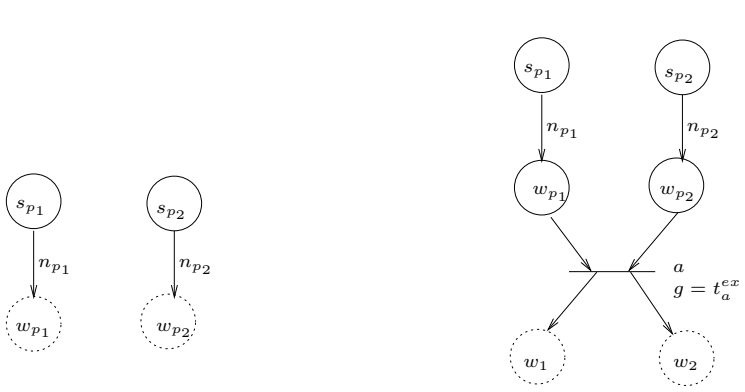
For offer/notifications interactions between Send/Receive components and the schedulers and for reservation/ok/fail interactions between schedulers and the conflict resolution protocol, the proof is similar to the proof of Lemma 4. We only need to prove that the property also holds for cancel/ack interactions between the schedulers and the conflict resolution protocol. By construction of the scheduler a cancel message $cancel_a$ is possible only after receiving an ok_a message from the conflict resolution protocol. The latter is being in w_a place where a cancel message could be received. Thus, whenever the send port $cancel_a$ in the scheduler is enabled, the corresponding receive port is also enabled. By construction of the conflict resolution protocol and the schedulers, the transition labeled by ack_a is a successive transition of the one labeled $cancel_a$. Thus, by construction, when the send port ack_a is enabled in the conflict resolution protocol, its corresponding receive port is also enabled in the scheduler. □

Weak simulation between Original and Transformed Models

Remark that the interactions execution are hidden in the obtained Send/Receive model B_{CP}^{SR}. In fact, in this model there are only transitions for planning interactions, bur no transitions for their executions. In order to make interactions execution visible in the Send/Receive model, we consider a new version of B_{CP}^{SR} denoted by B_{CP}^{SR*} where for each scheduler S_j in B_{CP}^{SR} we add the following (see Figure 6.13).

- we add for each port p, the place w_p with time progress condition $g \leq t_i^{ex}$ where i is the index of component B_i of port p.

- We add for each interaction a, the transition $\tau_a = (\{w_p\}_{p \in a}, a, \{w_i\}_{B_i \in part(a)})$.
- We modify for each port p, the output places of transition τ_{n_p} to w_p.
- We add the unary interaction $(S_j.a)$.



(a) Transitions in B_{CP}^{SR} (b) Transitions in B_{CP}^{SR*}

Figure 6.13: From B_{CP}^{SR} to B_{CP}^{SR*}

We show that the initial model B weakly simulates the model B_{CP}^{SR*}. The observational equivalence cannot be proven since we lose fairness with B_{CP}^{SR*}. In fact, the execution of conflicting interactions depend on the order of their scheduling. Thus an interaction executed in the original model, could be executed in the Send/Receive model with a particular order of scheduling. Nonetheless, we can prove that any interaction executed in the Send/Receive model, can be executed in the original model.

We consider the correspondence between actions of B and B_{CP}^{SR*} as follows. To each interaction $a \in \gamma$ of B, we associate the unary interaction a of B_{CP}^{SR*}. All other interactions of B_{CP}^{SR*} (offer, notification, reserve, ok, fail, cancel, ack) are unobservable and denoted by β.

We proceed as follows to complete the proof of weak simulation. Among unobservable actions β, we distinguish between β_1 actions, that are interactions between the atomic components and the schedulers (namely offer and notification) and the scheduling transition, and β_2 actions that are interactions between the schedulers and the conflict resolution protocol (namely reserve, ok, fail, cancel, ack). We denote by q^{SR} a state of B_{CP}^{SR} and q a state of B. A state of B_{CP}^{SR*} from where no β_1 action is possible is called a *stable state*, in the sense that any β action from this state does not change the state of the atomic components.

Lemma 9. *From any state q^{SR}, there exists a stable state $[q^{SR}]$ such that $q^{SR} \xrightarrow{\beta_1^*}_{\gamma^{SR}} [q^{SR}]$.*

Proof. The state $[q^{SR}]$ exists since each Send/Receive component B_i^{SR} can do at most two β_1 transitions: receive a notification and send an offer. □

Note that the state $[q^{SR}]$ is not unique, since it depends on the order of scheduling transitions. The state $[q^{SR}]$ verifies the property $q^{SR} \xrightarrow{\beta_1^*}_{\gamma^{SR}} [q^{SR}]$ and $[q^{SR}] \overset{\beta_1}{\not\to}_{\gamma^{SR}}$.

Lemma 10. *At a stable state $[q^{SR}]$, the Send/Receive model verifies the following properties:*

- *All atomic components are in ℓ or $\perp_\tau$ places.*
- *Tokens in the scheduler components are either in place w_p or r_i.*
- *The clock g and variables in the atomic components have the same value than their copies in the scheduler components.*

Proof. The two first points come from Lemma 8 that guarantees possible execution of a Send/Receive interaction if its send-port is enabled. Therefore no place s_p in the scheduler components (respectively $\perp_\ell$ in atomic components) can be active at $[q^{SR}]$, otherwise the answer n_p (respectively the offer from $\perp_\ell$) could occur. Furthermore, since all offers have been sent, no token can be in w_i place.

In each scheduler S_j, when a r_i is marked, the last transition executed is either an offer, $fail$ or *ack* messages reception, which does not modify variables in the scheduler S_j. When w_p is marked, the last transition executed is the notification which is resulted of scheduling an interaction that modifies the variables in the schedulers. By sending notifications, the scheduler sends updated variables to the components. Thus, each variable in the scheduler has the same value as the corresponding atomic component. The clock g has the same value in the atomic components and the schedulers as it is never reset. □

Lemma 11. *When B_{CP}^{SR*} is in a stable state, for each $i \in \{1, \ldots, n\}$, we have $n_i > N_i$.*

Proof. If the place r_i is marked, then the proof is similar to the proof of 7. If the place w_p is marked, the last transition executed is the notification which increments the variable n_i is the corresponding atomic component. □

We define a relation between the set of states Q^{SR} of B_{CP}^{SR*} and the set of states Q of B as follows. For each state $q^{SR} \in Q^{SR}$, we build a state $corr(q^{SR})$ by:

1. considering all possible stable states $[q^{SR}]$ reachable by doing β transitions,
2. considering the control location ℓ or $\perp_\tau$ where $\tau = (\ell, p, \ell')$ of B_i^{SR} as the control location ℓ for B_i in $equ(q^{SR})$,
3. restricting the valuation of variables of B_i^{SR} to a valuation of variables in B_i,
4. taking the valuation of original clock c in B_i as the valuation of $g - \rho_c$.

Theorem 4. $B_{CP}^{SR*} \subset B$.

Proof. We define the weak simulation relation R by taking:

$$R = \{(q^{SR}, q) \in Q^{SR} \times Q \mid q = corr(q^{SR})\}$$

The three next assertions prove that R is a weak simulation:

(i) If $(q^{SR}, q) \in R$ and $q^{SR} \xrightarrow{\beta}_{\gamma^{SR}} r^{SR}$ then $(r^{SR}, q) \in R$.

(ii) If $(q^{SR}, q) \in R$ and $q^{SR} \xrightarrow{\sigma}_{\gamma^{SR}} r^{SR}$ then $\exists r \in Q : q \xrightarrow{\sigma} r$ and $(r^{SR}, r) \in R$.

(i) If $q^{SR} \xrightarrow{\beta}_{\gamma^{SR}} r^{SR}$, then β is either β_2 action which does not change the state of the atomic components, thus all possible reachable state $[q^{SR}]$ are the same possible reachable state $[r^{SR}]$, or β is β_1 action, thus by definition all possible reachable state $[q^{SR}]$ are the same possible reachable state $[r^{SR}]$. Thus, we have $corr(q^{SR}) = corr(r^{SR})$.

(ii) The action σ in B^{SR*}_{CP} is either an interaction a or a delay step δ.

- If a is an interaction, it corresponds to executing a transition labeled by a unary port a in the scheduler handling interaction a. This transition is enabled only if scheduling interaction a is done through transition $sched_a$. If a is not externally conflicting, by construction of the scheduler, the transition labeled by $sched_a$ has the Boolean guard $G_a \bigwedge_{p \in a} g_p$, where g_p are Boolean guards sent by the atomic components for each port $p \in a$. By Lemma 10, the values involved in g_a are the same in atomic components, and by extension in $q = corr(q^{SR})$. Thus the guard of a evaluates to true at $q = corr(q^{SR})$. Also, by construction the transition labeled by a has the timing constraint $g = t^{ex}_a$ where t^{ex}_a satisfies $\mathsf{safe\text{-}tc}(t^{ex}_a)$. By Lemma 10, the values involved in $\mathsf{safe\text{-}tc}$ are the same in the atomic components and by extension in $q = corr(q^{SR})$. These timing constraints are expressed on clock g which could be equivalently expressed on original clocks c involved in the original timing constraints of B_i. The valuation of each clock c involved in the timing constraint of a is computed from the valuation of $g - \rho_c$ where ρ_c is the last clock reset date of c. Therefore, at state q the timing constraint of a expressed on its original clocks are also met at $t^{ex}_a - \rho_c$.

  If a is externally conflicting, the transition labeled by a in CP is possible only if the transitions rsv_a and ok_a are executed in the scheduler. The transition labeled by rsv_a has the guard $G_a \bigwedge_{p \in a} g_p$. Similarly to the previous case, if this transition is possible in the scheduler, the Boolean guard of a is met at q. Moreover, if transition labeled ok_a is enabled, this means that for each component B_i involved in a, $n_i > N_i$. In particular, for each involved component B_i, the offer corresponding to the number n_i has not been consumed yet. The transition labeled by ok chooses t^{ex}_a where t^{ex}_a satisfies $\mathsf{safe\text{-}tc}(t^{ex}_a)$. By Lemma 10, the values involved in $\mathsf{safe\text{-}tc}$ are the same in the atomic components and by extension in $q = corr(q^{SR})$. We conclude that in both cases, we have $q \xrightarrow{a} r$.

  Finally, if a is executed in B^{SR}, this is resulted from the execution of transition $sched_a$ which triggers the execution of the data transfer function F_a, followed by the computation in atomic component upon reception of the response. Thus at $[r^{SR}]$ the values in atomic components are the same as in r, which yields $(r^{SR}, r) \in R$.

- If a is a delay step δ, it is sufficient to consider only locations $\perp_{\ell_i}$ and ℓ_i and $\perp_{\tau_i}$ of B^{SR}_i. The time progress condition of these location are expressed on clock g.

These time progress conditions are expressed on clock g which could be equivalently expressed on original clocks c_i involved in the original time progress conditions B_i. The valuation of each clock c involved in the time progress condition of B_i is computed from the valuation of $g - \rho_c$ where ρ_c is the last clock reset date of c. The time progress conditions of $\perp_{\ell_i}$ and ℓ_i are defined by the time progress condition tpc_{ℓ_i} of the original model. Thus at this location, if δ is allowed then it is also allowed in the original model. The time progress conditions of $\perp_{\tau_i}$ is defined by $g \leq t_i^{ex}$. By construction, t_i^{ex} satisfies tpc_i as the latter is involved in the timing constraint of an interaction a involving B_i. As tpc_i corresponds to the time progress condition of location ℓ_i, thus t_i^{ex} satisfies also tpc_{ℓ_i} Thus, at location $\perp_{\tau_i}$, if δ is allowed then it is also allowed in the original model. Therefore $q \xrightarrow{\delta} r$. Executing δ has the same effect on clocks in both models, therefore $(r^{SR}, r) \in R$.

Similarly to B_{CP}^{SR*}, we consider new versions of B_{TR}^{SR} and B_{DP}^{SR} denoted by B_{TR}^{SR*} and B_{DP}^{SR*} where interactions appears in the scheduler components.

Proposition 3. *$B_{TR}^{SR*} \subset B$ and $B_{DP}^{SR*} \subset B$*

Proof. The proof is similar to the proof of Proposition 1. That is, we prove that $DP \subset CP$ and $TR \subset CP$. □

6.5 Optimizations

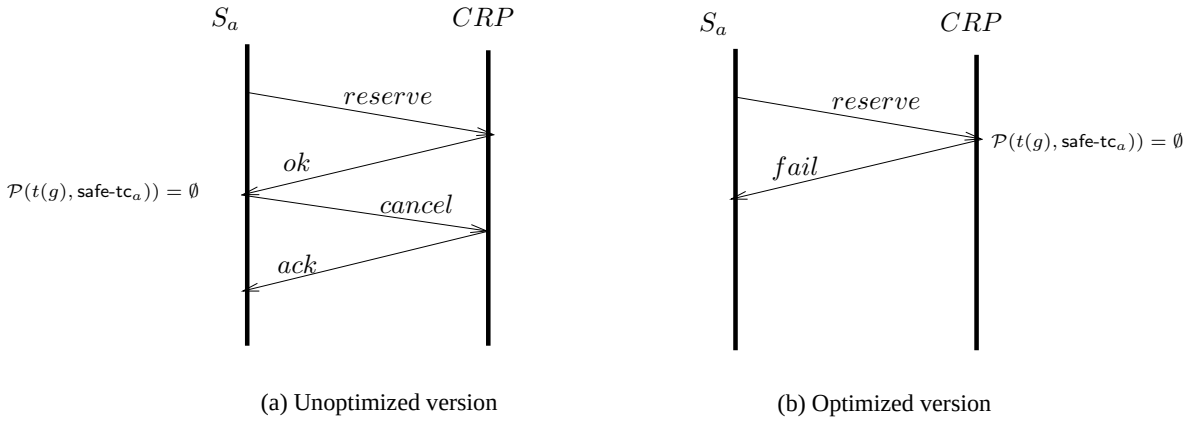
6.5.1 Minimizing Cancel Requests

Let a be an externally conflicting interaction handled by a scheduler S_a. In order to plan interaction a, S_a has to send a reservation request to the conflict resolution protocol. This request contains the participation numbers of components participating in a. In the case where the conflict resolution protocol succeeds to grant interaction a, it sends an *ok* message to S_a. In order to plan interaction a, the scheduler S_a has to choose a safe date from the set $\mathcal{P}(t, (g)\mathsf{safe\text{-}tc}_a)$ to be sent to participating components. In the case where the set $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a)$ is empty, S_a sends a cancel request to the conflict resolution protocol which is acknowledged by an *ack* message. Figure 6.14(a) shows such mechanism.

In order to minimize cancel requests, the conflict resolution protocol (CRP) may send a $fail$ message directly if it finds out that the set $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a)$ is already empty. Figure6.14(b) shows such optimization.

In order to implement this optimization, we need to:

- include the safe timing constraint $\mathsf{safe\text{-}tc}_a$ of a in the reservation request.
- add in the conflict resolution protocol transitions labeled by $fail$ guarded by $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a) = \emptyset$.
- add the guard $\mathcal{P}(t(g), \mathsf{safe\text{-}tc}_a) \neq \emptyset$ in each *ok* transitions of the conflict resolution protocol.



(a) Unoptimized version (b) Optimized version

Figure 6.14: Reducing Messages fro Cancel Mechanism

6.5.2 Refining Send/Receive Models

In the proposed Send/Receive model presented in this chapter, we suppose that communication delays between components are unknown which makes our proposed model general. The implementation of such models may lead to failure if the communication delays are too large. For instance, when a scheduler sends notifications to components participating in an interaction, the communication delays between the scheduler and those components should be short enough to meet the execution date and ensure the perfect synchronization of the components. If one of these components receives the notification message too late, it may miss the date for execution which lead to an error (some components executes at the execution date, and some others misses the date). In the implementation, we propose to detect such errors and stop the system execution.

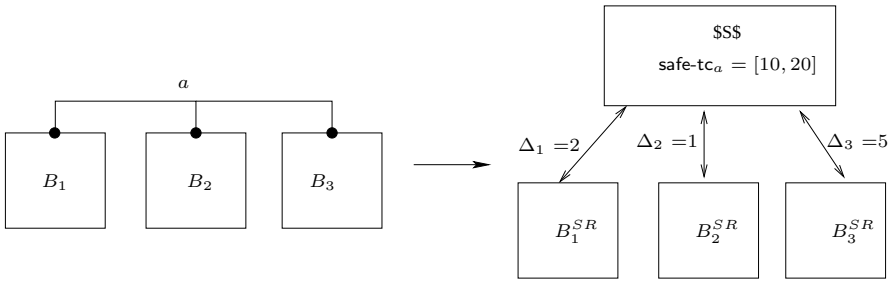
A more specific Send/Receive model could be proposed if the communication delays worst-case estimation of the target platform are known. The adopted scheduling policy takes into account communication delays worst-case estimation between the scheduler and the Send/Receive atomic components.

Let Δ_i^j be the communication delay worst-case estimation of the Send/Receive atomic component B_i^{SR} and scheduler S_j. Let a be an interaction handled in the scheduler S_j, and let safe-tc_a its safe timing constraint. The set of safe dates for scheduling a could be the following:

$$\mathcal{P}(t(g), \text{safe-tc}_a, \{\Delta_i^j\}_{B_i \in part(a)}) = \{t \mid \text{safe-tc}_a(t) \wedge t - t(g) \geq \mathbf{max}\ (\{\Delta_i^j\}_{B_i \in part(a)})\} \quad (6.5)$$

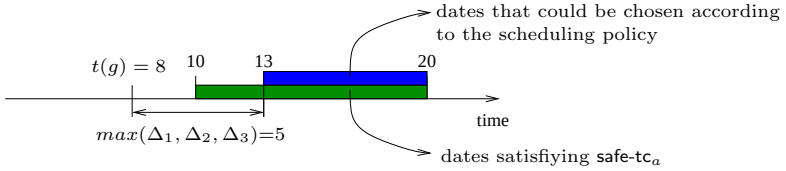
Choosing a date t that satisfy $t - t(g) \geq \mathbf{max}\ (\{\Delta_i^j\}_{B_i \in part(a)})\}$ ensures that each component $B_i \in part(a)$ receives the notification from the scheduler before the date expiration.

