Rigorous Design Flow for Programming Manycore Platforms

Paraskevas Bourgos

▶ **To cite this version:**

Paraskevas Bourgos. Rigorous Design Flow for Programming Manycore Platforms. Other [cs.OH]. Université de Grenoble, 2013. English. NNT : 2013GRENM012 . tel-01135186

HAL Id: tel-01135186
https://theses.hal.science/tel-01135186

Submitted on 24 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Paraskevas Bourgos

Thèse dirigée par **Saddek Bensalem**

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

Rigorous Design Flow for Programming Manycore Platforms

Thèse soutenue publiquement le **9 Avril 2013**,
devant le jury composé de :

M., Albert Cohen
Professeur, École Polytechnique, Rapporteur
M., Radu Grosu
Professeur, Vienna University of Technology (TUW), Rapporteur
M., Roberto Passerone
Professeur, University of Trento, Examinateur
M., Jean Claude Fernandez
Professeur, Université Joseph Fourier (UJF), Examinateur
M., Joseph Sifakis
Professeur, École Polytechnique Fédérale de Lausanne (EPFL), Examinateur
M., Saddek Bensalem
Professeur, Université Joseph Fourier (UJF), Directeur de thèse

...to my parents

Abstract

The advent of many-core platforms is nowadays challenging our capabilities for efficient and predictable design. To meet this challenge, designers need methods and tools for guaranteeing essential properties and determining tradeoffs between performance and efficient resource management.

In the process of designing a mixed software/hardware system, functional constraints and also extra-functional specifications should be taken into account as an essential part for the design of embedded systems. The impact of design choices on the overall behavior of the system should also be analyzed. This implies a deep understanding of the interaction between application software and the underlying execution platform.

We present a rigorous model-based design flow for building parallel applications running on top of many-core platforms. The flow is based on the BIP - Behavior, Interaction, Priority - component framework and its associated toolbox. The method allows generation of a correct-by-construction mixed hardware/software system model for many-core platforms from an application software and a mapping. It is based on source-to-source correct-by-construction transformations of BIP models. It provides full support for modeling application software and validation of its functional correctness, modeling and performance analysis of system-level models, code generation and deployment on target many-core platforms.

Our design flow is illustrated through the modeling and deployment of various software applications on two different hardware platforms; MPARM and platform P2012. MPARM is a virtual ARM-based multi-cluster manycore platform, configured by the number of clusters, the number of ARM cores per cluster, and their interconnections. On MPARM, the software applications considered are the Cholesky factorization, the MPEG-2 decoding, the MJPEG decoding, the Fast Fourier Transform and the Demosaicing algorithm. Platform 2012 (P2012) is a power efficient manycore computing fabric, which is highly modular and based on multiple clusters capable of aggressive fine-grained power management. As a case study on P2012, we used the HMAX algorithm.

Experimental results show the merits of the design flow, notably performance analysis as well as correct-by-construction system level modeling, code generation and efficient deployment.

Résumé

L'objectif du travail présenté dans cette thèse est de répondre à un verrou fondamental, qui est "comment programmer d'une manière rigoureuse et efficace des applications embarquées sur des plateformes multi-coeurs?". Cette problématique pose plusieurs défis : 1) le développement dune approche rigoureuse basée sur les modèles pour pouvoir garantir la correction; 2) le "mariage "entre modèle physique et modèle de calcul, c'est-à-dire, l'intégration du fonctionnel et non-fonctionnel; 3) l'adaptabilité. Pour s'attaquer à ces défis, nous avons développé un flot de conception rigoureux autour du langage BIP. Ce flot de conception permet l'exploration de l'espace de conception, le traitement à diffèrent niveaux d'abstraction à la fois pour la plate-forme et l'application, la génération du code et le déploiement sur des plates-formes multi-cœurs. La méthode utilisée s'appuie sur des transformations source-vers-source des modèles BIP. Ces transformations sont correctes-par-construction.

Nous illustrons ce flot de conception avec la modélisation et le déploiement de plusieurs applications sur deux plates-formes différentes. La première plate-forme considérée est MPARM, une plate-forme virtuelle, basée sur des processeurs ARM et structurée avec des clusters, o chacun contient plusieurs cœurs. Pour cette plate-forme, nous avons considéré les applications suivantes: la factorisation de Cholesky, le décodage MPEG-2, le décodage MJPEG, la Transformée de Fourier Rapide et un algorithme de demosaicing. La seconde plate-forme est P2012, une plate-forme multi-cœur, basée sur plusieurs clusters capable d'une gestion énergétique efficace. L'application considérée sur P2012 est l'algorithme HMAX.

Les résultats expérimentaux montrent l'intérêt de notre flot de conception, notamment l'analyse des performances ainsi que la modélisation au niveau du système, la génération de code et le déploiement.

Contents

Part

Context

- Chapter 1 -

Introduction - From Programs to Systems

General Context Embedded systems have become essential part of our daily lives. In contrast to general purpose computer systems, embedded systems integrate software and hardware, and are specifically designed to perform particular predefined tasks, which are often critical. Usually, they are not standalone devices, but they constitute the computerized part of a larger device. Mainly, we divide embedded systems in two categories: the reactive systems that continuously interact with the environment and the transformational systems that compute a function and terminate. Embedded systems appeared in the market in the early 1960s and since then they have become ubiquitous. Applications can be found in a tremendous variety of domains covering medical equipment, telecommunications, military applications, household appliances and consumer electronics, wireless sensor networks, transportation and avionics. The complexity of embedded systems varies from single micro-controller chip to multiple units and networks incorporated inside a larger system.

Efficiency of embedded systems is of paramount importance. To construct efficient systems optimizations are eligible upon design for energy cost, code size, execution time, weight and dimension, performance and reliability for both the software and the hardware part.

Embedded systems require a synergistic function between software and hardware design and development. The software is specific and often executed in a repeated fashion. However, for reusability reasons software application are often platform independent and immaterial. Conceptually, it is developed based on a high-level model which is decomposed into multiple components for the sake of complexity. A component-based model is normally well structured and is ideally characterized by formal semantics.

A software application is designed to execute on a logical time rather than on a real-time axis. Specifically, abstractions are considered about the behavior such as concurrent execution, instantaneous computation and communication steps, atomicity of actions and zero delays. The above abstractions should consider the functional constraints imposed by the program specifications such as deadlocks, throughput and jitter. Along with the software design, the hardware design should also be considered as a part of an efficient embedded system design. Small homogeneous processing units connected by a Network-on-Chip tend to succeed the complex superscalar architectures nowadays. The number of cores to be integrated in a single chip is expected to rapidly increase in the coming years, moving from multicore to manycore architectures. This leads to investigate new approaches for embedded systems design. Many-core computing architectures require correct design and programming methodologies which exploit parallelism, thus increasing

the performance, the scalability and the flexibility of the system. A correct design should also include successful resource allocation and run-time power management. It should also exploit the innovative capabilities of dynamic extension and reconfiguration, at run-time, of the architecture template depending on a possible variable workload and an interactive environment.

In the process of designing a mixed software/hardware system, functional constraints and also extra-functional specifications should also be taken into account as an essential part for the design of embedded systems. The extra-functional specifications concern the use of resources of the execution platform such as time, shared memory, semaphores, queues, energy, distribution and communication of tasks and scheduling policies. Interacting with the platform is real-time. Therefore, it becomes non-trivial the task of evaluating if a software/hardware system preserves the timing properties of its application software. It requires analysis of the impact of design choices on the overall behavior of the system. It also implies a deep understanding of the interaction between application software and the underlying execution platform. We currently lack approaches for modeling mixed hardware/software systems. There are no rigorous techniques for deriving global models of a given system from models of its application software and its execution platform.

System Level Design System design is the process leading to a mixed software/hardware system meeting given specifications. It involves the development of application software taking into account features of an execution platform. The latter is defined by its architecture involving a set of processors equipped with hardware-dependent software such as operating systems as well as primitives for coordination of the computation and interaction with the external environment. A simplified view of a system design is illustrated in Figure 1.1.

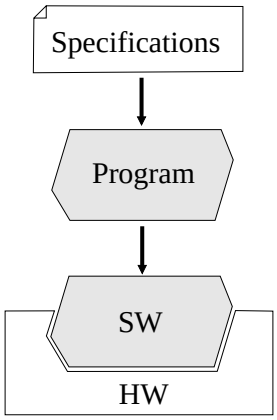


Figure 1.1: *Simplified View of a System Design*

Design approaches are developed based on the experience and expertise of the design teams. They tend to re-use, extend and improve existing solutions proven efficient and robust. This favors increased productivity since design methodologies are re-used. Reusability of components is an important issue. Systems are built by reusing and assembling components that are simpler sub-systems. This is the only way to master complexity and to ensure correctness of the overall design, while maintaining or increasing productivity. However, a design methodology may turn to be counter-productive and result to low adaptivity upon new system requirements. Better solutions may be a priori excluded because they do not fit the designers know-how. The main goal of a system design, albeit the heterogeneity of the assembling components and the difficulties upon integration of

different technologies, is the efficient prediction of the behavior of a software application running on an execution platform.

A system design flow consists of steps starting from specifications and leading to an implementation on a given execution platform. It involves the use of methods and tools for progressively deriving the implementation by making adequate design choices.

We consider that a system design flow must meet the following essential requirements:

Functionality and Performance. The design flow must allow the satisfaction of both functional and extra-functional properties. This means that functional properties such as deadlock freedom, jitter, throughput and resources such as memory, time and energy are first class concepts encompassed by formal models. Moreover, it should be possible to analyze and evaluate efficiency in using resources, which conform with the functional requirements, as early as possible along the design flow. Lack of adequate semantic models does not allow consistency checking for timing requirements, or meaningful composition of features.

Correctness. This means that the designed system meets its specifications. Ensuring correctness requires that the design flow relies on models with well-defined semantics. Semantics should be defined at both the execution and the interaction level of the models. The models should consistently encompass system description at different levels of abstraction from application software to its implementation. Correctness can be achieved by application of verification techniques. It is desirable that if some specifications are met at some step of the design flow, they are preserved in all the subsequent steps. This can be achieved by transforming the application software model to include the important physical aspects of the target platform.

Heterogeneity and Adaptivity. Heterogeneity in systems design can be supported by developing high level domain-specific languages ease of expression. System model semantics should encompass heterogeneity in computation, interaction and abstraction level to facilitate modeling mixed hardware/software systems. This should be accompanied by reusability allowing the definition of libraries of components to be reused and the development of component-based solutions. The design flow should not enforce any particular programming or execution model. Specific programming models or implementation principles may a priori exclude efficient solutions and parallelism. For instance, programming multimedia applications in plain C may lead to designs obscuring the inherent functional parallelism and involving built-in scheduling mechanisms that are not optimal. It is essential that designers use adequate programming models. The above characteristics along with a tool integration for programming, validation and code generation perceive a productive design flow.

We call *rigorous* a design flow which allows guaranteeing essential properties of the specifications. Most of the rigorous design flows privilege a unique programming model together with an associated compilation chain adapted for a given execution model. For example, synchronous system design relies on synchronous programming models and usually targets hardware or sequential implementations on single processors [Hal93]. Alternatively, real-time programming based on scheduling theory for periodic tasks, targets dedicated real-time multitasking platforms [BW01].

A rigorous design flow should be characterized by the following:

- it is *model-based*, that is, both application software and mixed hardware/ software system descriptions are modeled by using a single, semantic framework. As stated

in [BBB+11], this allows maintaining the coherency along with the flow by proving various transformations used to move from one description to another while preserving essential properties. This means that the semantic model is expressive enough to directly encompass various types of component heterogeneity arising along the design flow [HS06].

- it should be *component-based*, that is, it should provide primitives for building composite components as the composition of simpler components. The use of components reduces development time by favoring component reuse and provides support for incremental analysis and design, as introduced in [BBNS08, BBNS09, BBL+10].

- it should be *correct-by-construction*, that is, all design flow steps concerning the synthesis of the final model should be proven to guarantee the preservation of all properties of the initial input model.

- it should be *tool-supported*, that is, all steps in the design flow should be realized automatically by tools ensuring significant productivity gains.

1.1 System Design Flow

We propose a system construction method that is both rigorous and allows fine-grain analysis of system dynamics. It is rigorous because it is based on formal models described in BIP [BBS06], with precise semantics that can be analyzed by using formal techniques. A system model in BIP is derived by progressively integrating constraints induced on an application software by the underlying hardware architecture. In contrast to ad hoc modeling approaches, the system model is obtained, in a compositional and incremental manner, from BIP models of the application software and respectively, the hardware architecture, by application of source-to-source transformations that are proven correct-by-construction. The system model describes the behavior of the mixed hardware/software system and can be simulated and formally verified using the BIP toolset [bip]. The method for the construction of mixed hardware/software system models is illustrated in Figure 1.2. It takes as inputs: (i) the application software model, the hardware architecture and (iii) the mapping between them. It proceeds in four main steps.