Example 28. *Figure 6.15(a) depicts an interaction a involving three components, B_1, B_2 and B_3. The communication delays worst-case estimation between the scheduler handling interaction a and the Send/Receive atomic components are shown in Figure 6.15(b). Suppose that the safe timing constraint of a is $\text{safe-tc}_a = 10 \leq g \leq 20$ and the valuation of clock g is $t(g) = 8$. According to the scheduling policy given in (6.5), the dates that could be chosen*



(a) interaction a involving components B_1, B_2 and B_3

(b) Communication delays estimation between send/receive components and the scheduler handling a.



(c) Scheduling policy taking into account communication delays

Figure 6.15: Improving Scheduling Policy.

to execute a should satisfy $13 \leq g \leq 20$. *The dates satisfying* $10 \leq g < 13$ *could lead to a system error because component B_3 could miss its execution date due to the communication delay between the scheduler and component B_3.*

6.5.3 Reducing the Number of Observed Components

In order to plan an interaction a, the scheduler handling a has to wait to receive offers from components participating in a as well as offers from components observed by a. Since the execution of interaction a changes the state of the participant components, receiving their offers is mandatory for scheduling interaction a. However, receiving offers from observed components could be avoided. For instance, consider the example of Figure 6.4. Interactions a and b are conflicting, then $obs(a) = B_3$ and $obs(a) = B_1$. Consider interaction b. The scheduling timing constraint of b is sched-tc$_b = c_3 \leq 2$ (or equivalently $g \leq 2 - \rho_{c_2}$). In the observed component B_1, location ℓ_1 with time progress condition tpc$_{\ell_1} = c_1 \leq 10$, is the only location where the port p_1 involved in interaction a is enabled. Remark that for each valuation of clock c_2 that satisfies sched-tc$_b$ we have $c_1 \leq 10$ as c_1 and c_2 have the same valuations in components B_1 and B_3 at locations ℓ_1 and ℓ_3 respectively. That is, each date for scheduling b satisfies the time progress condition of location ℓ_1 of component B_1. In this case, it is unnecessary for interaction b to observe component B_1. For interaction a, the scheduling timing constraint of a is sched-tc$_a = c_1 \leq 10$. Remark that there exists valuations of clock c_1 that satisfies sched-tc$_a = c_1 \leq 10$ but not the time progress condition $c_3 \leq 2$ of ℓ_3 of component B_3 where ℓ_3 is the location in the observed component B_3 from which port p'_3 is enabled. Thus, it is mandatory for interaction a to observe component B_3.

In this section, we propose to use static analysis techniques in order to detect such situations and then, to reduce the number of observed components. For an interaction a, we define the predicate reduce_a characterizing the components that can be removed from the set of observed components $obs(a)$.It is fully defined in the following.

Observed Components Reduction Predicate

Let $\gamma(B_1, \ldots, B_n)$ be a BIP model, and let $a = (\{p_i\}_{i \in I}, G_a, F_a)$ be an interaction in γ. Recall that the predicate $\mathsf{sched\text{-}tc}_a[\ell_a]$ characterizing clocks valuations for which a could be scheduled from the configuration $\ell_a \in \mathcal{L}_a$ is defined as follows:

$$\mathsf{sched\text{-}tc}_a[\ell_a] = \bigwedge_{\ell_i \in \ell_a | \tau_i = (\{\ell_i, p_i, \ell_i') \in T_i\}} \mathsf{tc}_{\tau_i} \wedge \mathsf{tpc}_{\ell_i}$$

We denote by $\mathsf{confinter}_a(B_j)$ the set of interactions that are conflicting with a and involving component B_j. Formally, it is defined as follows:

$$\mathsf{confinter}_a(B_j) = \{b \in \gamma \mid b \# a \wedge B_j \in part(b)\}$$

Intuitively, the component B_j could be reduced from the set $obs(a)$ if for each configuration from which a is enabled, each valuation of clocks that satisfies $\mathsf{sched\text{-}tc}_a[\ell_a]$, satisfies also each time progress condition of locations enabling a port participating in each interaction b that is conflicting with a (see Figure 6.16).

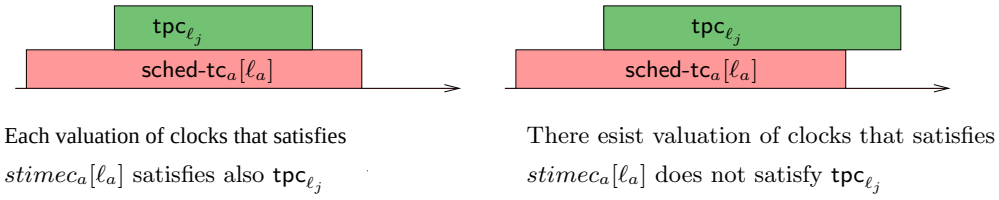


Figure 6.16: Illustrative example.

Definition 40. *Let $\gamma(B_1, \ldots, B_n)$ be a BIP model. We denote by C be the set of clocks in the BIP model. Consider an interaction a in γ and an observed component B_j in $obs(a)$. We define $\mathsf{reduce}_a(B_j)$ the predicate indicating whether B_j could be reduced from $obs(a)$. It is defined as follows:*

$$\mathsf{reduce}_a(B_j)$$

$$\equiv$$

$$\forall\ \ell_a \in \mathcal{L}_a,\ \forall p_j \in b \mid b \in \mathsf{confinter}_a(B_j), \forall\ \ell_j \in \mathcal{L}_{p_j}, \forall t \in \mathcal{T}(\mathsf{C}),\ \forall \delta > 0$$

$$\mathsf{sched\text{-}tc}_a[\ell_a](t + \delta) \Rightarrow \mathsf{tpc}_{\ell_j}(t + \delta)$$

According to Definition 40, interaction a reduces the observation of component B_j, if for each configuration of locations $\ell_a \in \mathcal{L}_a$ enabling interaction a, for each location ℓ_j of component B_j enabling port p_j participating in interaction b such that $b\#a$, for each clocks valuation $t \in \mathcal{T}(\mathsf{C})$ and for each time step $\delta > 0$, we have $\mathsf{sched\text{-}tc}_a(\ell_a)(t+\delta) \Rightarrow \mathsf{tpc}_{\ell_j}(t+\delta)$. The latter implication specifies that there is no clocks valuation $t \in \mathcal{T}(\mathsf{C})$ that satisfies $\mathsf{sched\text{-}tc}_a[\ell_a]$ and not tpc_{ℓ_j}.

Rewriting $\mathsf{reduce}_a(B_j)$

The predicate $\mathsf{reduce}_a(B_j)$ involves a non-constant variable δ and the clocks valuation t. Thus, static analysis techniques cannot be used at this step.

In order to obtain a static expression of $\mathsf{reduce}_a(B_j)$, we use explicit expressions of $\mathsf{sched\text{-}tc}_a[\ell_a]$ and tpc_{ℓ_j} to rewrite $\mathsf{reduce}_a(B_j)$.

Using (6.1), $\mathsf{sched\text{-}tc}_a[\ell_a]$ can be written into the following form: $\mathsf{sched\text{-}tc}_a[\ell_a] = \bigwedge_{c_a \in \mathsf{C}_a} l_{c_a}^{\ell_a} \leq c_a \leq u_{c_a}^{\ell_a}$, where C_a is the set of all clocks involved in $\mathsf{sched\text{-}tc}_a[\ell_a]$, and $l_{c_a}^{\ell_a}$ (resp. $u_{c_a}^{\ell_a}$) is the lower (resp. upper) bound value involving clock c_a in $\mathsf{sched\text{-}tc}_a[\ell_a]$. Similarly, tpc_{ℓ_j} can be written in the following form: $\mathsf{tpc}_{\ell_j} = \bigwedge_{c_j \in \mathsf{C}_j} c_j \leq d_{c_j}^{\ell_j}$, where C_j is the set of clocks of component B_j.

Proposition 4. *Given an interaction a and a component $B_j \in obs(a)$, the predicate $\mathsf{reduce}_a(B_j)$ can be rewritten in the following form:*

$$\mathsf{reduce}_a(B_j)$$

$$\equiv$$

$$\forall\ \ell_a \in \mathcal{L}_a, \forall \ell_j \in \mathcal{L}_{p_j} \mid p_j \in b \wedge b \in \mathsf{confinter}_a(B_j)$$

$$\bigwedge_{c_j \in \mathsf{C}_j} \bigvee_{c_a \in \mathsf{C}_a} c_j - c_a \leq d_{c_j}^{\ell_j} - u_{c_a}^{\ell_a} \bigvee_{c_a \in \mathsf{C}_a} \bigvee_{c'_a \in \mathsf{C}_a} c_a - c'_a < l_{c_a}^{\ell_a} - u_{c'_a}^{\ell_a}$$

Proof. We have to prove that:

$$\forall t \in \mathcal{T}(\mathsf{C}), \forall \delta > 0$$

$$\mathsf{sched\text{-}tc}_a(\ell_a)(t+\delta) \Rightarrow \mathsf{tpc}_{\ell_j}(t+\delta)$$

$$\Leftrightarrow$$

$$\bigwedge_{c_j \in \mathsf{C}_j} \bigvee_{c_a \in C_a} c_j - c_a \leq d_{c_j}^{\ell_j} - u_{c_a}^{\ell_a} \bigvee_{c_a \in \mathsf{C}_a} \bigvee_{c'_a \in \mathsf{C}_a} c_a - c'_a < l_{c_a}^{\ell_a} - U_{c'_a}$$

$\mathsf{sched\text{-}tc}_a[\ell_a](t+\delta) \Rightarrow \mathsf{tpc}_{\ell_j}(t+\delta)$

$\Leftrightarrow \quad \mathsf{tpc}_{\ell_j}(t+\delta) \vee \neg \mathsf{sched\text{-}tc}_a(\ell_a)(t+\delta)$

$\Leftrightarrow \quad \Big[\ \bigwedge_{c_j \in \mathsf{C}_j} t(c_j) \leq d_{c_j}^{\ell_j} - \delta\Big] \vee \Big[\bigvee_{c_a \in \mathsf{C}_a} t(c_a) > l_{c_a}^{\ell_a} - \delta\Big] \vee \Big[\bigvee_{c_a \in \mathsf{C}_a} t(c_a) > u_{c_a}^{\ell_a} - \delta\Big]$

$\Leftrightarrow \quad \Big[\bigwedge_{c_j \in \mathsf{C}_j} t(c_j) \leq d_{c_j}^{\ell_j} - \delta \bigvee_{c_a \in \mathsf{C}_a} t(c_a) > u_{c_a}^{\ell_a} - \delta\Big] \vee \Big[\bigvee_{c_a \in \mathsf{C}_a} t(c_a) < l_{c_a}^{\ell_a} - \delta \bigvee_{c_a \in \mathsf{C}_a} t(c_a) > u_{c_a}^{\ell_a} - \delta\Big]$

$$\Leftrightarrow \quad \Big[\bigwedge_{c_j\in\mathsf{C}_j}\bigvee_{c_a\in\mathsf{C}_a} t(c_j)-t(c_a)\le d_{c_j}^{\ell_j}-u_{c_a}^{\ell_a}-\delta\Big]\vee\Big[\bigvee_{c_a\in\mathsf{C}_a}\bigvee_{c'_a\in\mathsf{C}_a} t(c_a)<l_{c_a}^{\ell_a}-\delta\vee t(c'_a)>u_{c'_a}-\delta\Big]$$

$$\Leftrightarrow \quad \Big[\bigwedge_{c_j\in\mathsf{C}_j}\bigvee_{c_a\in\mathsf{C}_a} t(c_j)-t(c_a)\le d_{c_j}^{\ell_j}-u_{c_a}^{\ell_a}-\delta\Big]\vee\Big[\bigvee_{c_a\in\mathsf{C}_a}\bigvee_{c'_a\in\mathsf{C}_a}\neg[t(c_a)\ge l_{c_a}^{\ell_a}-\delta\wedge t(c'_a)\le U_{c'_a}-\delta]\Big]$$

$$\Leftrightarrow \quad \Big[\bigwedge_{c_j\in\mathsf{C}_j}\bigvee_{c_a\in\mathsf{C}_a} t(c_j)-t(c_a)\le d_{c_j}^{\ell_j}-u_{c_a}^{\ell_a}-\delta\Big]\vee\Big[\bigvee_{c_a\in\mathsf{C}_a}\bigvee_{c'_a\in\mathsf{C}_a}\neg[t(c_a)-t(c'_a)\ge l_{c_a}^{\ell_a}-u_{c'_a}^{ell_a}]\Big]$$

$$\Leftrightarrow \quad \Big[\bigwedge_{c_j\in\mathsf{C}_j}\bigvee_{c_a\in\mathsf{C}_a} c_j-tc_a\le d_{c_j}^{\ell_j}-u_{c_a}^{\ell_a}-\delta\Big]\vee\Big[\bigvee_{c_a\in\mathsf{C}_a}\bigvee_{c'_a\in\mathsf{C}_a}\neg[c_a-c'_a\ge l_{c_a}^{\ell_a}-u_{c'_a}]\Big]. \qquad \square$$

We use static analysis techniques to check the satisfiability of $\mathsf{reduce}_a B_i$ as defined in Proposition 4, for each interaction a and for each for each component $B_j \in obs(a)$. In the following, we present one of these techniques.

Use of Static Analysis Techniques

There exist several static analysis techniques for the verification of timed systems. The basic idea is to characterize the set reachable states of the system. This set is exactly computed or approximated. Computing the exact set of reachable state is done using state space exploration techniques, like in [79, 80] Approximating the set of reachable states is provided using invariants computation. Invariants are obtained with much lower complexity than computing the exact set of reachable states. We are interested in the use of invariants to approximate the set of reachable states. In particular, we focus on the method presented in [61], for compositional verification of BIP models.

Consider a BIP model $\gamma(B_1,\ldots,B_n)$. Let Ψ be a property of interest. Assume that the system could be characterized by the global invariant GI. That is, GI characterizing an over-approximation of reachable states of the system. The following inference rule (VR) summarizes the verification rule of the method.

$$\frac{\vdash GI \Rightarrow \Psi}{\gamma(B_1,\ldots,B_n) \models \Box\,\Psi}$$

Intuitively, VR can be understood as follows: if Ψ can be proved to be a logical consequence of the global invariant GI, then Ψ holds for the system.

GI is computed from invariants characterizing the components behaviors called component invariants, and also from the invariants characterizing the interactions between components called interaction invariant. We do not detail here how to compute exactly the global invariant, but we provide here a brief description. Interested readers may refer to [61], for more details.

A component invariant is characterized by its reachable symbolic set which consists of the finite symbolic reachable states of the component computed from its zone graph [81]. A symbolic state of a component B_i in a zone graph is the pair (ℓ_i, ζ_i) where ℓ_i is a location of

B_i and ζ_i is a *zone* which is a set of clocks valuations computed from the timing constraints and time progress conditions of B_i. A component invariant of B_i is the disjunction of $(\ell_i \wedge \zeta_i)$ for all symbolic states (ℓ_i, ζ_i).

The interaction invariant are over-approximation of the global state space. It involves only locations of components and does not capture any global timing of interactions. They are computed by the method explained in [82].

To better track global timing of interactions between components, the considered method makes use of auxiliary clocks called *history clocks*. Each component B_i is equipped with history clocks, a clock per port. When an interaction takes place, the clocks corresponding to the ports participating in that interaction are reset, and thus are equal. All remaining clocks values are then greater than those being reset. The relations between history clocks are defined allow to relate local clocks of components.

Recently, a tool called RTD-Finder [83] implementing this method has been developed. We are using this tool to check the satisfiability of $\mathsf{reduce}_a(B_j)$ predicate.

6.6 Conclusion

In this chapter, we proposed method for obtaining Send/Receive models dedicated to be implemented on platforms that do not provide fast communication delays. Optimization for reducing the number of messages could are proposed. In particular, we proposed to minimize observed components number using static analysis techniques.

In the next chapter we give an overview on the tools that are implemented in order to generate distributed real-time implementations

7 Implementation

This chapter discusses the implementation methods for generating distributed real-time implementations. First, we present in Section 7.1 the BIP language that provides a textual representation of BIP models. Then, we present in Section 7.2 the existing BIP tools. Finally, we present in Section 7.3 the different tools developed within this thesis.

7.1 The Real-Time BIP Language

The Real-Time BIP language is used to represent concrete models presented in Section 2.2. It provides syntactic constructs for describing composition of components with respect to interactions and priorities. In practice, variables, data type declarations, expressions, update functions and data transfer functions are written in C. The Real-Time BIP language incorporate a set of structural syntactic constructs for defining component behavior, specifying the coordination through connectors and describing the priorities. The basic constructs of the BIP language are the following:

- atomic component: to specify behavior, with an interface consisting of ports. Behavior is described as a set of transitions labeled by ports names.
- connector: to specify the synchronization between the ports of components.
- priority: to restrict the possible interactions, based on conditions depending on the state of the integrated components.
- composite component: to specify systems hierarchically, from other atoms or compounds, with connectors and priorities.
- model: to specify the entire system, encapsulating the definition of components and specify the top level instance of the system.

Existing Real-Time BIP Language. Previous work on Real-Time BIP [84] proposed a BIP model that describes timing constraints on transitions as time intervals with urgencies. In such models, time progress depends on the presence of transitions urgent at the current state. That is, time is not allowed to progress from a given state if there exists an urgent transition enabled from that state. In Appendix A, we present a high level description of such models as well as its semantics. In addition, we present the transformation of models with urgencies into models with time progress conditions. The actual version of Real-Time BIP language describes BIP models with urgencies. In this thesis, we developed a new version of Real-Time BIP language that describes BIP models with time progress conditions.