The first step is the generation of the *application software model in BIP*. This is achieved by an automatic translation of the input application software model which should be described in a process network with a well defined structure. The translation preserves intact the behavior and the characteristics of the initial application software.

The second step is the synthesis of the *hardware architecture model in BIP*. A library of BIP atomic components that characterize manycore architectures is defined, including models for hardware components (e.g., processor, memory) and for hardware-dependent software components (e.g., FIFO channel read/write, bus controllers, schedulers). Combining the hardware architecture specifications and the suitable BIP library components, we synthesize the hardware architecture model in BIP. The model is parametrized and allows flexible integration of specific target architecture features, such as arbitration policy, latency for buses and memories, scheduling policy etc.

The third step is the construction of the mixed software/hardware *system model*. This model represents the behavior of the application software running on the hardware architecture according to the mapping, but without taking into account execution times for the software actions. This step consists in progressively enriching the application software

model by doing: (1) Application of a sequence of source-to-source transformations to synthesize *hardware dependent software routines* implementing communication by using the hardware components. (2) Integration of hardware components used in the system model. The transformations are proved correct-by-construction, that is, they preserve functional properties of the application software.

In the final step, the (bounds for) execution times are obtained by analysis or simulation of the run of every software process in isolation on the platform. These bounds are injected into the *system model* and lead to the *calibrated system model*. This final model allows accurate estimation through simulation of real-time characteristics (response times, delays, latencies, throughput, etc.) and indicators regarding resource usage (bus conflicts, memory conflicts, etc.).

The above design flow sticks to the general principles of *rigorous design* introduced in the previous section. Namely, it is *model-based*, *component-based*, *correct-by-construction* and *tool-supported*.

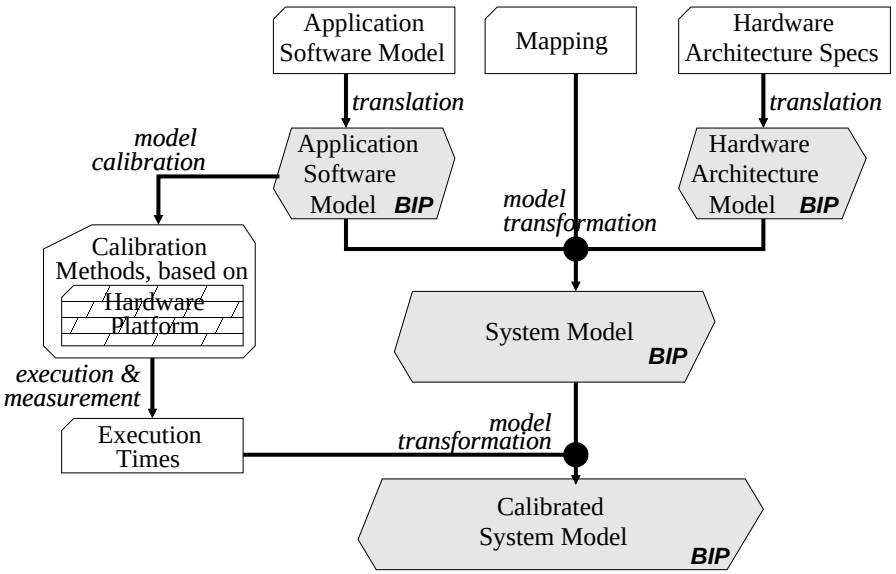


Figure 1.2: *System Model Design Flow*

1.2 Related Work in System Design

To the best of our knowledge, the BIP design flow is unique as it uses a single semantic framework to support application modeling, validation of functional correctness, performance analysis on system models and code generation for manycore platforms. Building faithful system models is mandatory for validation and performance analysis of concurrent software running on manycore platforms.

Synchronous languages [BB91] such as Esterel [BG92], Lustre [HCRP91] and Signal [BBGLG85] offer strong formal semantics that facilitate by construction verification and code generation, but they have remained limited to safety-critical domains such as aviation and automotive.

Simulink [Mat] and Stateflow [Sta] are synchronous formalisms that are mainly used to generate quickly an input for an FPGA prototype implementation through Verilog and VHDL.

Many of the frameworks, which we present below, follow the Y-chart design principle [BCG+97, KDVvdW97] as we do in our design flow. Namely, these are DOL, SPADE, Sesame, Polis, Metropolis, Artemis, Octopus and CoFluent. This means that they decouple application from architecture by recognizing two distinct models for them. According to the Y-chart approach, an application model -derived from a target application domain- describes the functional behavior of an application in an architecture-independent manner. The principle is illustrated in Figure 1.3.

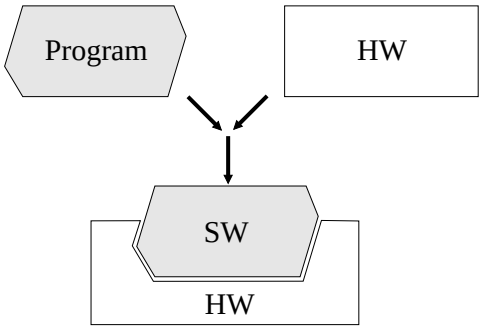


Figure 1.3: *Software Application - Hardware Platform mapping of a System Design*

DOL [TBHH07] introduces a framework for specifying and mapping parallel applications onto heterogeneous multiprocessor platforms. It defines abstraction models for the application and architecture, as well as a format for the mapping specification. They integrate an analytic performance analysis strategy to replace modeling and analyzing a multiprocessor system using other methods. The system level performance analysis is based on formal analysis techniques using Real Time Calculus [TCN02].

Polis [BCG+97] is considered as a pioneer method for platform-dependent design as it set the separation of concern principle for architecture and function, communication, and computation. Polis supports one model of computation described in finite-state machines (FSM). It is focused on automotive application domain supporting architecture based on a single microprocessor and peripherals. The supported tools were simulation, architectural exploration with accurate and fast code execution time evaluation using automatically generated from the FSMs model. However, Polis is considered limited both the model of computation and in the target architecture.

Metropolis [BLP+02, BWH+03] is a framework which follows the mapping of function to architecture paradigm in the Y-chart organization. Architectures are represented as computation and communication services. The association of functionality to architectural services allow Metropolis to evaluate characteristics (such as latency, throughput, power, and energy) of an implementation of a particular functionality with a particular platform instance. Metropolis has back-end tool connection with a SystemC simulator and to verification of LTL and LOC constraints. MetroII [Aea07] is a framework extending Metropolis to import heterogeneous IPs, to facilitates performance analysis and design space exploration.

Artemis workbench [Pim08, PHL+01] begins with a Simulink representation of the functionality of the design that is converted in Kahn Process Networks (KPN). KPNs are also used to capture the architecture based on a set of virtual processors. Eventually, Artemis generates VHDL to obtain an FPGA implementation.

Ptolemy [EJL+03, Lee09] develops another approach analyzing the behavior of interfaces to find whether two models can be composed in a semantic framework that support this form of heterogeneity. In fact, interfaces are finite-state machines used for hetero-

geneous modeling. Ptolemy uses Java as an imperative language and is more embedded software oriented rather than dealing with hardware architectures.

CoFluent Studio [CoF] is developed for system architecting. It supports the MCSE methodology (Méthodologie de Conception des Systémes Electroniques). It follows the mapping of function to architecture in the Y-chart paradigm. The approach is dual: one for the software developer to satisfy the functional specifications and one for the architecture designer. Simulation and platform prototyping is supported.

Octopus [Tea10] is specifically designed to support Design-Space Exploration (DSE). It allows the independent specification of applications, platforms, and mappings, following the Y-chart approach, and aims to integrate existing formal analysis and simulation tools in the DSE process.

SystemCoDesigner [HSKM08] is tool for Design Space Exploration (DSE) and prototyping. SystemCoDesigner starts from a behavioral SystemC model and generates hardware accelerators and hardware/software solutions for DSE. It also provides the capability for prototyping on an FPGA basis constructing a link between Electronic System Level (ESL) and Register Transfer Level (RTL).

LusSy [MMMcMc] is a tool for the analysis of SystemC transactional models. Starting from the source code of a SystemC design, it uses GCCs C++ front-end and the SystemC library itself for parsing, then transform it into a set of automata, and finally dump it in the Lustre language. Thus, it provides a way to express safety properties directly in SystemC, by using identifying operational semantics for TLM models written in full SystemC. Although the method has a working connection to verification tools, the semantics remain abstract because Lustre is less expressive than a general-purpose language which deals with dynamic data structures.

VISTA [MGN03], based on SystemC [Gro02], provides a methodology and tool for modeling SoC virtual platform for SW development and system level performance analysis and exploration. The SoC provides a cycle-accurate functional model of the architecture using the basic SystemC Transaction Level Modeling (TLM) components provided by VISTA. It supports cross-compilation on the target processor and back annotation, therefore bypassing the use of an Instruction Set Simulator (ISS). However, it may hardly be used for other purposes than performance analysis due to the lack of a formal specification of the system.

A simulation based method is presented in [KDVvdW97]. The authors specify software applications as Kahn graphs and they textually instantiate different architectures from an architecture template. They obtain performance numbers by using a configurable simulator in C++ that has been constructed for the architecture template, using multithreading and object oriented programming techniques. Even though the model is at a high level of abstraction, the simulator can efficiently execute different types of dataflow architectures at a level that is clock-cycle accurate.

The Sesame [EPTP07] modeling and simulation environment facilitates performance analysis of embedded media systems architectures according to the Y-chart design principle [BCG+97, KDVvdW97]. Sesame aims at early system evaluation and design space exploration using model calibration and trace-driven cosimulation.

Daedalus [NTS+08] offers a fully integrated tool-flow in which design space exploration (DSE), system-level synthesis, application mapping, and system prototyping of Multi-Processor System on Chips (MP-SoCs) are highly automated. It is based on Sesame framework [EPTP07] and automatically extends it to VHDL implementations of the MP-SoC platform architecture generated by the ESPAM tool [NSD08, NSD06]. The Daedalus high-level MP-SoC models aim at the accurate prediction of the overall system perfor-

mance.

SPADE [LSvdWD01] is a method and tool for architecture exploration of heterogeneous signal processing systems used to evaluate alternative multi-processor architectures. It follows the Y-chart paradigm for system level design, on which the application and architecture are modeled separately and mapped onto each other in an explicit design step. SPADE uses a trace-driven simulation technique and permits architectures to be modeled at an abstract level using a library of generic building blocks.

SymTA/S [Hea05] is a system-level performance analysis approach based on formal scheduling analysis techniques, event streams [RE02, RZJE02] and symbolic simulation. The tool supports heterogeneous architectures and determines system-level performance data such as end-to-end latencies, bus and processor utilization, and worst-case scheduling scenarios.

In [AAM06] the authors suggest a method of timed automata, for solving optimal scheduling problems. They demonstrate that the timed-automata-based methods can be used to synthesize scheduling strategies for applications with uncertain task durations. In [SBM09] the authors developed a methodology for automatic abstraction of systems modeled by timed automata allowing them to analyze timed automata of greater size and complexity. In [CMLS11], trade-offs between communication cost and computational workloads are modeled addressing the problem of mapping applications to processors in a multicore environment.

A hybrid approach for system level performance evaluation of embedded systems that combines formal analysis methods with a simulation framework in presented in [KPBT06]. However, the current approach is limited to small systems.

1.3 Organization of the Document

This document is composed of four parts, the first presenting the context of the work (Chapter 1 and Chapter 2), the second describing the system designer method which is the thesis contribution (Chapters 3, 4, 5, 6, 7 and 8), the third presenting the results of implementation and experimentation (Chapters 9, 10 and 11) describing the tool developed and applying the methodology presented in the contribution, and the last part (Chapter 12) drawing the conclusion and perspectives. The details of all Chapters are as follows. Chapter 2 describes the BIP component-based framework which is the foundation of this work. Chapter 3 presents the BIP Language Factory and the construction of software model in BIP using well structured models. The synthesis of HW Platforms in BIP and the library of HW components is provided in Chapter 4. In Chapter 5 we analyze the method and the necessary transformations used to efficiently map an application software model in BIP with the HW Platform model. Chapter 6 presents the mixed software/hardware system in BIP integrating all the HW platform constraints. Chapter 7 describes the two methods of integration the run-time HW/SW constraints which are the execution times of every software process in isolation. The method used for performance analysis and the comparison with related work is found in Chapter 8. Chapter 9 described the whole tool-flow developed to automatically generate an accurate system model in BIP. Chapter 10 and Chapter 11 present the two case studies considered in this work which respectively concern two different HW platforms. Chapter 12 draws the conclusion of this work and the futures perspectives.