In the following, we present different elements of BIP models described using the Real-Time BIP language. We rely on the example from Figure 2.10 to detail some elements of the Real-Time BIP language. In the Real-Time BIP language, we start by defining types (components, ports, connectors) which are instantiated later. For instance, in Figure 2.10, $\texttt{setter}_1$ and $\texttt{setter}_2$ components are instances of the same atomic component type *setter* shown in Figure 2.5. The latter is parametrized by the bound l of timing constraints of transition set and the bound u of time progress condition of location ℓ^2. The description of the `setter` atomic component type is illustrated as follows:

```
port type IntPort (int i )
port type EventPort

atomic type setter (int u, int l )
   data int x
   export port EventPort sync
   export port IntPort set(x)

   clock c unit 1 second

   place ℓ1
   place ℓ2 while c in [0 , u]

   initial to ℓ1 do { x = 0 ; }

   on sync from ℓ1 to ℓ2
     provided true
     reset {c}
     do {x = x+1 ;}
   on set from ℓ2 to ℓ1
     provided true
     when c in [l , -]
end
```

In the above example, two types of ports are defined, namely `EventPort` and `IntPort`. The port type `EventPort` is an event port and it is not associated with any variable. The port type IntPort associates to a port an integer variable `i`. The description of the atomic type `setter` starts with describing variables, ports and locations. A location may define

a time progress condition (after **while**). The variables that are associated with the port are explicitly given, and their types need to match those in the port type definition. The construct **initial to** specifies the initial location, and possibly some initial update function to execute. Each transition of the behavior is declared with a port (after **on**), a (set of) input and output place(s) (after **from** and **to** respectively), a Boolean guard (after **provided**), a timing constraint (after **when**) and an update function (after **do**). The update functions and the Boolean guards are written in C-like syntax.

As said before, components are composed using connectors. As an example, we present the connector type used to describe a_2 and a_3 interactions from Figure 2.10.

```
connector type SetGet(IntPort set, IntPort get)
  define set get
  on set get
    provided true
    down { get.i = set.i }
end
```

A connector type is parameterized by a list of port that describes its support. The construct **define** defines the type of ports, trigger of synchron, as explained in Subsection 2.2.3. For each interaction, a guard and a data transfer function can be defined. In the above example, `SetGet` connector defines a strong synchronization between two ports of type `IntPort`. The guard associated to the interaction is `true` and the data trasfer function is provided with the **down** construct. The notation `get.i` (resp. `set.i`) is used to access the variable `i` associated to the port `get` (resp. `set`), as defined in the port type declaration `IntPort`.

A compound component is a new component type defined by creating instances of existing components types which are glued by instantiating connectors and by specifying the priorities. We define below the compound type that corresponds to Figure 2.10, assuming that the atomic type `Getter` has been aleady defined.

```
compound type Compound
  component setter setter1
  component setter setter2
  component getter getter1

  connector SyncEvents a1 (setter1.sync1,getter.sync,setter2.sync2)
  connector SetGet a2 (setter1.set1, setter.get1)
  connector SetGet a2 (setter2.set2, setter.get2)

  priority prio a2 < a3
end
```

When `Compound` type is instantiated, the compound instance creates first the three components that constitute the system. For instance, "**component** `Getter getter`" creates an instance of `Getter` atomic component type, named `getter`. Second, the compound instance

creates connectors, associating the ports of instantiated components through the interactions defined by the connector type. For instance, "**connector** `SetGet` a_2 (`setter`$_1$`.set`$_1$, `getter.get`$_1$)" creates an instance of `SetGet` connector type, named a_2. The dotted notation `setter`$_1$`.set`$_1$ is used to denote the port `set` of the component instance `setter`$_1$. Finally, the priority rule " **priority** `prio` $a_2 < a_3$" specifies that interaction a_2 has lower priority than interaction a_3.

7.2 The BIP Toolbox

This section presents the toolbox available with the BIP framework. It consists of a rich set of tools for modeling, executing and verifying BIP models.

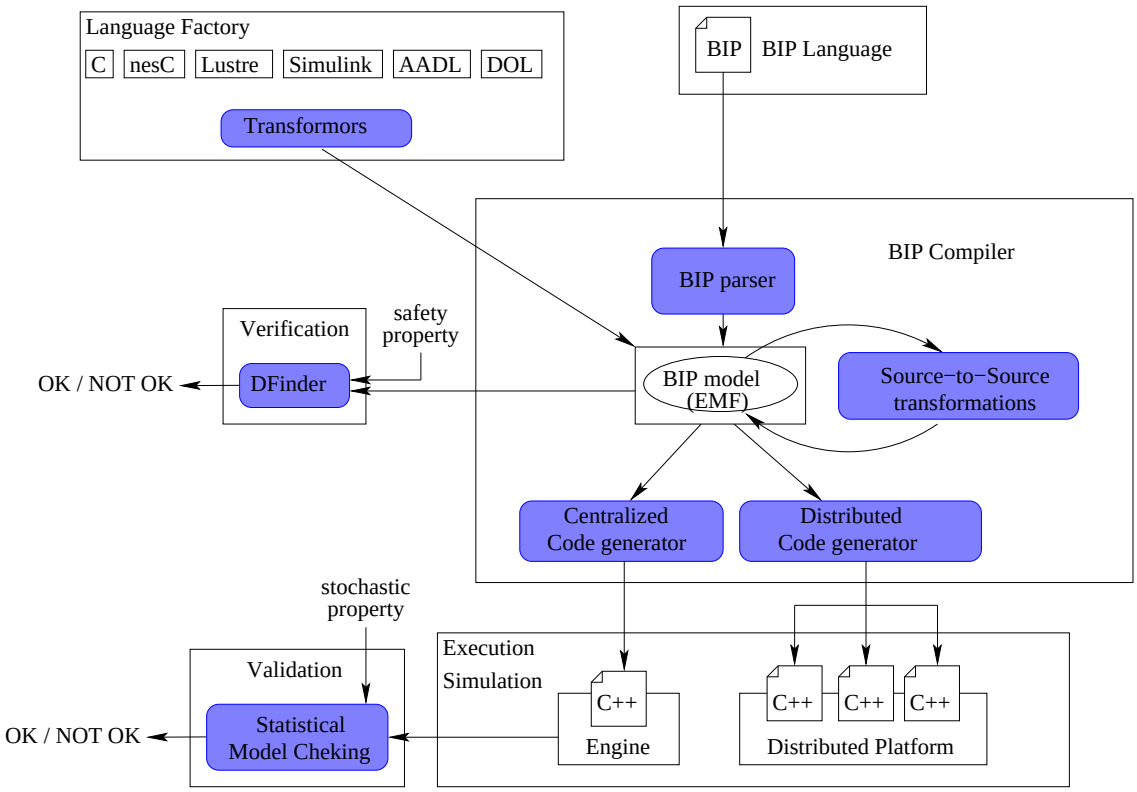


Figure 7.1: Overview of the BIP toolbox

Figure 7.1 shows an overview of the BIP toolbox. Mainly, there are four types of tools, namely Language Factory, Source-To-Source Transformations, Verification and Execution/Simulation. Below, we detail each of these types.

7.2.1 Language Factory

These tools are used to transform various programming models, using different languages, into BIP models. The translation allows representing different programming models in a BIP

model with a rigorous semantics. There exist different transformations including transformations from synchronous languages, i.e. transformations from Lustre [85] and Simulink [86]. These transformations target synchronous BIP [87], that is an extension of BIP dealing efficiently with synchronous models.

Other transformations target languages model that mix both the application software and the hardware architecture. These models can be transformed either into two separate models: one for the software and the other for the architecture or into a single model including both of them, called system model. Transformations to hardware model often rely on a library of hardware components such as memories, buses, processors, that are modeled in BIP. For instance, these transformations target the Architecture Analysis and Design Language (AADL) [88], nesC/TinyOS [89] and the Distributed Operation Layer (DOL) [90].

7.2.2 Verification

The BIP toolbox is completed by verification and validation tools for both checking system correctness and performance evaluation.

D-Finder [82, 91] is a verification tool targeting safety properties, e.g. deadlock freedom or mutual exclusion of untimed BIP models. Note that untimed BP models are models that have not timing features (clocks, timing constraints, time progress conditions). The verification method implemented by D-Finder is based on the computation of 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 for interactions. The approach is compositional and can be applied incrementally, allowing to better scale to large systems than traditional verification techniques.

RT-DFinder [61] is an extension of D-finder tool for the verification of BIP models. RT-DFinder 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. The generated invariant for timed models using RT-DFinder proved to be accurate enough to capture non trivial properties of several case studies, as shown in [92].

In addition to the verification tools, the BIP toolbox includes the statistical model-checker SMC-BIP [93] for checking stochastic properties expressed as probabilistic bounded linear temporal logic (PBLTL) formulas. Given a stochastic BIP model, a PBLTL formula and confidence parameters, SMC-BIP computes execution sequences until the formula can be proven with the target degree of confidence. Such a tool is particularly suited for evaluating quantitative properties including system performance related metrics.

7.2.3 Source-to-Source Decentralization

In this thesis, we presented method for decentralizing BIP models into 3-layer Send/Receive BIP models. Actually, the 3-layer design was considered also for decentralizing untimed BIP models. Figure 7.2 shows the tools for obtaining Send/Receive untimed BIP models from untimed BIP models with and without priorities. They consist of the following.

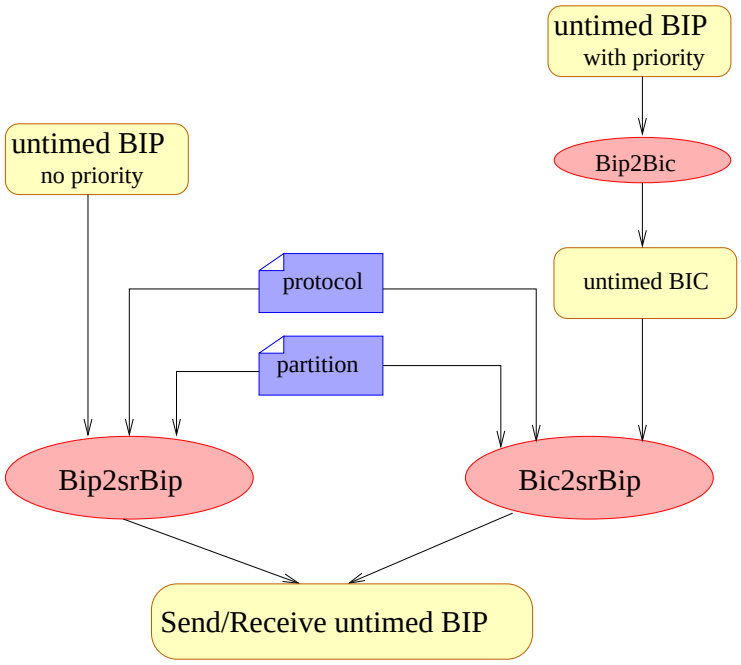


Figure 7.2: Source-to-Source Decentralization design flow for untimed BIP models

- **Bip2SrBip**: This tool generates an untimed BIP model into a 3-layer Send/Receive untimed BIP model. Priority is not supported. The 3-layer Send/Receive untimed BIP model architecture is the same as described in this thesis. That is, it consists of transformed atomic components, a set of schedulers and a conflict resolution protocol. However, timing features are not supported. This tool is parametrized by an interactions partition.

- **Bip2Bic**: This tool transforms an untimed BIP model into a untimed BIC model. Untimed BIC is an untimed BIP model where priorities are rewritten as Condition predicates [94]. Condition associates a predicate to each interaction. This predicate characterizes the system state where an interaction could execute. That is, states where there is no higher priority interaction is enabled. This predicate depends on the global state of the system and not only on the states of the participants in the interaction. Thus, instead of writing explicit priority, BIC transforms priorities into "guards" on interactions.

- **Bic2SrBip** This tool is obtained by extending Bip2srBip tool to support Condition. Given a BIC model and an interactions partition, this tool generates a Send/Receive untimed BIP model.

7.2.4 Execution/Simulation

Execution/Simulation of Untimed BIP models

BIP toolbox provides code generators for execution/simulation of untimed BIP models on target platforms. One option is to execute the generated C++ code using a centralized scheduler, implemented in C++ as well, that enforces the BIP operational semantics. The scheduler plays the role of the coordinator between components. Executing an untimed BIP model using a centralized scheduler can be done in a single thread or multi-thread mode.

For single-thread mode, the scheduler as well as the atomic components executes in a single thread. The execution cycle of single thread scheduler executes an interaction, moving the system from a global state to another global state. This ensures sequential execution of BIP models.

For multi-thread mode, each atomic component is assigned to a different thread, the scheduler being assigned a thread as well. Contrarily to the single-thread version, the global state is not known to the scheduler as an atomic component performing an internal computation has an undefined state. Therefore the scheduler executes according to the partial state semantics for untimed BIP modeols [74]. When an atomic component completes and reaches a stable state, it notifies the scheduler about the ports on which it is willing to interact. The scheduler is parametrized by an oracle. The oracle allows the execution of an interaction from a given partial state, only if there is no interaction with higher priority that could be enabled from the corresponding global state.

For distributed implementations of untimed BIP models, the BIP tool-box provides a code generator for generating C++ for each Send/Receive component of the Send/Receive untimed BIP models.

Execution/Simulation of BIP models

Regarding BIP models with timing features, the BIP tool-box provides a single-thread real-time scheduler allowing sequential execution [84].

7.3 Tools Developed in this Thesis

We developed tools that implements method described in this thesis. In Chapter 4, we presented Send/Receive BIP models with a centralized scheduler. It corresponds to an implementation of partial state models presented in Chapter 3. In practice, this model is never built, but has been used for sake of clarity. Instead, we have developed a multi-thread real-time scheduler that implements interactions and priorities of partial state models. A code generator has been also developed to generate C++ code for atomic components. Each atomic component as well as the scheduler execute each one in a separate thread.

Regarding 3-layer BIP models presented in Chapter 5 and Chapter 6, we developed tools for generating such models from given BIP models. These tools are parametrized by the interactions partition and the protocol for conflict resolution. For distributed implementations, we have developed a common code generator that generates C++ code for each component of the 3-layer BIP models.

In the following, we give details of the implementation of these tools.

7.3.1 Multi-threaded Real-Time Scheduler

We implemented a multi-thread real-time scheduler that implements interactions and priorities of the partial state models presented in Chapter 3 (this work appears in [95]). Unlike the centralized scheduler component presented in Chapter 4, the multi-thread real-time scheduler is not specific to a given BIP model. It corresponds to a general implementation of a scheduler for any BIP model for parallel execution.

We developed a compiler to generate C++ code from a given BIP model. This code is compiled to run with the multi-thread real-time scheduler Each atomic component is assigned to a thread, the multi-thread real-time scheduler being a thread itself. Communication between the scheduler and the atomic components is ensured using message passing through FIFO channel.s

The multi-thread real-time scheduler coordinates between atomic components and computes enabled interactions on-line. The protocol between the atomic components and the multi-thread real-time scheduler is similar to the one presented in Chapter 4. That is, atomic components sends offers to the scheduler which are acknowledged by notifications.

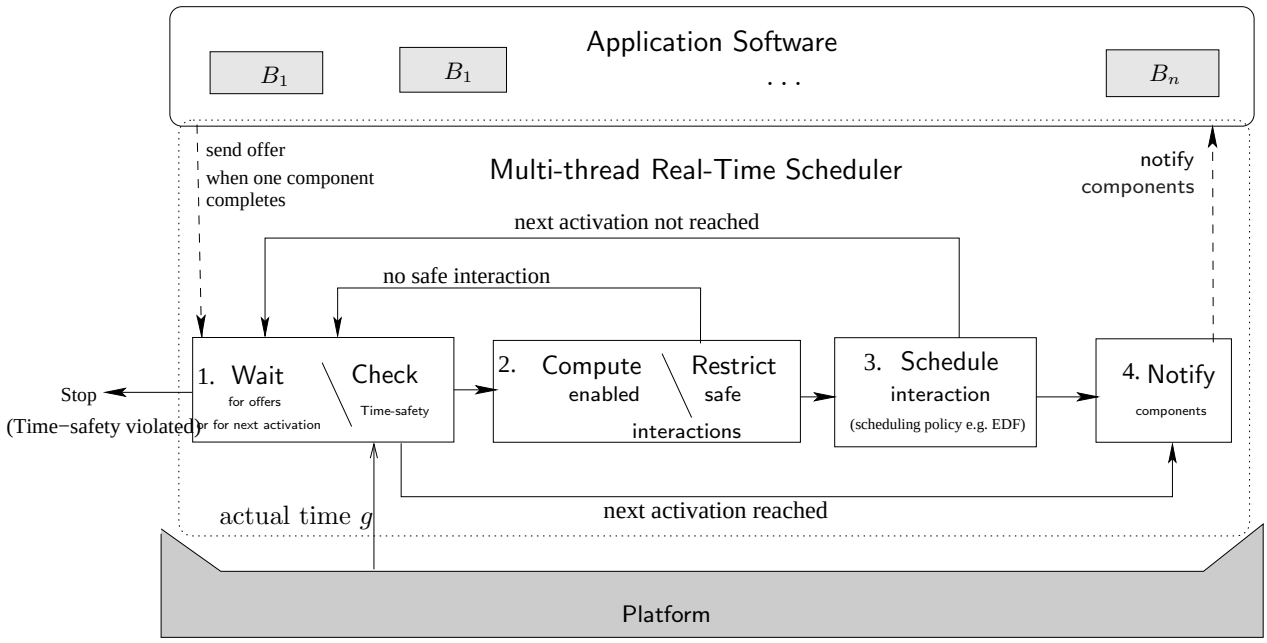


Figure 7.3: Multi-Threaded Real-time Scheduler

Figure 7.3 shows the execution cycle of the multi-thread real-time scheduler. Given a state q, the scheduler computes the next enabled interaction as follows:

1. It waits for offers from components finishing their execution or also for next activation date of a scheduled interaction. When the scheduler receives an offer from a component

B_i, it checks its next deadline $D_i(q) = \mathbf{max}\ \{t > g \mid \mathsf{tpc}_i(t)\}$ with respect to the current value of clock g. If $g > D_i(q)$ its stops the execution and reports a deadline miss. Otherwise it continues the execution.