- Chapter 2 -

The BIP Framework

The BIP –Behaviour/Interaction/Priority– framework [BBS06] is aiming at design and analysis of complex, heterogeneous embedded applications. BIP is a highly expressive, component-based framework with rigorous semantical basis. It allows the construction of complex, hierarchically structured models from atomic components characterized by their behavior and their interfaces. Such components are transition systems enriched with data. Transitions are used to move from a source to a destination location. Each time a transition is taken, component data (variables) may be assigned new values, computed by user-defined functions (in C/C++). Atomic components are composed by layered application of interactions and priorities. Interactions express synchronization constraints and define the transfer of data between the interacting 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.

This chapter is structured as follows. The abstract model of BIP is described in Section 2.1 as an abstract formalization of the layers of *Behavior*, *Interactions* and *Priorities*. Section 2.2 describes the concrete model of BIP extended with data. We introduce the concepts of *Components* and *Connectors* to build system models and we define the operational semantics of all three layers (behavior, interaction, priorities). Section 2.3 describes the basic constructs of the BIP Language. Section 2.3 presents the BIP Tool-chain, the BIP execution engines and the Distributed Implementation. Conclusions are given in the last section.

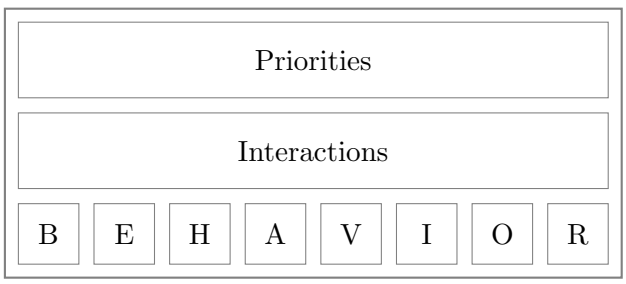


Figure 2.1: *Structure of a BIP Model*

2.1 Abstract Model of BIP

We provide a formalization of the BIP model focusing on the individual layers of behavior, interaction and priority glue (see Figure 2.1). In this section, we provide for each layer its abstract model.

2.1.1 Modeling Behavior

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

1 Definition (Behavior)
A behavior B is a labeled transition system represented by a triple $(Q, P, \rightarrow)$, where:

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

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

2.1.2 Modeling Interactions

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

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

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

2.1.3 Modeling Priorities

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

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

3 Definition (Priority)
A priority is a partial order $\prec \gamma \times \gamma$, where:

- *γ is the set of interactions,*

For $\alpha \in \gamma$ and $\alpha' \in \gamma$, the priority $(\alpha, \alpha') \in \prec$ is denoted as $\alpha \prec \alpha'$. That is, interaction α has less priority than α'.

2.1.4 Composition of Abstract models

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

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

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

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

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

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

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

2.2 Concrete Model of BIP

2.2.1 Atomic Components

In BIP, *atomic components* are automata equipped with a set of ports and a set of variables. Each transition is guarded by a predicate on the variables, triggers an update function, and is labelled by a port. Ports are used for communication among different components and are associated with variables of the component.

6 Definition (Port)
Each port is a pair (p, X_p), where p is the label and X_p is the set of variables associated with p. For the sake of simplicity we denote a port (p, X_p) by p. We refer to internal ports using the β character instead of p, which refers to communication ports.

7 Definition (Atomic Component: Syntax)
An atomic component is a labelled transition system extended with data $B = (L, X, P, \mathcal{T})$ where:

- *$L = \{\ell_1, \ell_2, ..., \ell_k\}$ is a set of control locations,*
- *$X = \{x_1, x_2, ..., x_n\}$ is a set of variables,*

- *P is a set of communication ports. Each port is a pair (p, X_p), where p is a label and $X_p \subseteq X$ is the set of variables associated with p. For the sake of simplicity we denote a port (p, X_p) by p and we refer to port p that belongs to component B by $B.p$,*
- *$\mathcal{T}$ is a set of transitions of the form $\tau = (\ell, p, g, f, \ell')$ or $(\ell, \beta, g, f, \ell')$, where $\ell, \ell' \in L$ are control locations, $p \in P$ is a communication port, β is an internal port, g is a guard, a predicate on X which can be true or false, and $f(X, X')$ is an update relation, a predicate on X (current) and X' (next) state variables. We represent concretely update relations as sequential programs operating on data X. We use the term skip to denote $f(X, X')$ as empty, where $X = X'$.*

Let $\mathcal{D}$ be a universal data domain. Given a set of variables X, we define valuations for X as functions $v : X \to \mathcal{D}$. The set of valuations is denoted as $\mathcal{D}^X$. Given two valuations $u : X \to \mathcal{D}$ and $v : Y \to \mathcal{D}$, we define the substitution $u \oslash v : X \cup Y \to \mathcal{D}$ as a valuation defined by:

$$(u \oslash v)(x) = \begin{cases} u(x) & :\text{if } x \in X \setminus Y \\ v(x) & :\text{if } x \in Y \end{cases}$$

8 Definition (Atomic Component: Semantics)
The semantics of $B = (L, P, X, \mathcal{T})$ is a transition system $(Q, \Sigma, \to_B)$ such that:

- *$Q = L \times \mathcal{D}^X$,*
- *$\Sigma = \{p[v''] | p \in P, v'' \in \mathcal{D}^{X_p}\} \cup \{\beta\}$ is the set of labels. A label $p[v'']$ marks instantaneous data change through the port p.*
- *$\to_B$ is the set including transitions*
  - *$((\ell, v), p[v''], (\ell', v'))$ such that $g(v) \wedge f(v \oslash v'', v')$ for some $\tau = (\ell, p, g, f, \ell') \in \mathcal{T}$. As usual, if $((\ell, v), p[v''], (\ell', v')) \in \to_B$ we write $(\ell, v) \xrightarrow{p[v'']}_B (\ell', v')$,*
  - *$((\ell, v), \beta, (\ell', v'))$ such that $g(v) \wedge f(v, v')$ for some $\tau = (\ell, \beta, g, f, \ell') \in \mathcal{T}$. As usual, if $((\ell, v), \beta, (\ell', v')) \in \to_B$ we write $(\ell, v) \xrightarrow{\beta}_B (\ell', v')$.*

For a model built from a set of n atomic components $\{B_i = (L_i, P_i, X_i, T_i)\}_{i=1}^n$,we assume that their respective sets of ports and variables are pairwise disjoint, i.e. for any two $i \neq j$ in $\{1...n\}$, we require that $P_i \cap P_j = \emptyset$ and $X_i \cap X_j = \emptyset$. Thus, we define the set $P = \bigcup_{i=1}^n P_i$ of all ports in the model as well as the set $X = \bigcup_{i=1}^n X_i$ of all variables.

1 Example
Figure 2.2 shows a graphical representation of two atomic components in BIP, the Sender and the Receiver. The behavior of Sender is described as a transition system with control locations ℓ_1 and ℓ_2. It communicates through ports tick and out. Port out exports the variable x. Initially, the Sender communicates through port out exporting the variable x. Then, it ticks through the tick port at a maximum number of ten times. Finally, the guard $[10 \leq c]$ enables the internal transition β. At the execution of the β transition the variable x is reevaluated depending on a user-defined function $f()$. Respectively, the behavior of Receiver has control locations ℓ_5 and ℓ_6 and communicates through port tick and port in, which exports the variable z. Initially, the Receiver ticks through the tick port at a maximum number of twenty times. Then, the guard $[c \leq 20]$ enables the communication port in which exports the variable z. Finally, the internal transition β is enabled returning

the component to the initial state. At the execution of the β transition the variable x is reevaluated depending on a user-defined function $f()$.

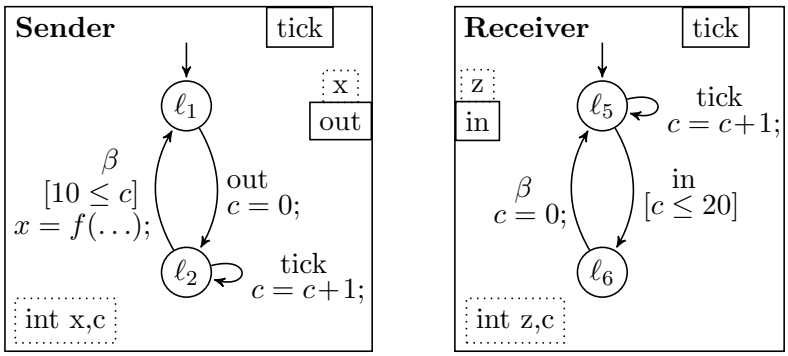


Figure 2.2: *Sender (left) and Receiver (right) BIP atomic components*

2.2.2 Interactions

9 Definition (Interaction)
An interaction α is a triple $(P_\alpha, g_\alpha, f_\alpha)$, where $P_\alpha \subseteq P$ is a set of ports, g_α is a guard, and f_α is a data transfer function. We restrict P_α so that it contains at most one port of each component, therefore we denote $P_\alpha = \{p_i\}_{i \in I}$ with $p_i \in P_i$ and $I \subseteq \{1...n\}$. g_α and f_α are defined on the variables available on the interacting ports $\bigcup_{p \in \alpha} X_p$.

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

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

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

In BIP, we distinguish two models of synchronization on connectors:

- Strong synchronization or rendezvous, where the only feasible interaction of γ is the maximal one, i.e., it contains all the ports of γ. We note $A_\gamma = P_\gamma$.
- Weak synchronization or broadcast, where all feasible interactions are those containing a particular port p_{trig} which initiates the broadcast. We note $A_\gamma = \{\alpha \in \gamma | \alpha \cap \{p_{trig}\} \neq \emptyset\}$ where $p_{trig} \in P_\gamma$ is the port that initiates the broadcast.

There is a graphical notation for interactions. In a rendezvous interaction all ports (known as synchrons) are denoted by bullets. In a broadcast interaction, the port that initiates the interaction, also called trigger, is denoted by a triangle and all the rest with bullets.

Hierarchical connectors We have seen that a connector has an option to define a port and export it. This allows a connector to be used as a port in other connectors, and create structured connectors. The representation of structured connectors require connectors to be treated as expressions with typing and other operations on groups of connectors. This led to a formalization of the algebra of connectors defined in [BS08a, BS08b]. The Algebra of Connectors is a compact notation for algebraic representation and manipulation of connectors and formalizes the concept of connectors supported by the BIP component model.

2.2.3 Priorities

11 Definition (Priority)
A priority is a tuple $(C, \prec)$ where C is a state predicate (boolean condition) characterizing the states where the priority applies and $\prec$ gives the priority order on the set of interactions A_γ.

For $\alpha_1 \in A_\gamma$ and $\alpha_2 \in A_\gamma$, a priority rule is textually expressed as $C \to \alpha_1 \prec \alpha_2$. When the state predicate C is true and both interactions α_1 and α_2 specified in the priority are enabled, the higher priority interaction, i.e., α_2 is selected for execution.

2.2.4 Composition of Components

12 Definition (Composite Component: Syntax)
A composite component is defined by a set of atomic components, composed by a set of interactions γ and a priority $\pi \subseteq \gamma \times \gamma$. We denote by $B \stackrel{def}{=} \pi\gamma(B_1, ..., B_n)$ the component obtained by composing components $B_1, ..., B_n$ using the interactions γ and priority π. If π is the empty relation, then we may omit π and simply write $\gamma(B_1, \cdots, B_n)$.

A state of $\pi\gamma(B_1, ..., B_n)$ is defined by a pair (ℓ, v), where $\ell = (\ell_1, ..., \ell_n)$ is the control state of each component and $v = (v_1, ..., v_n)$ is a valuation of component variables.