2. It computes the set of enabled interactions γ_q based on the received offers. Notice that they involve only ready components. It restricts the enabled interactions into safe ones. As explained in Section 4.2.2, in order to enable only safe execution, the scheduler restricts each timing constraint tc_a of each interaction a in γ_q into $\mathsf{tc}_a \wedge K_a^*$. Details on how to compute tc_a and K_a^* are presented in Section 4.2.2. If no such interaction exists, the scheduler goes to step 1.

3. It chooses a safe interaction among the enabled ones as follows. It computes for each interaction a its next activation $N_a(q)$ that corresponds to the next valuation of lock g where a is enabled. It is computed as follows: $N_a(q)$=**min** $\{t > g \mid \mathsf{tc}_a(t) \wedge K_a^*(t)\} \cup \{+\infty\}$. It chooses an enabled interaction a, that is, such that its next activation date is less than all Deadlines $D_i(q)$ of participating components $B_i \in part(a)$ that is, $N_a(q) \leq \mathbf{min}\ \{D_i(q) \mid B_i \in part(a)\}$. The chosen interaction a is executed as soon as possible, i.e. at the global time $N_a(q)$. If $N_a(q)$ is not reached by the global clock g, the scheduler goes to step 1 to wait for the next activation of the selected interaction. Otherwise it goes to the next step.

4. When the next activation of the selected interaction is reached, the scheduler notify the components participating in the selected interaction to execute.

7.3.2 Tools for Generating Send/Receive BIP models

The methods presented in Chapter 5 and Chapter 6 have been implemented through a set of tools that we present here. Figure 7.4 presents an overview of the different tools used to generate 3-layer BIP models from a BIP models. These tools are the following:

- **timedBip2SrBip** Given a BIP model and a partition of interaction, this tool generates a timed Send/Receive BIP model as described in Section 5.3.

- **Check Non-Decreasing Deadlines**. It checks whether the input BIP model satisfies the non-decreasing deadline property. More precisely, this tool checks conditions of Proposition 2.

- **Observed Components Reduction**(under construction). It implements the technique presented in Section6.5.3. Given an interaction a and an observed component B_j, this tool generates the property $\mathsf{reduce}_a(B_j)$ as defined in Proposition4. This property is then checked using RTD-Finder tool presented in Section7.2.2.

- **Earlier Decision Making timedBip2SrBip** Given a BIP model and a partition of interaction, this tool generates a Send/Receive BIP model as described in Section 6.3.

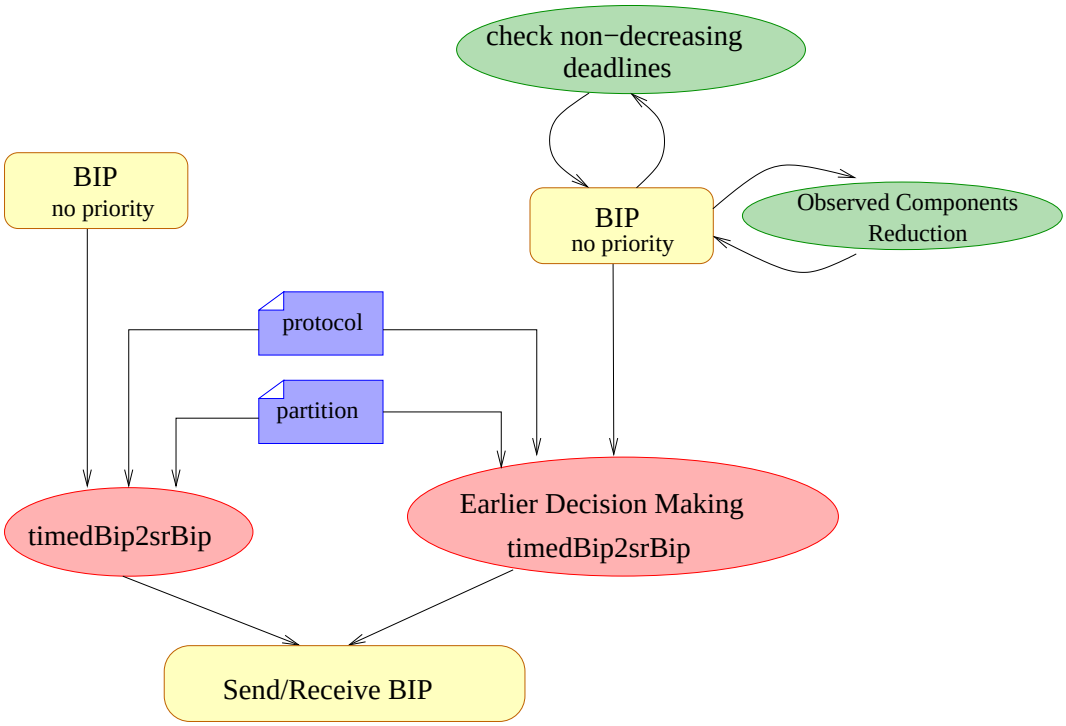


Figure 7.4: Source-to-Source Decentralization of timed BIP

7.3.3 Code generation for Send/Receive BIP models

In order to generate distributed implementations from Send/Receive BIP models, we developed a code generator that takes a Send/Receive model as input and outputs a distributed implementation. Our implementation automatically generates C++ code for each component in the Send/Receive BIP model where Send/Receive interactions are implemented by TCP sockets. Each component has access to a clock g giving access to the actual time.

Consider a Send/Receive BIP model. Each component is a Petri net whose transitions are labeled by ports. According to Definition 22, there are three types of ports in each component in a Send/Receive BIP model: send ports, receive ports and unary ports. A send port is used to trigger the emission of a message to the receive ports. The code generated from a transition labeled by a send port p includes a call to a $\mathsf{send}(p)$ function. This function sends the set of variables associated with port p to receive ports.

Unary ports correspond to computation actions in which only the component is involved. They may involve timing constraints. Therefore, in the generated code they are executed only when the corresponding timing constraint is true with respect to global clock g.

Receive ports are executed only if there are incoming messages corresponding to that ports. Therefore, if there is an enabled transition labeled by a receive port, the component has to wait for an incoming message that matches with that port.

In the generated code, we assume minimal waiting for the execution of ports. That is, we execute a port as soon as it is enabled. In addition, we give priority to execute send ports

Algorithm 1: Code generated for SR-BIP components.

```
1  Initialize()            // initialize the Petri net and connections
2  g := 0                  // reset real-time clock
3  while true do
4      while P_s ∩ enabled_q ≠ ∅ do                          // send messages
5          choose p ∈ P_s ∩ enabled_q
6          send(p)
7          s:=NextState()
8
9      D :=nextDeadline(q)
10     N := min_{p∈P_u} {nextEnabled_q(p)}
11
12     if N > g ∧ D ≥ g then
13         wait newMessage() ∨ g ≥ min(N, D)
14         if newMessage() then                              // received message
15             recv(message())
16             q:=NextState()
17             continue
18
19     else
20         if g ≥ N ∧ D ≥ g then
21         choose p ∈ P_u such that nextEnabled(p) = N
22         DoInternalComputation(p)          // internal transition
23         q:=NextState()
24
25         else
26             exit(DEADLINE_MISS)
27
```

over receive and unary ports.

Recall that we assume that components of the original BIP models can not reach states where their time progress conditions are false. In the implementation of Send/Receive models, we propose to check only the time progress conditions of the original models and to stop the execution if a deadline is missed, i.e., if there is a time progress condition (of the original model) that is evaluated to `false`.

Generated Code Description

The generated C++ code for each component in the Send/Receive model is shown in Algorithm 1. The function NextState() returns the state q of the component upon execution of a transition. We denote by $\mathsf{enabled}_q$ the set of enabled ports at the current state q. Given a port p in $\mathsf{enabled}_q$, we denote by $\mathsf{nextEnabled}_q(p)$ the next global time value at which the

port p is enabled at state q. This value is computed from the timing constraint tc_p of p as follows: $\mathsf{nextEnabled}_q(p)$=**min** $\{t > g \mid \mathsf{tc}_p(t)\} \cup \{+\infty\}$. We denote by $\mathsf{nextDeadline}(q)$ the maximal global time value until which the component is allowed to stay at q. It is computed as follows: $\mathsf{nextDeadline}(q)$=**max** $\{t > g \mid \bigwedge_{\ell \in q} \mathsf{tpc}_\ell(t)\} \cup \{g\}$

First, the component is initialized (Lines 1 to 2). Then, it enters an infinite loop that executes the transitions of the component. It scans the list of enabled ports and and executes one of them, considering that send-ports have more priority (Lines 4 to 7). Unary ports P_u may have timing constraints, and have to be executed in real-time. To this end, it computes the next activation date N corresponding to the minimal value of $\mathsf{nextEnabled}_q(p)$ for each port $p \in P_u$, and the next deadline D corresponding to the value of $\mathsf{nextDeadline}(q)$ (Lines 9 to 10). If no unary port can be executed at the current value of g (i.e. $N > g$), its waits for a unary port to be enabled (i.e. g reaches N), for the deadline to expire (i.e. g reaches D) or for a message to be received (Line 13). In the later case, it executes the corresponding port to take the message into account (Lines 14 to 17). If no message has been received until N (i.e $g \geq N$) and the deadline did not expire ($g \leq D$), it executes a unary port enabled at N (Lines 10 to 23). Otherwise, it reports a deadline miss if $g > D$ (Line 26).

8 Experimental Results

In this chapter, we present our experimental results to show how our implementations behave. We consider 3 type of applications: multi-thread, multi-process and distributed applications.

In Section 8.1, we consider a muli-thread application consisting on an obstacle avoidance application running on a MarXbot platform. With this application we compare the performance of the multi-threaded real-time scheduler against the single-thread real-time scheduler.

In Section 8.2, we consider a multi-process application consisting on an implementation of demosaicing algorithm. In order to implement this application, we use the transformations presented in Chapter 5. We compare the performance of an implementation with a centralized scheduler against the one with conflict-free schedulers.

In Section 8.3, we consider two distributed applications. The first consists on an implementation of the dining philosophers application. The second is a robotic application consisting of a set of robots that collaborate to perform a given task. Both applications are implemented using the approach presented in Chapter 6. We show the influence of the choice of the conflict resolution protocol for the implementation of such applications. Also, we show how to improve performance by using our method for optimizing observed components.

8.1 Multi-Thread Application: Avoidance obstacle

This case study evaluates the performance of the multi-thread real-time scheduler proposed in Section 7.3.1. Recall that the multi-thread real-time scheduler implements interactions and priorities of the partial state models presented in Chapter 3. The objective of this experiment is to show how preferment our proposed scheduler is, against the single-thread real-time scheduler proposed in [77].

8.1.1 MarXbot Robot

We made experiments on the marXbot platform [96], a miniature mobile robot embedding a multitude of sensors and actuators. The management of sensors and actuators is ensured via a network of distributed microcontrollers communicating through CAN bus. These microcontrollers communicate also with a central computer via CAN bus (see Figure 8.1).

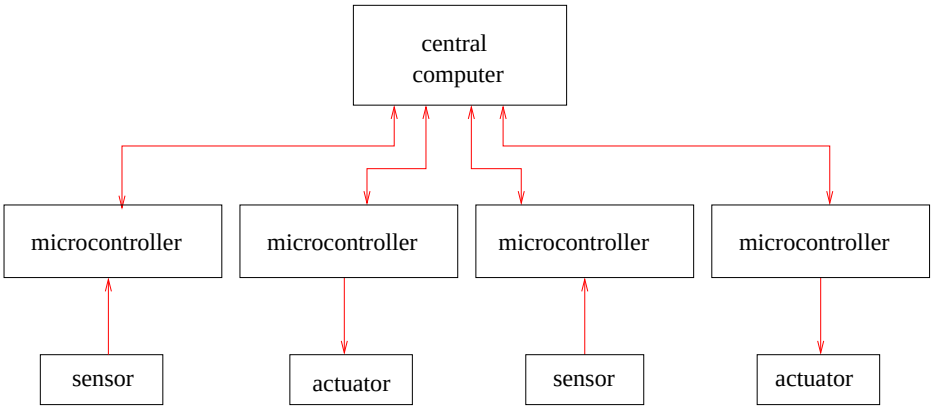


Figure 8.1: The hardware architecture of the MarXbot robot.

Mainly, the marXbot robot (see Figure 8.2) is composed of the following modules.

- The base module providing rough-terrain mobility thanks to treels (combination of tracks and wheels). It embeds also 16 infrared proximity sensors for detection of obstacles. In addition, the base module contains a slot for a swappable battery that powers the robot.
- A range and bearing module allows the robot to compute a rough estimate of the direction and the distance of the neighboring robots.
- A rotating distance scanner module including 4 infrared long range sensors is used to build 2D map of its environment.
- An attachment module provides self-assembling capabilities with peer marXbots. This module allows the docking of the other robots and can feel the force they apply.
- A main processor module which is an ARM11 running Linux-based operating system and communicating through CAN bus with 10 micro-controllers (dsPIC33) managing sensors and actuators. It drives also two cameras, one looking front and one oriented towards an omnidirectional hyperbolic mirror.

The main processor polls the microcontrollers at regular intervals to read the sensor values and to set the actuator commands. These read and write operations transit through the CAN bus. An event-based architecture, called ASEBA [97], is implemented to make the main computer communicate with the micro-controllers. ASEBA is a scripting language used to program

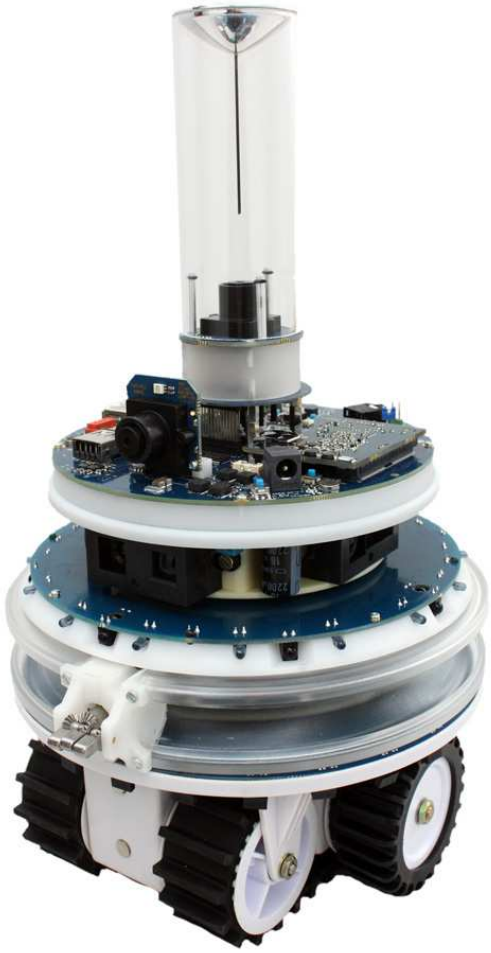


Figure 8.2: The Marxbot robot.

micro-controllers so that they are able to emit and receive events. The microcontrollers communicate through events thanks to the asynchronous communication capabilities of the CAN bus. When an event is detected by a sensor, the information is dispatched by its manager microcontroller to the rest of the robot. This is done only if the information is relevant to the application. In ASEBA, the description of the robot behavior and the events emission and reception policy is described in a scripting language. The language is a simple imperative programming language with a single basic data type (16 bit signed integers) and arrays. ASEBA provides also an IDE that provides each microcontroller a tab with the list of variables for the real-time edition of the values of the sensors, the actuators and the user-defined variables, a script editor and debug controls. The IDE compiles scripts into bytecodes and loads them to the microcontrollers, through the communication bus, to be executed in a lightweight virtual machine.

The inconvenient of ASEBA is that it does not enable the writing of complex applications

and memory allocations are limited in the micro-controllers. The ASEBA scripting language cannot provide for instance the use of 32-bits integers since it has 16-bits signed integers as its only data type. The central computer that has a Linux-based operating system, communicates with the microcontrollers via a software switch, that extends the communication bus to local TCP/IP connections. With ASEBA, it is not possible to perform complex computations or construct a big application. Thus, we use the BIP framework in order to build applications running on the centralized computer of the robot.

8.1.2 Modeling obstacle avoidance application using BIP

We consider an experimental setup for an obstacle avoidance scenario. Initially, the robot moves straight and turns whenever it detects an obstacle. Turning the robot is ensured by updating its direction and its speed. In our experiments, we make use of the three modules provided by the robot. We use the base module that contains the wheels and the proximity sensors. Also, we use the scanner module including long range sensors. Reading sensors values is ensured using predefined functions that call their manager microcontrollers. The third module that we use is the computer module running on Linux that executes the application. The BIP model of the application is composed of the following components (see Figure 8.3):

- Components *AvoidObstProxy* and *AvoidObstLRang* responsible for reading the values of the proximity and long range sensors. If one of these components detects the presence of an obstacle, it transmits its direction to component *Arbiter* through interaction *obs*. Otherwise, it sends message $free$ indicating the absence of obstacle.

- From messages received from *AvoidObstProxy* and *AvoidObstLRang*, *Arbiter* computes the new direction of the robot, which is sent to components $CtrlMotorLeft$ and *CtrlMototRight* which are the controllers of the motors

- $CtrlMotorLeft$ and *CtrlMototRight* determine the speed to apply to the left and right treels, based on the direction received from *Arbiter*. To avoid collisions, we give priority to obstacles detected by *AvoidObstProxy* over the ones detected by *AvoidObstLRang*, which is implemented by rule $obsL\ \pi\ obsP$. We also give priority to presence of obstacles over than their absence, corresponding to rules $freeP\ \pi\ obsL$ and $freeL\ \pi\ obsP$.

8.1.3 Results

Using BIP, we generated C++ code for the main processor. We compared the application running with the Multi-threaded Real-time scheduler proposed in Section 7.3.1 allowing parallel execution, with the same application running with the Single-thread Real-time scheduler of [77] allowing sequential execution. Its performance is measured by varying the period used for reading sensors in *AvoidObstProxy* and *AvoidObstLRang*. For each tested period, we ran the application 5 times under similar conditions. As shown in Figure 8.4, with the single-thread real-time scheduler the minimal period for a correct operation of the robot is 130 ms. For smaller periods time-safety may be violated which stops the application. The minimal

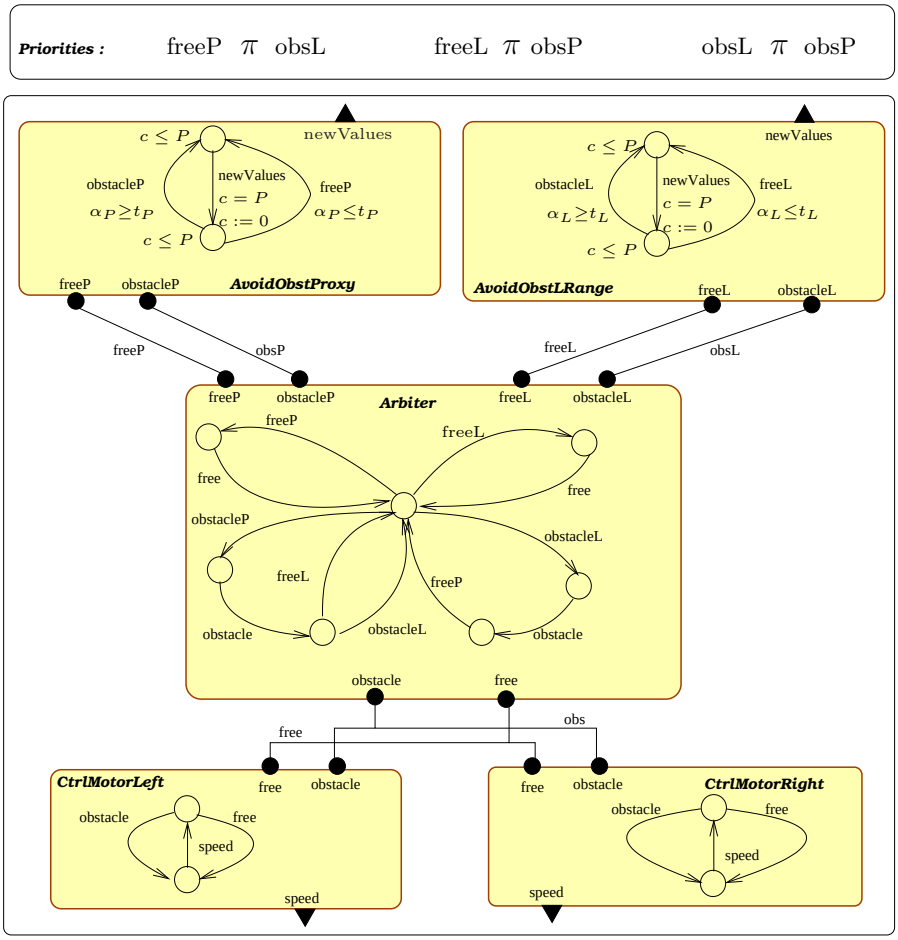


Figure 8.3: The obstacle avoidance application.

period with the multi-thread real-time scheduler is 60 ms, which drastically improved the reactivity of the robot.

The multi-thread real-time scheduler executes each component in a thread, allowing *AvoidObstProxy* and *AvoidObstLRang* to wait in parallel for new values of the sensors sent by the microcontrollers. In contrast, the single-thread real-time scheduler treats the interaction with the microcontrollers sequentially leading to the addition of the waiting times.

8.2 Multi-Process Application: Democising

This use case evaluates the framework described in Chapter 5. Recall that in Chapter 5, we transform BIP models into Send/Receive BIP models implementable on platforms providing fast communications.

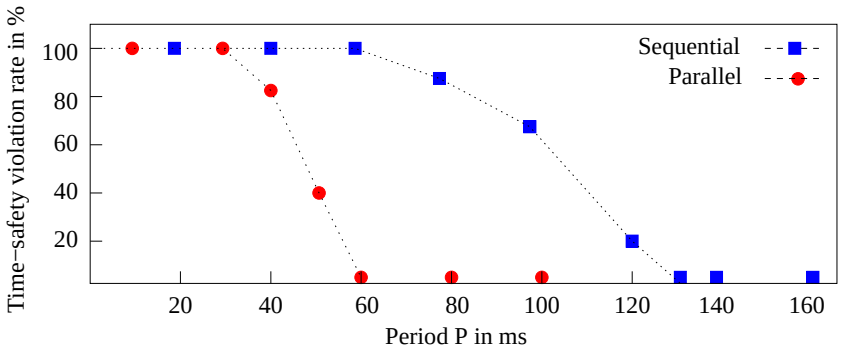


Figure 8.4: Time-safety violations for an execution of the obstacle avoidance application with the multi-thread real-time Scheduler and the single-thread real-time scheduler.

Multi-process platforms are adequate for the implementation of such models. Indeed, processes are a useful choice for parallel application with workloads where tasks take significant computing power. For that, we consider an application that implements demosaising algorithm which provides intensive parallel computation.

8.2.1 Demosaicing Algorithm

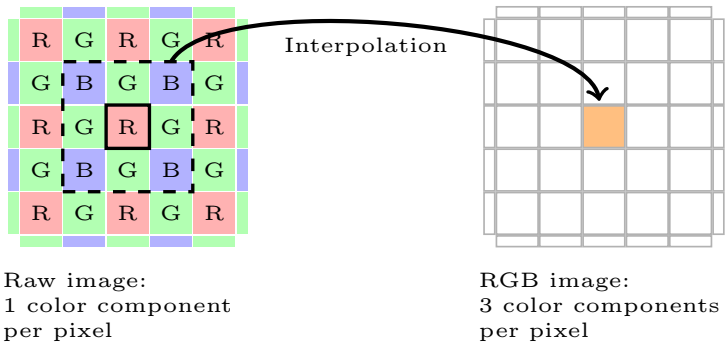


Figure 8.5: Demosaicing raw data to RGB image.

A demosaicing algorithm [98] transforms the raw data from a camera sensor into an actual image. A camera sensor is an array of light sensors, each of them outputting a single value. A color filter placed over the sensor array ensures that each sensor receives either red, green, or blue light. The filtering is done according to the pattern presented on the left of Figure 8.5. The obtained raw image contains a single color component for each pixel. Demosaicing yields an RGB image by interpolating for each pixel the missing color components from the values of the neighbor pixels. In Figure 8.5, the neighborhood contains only adjacent pixels. Depending upon the interpolation algorithm used, this neighborhood may change. In our

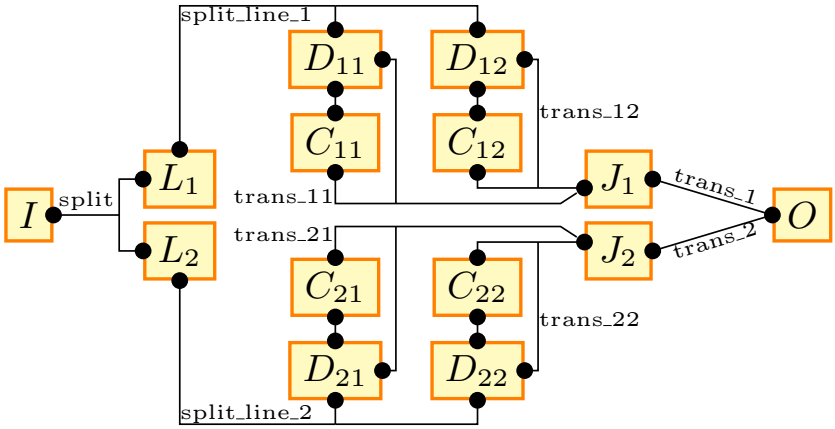


Figure 8.6: BIP model for demosaicing.

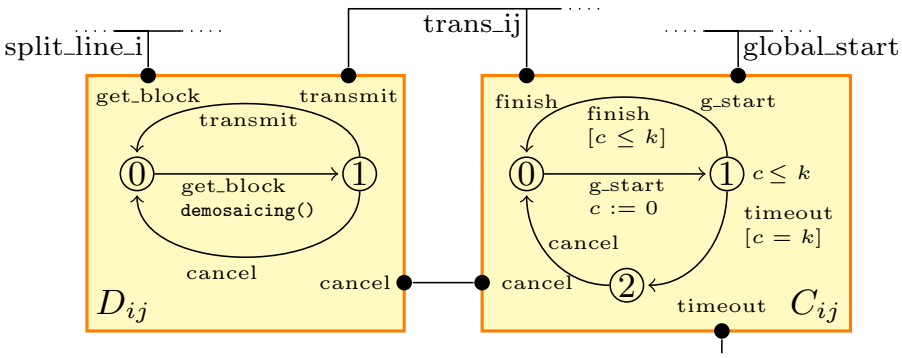


Figure 8.7: Detail of the C_{ij} and D_{ij} components.

case, we use a neighborhood of size 5×5, centered on the interpolated pixel. We do not detail here the interpolation function used.

8.2.2 Modeling Demosaicing Application in BIP

Figure 8.6 shows the BIP model of demosaicing for 4 parallel blocks. Initially, the image is loaded by the component I, which trigger a global_start interaction (not shown on the Figure) between components D_{ij}, J_i, I and O. These components reset their respective clocks upon execution of global_start.

This algorithm is parallelized by cutting the raw image into blocks, each being demosaiced concurrently. The image is first split into lines by the interaction named split, then each line i is split into blocks by the interaction named split_line_i. The detailed behavior of components D_{ij} and C_{ij} is depicted in Figure 8.7.

Component D_{ij} performs actual demosaicing of the image block located at i, j, whenever it receives an image block through its port get_block. Component C_{ij} controls timing of the component D_{ij}. We use timing constraints and time progress conditions to enforce delivery of a RGB image within a given amount of time after the raw image has been provided.

The parameter k allows to control the time to demosaice a block. When D_{ij} finishes the demosacing of a block, if less time than k elapsed since the last global start, C_{ij} allows interaction trans_ij that transmits the demosaiced block to J_i. Otherwise, the block is not transmitted, C_{ij} declares a timeout through unary interaction timeout and returns with D_{ij} to the initial state through interaction cancel. The component J_i joins blocks to form the line i and transmits it through trans_i. Its transition from the receiving state to the transmitting state is triggered if all blocks are received or if too much time elapsed since global_start.

Finally, component O merges received lines and outputs the image. The image is outputted either if all lines are received or if too much time elapsed since global_start. If some blocks or lines were not transmitted, the outputted image is incorrect on the corresponding blocks.

8.2.3 Results

We consider two different partitions for generating the Send/Receive model. The first one, called *centralized* partition puts all interactions in the same partition class. The second one is called *conflict-free* partition, and is such that:

- Interaction split, start and all interactions split_line_i define the first class of the partition.
- For each i, the interactions trans_ij are grouped in the same class, together with reset interactions.
- The interactions trans_i form the last class.

We demosaice raw images of size 25×10^6 pixels and 6×10^6 pixels. The BIP model splits each image into 9 blocks, that are demosaiced concurrently. We generate distributed code for both the centralized and the conflict-free versions. We run the code on a UltraSparc T1 that allows parallel execution of 24 processes. The parameter k in the component C_{ij} controls the amount of time after which the image must be outputted.

Figures 8.8 and 8.9 show the number of blocks processed depending on the amount of time allowed k, respectively for 25M pixels and 6M pixels raw images, for both centralized and conflict-free partition. With the centralized partition, to treat all nine blocks of the 25MP image is 29s, whereas it reduces to 17s for the conflict-free partition. Similarly, treating all nine blocks of the 6MP image requires a time budget of 7s, against 4.5s for the conflict-free partition. The conflict-free partition exhibits a speedup ranging between 1.5 to 2 comparatively to the centralized partition. The conflict-free partition allows more parallelism between interactions, for instance, interactions split_line_i can be executed in parallel, where the centralized does not. Since demosaicing components run concurrently in both implementations, the speedup stems solely from the possible parallelism between interactions.

8.3 Distributed Applications

In this section, we experiment the framework presented in Chapter 6. Recall that in Chapter 6, we transform BIP models into Send/Receive BIP models that could be implemented on distributed platforms providing slow communications.

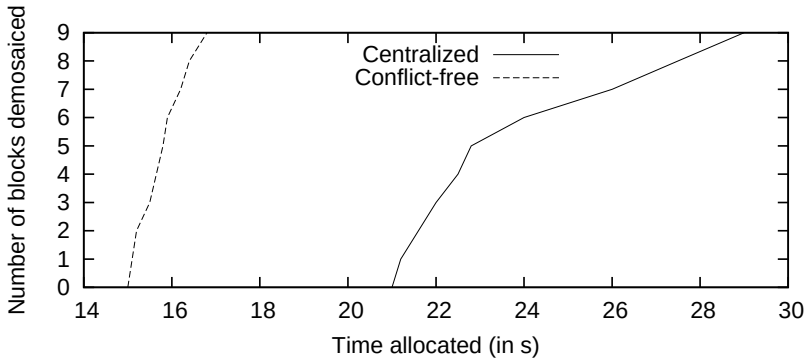


Figure 8.8: Time needed to process a 25MP image.

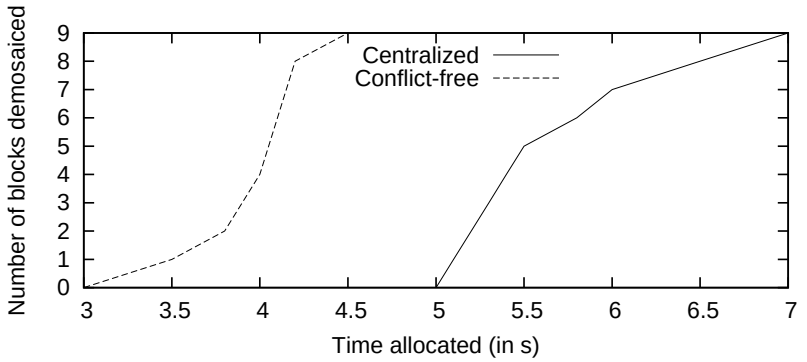


Figure 8.9: Time needed to process a 6MP image.

8.3.1 Dining "professors"

We consider a variation of the well-known dining philosophers problem. To avoid ambiguity with the dining philosophers protocol used in our design, we call this application dining professors. The scenario is described as follows: N professors sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers. Each professor must alternately think, eat and clean. A professor can eat only when he has both left and right forks. Each fork can be held by only one professor and so a professor can use the fork only if it is not being used by another professor. After he finishes eating, he needs to clean both forks so they become available to others. The problem to solve is to design the system such that no professor will starve.

In our setting, we consider that the professors are numbered from 1 to N. In addition, we consider that the professors are sit such that each two neighbors have successive numbers (professors 1 and N are neighbors). We suppose that each professor can eat with a period P. In order to avoid starvation, we suppose that professors having odd numbers can eat at the same time (similarly for even numbers). To do so, we shift the eating starting date of professors with odd numbers by $P/2$. This ensures that two neighbors are not able to eat at the same time. Figure 8.10 shows the eating points of professors with odd and even numbers.

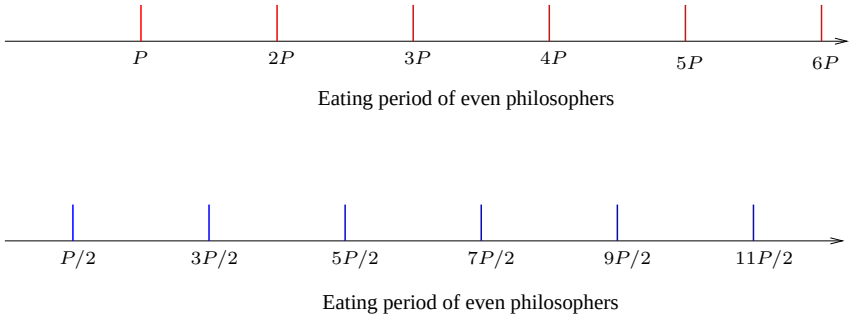


Figure 8.10: Eating points of professors with odd and even numbers.

Modeling Dining professors Application in BIP

Figure 8.11 shows a fragment of the BIP model of dining professors application. Components P_{i-1} and P_{i+1} are professors having odd numbers (i.e. i-1 and i+1 are odd numbers). These components have transition labeled delay needed to shift their starting eating date with even professors by $P/2$. Interaction eat_i involves even professor P_i and the adjacent forks. This interaction executes when the clock c_i of professor component P_i reaches the period P. When executing interaction eat_i, the clock c_i of the professor component P_i is reset. After eating, P_i has to clean the forks F_i, F_{i+1}. The time for cleaning the forks should not exceed $P/2$ units time. This is because odd professors P_{i-1} and P_{i+1} reach their periods for eating after $P/2$ units time. That is, P_i should clean the forks before that professor components P_{i-1} and P_{i+1} become ready to eat.

Notice that all components in the dining professors example have non-decreasing deadlines since they satisfy the conditions of Proposition 2.