13 Definition (Composite Component: Semantics)
The behavior of a composite component $\pi\gamma(B_1, ..., B_n)$ without priority, where $B_i = (L_i, X_i, P_i, \mathcal{T}_i)$, is a labeled transition system $(Q, \gamma, \to_\gamma)$, where $Q = \bigotimes_{i=1}^n L_i \times \bigotimes_{i=1}^n \mathcal{D}^{X_i}$. We define $\to_\gamma$ the least transition relation containing the interleaving of internal β-transitions from components B_i and moreover, satisfying the interaction rule:

$$\frac{\alpha = (\{p_i\}_{i\in I}, g_\alpha, f_\alpha) \in \gamma \qquad g_\alpha(\{v_i\}_{i\in I}) \qquad \{v_i''\}_{i\in I} = f_\alpha(\{v_i\}_{i\in I}) \qquad \forall i \in I(\ell_i, v_i) \xrightarrow{p_i[v_i'']}_{B_i} (\ell_i', v_i') \qquad \forall i \notin I.\ (\ell_i, v_i) = (\ell_i', v_i')}{((\ell_1, \ldots, \ell_n), (v_1, \ldots, v_n)) \xrightarrow{\alpha}_\gamma ((\ell_1', \ldots, \ell_n'), (v_1', \ldots, v_n'))}$$

Intuitively, this inference rule specifies that a composite component $B = \gamma(B_1, ..., B_n)$ can execute an interaction $\alpha \in \gamma$, iff(1) for each port $p_i \in P_\alpha$, the corresponding atomic component B_i allows a transition from the current state labelled by p_i (i.e. the corresponding guard g_i evaluates to true), and (2) the guard g_α of the interaction evaluates to true. If these conditions hold for an interaction α at state (ℓ, v), α is *enabled* at that state. Execution of α modifies participating components' variables by first applying data transfer function f_α on variables of all interacting components and then update functions inside each interacting component. The (local) states of components that do not participate in the interaction remain unchanged. In order to comply with the trace equivalence terminology we can simply refer to interactions as actions.

We define the behavior of the composite component $B = \pi\gamma(B_1, \ldots, B_n)$ with priority, as the labeled transition system $(Q, \gamma, \rightarrow_{\pi\gamma})$ where $\rightarrow_{\pi\gamma}$ is the least set of transitions satisfying the rule:

$$\frac{(\ell, v) \xrightarrow{\alpha}_{\gamma} (\ell', v') \qquad \forall \alpha' \in \gamma.\ \alpha \prec \alpha' \qquad (\ell, v) \not\xrightarrow{\alpha'}_{\gamma}}{(\ell, v) \xrightarrow{\alpha}_{\pi\gamma} (\ell', v')}$$

The inference rule filters out interactions which are not maximal with respect to the priority order. An interaction is executed only if no other one with higher priority is enabled.

2 Example
Figure 2.3 shows a graphical representation of a composite component in BIP. It consists of three atomic components, the Sender, the Buffer and the Receiver. The behavior of Sender and Receiver is already introduced in Example 1. The behavior of Buffer has control locations ℓ_3 and ℓ_4. It communicates through ports tick, in and out. Ports in, out export the variable y. The Buffer ticks through the tick port to synchronize with the Sender and the Receiver. It communicates through port in which modifies the variable y interacting with Sender via connector io1. Then, the Buffer forwards the same variable to Receiver through port and connector io2. The hierarchical connector tick2 which includes connector tick1, synchronizes all the available components. The tick2 connector has also the least priority among the connectors of the composite component.

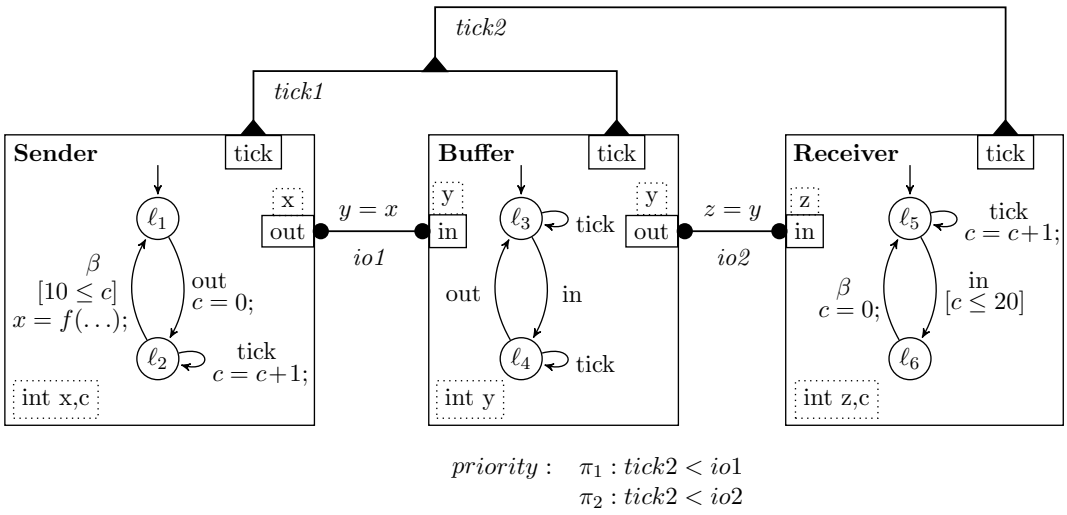


Figure 2.3: *Sender/Buffer/Receiver model as a composition of BIP atomic components*

14 Definition (Transition Sequence)
We define:

- *A **run** θ is a finite sequence of transitions:*

$$q_0 \xrightarrow{a_1}_{\gamma} q_1 \xrightarrow{a_2}_{\gamma} q_2 \xrightarrow{a_3}_{\gamma} \ldots \xrightarrow{a_n}_{\gamma} q_n \text{ , with } a_i \in \gamma \cup \{\beta\}$$

  *We say that for each **run** θ*

- *Runs(C) is the set of all runs observed on a composition model in BIP, such as $C = \pi\gamma(B_1, \ldots, B_N)$.*

- *$Runs(q_0)$ is the set of all runs started from q_0.*
- *A run is maximal if q_n has no successor.*
- **trace**(θ) *is the sequence of labels $a_1.a_2 \ldots a_n$ for a given run θ.*
- *$Traces(C)$ is the set of all traces which correspond to the set of $Runs(C)$, for a given composition model in BIP, such as $C = \pi\gamma(B_1, \ldots, B_N)$.*

2.3 The BIP Language

The BIP language represents components of the BIP framework [BBS06]. BIP language is a user-friendly textual language which provides syntactic constructs for describing systems. It leverages on C style variables and data type declarations, expressions and statements, and provides additional 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: to specify behavior, with an interface consisting of ports. Behavior is described as a set of transitions.
- connector: to specify the coordination between the ports of components, and the associated guarded actions.
- priority: to restrict the possible interactions, based on conditions depending on the state of the integrated components.
- compound: to specify systems hierarchically, from other atoms or compounds, with connectors and priorities.
- model: to specify the entire system, encapsulating the definition of the components, and specify the top level instance of the system.

3 Example
The BIP descriptions of the Sender atomic component of Figure 2.2(left) and Port types used are illustrated below:

model *S_R_Buffer*

port type *DataPort (***int** *i)*
port type *EventPort*
port type *InternalPort*

atomic type *Sender*

> **export port** *EventPort tick=tick*
> **export port** *DataPort out(x)=out*
> **port** *InternalPort* β
>
> **place** ℓ_1,ℓ_2
> **initial to** ℓ_1 **do** {}

on *out* **from** ℓ_1 **to** ℓ_2
do $\{c = 0;\}$

on *tick* **from** ℓ_2 **to** ℓ_2
do $\{c = c + 1;\}$

on τ **from** ℓ_2 **to** ℓ_1 (**provided** $10 \leq c$)
do $\{x = f();\}$

end

Three types of ports are defined: `DataPort`, `EventPort` *and* `InternalPort`. *A port type* `DataPort` *associates a port to an integer variable i. Variables associated to ports may be modified when executing the interaction in which the port participates. The port* `out` *is an instance of the type DataPort. A port type* `EventPort` *is an event port and it is not associated with any variable. The port* `tick` *is an instance of the type* `EventPort`. *All ports are exported at the interface of the component. Initially, the state of the component is at the place* $\ell 1$, *the only place with token. The BIP code uses the constructs "on...from...to"to represent transitions from one place to the other. The construct "provided"is used when the execution of a transition is restricted by a guard. Moreover, if the transition is associated with a function, the C code inside the constructs "do {...}"is executed .*

Components are composed by using connectors. A connector defines the set of possible interactions between ports of components and the corresponding data transfer between the variables associated with the ports. The BIP language allows the definition of connector types.

4 Example
Below is presented the syntax of two different types of connectors, `RendezVous- Data` *and* `BroadcastEvents` *connector.*

connector *type RendezVousData(DataPort out, DataPort in)*
define *out in*
on *out in*
up *;*
down *in.i=out.i;*
end

connector *type BroadcastEvents(EventPort e1, EventPort e2)*
define *e1' e2'*
on *e1*
on *e2*
on *e1 e2*
export *port EventPort e*
end

The `RendezVousData` *connector defines a strong synchronization between two ports of type* `DataPort`, *in and out. The value i is copied from the port in to the port out each time*

the connector is executed. The `BroadcastEvent` *connector defines a weak synchronization between the ports* $e1$ *and* $e2$ *of* `EventPort` *type. At least one of the ports is required to initiate the interaction. This interaction is exported to the environment through the* `EventPort` e.

A compound component is a new component type defined from existing components by creating their instances, instantiating connectors between them and specifying the priorities. A compound offers the same interface as an atom, hence externally there is no difference between a compound and an atomic component.

5 Example
The BIP description of the Send/Buffer/Receiver compound component of Figure 2.3 is given below.

compound type *Compound_S_R_Buffer*

component *Sender sender*
component *Receiver receiver*
component *Buffer buffer*

connector *RendezVousData io1 (sender.out, buffer.in)*
connector *RendezVousData io2 (buffer.out, receiver.in)*

connector *BroadcastEvent tick1 (sender.tick, buffer.tick)*
connector *BroadcastEvent tick2 (tick1.e, receiver.tick)*

priority π_1 **if***(true) tick2<io1*
priority π_2 **if***(true) tick2<io2*

end

The three atomic components that constitute the Send/Buffer/Receiver model are instantiated. For example **component** `Sender sender`*, creates an instance of* `Sender` *component named* `sender`*. Connectors are also instantiated, associating the ports of instantiated components through the interactions defined by the connector type. Finally, priorities are defined specifying an order between a pair of interactions.*

2.4 The BIP Tool-Chain

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

The overview of the BIP tool-chain is shown in Figure 2.4. It includes the following tools:

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

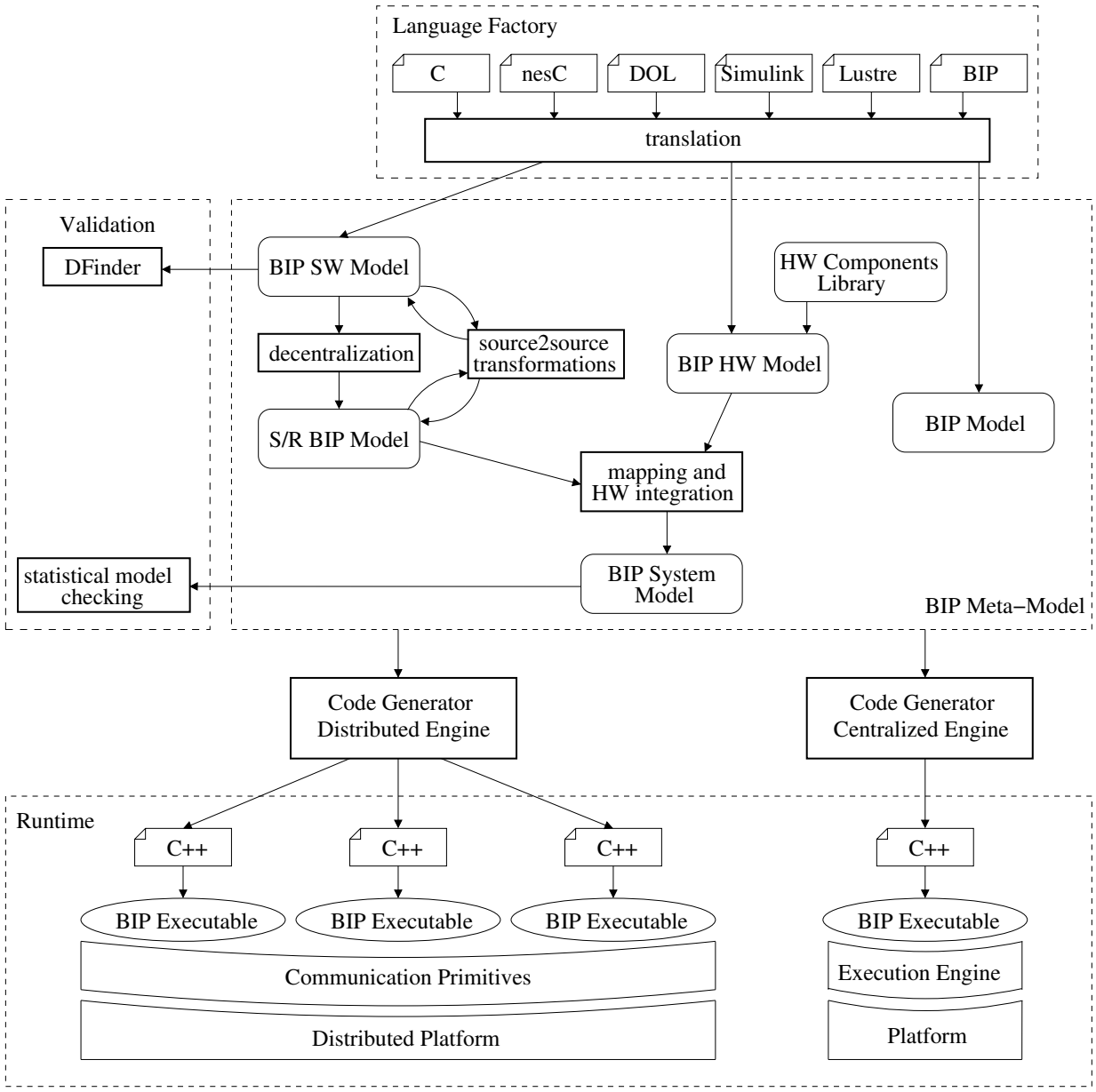


Figure 2.4: *The BIP Tool-Chain.*

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

- *The compiler.* It generates a BIP model from the BIP description source. It uses The BIP meta-model as the intermediate representation of BIP models and to implement model transformations. It includes :