Results

Using the transformations presented in Section 6.3, we obtain a Send/Receive BIP model of the dining professors application. As already explained, our Send/Receive BIP model is structured into 3 layers. The second layer is parametrized by a partition of interactions. In this dining professors example, we consider the following partition.

- Each interaction eat_i forms a partition class.
- Each pair of interactions $clnR_{i-1}$ and $clnL_i$ forms a partition class.

Influence of the conflict resolution protocol choice.

We study the influence of the conflict resolution protocol choice for the construction of the Send/Receive BIP model. We simulate the environment by adding communication delays. We consider the following communications delays values between the different components of the Send/Receive BIP model.

- The communication delay between the atomic components and the schedulers is 10 ms.

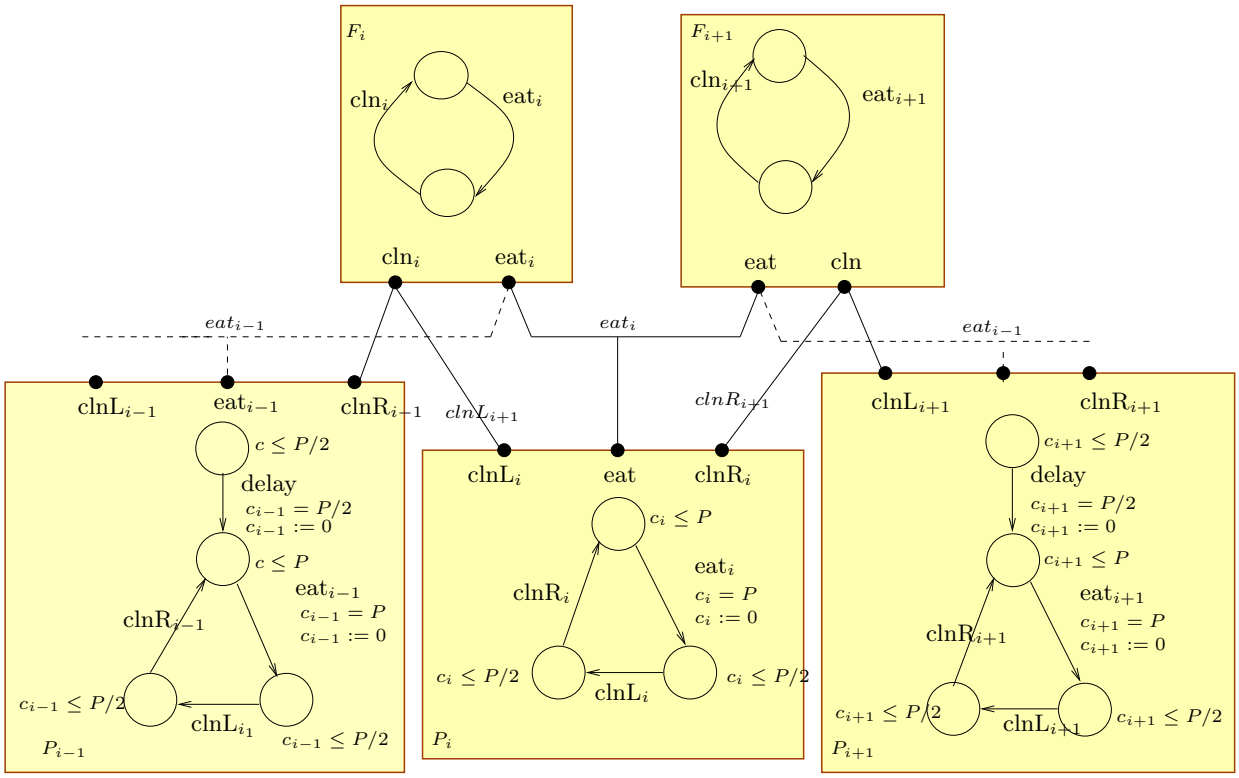


Figure 8.11: Fragment of the BIP model of dining professors application.

- The communication delay between the schedulers and the conflict resolution protocol is 10ms.
- The communication delay between different components of the conflict resolution protocol is 2ms.

We simulate communication delays by temporarily suspending the communicating components. We measure the minimum period P that allows correct execution of the application, i.e. without time-safety violation. Figure 8.12 and Figure 8.13 shows simulation results for the application with 4 and 50 professors respectively. Recall that we denote by RP the centralized implementation of the conflict resolution protocol , by TR token ring based implementation and by DP the dining philosophers based implementation. For 4 professors, the RP outperforms TP and DP. The latters show comparable performance. This is because for TP and DP, the conflict resolution protocol requires a similar amount of communication for acquiring either the fork or the token. Indeed, for 4 professors, we have 4 conflicting interactions and therefore, 4 components inside the conflict resolution protocol for TR and DP. For TR, the token has to travel through all these 4 components. For DP, each conflict resolution component acts in their local neighborhood, by sending requests for acquiring the fork. This has a similar effect in terms of communications. Although RP schedules interactions sequentially, this has less impact regarding the effect of communications in DP and

RP. For 50 philosophers, the DP outperforms TP and RP. In fact, increasing the number of

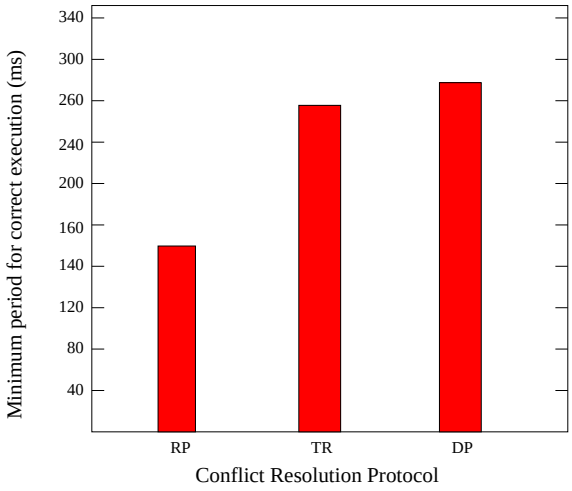


Figure 8.12: Simulation For 4 professors.

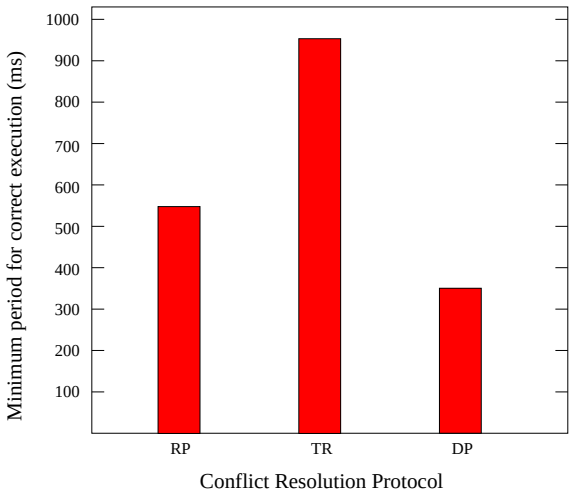


Figure 8.13: Simulation For 50 professors.

professors and thus increasing the number of conflicting interactions, does not affect so much the performance of the DP based implementation. This is because, for DP, each conflict resolution component acts always in their local neighborhood components, regardless the total number of components in the conflict resolution protocol. However, for TR, the communications amount required for circulating the token depends on the number of component in the conflict resolution protocol, which corresponds to the total number of conflicting interactions. For RP implementation, scheduling interactions sequentially slows down the application execution, but still, this has not the same effect compared to communication required for TR

implementation.

Optimizing Observed Components.

As explained in Subsection 6.2.1, in order to schedule correctly an interaction, each interaction should observe components participating in conflicting interactions. In the dining professors example, each interaction eat_i is in conflict with its neighbor interactions eat_{i-1} and eat_{i+1}. Therefore, interaction eat_i has to observe components components professor components P_{i-1} and P_{i+1} and fork components F_{i-1} and F_{i+1}.

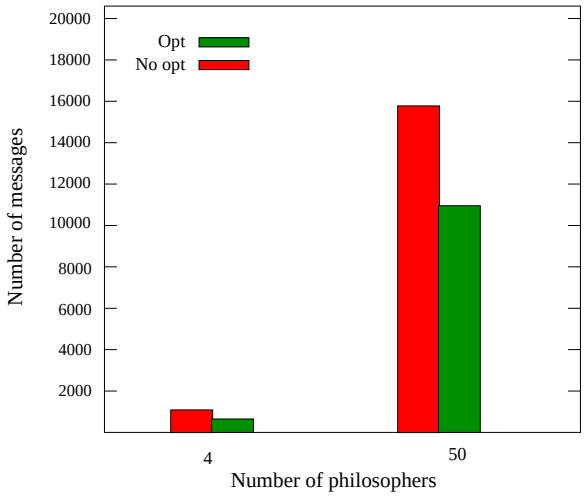


Figure 8.14: Number of messages exchanged for optimized and non-optimized implementations for the dining professors example.

In order to optimize the number of observed components of interaction eat_i, we check for each component $B_i \in \{P_{i-1}\ ,\ P_{i+1},\ F_{i-1}\ ,\ F_{i+1}\}$ the satisfiability of $\mathsf{reduce}_{eat_i}(B_i)$ property defined in Definition 40. We report the following results. Fork components F_{i-1}, F_{i+1} satisfy the property and thus, they could not to observed by eat_i interaction. However, professor components P_{i-1}, P_{i+1} do not satisfy the property, and thus, they have to be observed by eat_i. Indeed, the property checks if there is clocks valuation that satisfy eat_i timing constraint and do not satisfy time progress conditions of observed components when being in a location enabling conflicting interactions. Since fork components have `true` as time progress conditions on their locations, in particular on locations enabling eat transition, then any valuation of clocks satisfying the timing constraint of interaction eat_i satisfy necessarily time progress conditions of Fork components F_{i-1} and F_{i+1}.

For the experimentations, we measure the number of messages needed to let each professors eat 10 times, for both optimized and non optimized version of the application for 4 and 50 philosophers. Figure 8.14 shows the simulation results. We remark an improvement of performance for the optimized version of the application with both 4 and 50 professors. This is explained by the fact that for unoptimized version, the scheduler handling eat_i should receive observation messages (offers) from fork components F_{i-1} and F_{i+1} and professor

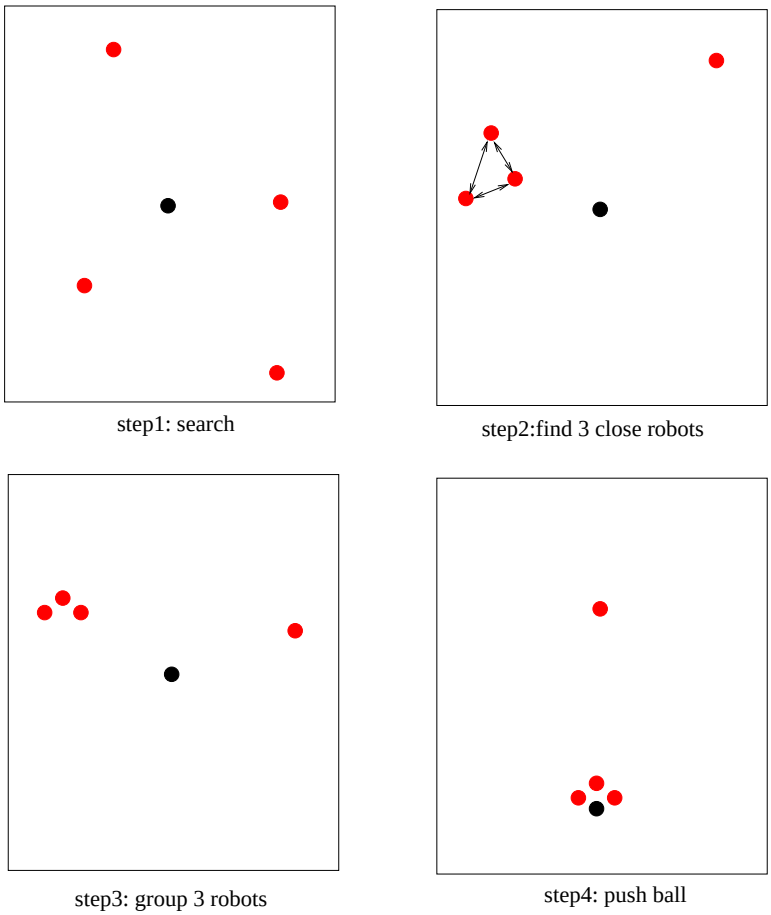


Figure 8.15: Collaborating Robot application scenario.

componentsP_{i-1} and P_{i+1}. Whereas, for the optimized version the scheduler handling eat_i should receive observation messages only from professor components P_{i-1} and P_{i+1}. This explains the decrease of the number of exchanged messages for optimized version.

8.3.2 Collaborating Robots

We consider a robotic application that consists of N communicating robots that collaborate to perform a given task. The scenario is described as follows (see Figure 8.15 for the description of the scenario with 4 robots): initially, the robots (red circles) are randomly distributed over arena. They start by exploring the arena in order to find each others. When 3 robots become sufficiently close, they group themselves by forming "V" shape. Then, they go towards a ball (black circle) which is positioned at the center of the arena, and push it. The number of combination for "grouping" 3 robots from a set of N robots is given by the binomial coefficient $\binom{N}{3}$. We denote by $P_3(N)$ the set of 3-combinations from the set of N robots.

We assume that the robots are equipped with proximity sensors to detect obstacles

(arena's walls and the ball) and a camera to detect the robots. Moreover, we assume that the robots can communicate with each others at any time (e.g. using wireless communications). This robotic application could be implemented on a set of marXbot robots (see Subsection 8.1.1 for the description of such platform) since these robots provide all features (proximity sensors, camera, wifi communication) required to implement the application. In this work, we rely on simulations as we do not have sufficient number of robots to implement the application.

Modeling the application in BIP

Figure 8.16 shows the BIP model of a single robot. We use timing constraints and time progress conditions to express a periodic reading sensors. Notice that such a component has non-decreasing deadlines since it satisfies the conditions of Proposition 2. Note that transitions *turn*, *continue* and *group* have mutually exclusive guards. Therefore, they cannot be enabled at the same time.

Figure 8.17 shows the BIP model of the application composed of 4 robots. The grouping action is modeled by an interaction that synchronizes *group* transitions of any 3 robots. The "grouping" interaction is enabled only if there are 3 robots which are sufficiently close to each others. Similarly, "pushing" action is modeled by an interaction that synchronizes *push* transitions of any group of 3 robots. By construction, this interaction is enabled only if the corresponding robots successfully grouped themselves.

Generally, there are $\binom{N}{3}$ "grouping" interactions g_k and $\binom{N}{3}$ "pushing" interactions p_k where $k \in P_3(N)$ is the robots group identifier. These interactions are pairwise conflicting as for any two interactions there is at least one shared port. In Figure 8.17, there are 4 "grouping" interactions and 4 "pushing" interactions as the number of 3-combinations from a set of 4 robots is 4. Note that for each robot, there are 3 unary interactions which are sense_i, continue_i and turn_i. Note that interaction continue_i and turn_i are not conflicting with interaction g_k due to their mutually exclusive guards.

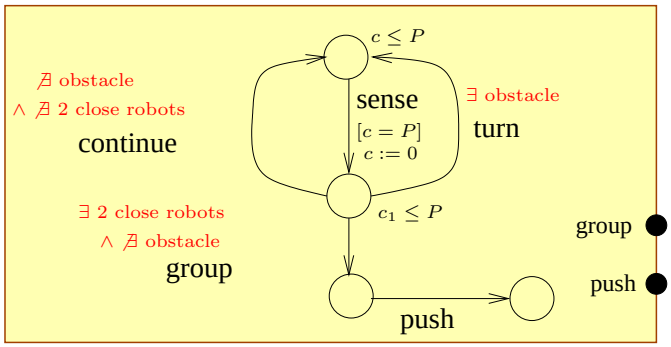


Figure 8.16: Model of a single robot.

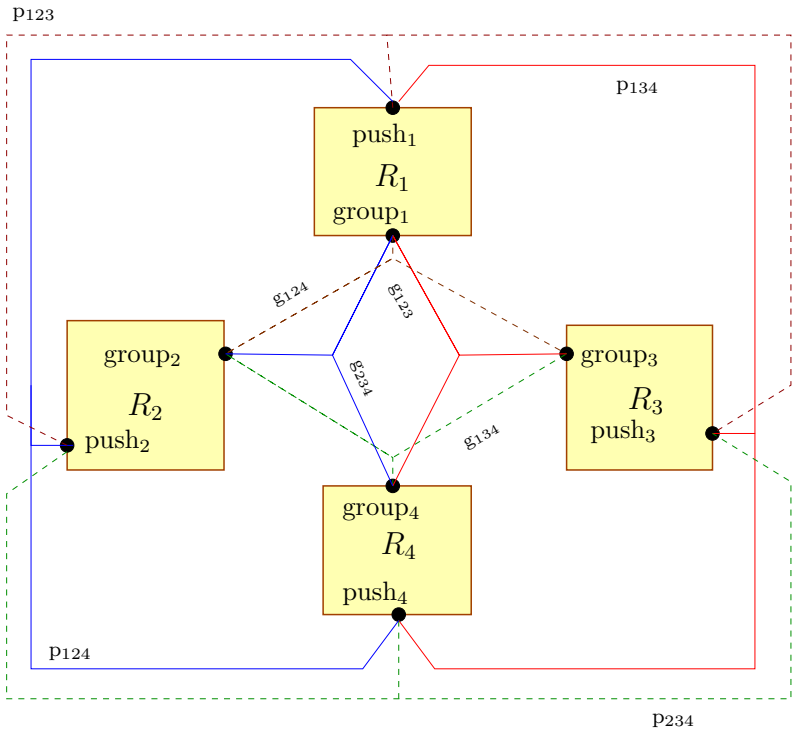


Figure 8.17: The BIP model of the application with 4 robots

Results

Using the transformations described in Section 6.3, we transform the BIP model of our application into Send/Receive BIP where:

- all unary interactions of robot R_i, $i \in \{1 \dots N\}$ are handled by a single scheduler S_i
- "grouping" and "pushing" interactions of the same group $k \in P_3(N)$ are handled by a single scheduler S_k

In this example, we use DP and TR implementation of the conflict resolution protocol.

We simulate the robots platform by considering that each robot is running on a dedicated machine. The number of machines corresponds to the number of robots used in the application. We generate C++ code corresponding to each component of the Send/Receive BIP model and we embedded each C++ executable on each machine. We distribute the C++ code as follows: each machine i hosts a robot's code R_i^{SR}, its corresponding scheduler S_i and one scheduler S_k and conflict resolution protocol components CRP_{g_k} and CRP_{p_k} such that the robot R_i is participating in interactions managed by S_k. Table A.1 shows the distribution of the C++ code corresponding to the application with 4 robots.

In order to make simulations as realistic as possible, we also modeled robot's behavior, such as robot's movement, sensor's reading and camera image processing. Figure 8.18 presents

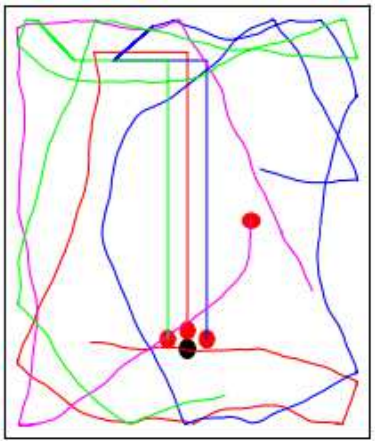


Figure 8.18: Simulation Results for the application with 4 robots

the simulation results for P=200 ms. The red circles represent the final positions of the robots and the black one represents the ball. It shows that the robots effectively managed to group themselves and push the ball.

Influence of the conflict resolution protocol choice.

We study the impact of communications delays and the choice of the conflict resolution protocol on time-safety. We simulate communication delays by temporarily suspending the communicating components. We assume that between any two machines i and j, we have the same communication delay K. We set the sensing period P to 200 and we vary the communication delays between machines. We measure the maximum value of communication delays that allows correct execution of the application.

Figure 8.19 and Figure 8.20 shows the simulation results for the application with 4 and 10 robots. For 4 robots, TR (Token Ring) outperforms DP (dining philosophers). This is because for few number of conflicting interactions, TR requires less communication than DP. Indeed, in DP, each conflict resolution protocol component has to negotiate the forks with all other components in DP which requires more communications than required for traveling the token in TR. However, for 10 robots, DP outperforms TR. Indeed, for a large number

Table 8.1: Send/Receive components code distribution on 4 machines

machine#1	machine#2	machine#3	machine#4
R_1^{SR}	R_2^{SR}	R_3^{SR}	R_4^{SR}
S_1	S_2	S_3	S_4
S_{123}	S_{124}	S_{134}	S_{234}
$CRP_{g_{123}}$	$CRP_{g_{124}}$	$CRP_{g_{134}}$	$CRP_{g_{234}}$
$CRP_{p_{123}}$	$CRP_{p_{124}}$	$CRP_{p_{134}}$	$CRP_{p_{234}}$

of conflicting interactions, TR requires lot of communications as the token has to travel in all conflict resolution components sequentially before arriving to the component that takes the decision to execute the interaction. That is, the time required for the token to arrive corresponds to the sum of communications delays between components in TR. Whereas in DP, the forks negotiation is done in parallel. That is each conflict resolution protocol component sends fork requests to all other components almost at the same time.

Optimizing Observed Components

In our example, each two "group" interactions g_k and $g_{k'}$ are in conflict. Therefore, each g_k interaction involves its participant components and observes all remaining components. Our method for optimization allows removing all observed components for each g_k interaction. Indeed, in the BIP model, the robots have the same behavior and the same period for sensing which make the robots having the same deadline when searching for each others. Thus, when 3 robots try to group themselves, they have to do it before the expiration of their periods of sensing, which is the same for the other (observed) robots.

We measure the number of exchanged messages needed or the execution of the application during 10s. We require that the robot remains at searching phase during the execution. To do so, we put `false` as guard on each g_k interaction in order to forbid robots grouping them selves. Figure 8.21 shows the number of exchanged messages needed or the execution of the application with 4 and 10 robots. We remark that the performance is improved in the optimized version especially for the application with 10 robots. For instance, in the non optimized version with 10 robots, the scheduler S_k handling g_k interaction have to receive messages from all robots: 3 participating in the interaction and 7 other observed. In the optimized version, S_k receives only messages from the participant robots.

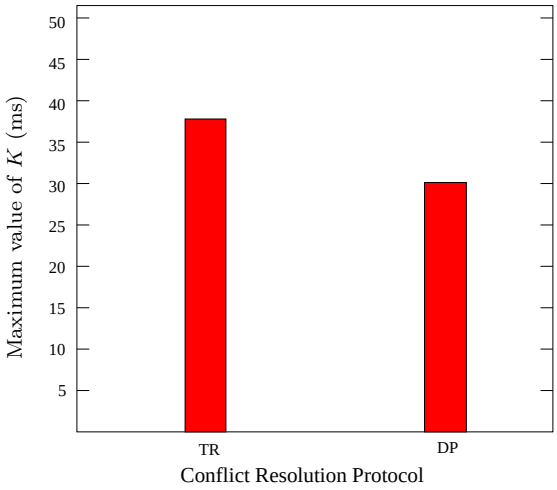


Figure 8.19: Simulation for 4 robots.

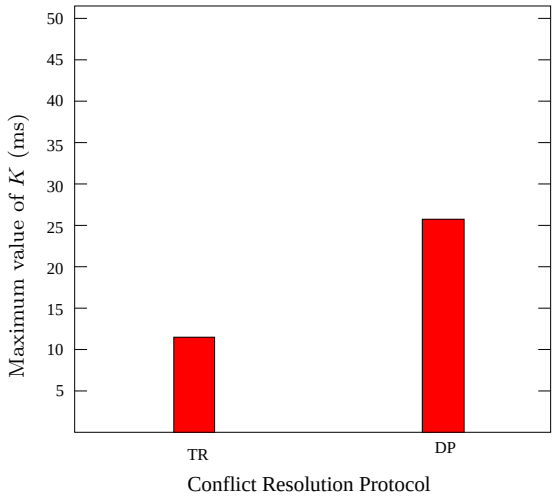


Figure 8.20: Simulation for 10 robots.

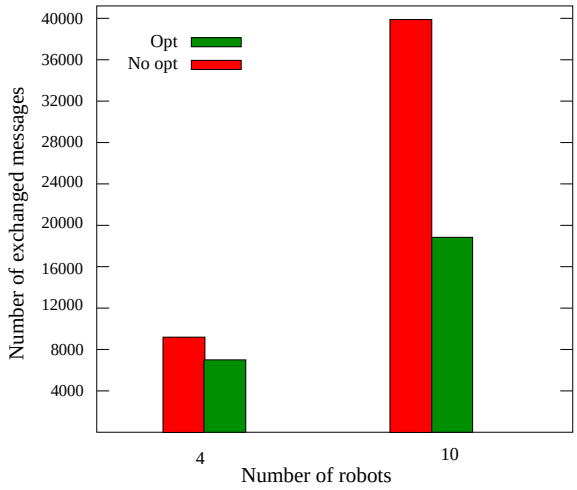


Figure 8.21: Number of exchanged messages needed or the execution of the application during 10s.

9 Conclusions

In this chapter, we conclude the thesis by describing the main achievements and the future work directions and its perspectives.

9.1 Achievements

Rigorous Design Flow

In this thesis, we presented a rigorous design flow starting by a centralized high-level BIP models and leading to Send/Receive BIP models that can be directly implementable on a given platform. The design flow involves the use of model transformations, each one takes into account a particular aspect of the decentralization while preserving the functional properties of the original BIP model.

We presented three solutions for obtaining Send/Receive BIP models.

- In the first solution, we propose Send/Receive models with a centralized scheduler that implements interactions and priorities. Atomic components of the original models are transformed into Send/Receive components that communicate with the centralized scheduler via Send/Receive interactions. We showed that the centralized scheduler should satisfy some conditions to ensure safe interactions execution. These conditions was identified in the partial state models to be observationally equivalent to original BIP models. Recall that partial state models are high level-representation for parallel execution of BIP models in which the assumption of atomic execution of transitions is discarded.

- In the second solution, we proposed Send/Receive models with decentralized the schedulers. With this solution, we considered BIP models without priorities. The solution is based on transforming BIP models into 3-layer Send/Receive BIP models which consist

of Send/Receive atomic components, a set of schedulers each one handling a subset of interactions, and a set of components implementing a conflict resolution protocol. Components in the conflict resolution protocol are called by the schedulers only in case of conflicting interactions execution.

With the above solutions, we assumed that the obtained/Send Receive models are implemented on platforms that provide fast communications (e.g. multi-process platforms) to meet perfect synchronization in components. This is because our obtained Send/Receive model is modeled such that there is no delay between the decision to execute an interaction in a scheduler and its execution in the participating components.

- In the third solution, we proposed Send/Receive models that execute correctly even if communications are not fast enough. This method is based on the fact that schedulers plan interactions execution and notify components in advance. In particular, the schedulers choose as soon as possible, according to a scheduling policy, a date for executing a selected interaction and send it to the participating components. We proposed also a cancel mechanism whenever the scheduler is not able to find a valid date to execute a conflicting interaction when the conflict resolution protocol confirms its execution.

  We showed that with this solution, the scheduler needs to observe additional components not participating in the scheduled interactions. In particular, the schedulers observe time progress conditions of observed components to ensure safe interactions scheduling. These time progress conditions represent deadlines of observed components. In order to maintain valid observations of deadlines in the schedulers, we assumed that the components have non-decreasing deadlines. That is, observations are valid even if the components change their states by executing interactions.

  We proposed an optimization method in order to minimize the number observed components. In fact, we identified the condition for which a component could not be observed by a given interaction. We expressed this condition in terms of a property of the system. We used static analysis technique based of approximations of system's reachable states to verity satisfaction of the property.

Implementation and Results

Actually, our first solution consisting on the centralized scheduler that implements interactions and priorities was presented only for sake of clarity and to make connections with the rest of the thesis. Instead, we have developed a multi-thread real-time scheduler that implements interactions and priorities of partial state models. A code generator have been also developed to generate C++ code for atomic components.

For distributed implementations, we developed tools for generating 3-layer Send/Receive BIP models. We developed a code generator that takes a Send/Receive model as input and outputs a distributed implementation. The code generator automatically generates C++ code for each component in the Send/Receive BIP model where Send/Receive interactions are implemented by TCP sockets.

Regarding experiments, we considered three types of applications: multithreaded, multiprocess and distributed applications.

- With the multi-threaded application, we evaluated the performance of the multi-thread real-scheduler. We showed that the latter outperforms the single-thread real time scheduler presented in [77].
- With multi-process application, we evaluated the method of our second solution. We considered an application that needs intensive parallel computation. We compared the application implementations obtained with a centralized scheduler against the one obtained with conflict-free schedulers.
- With distributed applications, we evaluated the method of our third solution. We studied the influence of the chosen conflict resolution protocol with different simulated platforms. We showed that the best performance depends on the application size and also on the chosen conflict resolution protocol.

  Also, we studied the influence of optimizing the number of observed components in Send/Receive models. We showed that this optimization allows reducing the number of exchanged messages which increases performance.

9.2 Perspectives

For future work, we are considering several research directions.

- In this thesis, priorities are handled only in Send/Receive models with centralized scheduler. An important future direction is to handle priorities in the 3-layer Send/Receive models. For the case of our 3-layer Send/Receive models implemented on fast platforms, the solution is not very difficult. In fact, the schedulers may observe components participating in higher priority interactions. As in the case of Send/Receive models with centralized scheduler, the observation brings approximations of next reachable states. Based on these approximations, the schedulers may decide the execution of a lower priority interaction.

  For the case of Send/Receive models with planning interactions, the problem is quiet difficult. The problem resides in the fact interactions are planned in advance. In order to plan correctly an interaction, one has to be sure that when the interaction executes at the selected date, no higher interaction is enabled at that date.
- Our third solution for obtaining Send/Receive models with planing interactions is based on components observation. This method is applied on BIP models whose components provide non-decreasing deadlines. In fact, considering such models allows the schedulers to maintain a valid observation of components deadlines. We are working on extending this method for more general models where components do not necessarily have non-decreasing deadlines. This is not trivial since the scheduler cannot be sure that its observation could be valid in the future. For example, assume that the scheduler observes a component at its current state, indicating that the component's deadline is D. Based in this observation, the scheduler may schedule an interaction at a date less or equal to than D. However, this is not safe since the component may change its state providing a deadline less than D.

A pragmatic approach, where the scheduler blocks the execution of the observed component until the date t of the selected interaction execution, is not safe. In fact, the observed component may need to synchronize with other components having deadlines less than t.

The solution could be the use of static analyses techniques to identify states where it is safe to schedule a given interaction.

- Another research direction could be to propose a method for obtaining Send/Receive BIP models that handle clocks drift. The problem of clocks drift may influence the coherence of interactions execution. In particular, the interactions order could be inverted. For instance, consider two interactions a and b handled in different schedules having access to drifted clocks. Suppose that a and b are scheduled to execute at date t_a and t_b respectively . Here, t_a and t_b are computed based on different clocks. Logically talking, if $t_a \leq t_b$, a should be produced before b. However, in reality, it could be possible that b executes before a. That is a and b executes at real date t_a^r and t_b^r respectively, where $t_b^r \leq t_a^r$. These real dates could be computed based on a reference clock.

  A solution to handle this problem is to introduce perturbations in the BIP models, and then study implementability issues for these models. The perturbation consist on enlarging the timing constraints. For timed automata, the model of timing constraint enlargement has been studied in [99]. In this paper, it is proven that the model with perturbation covers the issue of clocks drift, by reducing the implementability problem to the analysis of the enlarged semantics.

- Another work direction is the issue of mixed critical systems [100]. A mixed critical application allows different levels of criticality to interact on the same platform. However, those applications are very difficult to certificate. This is because mixed criticality concept requires that even the components of less criticality be certified at the highest criticality level. Thus, it is necessary to provide strong scheduling theories for real-time and safety-critical system design and implementation.

A

From models with time progress cnditions to urgency-based models

A.1 Urgency-based Abstract Models

Time progress conditions are used to enforce progress in the system. Indeed, time progress conditions constraints the system to stay infinitely at a state without moving to another one. There exists different manners to enforce progress in the system. In [101], the authors consider abstract models with urgency types on transitions to enforce transitions execution. Urgency types are used to specify the need for a transition to progress when it is enabled (i.e. when its timing constraint is true) [102]. There are three urgency types that can be used to specify urgency of transitions: *Lazy* transitions (i.e. non-urgent) are denoted by `l`, *delayable* transitions (i.e. urgent just before they become disabled) are denoted by `d`, and *eager* transitions (i.e. urgent whenever they are enabled) are denoted by `e`. A transition urgency is computed from the timing constraint and the urgency type of the transition. We define a *time guard* tg which corresponds to a combination of timing constraint tc with an *urgency type* $\nu \in \{\ \mathtt{l}, \mathtt{d}, \mathtt{e}\ \}$, denoted by $\mathsf{tg} = [\mathsf{tc}]^{\nu}$. The predicate $\mathsf{urg}[\mathsf{tg}]$ that characterizes the valuations of clocks for which the time guard tg is *urgent* is defined by:

$$\mathsf{urg}[\mathsf{tg}](t) \iff \begin{cases} \mathtt{false} & \text{if } \mathsf{tg} \text{ is lazy, i.e. if } \nu = \mathtt{l} \\ \mathsf{tc}(t) \wedge \neg \mathsf{tc}(t+1) & \text{if } \mathsf{tg} \text{ is delayable, i.e. if } \nu = \mathtt{d} \\ \mathsf{tc}(t) & \text{if } \mathsf{tg} \text{ is eager, i.e. if } \nu = \mathtt{e}. \end{cases}$$

Definition 41 (Urgency-based Abstract Model)**.** *An urgency-based abstract model is defined by a labeled transition system* $B = (L, P, T, \mathsf{C})$*, where:*

- L *is a finite set of control locations,*
- P *is a finite set of communication ports,*

- C *is a finite set of clocks,*
- $T \subseteq Q \times P \times Q$ *is a finite set of labeled transitions. A transition* τ *is a tuple* (q, p, q') *where* p *is a communication port.* τ *has a time guard over* C tg_τ *and resets a subset of clocks* r_τ*.*

Definition 42 (Semantics of Urgency-Based Abstract Model)**.** *An urgency-based abstract Model* $B = (L, P, T, \mathsf{C})$ *defines a transition system* $S_B = (Q_B, P_B, \longrightarrow_B)$ *where:*

- $Q = L \times \mathcal{T}(C)$ *is the set of states*
- $P_B = P \cup \mathbb{Z}_{\geq 0}$
- $\underset{B}{\longrightarrow} \subseteq Q_B \times P_B \times Q_B$ *is the set of labeled transitions defined as follows: Let* (ℓ, t) *and* (ℓ', t') *be two states,* $p \in P$*, and* $\delta \in \mathbb{Z}_{\geq 0}$ *be a delay*
  - Jump transitions: *We have* $(\ell, t) \xrightarrow[B]{p} (\ell', t')$ *where* $t' = t[r \mapsto 0]$ *if* $\tau = (q, p, q')$ *is in* T *and its corresponding timing constraint is* `true` *with respect to the valuation* t*, i.e.* $\mathsf{tc}_\tau(t) = \mathtt{true}$*.*
  - Delay transitions: *We have* $(\ell, t) \xrightarrow{\delta} (\ell, t + \delta)$ *if for all transitions* $\tau = (\ell, a, \ell')$ $\in T$ *and for all* $\delta' \in [0, \delta[$*,* $\neg \mathsf{urg}[\mathsf{tg}_\tau](t + \delta')$*.*

Compared to abstract models with time progress conditions, abstract models with urgency differ in that the condition for time progress is given implicitly and rather derived from time guards (combination of timing constraint and urgency) of transitions. Thus, abstract models with urgency are subclass of Abstract Models with time progress condition [103]. The transformation of an abstract model with urgency into an abstract model with time progress condition could be done by deriving *deadline* formula from any time guard of transitions . Given a time guard $\mathsf{tg} = [\mathsf{tc}]^\nu$ where $\mathsf{tc} = \bigvee_{i=1}^{n} \bigwedge_{c \in \mathsf{C}} l_c^i \leq c \leq u_c^i$ is given by (2.1), we define deadline d characterizing valuation of clocks for which time is enforced to stop progress. d is defined as follows:

$$d(\mathsf{tg}) \;=\; \begin{cases} \mathtt{false} & \text{if } \mathsf{tg} \text{ is lazy, i.e. if } \nu = \mathtt{l} \\ \bigvee_{i=1}^{n} \bigvee_{c \in \mathsf{C}} \;\; c = u_c^i & \text{if } \mathsf{tg} \text{ is delayable, i.e. if } \nu = \mathtt{d} \\ \mathsf{tc} & \text{if } \mathsf{tg} \text{ is eager, i.e. if } \nu = \mathtt{e}. \end{cases}$$

Let ℓ be a state in L and $\tau_k = \{(q, p_k, q_k)\}_{k \in K}$ be the set of all transitions issued from ℓ. The time progress condition associated to ℓ and derived from $\{\mathsf{tg}_{\tau_k}\}_{k \in K}$ is:

$$\mathsf{tpc}_\ell = \neg \bigvee_{k \in K} d(\mathsf{tg}_{\tau_k})$$

Example 29. *Consider the abstract model with urgency in Figure A.1. In table 29, we give the time progress condition associated to control location* ℓ *for different types of urgency* ν_1 *and* ν_2*.*

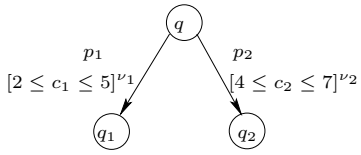


Figure A.1: Example of transitions with urgency.