  - *The BIP meta-model.* It represents a template of the structure of the intermediate model to be generated from a BIP program, using EMF. All the modeling elements, presented in the BIP language, have a representation in the BIP model in the form of the data-structure. Class diagrams are used to define the relations between the different modeling elements, through inheritance and containment.
  - *The parser.* It analyzes a BIP description source and generates an intermediate model conforming to the BIP meta-model. It performs syntactic analysis of the input program conforming to the BIP grammar and reports the programming errors.
  - *Model-to-model transformation tools.* They are used in order to perform useful static transformations for systems optimizations including run-time. The transformations use a set of correct-by-construction models and preserve functional properties. Moreover, they can take into account extra functional constraints. There exist three types of transformations, architecture optimizations, such as flattening the hierarchy and transforming structured connectors to flat connectors [BJS09], distributed implementation [BBJ$^+$10], such as the replacement of atomic multiparty interactions by protocols using asynchronous message passing (send/receive primitives) and memory management.
  - *The code generator.* It generates C++ code from the model produced by the parser. The code generator has options for generating application code for the single-threaded BIP Engine, the multi-threaded BIP Engine and the distributed BIP implementation.

- *D-Finder.* It is a compositional verification tool for deadlock detection and generation of invariants [BBN$^+$09, BGL$^+$11]. Verification is applied only to high level models for checking safety properties such as invariants and deadlock-freedom.

- *The BIP Execution engines.* They are middleware responsible for the coordination of atomic components, that is, they apply the semantics of the interaction and priority layers of BIP. Execution engines are used for execution, simulation, run-time verifications, debug or state-space exploration(i.e. all traces) of BIP models. There are currently three engines available, the single-threaded engine, the multi-threaded engine and the engine supporting the distributed implementation of BIP.

2.4.1 The BIP Execution Engines

The BIP execution Engines and the distributed BIP implementations directly implement the BIP operational semantics. It plays the role of the coordinator in selecting and executing interactions between the components, taking into account the glue specified in the

input component model. It monitors the state of the components and considering the interaction model, finds all the enabled interactions. It then applies the priority rules to eliminate the interactions with low priority, and selects one amongst the maximal enabled, for execution.

Here is the presentation of the current Engines.

The Single-Threaded BIP Engine

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

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

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

Algorithm 1 Single Threaded Execution Engine

Require: Model $M^i = (Q_i, \rightarrow_i), 1 \leq i \leq n$, initial control location $(q_0^1, \ldots, q_0^n)$, set of interactions γ
  $s = (q^1, \ldots, q^n) \leftarrow (q_0^1, \ldots, q_0^n)$
  loop
    $\gamma_s = EnabledInteractions(s)$
    if $\exists \alpha \in \gamma_s$ **then**
      $\alpha = \{\alpha_i | i \in I\} \leftarrow EDFScheduler(\gamma_s)$
      $ExecuteDataTransfer(F_\alpha)$
      for all $i \in I$ **do**
        $Execute(\alpha_i)$
        $q_i \leftarrow q_i'$
      end for
    else
      exit(DEADLOCK)
    end if
  end loop

The Multi-Threaded BIP Engine

The implementation of the multi-threaded implementation with centralized engine is based on the notion of partial state semantics where interactions are allowed to fire as soon as only the involved components are stable [BBB+08]. Each atomic component is assigned to a different thread (process), the engine being assigned to a thread as well. Each atomic

component performs its computations locally and then, when it reaches a stable state, it notifies the engine about the ports on which it is willing to interact. It waits for the engine to select the port to be executed upon the chosen interaction.

The Engine is parametrized by an oracle. As depicted in Algorithm 2, the engine computes feasible interactions available on state components. Then, if such interactions exist and the oracle allows them, the engine selects one for execution and notifies the involved components.

Iteratively, the Engine receives the sets of ports and the local states of components ready to interact. Depending on this information, the engine computes the feasible interactions. It chooses a feasible interaction, which is allowed by the oracle O. If such an interaction exists, the engine executes it by notifying sequentially, in some arbitrary order, all the involved components. Otherwise, it is a deadlock.

Algorithm 2 Multi-Threaded Execution Engine

Require: Model $M^i = (Q_i, \rightarrow_i), 1 \leq i \leq n$, initial control location $(q_0^1, \ldots, q_0^n)$, set of interactions γ
  $s = (q^1, \ldots, q^n) \leftarrow (q_0^1, \ldots, q_0^n)$
  loop
    wait(P_i)
    $\gamma_s = EnabledInteractions(P_i)$
    $\gamma_o = restriction(\gamma_s, O)$
    if $\exists \alpha \in \gamma_o$ **then**
      $\alpha = \{\alpha_i | i \in I\} \leftarrow EDFScheduler(\gamma_s)$
      $ExecuteDataTransfer(F_\alpha)$
      for all $i \in I$ **do**
        $notify(M_i, \alpha_i)$
        $q_i \leftarrow q_i'$
      end for
    else
      exit(DEADLOCK)
    end if
  end loop

2.4.2 The Distributed BIP Implementation

Currently, powerful hardware platforms are needed for executing applications on multicore or many-core platforms. The application code should be optimally distributed over the platform to take advantage of its computing power. Although distributed systems are widely used nowadays, their implementation is still time-consuming and an error-prone task. The distributed implementation in BIP provides a method for automatic generation of efficient and correct-by-construction distributed model from a given application software in BIP. Coordination in BIP is achieved through multi-party interactions (i.e., those across multiple components), and scheduling by using dynamic priorities. Transforming the semantics of BIP, which is based on a global state model, into a distributed implementation is clearly a non-trivial task.

A generic framework allowing the transformation of high-level BIP models into distributed implementations has been recently developed [BBJ+10]. The method involves BIP to BIP transformations preserving observational equivalence. It transforms multiparty interactions into asynchronous message passing, that is, send/receive primitives.

The target Send/Receive BIP model is structured in three layers (see Figure 2.5): (i) the component layer corresponds to a modified behavior of the components of the original model; (ii) the interaction protocol consists of a set of components such that each component detects enableness of a subset of interactions of the original model using partial-state knowledge, and executes them after resolving conflicts (e.g., regarding which interaction to execute when there is more than one involving the same port) either locally or by the help of the third layer; (iii) the reservation protocol resolves conflicts between components of the interaction protocol layer using committee coordination algorithms such as the token-ring distributed algorithm or the distributed dining philosophers algorithm. Notice that the obtained Send/Receive BIP model depends on a user-defined partition of the interactions of the original model, associating subsets of interactions to components of the interaction protocol layer.

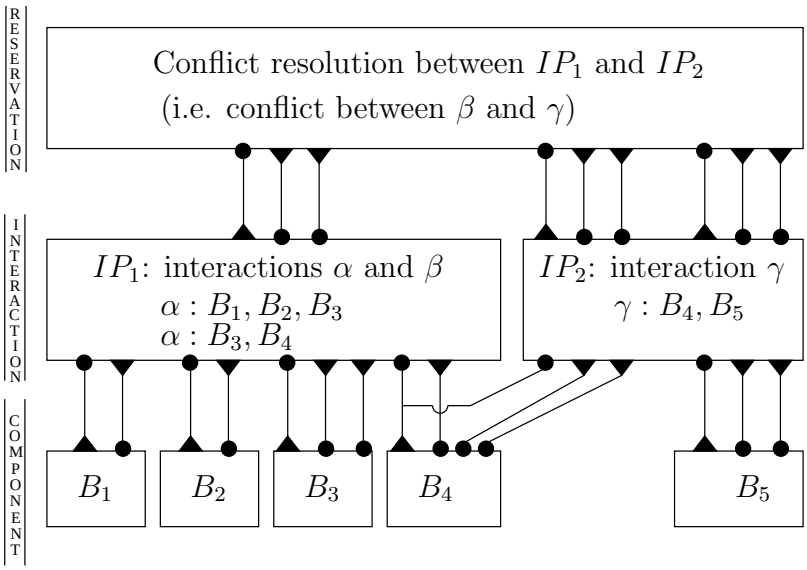


Figure 2.5: *Send/Receive BIP model obtained from BIP to BIP transformations.*

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

The method has been fully implemented in a toolset allowing the automatic generation of distributed implementations from BIP models. It is parametrized by the partitioning of interactions, a committee coordination algorithm, and the mapping of components.

2.5 Conclusion

BIP [BBS06] (Behavior, Interaction, Priority) is a general framework encompassing rigorous design. It uses the BIP language and an associated toolset supporting the design flow. The BIP language is a notation which allows building complex systems by coordinating the behavior of a set of atomic components. Behavior is described as a finite-state automaton extended with data and functions described in C/C++. The transitions of the automata are labeled with guards (conditions on the state of a component and its environment) as

well as functions that describe computations on local data. The description of coordination between components is layered. The first layer describes the interactions between components. The second layer describes dynamic priorities between the interactions and is used to express scheduling policies. The combination of interactions and priorities characterizes the overall architecture of a component. It confers BIP strong expressiveness that cannot be matched by other languages [BS08b]. BIP has clean operational semantics that describe the behavior of a composite component as the composition of the behaviors of its atomic components. This allows a direct relation between the underlying semantic model (transition systems) and its implementation.

The BIP design flow uses a single language to ensure consistency between the different design steps. This is mainly achieved by applying source-to-source transformations between refined system models. These transformations are proven correct-by-construction, that means, they preserve observational equivalence and consequently essential safety properties. Functional verification is applied only to high level models for checking safety properties such as invariants and deadlock-freedom. To avoid inherent complexity limitations, the verification method applies compositionality techniques implemented in the D-Finder tool. BIP has been successfully used to model complex systems and software applications like the DALA robot [dal], the Heterogeneous Communication System (HCS) [BBB$^+$12], the NesC/TinyOS applications [BMP$^+$07] and others which we refer in the next chapter. In the next chapters we analyze the method of designing complex mixed hardware/software systems using the BIP component-based framework.

Part

System Designer

- Chapter 3 -

BIP Language Factory

In this chapter, we present the methods for generating BIP models out of languages and various programming models. The ensemble of these methods consists the BIP Language Factory. Most importantly, we describe the generation of Kahn Process Network (KPN) software models in BIP. These models are used to model the software application part of our system design.

The chapter structure is as follows. In Section 3.1, we epigrammatically present the methods developed in Verimag which generate BIP models using different programming models. In Section 3.2, we describe the KPN models using BIP. In the final Section 3.3, we provide the method that we used to automatically generate KPN models in BIP and then, we conclude the chapter.

3.1 Construction of Software Models

A general method for generating BIP models from languages is developed in Verimag laboratory. The method is illustrated in Figure 3.1. The BIP semantic model is used to structurally represent different programming models or domain specific languages and enable the available analysis and verification techniques provided by the BIP toolchain [bip]. In this section, we provide an overview of the set of languages and programming models translated into BIP models and analyzed by the BIP toolchain.

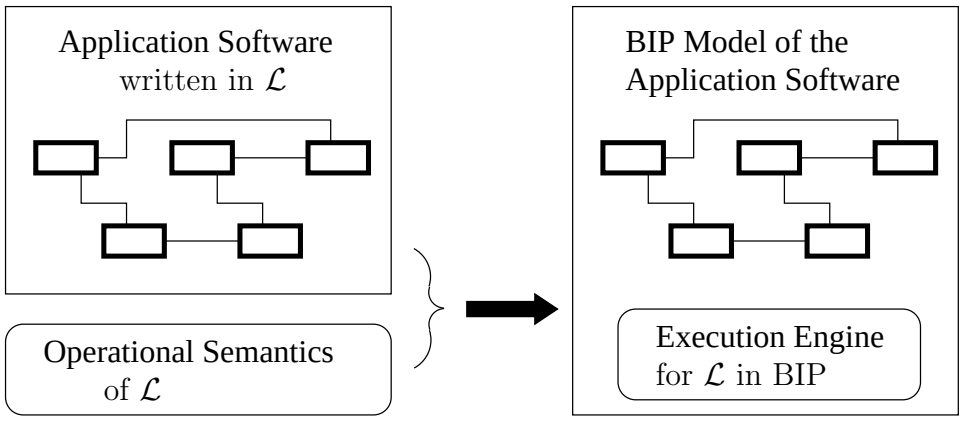


Figure 3.1: *Translation method for a language in BIP*

From AADL to BIP AADL (Architecture Analysis and Design Language) [SAE09] is a language dedicated to modeling and specification of complex Real-time embedded systems. It is used to describe component-based systems, where each component represents the physical hardware or the application software. There are several types to describe, on the one hand, the physical hardware such as processors, memories, buses and devices, and on the other, the application software, such as processes, threads, data and functions. In [CRBS08] the authors provide a translation from AADL to BIP. The BIP framework provides a series of advantages compared to AADL. Such as concrete operational semantics, an execution environment and formal verification techniques.

From Lustre to BIP Lustre [HCRP91] is a dataflow language for programming synchronous reactive systems. In [BSS09] the authors present a modular translation of Lustre into modal flow graphs described in synchronous BIP. The modal flow graphs are acyclic graphs representing three different types of dependency between two events p and q: strong dependency (p must follow q), weak dependency (p may follow q), conditional dependency (if both p and q occur then p must follow q). Synchronous BIP is a subset of BIP which describes systems of components which are strongly synchronized by a common action that triggers the execution steps. The advantage is that the translation is modular and exhibits not only data-flow connections between nodes but also their synchronization by using clocks.

From MATLAB/Simulink to BIP MATLAB/Simulink [Mat] is a simulation environment developed by Mathworks for analysis and model-based design of dynamic and embedded systems. In [STS+10] a discrete-time fragment of Simulink is used to develop a method for translation into synchronous BIP. There are several advantages, concerning both MATLAB/Simulink to BIP and Lustre to BIP translations, related to the obtained modal flow graphs in BIP. They are well-triggered, a property of modal flow graphs that expresses consistency between the three types of dependency. It guarantees deadlock freedom and deterministic behavior under some conditions of non interference of concurrent computations. The translations of these two synchronous formalisms open the way for exploring problems regarding relations between synchronous and asynchronous systems. They allow integration of synchronous systems theory in an all encompassing component framework without losing advantages such as correctness-by-construction and efficient code generation. This makes possible modeling mixed synchronous/asynchronous systems without artifacts.

From LAAS/GeNom applications to BIP LAAS [ACF+98] is a framework used to describe both the functional and the execution control level of a robot. The LAAS framework is based on the componentization of GenoM [FHC97] Functional Modules. Each module can integrate synchronous and asynchronous processes. It has a predictable behavior and standard communication interfaces. A module description language is associated with an automatic module generator according to GeNom generic model. In [BGL+08], the modeling of the functional part of a robotic system into a BIP model and the synthesis of an execution controller are presented. Both the functional and the execution control level of the robot are described with the LAAS [ACF+98] framework and the GenoM [FHC97] Functional Modules. The goal of the above construction methodology is the verification and validation of essential "safety" properties.

From NesC/TinyOS applications to BIP TinyOS [Tin] is an embedded operating system and platform written in the NesC programming language, which is a subset of the C language optimized for sensor networks with strict memory limits. It is designed as a set of cooperating tasks and processes and it targets low-power wireless devices. In [BMP+07], the authors present a methodology for construction, analysis and verification of network system models in BIP using the TinyOS operating system. The corresponding BIP model is constructed based on the modeling of a NesC program describing the application and based on models of TinyOS components. Different types of BIP connectors are used to model the composition of components into network models. The use of BIP opens the way for enhanced analysis and early error detection by using verifications techniques.

From C language to BIP A translation process from C code to a BIP model is developed. The translator currently supports the C language. Any C function can be translated into a BIP atomic component. The BIP component that models the function call uses a BIP port and a BIP connector that interacts with the BIP component modeling the body of the invoked function. This translation is used as a building block for the Kahn Process Network to BIP model language factory described in the following sections.

3.2 From Kahn Process Networks to BIP

A Kahn Process Network (KPN) model [Kah74] is a set of autonomous processes that communicate through unidirectional software channels. The channels are first-in first-out (fifo) queues with blocking read and non-blocking write operations. The read operation is blocking since the process suspends if the fifo queue is empty. Assuming that the queue has an infinite size, the write operations are non-blocking. The communication through the channels must occur in a finite and unspecified amount of time. A Kahn Process Network program is deterministic; the result of a computation is independent from execution order. Sequential or parallel executions produce the same outcome. Determinism separates the functionality of the application from the target hardware platform. Moreover, a KPN model provides separate analysis of computation and communication and exposes functional parallelism. More restrictive models can be derived from KPNs such as Synchronous Data Flows (SDF) [LM87] [LP95], Marked Directed Graphs [CHEP71] and SW/HW Integration Medium (SHIM) [ET05].

In the next section, we present a construction of Kahn Process Networks using BIP models. We consider KPNs with bounded size queues where both read and write operations are blocking. The derived BIP model consists of process components and FIFO channel components connected via send/receive data connectors.

3.2.1 BIP Process Component

A process component in BIP models the behavior of a KPN process. It contains *Read* and *Write* communication primitives modeled as interactions with external components. A *Read* operation retrieves the top most value stored in a the queue. The *Write* operation stores a value in the next available cell of the queue. A process component in BIP uses a set of ports P_w for *Write*, which are associated with the variable *wr_data* and a set of ports P_r for *Read* associated the the variable *rd_data*. A generic process component is defined below.

15 Definition (Process Component)
We define the process component $G = (L, X, P, \mathcal{T})$, with Read and Write communication primitives where:

- $L = \{\ell_{ini}\} \cup \{\ell_1, \ell_2, ..., \ell_k\} \cup L_{fin}$ *is the set of control locations, where* ℓ_{ini} *is the initial control location,* $\{\ell_1, \ell_2, ..., \ell_k\}$ *are the intermediate control locations and* L_{fin} *the set of possible final control locations such that:*

$$L_{fin} = \begin{cases} \emptyset & \text{:if process never terminates} \\ \{\ell_{k+1}, \ell_{k+2}, ..., \ell_{k+m}\} & \text{:if process terminates} \end{cases}$$

- $X = \{wr\_data, rd\_data\} \cup \{x_1, x_2, ..., x_n\}$ *is a set of variables,*
- $P = P_w \cup P_r$, *where* P_w *the set of ports used for Write and* P_r *the set of ports used for Read respectively, for each* $w \in P_w$, w *are associated with* $\{wr\_data\}$ *and for each* $r \in P_r$, r *are associated with* $\{rd\_data\}$
- $\mathcal{T}$ *is the set of transitions of the form* $\tau = (\ell, w, true, skip, \ell')$, $\tau = (\ell, r, true, skip, \ell')$ *or* $(\ell, \beta, g, f, \ell')$,

6 Example (Producer Component)
We assume a Producer Component that generates an integer value and sends it to an other component. We define the BIP Producer Component $G_P = (L_{G_P}, X_{G_P}, P_{G_P}, \mathcal{T}_{G_P})$, *with one Write communication primitive. There are two control locations* ℓ_1,ℓ_2, *the wr_data variable for the integer value, a Write port w associated with wr_data and two transitions,* w *and* β. *On transition w, the Producer exports the wr_data variable and on transition* β *it re-evaluates the wr_data variable. Transition* β *is controlled by the guard* $g : [wr\_data < 2]$. *The Producer Component is illustrated in Figure 3.2.*

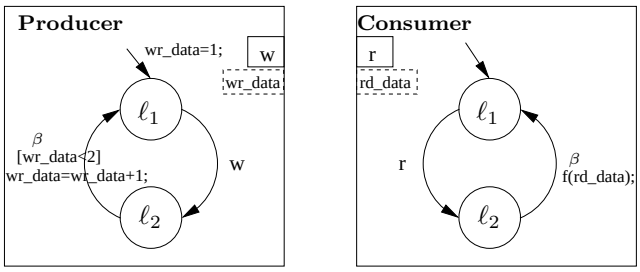


Figure 3.2: *Models of the Producer and Consumer Components in BIP*

7 Example (Consumer Component)
We assume a Consumer Component that receives and prints an integer. We define the Consumer Component $G_C = (L_{G_C}, X_{G_C}, P_{G_C}, \mathcal{T}_{G_C})$, *with one Read communication primitive as it is illustrated in Figure 3.2. There are two control locations* ℓ_1,ℓ_2, *the rd_data variable to store an integer value, a Read port r associated with rd_data and two transitions,* r *and* β. *On transition r, the Consumer exports the rd_data variable and on transition* β *it reads the rd_data variable.*

3.2.2 BIP FIFO Component

A FIFO channel in KPN is characterized by its size k, that is the maximal number of values that can be stored. It has ports w (write) and r (read), and a single control location ℓ.

The component contains an array of values $buff$ parametrized by k. The variable wr_data is associated with the port w and the rd_data with r. On w, the received value is inserted into $buff$. On r, the least recent value is sent and removed from the buffer. The variable count records the number of values stored in the buffer. It is increased on a w (write) and it is decreased on a r (read), respectively. The FIFO policy is implemented by using two indices i and j, for respectively insertion/deletion into/from the (circular) buffer $buff$. A formal definition is given below.

16 Definition (FIFO Component)
We define the FIFO atomic component $F = (L_F, X_F, P_F, \mathcal{T}_F)$, *where the behavior is described in Figure 3.3.*

Finally, we have:

- $L_F = \{\ell\}$,
- $X_F = \{wr\_data, rd\_data, buff, i, j, k, count\}$,
- $P_F = \{w, r\}$, *where* w *and* r *associated with* $\{wr\_data\}$ *and* $\{rd\_data\}$ *respectively.*
- $\mathcal{T}_F = \{\tau_w = (\ell, w, g_w, f_w, \ell), \tau_r = (\ell, r, g_r, f_r, \ell)\}$

with $g_w : [count < k]$ *and* f_w *defined as a sequence of functions* $f_w = f_{wm}; f_{wn};$, *where:*

- $f_{wm} : buff[i] = wr\_data;$
- $f_{wn} : count = count + 1; i = (i + 1)\%k;$

and with $g_r : [count > 0]$ *and* f_r *defined as a sequence of functions* $f_r = f_{rm}; f_{rn}$ *where:*

- $f_{rm} : rd\_data = buff[j];$
- $f_{rn} : count = count - 1; j = (j + 1)\%k;$

A write operation always precedes the read operation on a given buffer place.

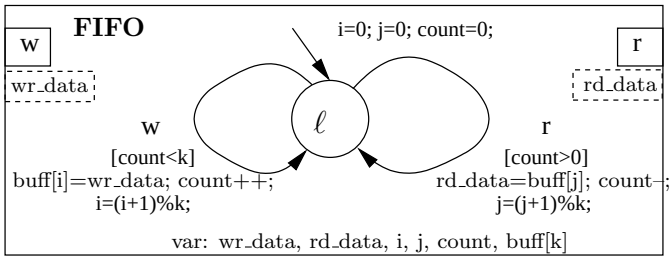


Figure 3.3: *Model of FIFO channel in BIP*

3.2.3 BIP KPN model

Using the Process Component and the FIFO Component in BIP, defined above, we describe below the composition of a Process Network in BIP modeling Kahn Process Network. We will also further refer to the Process Network in BIP as application software model in BIP.

17 Definition (Process Network Composition)
We define a Process Network in BIP as the composition N of Process Component G and FIFO Components F such that $N = \pi\gamma(G_1, \ldots, G_n, F_1, \ldots, F_k)$, where π is the priority rule applied in the interactions γ, where the set γ contains two categories of interactions such that $\gamma = (\alpha_r, \alpha_w)$:

- *read interactions α^r of the form:*
  $(\{F.r, G.r_k\}, true, f_r)$, with $r_k \in P_r(G)$, $f_r : F.rd\_data = G.rd\_data$;
- *write interactions α^w of the form:*
  $(\{F.w, G.w_k\}, true, f_w)$, with $w_k \in P_r(G)$, $f_w : G.wr\_data = F.wr\_data$;

Every read/write port in the FIFO/processes are used in only one interaction and moreover, every internal transition β has higher priority than any other interaction $\alpha \in \gamma$.