Table A.1: Time progress conditions of ℓ for different types of urgency

ν_2, ν_2	`l`	`d`	`e`
`l`	`false`	$c_1 = 5$	$2 \leq c_1 \leq 5$
`d`	$c_2 = 7$	$c_1 = 5 \vee c_2 = 7$	$2 \leq c_1 \leq 5 \vee c_2 = 7$
`e`	$4 \leq c_2 \leq 7$	$c_1 = 5 \vee 4 \leq c_2 \leq 7$	$2 \leq c_1 \leq 5 \vee 4 \leq c_2 \leq 7$

References

[1] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer, 2011. ISBN 978-1-4419-8236-0. URL `http://dx.doi.org/10.1007/978-1-4419-8237-7`.

[2] Yuh-Jzer Joung and Scott A Smolka. A comprehensive study of the complexity of multiparty interaction. *Journal of the ACM (JACM)*, 43(1):75–115, 1996.

[3] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems - concepts and designs (3. ed.)*. International computer science series. Addison-Wesley-Longman, 2002. ISBN 978-0-201-61918-8.

[4] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[5] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49. ACM, 2004.

[6] K Lee, John C Eidson, Hans Weibel, and Dirk Mohl. Ieee 1588-standard for a precision clock synchronization protocol for networked measurement and control systems. In *Conference on IEEE*, volume 1588, page 2, 2005.

[7] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. ISBN 0-201-05866-9.

[8] Rajive Bagrodia. A distributed algorithm to implement n-party rendevouz. In *Foundations of Software Technology and Theoretical Computer Science, Seventh Conference (FSTTCS)*, pages 138–152, 1987.

[9] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering (TSE)*, 15(9):1053–1065, 1989.

[10] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.

[11] K. Mani Chandy and Jayadev Misra. *Parallel program design - a foundation*. Addison-Wesley, 1989. ISBN 978-0-201-05866-6. I-XXVIII, 1-516 pp.

[12] Marcin Jurdzinski. Small progress measures for solving parity games. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 290–301, 2000.

[13] D. Kumar. An implementation of n-party synchronization using tokens. In *ICDCS*, pages 320–327, 1990.

[14] J. Parrow and P. Sjödin. Multiway synchronizaton verified with coupled simulation. In *International Conference on Concurrency Theory (CONCUR)*, pages 518–533, 1992.

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

[16] O. Babaoglu. On the reliability of consensus-based fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems*, November 1987.

[17] Hermann Kopetz. Time-triggered real-time computing. *Annual Reviews in Control*, 27 (1):3–13, 2003. URL `http://dx.doi.org/10.1016/S1367-5788(03)00002-6`.

[18] Christophe Aussaguès and Vincent David. A method and a technique to model and ensure timeliness in safety critical real-time systems. In *4th International Conference on Engineering of Complex Computer Systems (ICECCS '98), 10-14 August 1998, Monterey, CA, USA*, pages 2–12, 1998. URL `http://doi.ieeecomputersociety.org/10.1109/ICECCS.1998.706651`.

[19] Arkadeb Ghosal, Thomas A. Henzinger, Christoph M. Kirsch, and Marco A. A. Sanvido. Event-driven programming with logical execution times. In *Hybrid Systems: Computation and Control, 7th International Workshop, HSCC 2004, Philadelphia, PA, USA, March 25-27, 2004, Proceedings*, pages 357–371, 2004. URL `http://dx.doi.org/10.1007/978-3-540-24743-2_24`.

[20] 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.

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

[22] 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 *Computer Safety, Reliability and Security, 17th International Conference, SAFECOMP'98, Heidelberg, Germany, October 5-7, 1998, Proceedings*, pages 45–59, 1998. URL `http://dx.doi.org/10.1007/3-540-49646-7_4`.

[23] Stéphane Louise, Vincent David, Jean Delcoigne, and Christophe Aussaguès. OASIS project: deterministic real-time for safety critical embedded systems. In *Proceedings*

of the 10th ACM SIGOPS European Workshop, Saint-Emilion, France, July 1, 2002, pages 223–226, 2002. URL http://doi.acm.org/10.1145/1133373.1133419.

[24] Stéphane Louise, Matthieu Lemerre, Christophe Aussaguès, and Vincent David. The OASIS kernel: A framework for high dependability real-time systems. In *13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011, Boca Raton, FL, USA, November 10-12, 2011*, pages 95–103, 2011. URL http://dx.doi.org/10.1109/HASE.2011.38.

[25] 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.

[26] Mathieu Jan, Jean-Sylvain Camier, and Vincent David. The oasis-d transparent approach for safety-critical distributed real-time systems.

[27] Mathieu Jan, Jean-Sylvain Camier, and Vincent David. Scheduling safety-critical real-time bus accesses using time-constrained automata. In *RTNS*, pages 87–96. Citeseer, 2011.

[28] Damien Chabrol, Vincent David, Christophe Aussaguès, Stéphane Louise, and Frédéric Daumas. Deterministic distributed safety-critical real-time systems within the oasis approach. In *IASTED PDCS*, pages 260–268, 2005.

[29] David D Falconer, Fumiyuki Adachi, and Bjorn Gudmundson. Time division multiple access methods for wireless personal communications. *Communications Magazine, IEEE*, 33(1):50–57, 1995.

[30] David B. Stewart, Richard Volpe, and Pradeep K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. Software Eng.*, 23(12):759–776, 1997. URL http://doi.ieeecomputersociety.org/10.1109/32.637390.

[31] Lory D. Molesky, Chia Shen, and Goran Zlokapa. Predictable synchronization mechanisms for multiprocessor real-time systems. *Real-Time Systems*, 2(3):163–180, 1990. URL http://dx.doi.org/10.1007/BF00365325.

[32] Jan Lunze and Daniel Lehmann. A state-feedback approach to event-based control. *Automatica*, 46(1):211–215, 2010.

[33] Edward A Lee and II John. Overview of the ptolemy project, 1999.

[34] S. Tsang and E. Magill. Detecting feature interactions in the intelligent network. *Feature Interactions in Telecommunications Systems II, IOS Press*, 1994.

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

[36] Edward A Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7(1-4):25–45, 1999.

[37] Patricia Derler, Edward A Lee, and Slobodan Matic. Simulation and implementation of the ptides programming model. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 330–333. IEEE Computer Society, 2008.

[38] Yang Zhao, Jie Liu, and Edward A Lee. A programming model for time-synchronized distributed real-time systems. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE*, pages 259–268. IEEE, 2007.

[39] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of computer programming*, 16(2):103–149, 1991.

[40] 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.

[41] Nicolas Halbwachs. About synchronous programming and abstract interpretation. *Sci. Comput. Program.*, 31(1):75–89, 1998. URL `http://dx.doi.org/10.1016/S0167-6423(96)00041-X`.

[42] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, pages 1–16, 1998. URL `http://dx.doi.org/10.1007/BFb0028726`.

[43] Nicolas Halbwachs. About synchronous programming and abstract interpretation. In *SAS*, pages 179–192, 1994. URL `http://dx.doi.org/10.1007/3-540-58485-4_40`.

[44] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*. LNCS 1427, Springer Verlag, Vancouver (B.C.), June 1998.

[45] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, 2008.

[46] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[47] Daniel Pilaud, N Halbwachs, and JA Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[48] Christos Sofronis. *Embedded Code Generation from High-level Heterogeneous Components*. PhD thesis, Université Joseph-Fourier-Grenoble I, 2006.

[49] Safouan Taha, Ansgar Radermacher, Sébastien Gérard, and Jean-Luc Dekeyser. MARTE: uml-based hardware design from modelling to simulation. In *Forum on specification and Design Languages, FDL 2007, September 18-20, 2007, Barcelona, Spain, Proceedings*, pages 274–279, 2007. URL `http://www.ecsi-association.org/ecsi/main.asp?l1=library&fn=def&id=269`.

[50] OMG. Uml superstructure, v2.4.1. object management group. August 2011.

[51] Sébastien Demathieu, Frédéric Thomas, Charles André, Sébastien Gérard, and François Terrier. First experiments using the uml profile for marte. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 50–57. IEEE, 2008.

[52] Antonio Wendell De Oliveira Rodrigues, Guyomarc'H Frédéric, and Jean-Luc Dekeyser. An mde approach for automatic code generation from marte to opencl. 2011.

[53] Edward A Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, 1998.

[54] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[55] Huafeng Yu, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier, Frédéric Mallet, Charles André, and Robert De Simone. Polychronous analysis of timing constraints in uml marte. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2010 13th IEEE International Symposium on*, pages 145–151. IEEE, 2010.

[56] Charles André, Frédéric Mallet, and Robert De Simone. Modeling time (s). In *Model Driven Engineering Languages and Systems*, pages 559–573. Springer, 2007.

[57] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (aadl): An introduction. Technical report, DTIC Document, 2006.

[58] Zhibin Yang, Kai Hu, Dianfu Ma, Jean-Paul Bodeveix, Lei Pi, and Jean-Pierre Talpin. From AADL to timed abstract state machines: A verified model transformation. *Journal of Systems and Software*, 93:42–68, 2014. URL `http://dx.doi.org/10.1016/j.jss.2014.02.058`.

[59] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 (2):183–235, 1994.

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

[61] Lacramioara Astefanoaei, Souha Ben Rayana, Saddek Bensalem, Marius Bozga, and Jacques Combaz. Compositional invariant generation for timed systems. In *TACAS*, pages 263–278, 2014.

[62] Joseph Sifakis. A framework for component-based construction extended abstract. In *SEFM*, pages 293–300, 2005.

[63] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Software Engineering and Formal Methods (SEFM)*, pages 3–12, 2006.

[64] Joseph Sifakis. Component-based construction of real-time systems in bip. In *CAV*, pages 33–34, 2009.

[65] S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *Concurrency Theory (CONCUR)*, pages 508–522, 2008.

[66] R. Alur and D. Dill. Automata for modeling real-time systems. *ICALP*, 1990.

[67] Rajeev Alur and David L. Dill. The theory of timed automata. In *REX Workshop*, pages 45–73, 1991.

[68] Stavros Tripakis. Verifying progress in timed systems. In Joost-Pieter Katoen, editor, *ARTS*, volume 1601 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 1999. ISBN 3-540-66010-0.

[69] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989. ISSN 0018-9219.

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

[71] S. Bliudze and J. Sifakis. Causal semantics for the algebra of connectors. In *Formal Methods for Components and Objects (FMCO)*, pages 179–199, 2007.

[72] Mohamad Jaber. *Implémentations Centralisée et Répartie de Systèmes Corrects par construction á base des Composants par Transformations Source-á-source dans BIP*. PhD thesis, Université Joseph-Fourier - Grenoble I, 2010.

[73] M. Bozga, M. Jaber, and J. Sifakis. Source-to-source architecture transformation for performance optimization in BIP. In *Symposium on Industrial Embedded Systems (SIES)*, pages 152–160, 2009.

[74] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *FORTE*, pages 116–133, 2008.

[75] R. Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

[76] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[77] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-based implementation of real-time applications. In Luca P. Carloni and Stavros Tripakis, editors, *EMSOFT*, pages 229–238. ACM, 2010. ISBN 978-1-60558-904-6.

[78] Jacques Combaz Ahlem Triki, Borzoo Bonakdarpoor. Automated conflict-free concurrent implementation of timed component-based models. Technical Report TR-2015-2, Verimag Research Report.

[79] K. G. Larsen, P.Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[80] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 298–302. Springer, 1998.

[81] Thomas A Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244, 1994.

[82] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In *ATVA*, pages 64–79, 2008.

[83] RTD-Finder Tool. `http://www-verimag.imag.fr/RTD-Finder.html?lang=`. Accessed: 2015-04-09.

[84] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Rigorous implementation of real-time systems - from theory to application. *Mathematical Structures in Computer Science*, 23(4):882–914, 2013. URL `http://dx.doi.org/10.1017/S096012951200028X`.

[85] Marius Bozga, Vassiliki Sfyrla, and Joseph Sifakis. Modeling synchronous systems in bip. In *EMSOFT*, pages 77–86, 2009.

[86] Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga, and Joseph Sifakis. Compositional translation of simulink models into synchronous bip. In *SIES*, pages 217–220, 2010.

[87] Vassiliki Sfyrla. *Modeling Synchronous Systems in BIP*. PhD thesis, Université de Grenoble, 2011.

[88] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. Translating aadl into bip - application to the verification of real-time systems. In *MoDELS Workshops*, pages 5–19, 2008.

[89] Ananda Basu, Laurent Mounier, Marc Poulhiès, Jacques Pulou, and Joseph Sifakis. Using bip for modeling and verification of networked systems – a case study on tinyos-based networks. In *NCA*, pages 257–260, 2007.

[90] 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 *MEMOCODE*, pages 11–20, 2011.

[91] T.H. Nguyen S. Bensalem, M. Bozga and J. Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *Computer Aided Verification (CAV)*, pages 614–619, 2009.

[92] Saddek Bensalem, Marius Bozga, Jacques Combaz, and Ahlem Triki. Rigorous system design flow for autonomous systems. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, pages 184–198, 2014. URL `http://dx.doi.org/10.1007/978-3-662-45234-9_13`.

[93] Saddek Bensalem, Marius Bozga, Benoît Delahaye, Cyrille Jégourel, Axel Legay, and Ayoub Nouri. Statistical model checking qos properties of systems with sbip. In *ISoLA (1)*, pages 327–341, 2012.

[94] Borzoo Bonakdarpour, Marius Bozga, and Jean Quilbeuf. Model-based implementation of distributed systems with priorities. *Design Autom. for Emb. Sys.*, 17(2):251–276, 2013. URL `http://dx.doi.org/10.1007/s10617-012-9091-0`.

[95] Ahlem Triki, Jacques Combaz, Saddek Bensalem, and Joseph Sifakis. Model-based implementation of parallel real-time systems. In *Fundamental Approaches to Software Engineering*, pages 235–249. Springer, 2013.

[96] Michael Bonani, Valentin Longchamp, Stéphane Magnenat, Philippe Rétornaz, Daniel Burnier, Gilles Roulet, Florian Vaussard, Hannes Bleuler, and Francesco Mondada. The marxbot, a miniature mobile robot opening new perspectives for the collective-robotic research. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 18-22, 2010, Taipei, Taiwan*, pages 4187–4193, 2010. URL `http://dx.doi.org/10.1109/IROS.2010.5649153`.

[97] Stéphane Magnenat, Basilio Noris, and Francesco Mondada. Aseba-challenge: An open-source multiplayer introduction to mobile robots programming. In *Fun and Games, Second International Conference, Eindhoven, The Netherlands, October 20-21, 2008. Proceedings*, pages 65–74, 2008. URL `http://dx.doi.org/10.1007/978-3-540-88322-7_7`.

[98] Henrique S. Malvar, Li-wei He, and Ross Cutler. High-quality linear interpolation for demosaicing of bayer-patterned color images. In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2004, Montreal, Quebec, Canada,*

May 17-21, 2004, pages 485–488, 2004. URL http://dx.doi.org/10.1109/ICASSP.2004.1326587.

[99] Martin De Wulf, Laurent Doyen, and Jean-François Raskin. Systematic implementation of real-time models. In *FM 2005: Formal Methods*, pages 139–156. Springer, 2005.

[100] Sanjoy K Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Mixed-criticality scheduling of sporadic task systems. In *Algorithms–ESA 2011*, pages 555–566. Springer, 2011.

[101] Tesnim Abdellatif. *Rigorous Implementation of Real-Time Systems*. PhD thesis, Universié de Grenoble, France, 2012.

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

[103] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In *COMPOS*, pages 103–129, 1997.