8 Example (Producer-Consumer Composition)
We construct a composition using the Producer $G_P = (L_{G_P}, P_{G_P}, X_{G_P}, \mathcal{T}_{G_P})$, the Consumer $G_C = (L_{G_C}, P_{G_C}, X_{G_C}, \mathcal{T}_{G_C})$ and the FIFO $F = (L_F, X_F, P_F, \mathcal{T}_F)$ Components defined earlier. We define the composition $PC = \gamma(G_P, F, G_C)$, as illustrated in Figure 3.4, where $\gamma = \{\alpha^w, \alpha^r\}$, $\alpha^w = (\{G_P.w, F.w\}, true, f_{\alpha^w})$, $\alpha^r = (\{F.r, G_C.r\}, true, f_{\alpha^r})$,
$f_{\alpha^w} : F.wr\_data = G_P.wr\_data$;,
$f_{\alpha^r} : F.rd\_data = G_P.rd\_data$;,

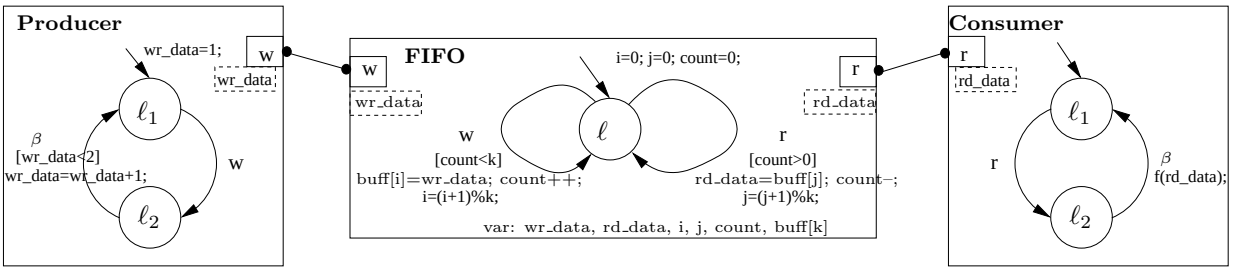
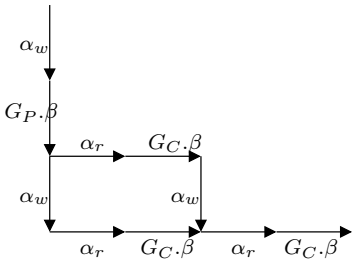


Figure 3.4: *Producer-Consumer Composition in BIP*

The set of traces as defined in Definition 14 is represented as the interleavings below:



3.3 Implementation using the DOL Framework

3.3.1 Distributed Operation Layer (DOL) Framework

DOL (Distributed Operation Layer) [TBHH07] is a framework devoted to the specification and analysis of mixed software/hardware systems. DOL provides languages for the representation of particular classes of applications software, multi-processor architectures and their mappings. In addition, DOL provides tools for performance analysis and

design-space exploration based on a combination of analytical and simulation-based techniques [TCN02, KPBT06]. In DOL, application software is defined using a variant of Kahn process network model [Kah74]. It consists of a set of deterministic, sequential processes communicating asynchronously through FIFO channels. The hardware architecture is described as interconnections of computational and communication devices such as processors, buses and memories. The mapping associates application software components to devices of the hardware architecture, that is, processes to processors and FIFO channels to memories.

Application Software in DOL The application software in DOL consists of three basic entities: *Processes*, *FIFO* channels, and *Connections*. The network structure is described in XML. Each *Process* has input, output ports and sequential behavior. Processes communicate by using *FIFO* channels. Each *FIFO* has a single input port and a single output port, uniquely associated with ports of processes.

9 Example

We present in Figure 3.5 the process network model derived from the right-looking variant of Cholesky factorization. The Cholesky application is an algorithm for solving numerically linear equations and it is thoroughly described later in Section 10.7. It contains processes Splitter, Joiner and three "block" computational processes P_{11}, P_{21} and P_{22}. Process Splitter splits the initial matrix into blocks and dispatches them to computational processes. Each process P_{ij} implements the computation required on a corresponding matrix block A_{ij}. The final matrix is re-constructed by process Joiner. Explicit communication between P_{ij} processes is used to enforce data dependencies. In this model, a dedicated FIFO (F) is used for every pair of dependent processes to transfer the result block from the source to the target process. In Figure 3.6, we present a fragment of the DOL specification of the Cholesky process network in XML. For each process, we specify the name of the process, the number of input and output ports, the names of the ports, the respective types and the location of the source C code describing the process behavior. For each software channel (F) we specify the name, the type the maximum capacity of data and the input and output port. Finally, we define the connections between the processes and the software channels by specifying the input and output ports which contribute in each connection.

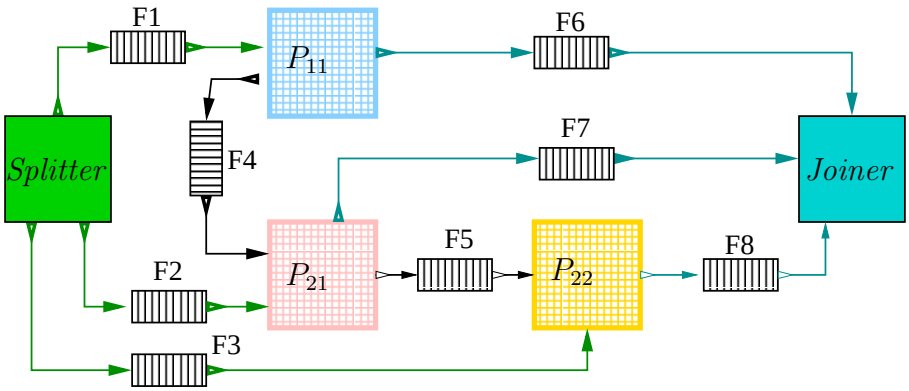


Figure 3.5: *Cholesky application in DOL*

Process behavior is described using sequential C programs with a particular structure (see Figure 3.7 for a concrete example). For a process P, its state is defined as an arbitrary

```
<processnetwork>
    <process name="splitter" basename="splitter">
      <port name="OUT_0_0" type="output" basename="OUT" range="2;2"/>
      <port name="OUT_0_1" type="output" basename="OUT" range="2;2"/>
      <port name="OUT_1_0" type="output" basename="OUT" range="2;2"/>
      <port name="OUT_1_1" type="output" basename="OUT" range="2;2"/>
      <source location="splitter.c" type="c"/>
    </process>
      . . .
    <process name="joiner" basename="joiner">
      <port name="IN_0_0" type="input" basename="IN" range="2;2"/>
      <port name="IN_0_1" type="input" basename="IN" range="2;2"/>
      <port name="IN_1_0" type="input" basename="IN" range="2;2"/>
      <port name="IN_1_1" type="input" basename="IN" range="2;2"/>
      <source location="joiner.c" type="c"/>
    </process>
    <sw_channel name="FIFO_GEN_1_1" type="fifo" size="72000" basename="FIFO_GEN_1_1">
      <port name="0" type="input" basename="0"/>
      <port name="1" type="output" basename="1"/>
    </sw_channel>
      . . .
    <sw_channel name="FIFO_2_1_2_2" type="fifo" size="72000" basename="FIFO_2_1_2_2">
      <port name="0" type="input" basename="0"/>
      <port name="1" type="output" basename="1"/>
    </sw_channel>
    <connection name="g-f_gen_1_1">
      <origin name="splitter">
        <port name="OUT_0_0"/>
      </origin>
      <target name="FIFO_GEN_1_1">
        <port name="0"/>
      </target>
    </connection>
      . . .
    <connection name="f-p_2_1_2_2">
      <origin name="FIFO_2_1_2_2">
        <port name="1"/>
      </origin>
      <target name="p_2_2">
        <port name="INx1x0"/>
      </target>
    </connection>
</processnetwork>
```

Figure 3.6: *Fragment of the DOL description of the Cholesksy process network*

C data structure named *P_state* and its behavior as the program *P_init(); while (true) P_fire();* where *P_init(), P_fire()* are arbitrary functions operating on the process state. The initial call of the *P_init()* function is followed by an endless loop calling the *P_fire()* function. Communication is realized by using two particular primitives, namely *write* and *read* for respectively sending and receiving data to FIFO channels. A *read* operation reads data from an input port, and a *write* operation writes data to an output port. Moreover, the *P_fire()* method invokes a *detach* primitive in order to terminate the execution of the process.

10 Example

The description of the process P_{22} is shown in Figure 3.7. It defines the function p_2_2_init() to initialize the process state and the function p_2_2_fire() to describe the cyclic behavior of the process. A call to p_2_2_fire() implements all operations required by a factorization. It reads the input block A_{22} from port IN_SPLT, reads the result block of P_{21} from port IN_2_1, performs the computation and writes the resulting L_{22} block to the port OUT_JOIN. The process terminates after a fixed number of operations, when the local variable index exceeds len.

```
void p_2_2_init(DOLProcess *p) {
  p->local->index = 0;
  p->local->len = LENGTH;
}
int p_2_2_fire(DOLProcess *p) {
  if (p->local->index < p->local->len) {
    // read input block A22 from splitter
    read((void*)IN_SPLT, p->local->A,
      (K)*(K)*sizeof(double), p);
    // read result block L21 from P21
    read((void*)IN_2_1, p->local->X,
      (K)*(K)*sizeof(double), p);
    // compute A22 = A22 − L21 × L21^t
    SubtractTProduct(p->local->A,
        p->local->X, p->local->X);
    // compute L22 = seq-cholesky(A22)
    Cholesky(p->local->L, p->local->A);
    // send the result L22 to the joiner
    write((void*)OUT_JOIN, p->local->L,
      (K)*(K)*sizeof(double), p);
    p->local->index++;
  else {
    // termination
    detach(p);
    return -1;
  }
  return 0;
}
```

Figure 3.7: *C code for the P_{22} process*

3.3.2 DOL based representation in BIP (DOL to BIP translation)

The construction of the application software model in BIP requires the translation of the software processes, FIFO channels and their connections. The construction is structure-preserving: each process and each FIFO are independently translated to atomic components in BIP and then connected according to their connections in the process network.

Translation of Software Processes

The translation converts each software process to an atomic component in BIP. Each atomic component port corresponds to a port in the process. The translation requires the extraction of a control-flow graph from the C code. It starts by parsing the process code into an intermediate, annotated abstract syntax tree (AST). The translation to BIP is then completed in two steps.

1. In the first step, the interaction points in the AST are identified, that is, each call to a *read/write* primitive is registered as an interaction point.

2. The second step involves the construction of an explicit control flow graph and its representation as a finite state automaton extended with data in BIP. For each interaction point, a control location is created. An outgoing transition is added from this location, labeled by the port used in the *read/write* call. The transition models the primitive call and requires synchronization with a FIFO channel.

The port of the transition is associated with data that is read/written by the primitive invocation. Additional assignment statements are added to load/store the data into the local variables in the function.

A block statement that contains interaction points is transformed into sequence of control locations and transitions in the automaton. For such statements, e.g., conditional (if-else, switch) or loop (for, while) or control statement (break, continue, return), additional control locations are created and internal transitions guarded by the control condition are added to model the control automaton.

For a conditional statement, a new control location is created with an incoming transition where the branch condition evaluation action is added. Outgoing transitions, one for the positive branch and another for the negative branch are created. The branches are finally merged to a new control location.

For a loop statement, a new control location is created with an incoming transition where the loop initialization action and exit condition are added. Outgoing transitions, one for the positive exit condition and the other for the negative exit condition are created. For the negative exit branch, a transition back to the starting location of the loop is added, with the exit condition action.

From the last control location generated in the automaton, a transition to the starting control location is added. This models the invocation of the process behavior in a loop at run-time. The termination of the process behavior is modeled as a move to a deadlocked location, that corresponds to the *detach* primitive call.

Notice that functions that contain *read/write* calls (either directly or through nested calls) are *inlined* in the BIP behavior. Consequently, the translation is restricted to programs without communication calls occurring within recursive functions. Additional restrictions are: no use of global variables, and no *goto* statements.

11 Example

Figure 3.8 shows the translation of the P_{22} process into an atomic component in BIP. The C code for P_{22} is provided in Figure 3.7. The generated BIP component has ports *in_split*, *in_2_1*, *out_join*, control locations $\ell_1 \dots \ell_6$ and variables *index*, *len*, A, L and X. Transitions are labeled by ports *in_split*, *in_2_1*, *out_join* and β (internal). At ℓ_2, P_{22} awaits synchronization through *in_split* corresponding to the *read* primitive call, where it reads the matrix block denoted by the variable A. It then synchronizes through *in_2_1*

```
void p_2_2_init(DOLProcess *p) {
  p→local→index = 0;
  p→local→len = LENGTH;
}
int p_2_2_fire(DOLProcess *p) {
  if(p→local→index < p→local→len) {
    // read input block A22 from splitter
    read((void*)in_split, p→local →len) {
      (K)*(K)*sizeof(double), p);
    // read result block L21 from P21
    read((void*)in_2_1, p→local →X) {
      (K)*(K)*sizeof(double), p);
    // compute A22 = A22 − L21 × L21^t
    SubtractTProduct(p→local →A,
      p→local→X, p→local→X);
    // compute L22 = seq-cholesky(A22)
    Cholesky(p→local→L, p→local→A);
      // send the result L22 to the joiner
    write((void*)out_join, p→local→L,
      (K)*(K)*sizeof(double), p);
    p→local →index++;
  else {
    // termination
    detach(p);
    return -1;
  }
  return -1;
}
```

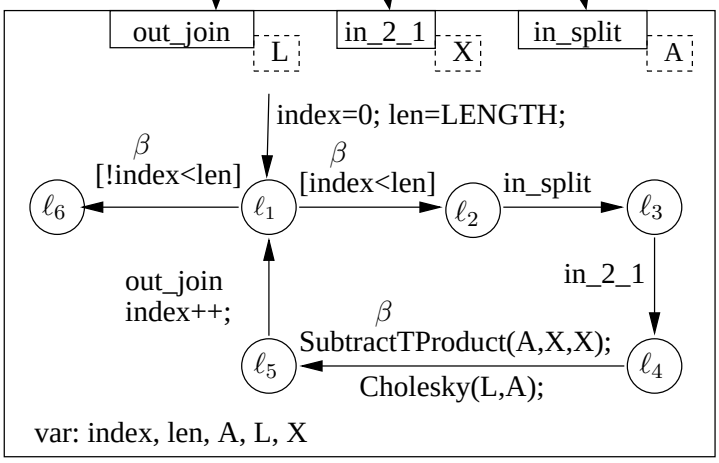


Figure 3.8: *C code and the corresponding BIP model of P_{22} process*

and obtains the result from P_{21}. On the β transition from ℓ_4 to ℓ_5, it performs the actual computations. Finally at ℓ_5, it awaits synchronization through *out_join* corresponding the *write* primitive call. At ℓ_1, the guarded outgoing internal transitions β models the conditional (if) statement. Exit of the process on the *detach* is modeled by the final location ℓ_6.

Translation of FIFO Channels and Connections

A FIFO channel in DOL is translated into a predefined BIP atomic component, as presented in Section 3.2.2. Each connection in the application software is translated into a BIP interaction which strongly synchronizes the corresponding ports. Interactions provide the transfer of data implementing the *read* and *write* operations. An interaction implementing *write* transfers data from a process to a FIFO, whereas the one implementing *read* transfers data from a FIFO to a process.

12 Example
Figure 3.9 depicts the architecture of the BIP model obtained from the process network example given in Figure 3.5.

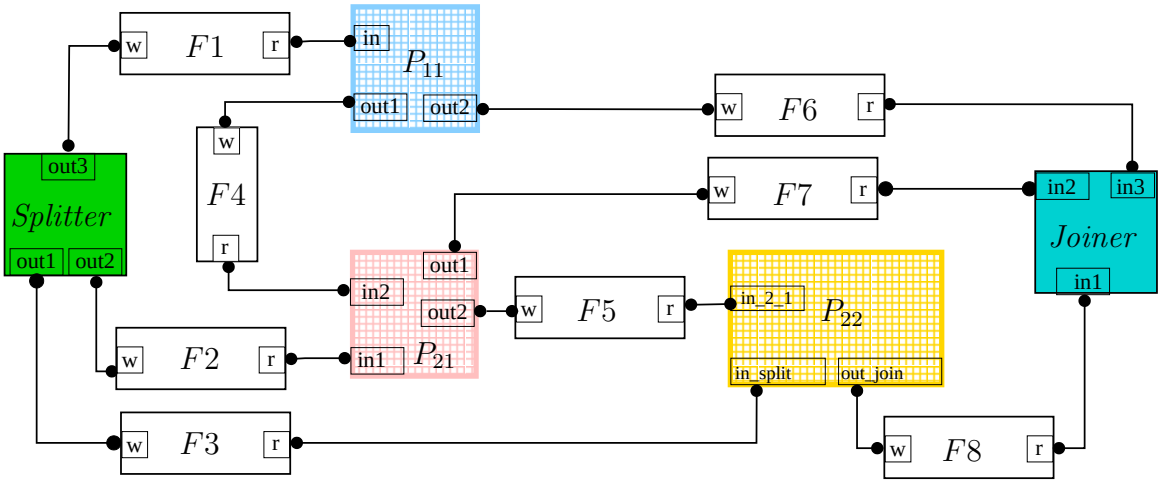


Figure 3.9: *Cholesky(2) application software model in BIP*

3.4 Conclusion

In this chapter, we presented the methods for generating BIP models out of languages and various programming models. The ensemble of these methods consists the BIP Language Factory. The semantic models of BIP preserve the structural representation of the input models and provide analysis and verification techniques included in the rich BIP toolchain. We analyzed the generation of KPN software models in BIP. The method receives as input a KPN application software model described in DOL and produces the equivalent representation in a BIP model. The construction is automated and fully preserves the behavior of the software application. The characteristic of determinism of KPN process networks enable separate analysis of computation and communication, exposes functional parallelism and separates the functionality of the application from the target hardware platform.

In the next chapter, we describe the hardware platform models used as the target platforms which the application software will be mapped and run on.

- Chapter 4 -

Modeling of HW Platforms in BIP

In the previous chapter, we presented the methods for generating BIP models out of languages and various programming models. In this chapter we focus on the hardware platforms by describing the Abstract Model of HW Platform in BIP. A BIP Hardware Model should integrate both computation and communication constraints in a unified model. The computation constraints are added with the use of processor components and the profiling of the software processes. The communication constraints are integrated with the use of cluster components modeling the communication paths using interconnects, buses, Network-on-Chips and memories.

The chapter is structured as follows. First, we provide an introductory text describing abstract models of manycore platforms in Section 4.1. Second, we specify the abstract models of hardware platforms in BIP in Section 4.2. We describe the BIP components needed to cover both computational and communication aspects of the hardware platforms. In the final section, we conclude the chapter.

4.1 Abstract Model of Manycore Platforms

The growing need of efficient and fast execution of parallel applications has led to the development of HW platforms designed to extend the capability of high parallelism in both computation and communication level. These high performance platforms are composed of multicore clusters which can be used as standard processing units or specific accelerators. The construction of the platforms is realized by a synthesis of HW/SW resources. These resources can be homogeneous or heterogeneous including clusters, shared memories, I/O devices or sensors.

The clusters are computational units of the platform containing many processing elements (PE). According to the specification and the characteristics of a cluster, the number of PEs can vary. In addition, there are different types of cluster buses, including the local bus, the crossbar switch and the multiplexing interconnect. There are also local memories for data and instruction caching. Figure 4.1 illustrates an example of a crossbar switch interconnecting PEs with a shared memory. Figure 4.2 shows an example of a multiplexing interconnect attached to a multi-bank memory.

The interconnecting structure needed for exchanging information on a chip is a major characteristic. In the current work, we focus on the description of a Network-on-Chip (NoC) composed by homogeneous manycore cluster IPs, as depicted in Figure 4.3. The NoC offers a high speed, low latency, low power and reliable communication solution

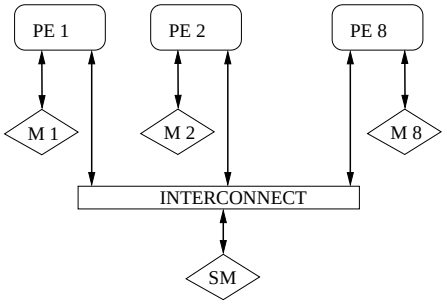


Figure 4.1: *Cluster Description*

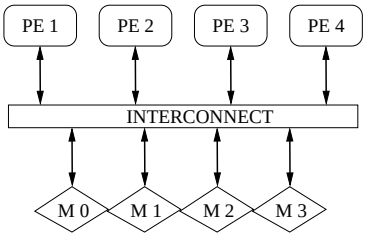


Figure 4.2: *Cluster Description*

based on packet transmission. Each packet contains routing and arbitrating information in order to be transported across the NoC. The packet routing is executed with the use of Routers. A Router contains input and output ports towards each direction (North, South, East, West), as well as towards the Local Cluster, which is connected to it via a Network Interface (NI).

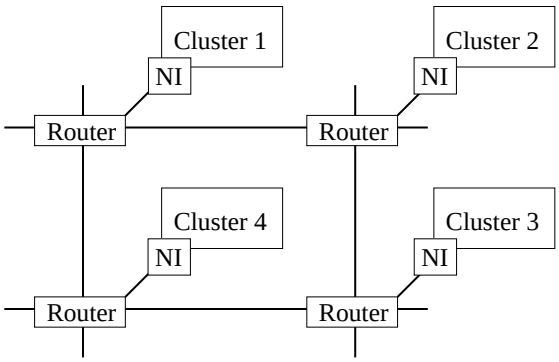


Figure 4.3: *NoC Description*

4.2 Abstract Models of HW Platforms in BIP

A HW platform consists of computational resources interconnected according to communication paths. Resources are used for computation (processors, memories) or for communication (buses). Communication paths define the connections between computational resources. More formally, we consider the family of HW platforms that can be represented by the following grammar:

$$
\begin{array}{rcl}
\textit{HW-Platform} & ::= & \textit{HW-Resource}^{+} \ . \ \textit{HW-Comm-Path}^{+} \\
\textit{HW-Resource} & ::= & \textit{HW-Processor} \mid \textit{HW-Memory} \mid \textit{HW-Cluster-InterConnect} \mid \\
 & & \textit{HW-NI} \mid \textit{HW-Router} \\
\textit{HW-Comm-Path} & ::= & \textit{HW-Processor} \ . \ \textit{HW-InterConnect} \ . \ \textit{HW-Memory} \\
\textit{HW-InterConnect} & ::= & \textit{HW-Cluster-InterConnect} \mid \\
 & & \textit{HW-Cluster-InterConnect} \ . \ \textit{HW-NI} \ . \ \textit{HW-Router}^{+} \ . \\
 & & \textit{HW-NI} \ . \ \textit{HW-Cluster-InterConnect}
\end{array}
$$

The BIP model constructed from the HW platform represents explicitly, in an operational manner, the interconnect between the different resources as defined by the communication paths. This model is organized as a collection of interconnect, network interface, router, bus, processor and memory components.

The goal of the HW model in BIP is to capture the HW constraints which arise upon the execution of a software application on a given platform. More specifically, these constraints are characterized as non-functional and can include parameters such as time delays, temperature and energy consumption. They are all affected by scheduling policies, conflicts, throughputs and response times. Therefore, the constraints exist both on computation and on communication level. In the current model, we focus on integrating non-functional constraints concerning time delays.

On the computation level, accurate measurements on the effects of a software application running on a HW platform demand fine-grained analysis of the executable code and the underlying processing elements of the platform. Initially, the executable code should be analyzed at the instruction level. A pipeline model should be constructed along with instruction and data caches. The above model will permit cycle-accurate performance analysis of the basic operations of the processing element and the cache hit/miss costs. In addition, the scheduling policies applied on the different processes can dramatically affect the performance. The computation measurements are achieved with the use of a Processor model described in the next section and the profiling of the software processes with the corresponding timing delays as described in Chapter 7.

On the communication level, significant constraints derive from the use of buses, interconnects and memories. Namely, we have bus conflicts, bus throughputs and scheduling, routing delays, routing policy and network throughput, memory data access, memory access response and memory conflicts. The communication constraints are integrated with the use of components modeling the communication paths from PEs towards the platform memories and via versa. These components are crossbar switches, shared memories, multiplexing interconnects and multi-banked memories used to model intra-cluster communication and components such as network interfaces and routers for off-cluster communication.

4.2.1 Processor Abstract Model for Computation Constraints and Scheduling

A *Processor* is a composite placeholder component with ports *wr_begin*, *wr_end*, *rd_begin*, *rd_end*, corresponding to the initiation and termination of write and read operations. It is composed by a set of software processes connected to the *Processor Scheduler* Component, as it is illustrated in Figure 4.4. Although, it is filled with the software processes during the next step, that is, the construction of the system model, we present below the *Processor Scheduler* Component and its functionality. The *Processor Scheduler* is responsible for modeling the processor resource. It uses the *acq* (*Acquire*), *rel* (*Release*) ports to grant and remove the control from the different processes which operate on the *Processor*. Initially, the *Processor Scheduler* uses the *acq* port to authorize a software process to run. Each β transition of the process is profiled with a delay to correspond with the amount of computational workload executed. Using the *get_d* (*Get-Delay*) port the *Processor Scheduler* measures the computational delay and continues to the next β transition through the *next* port. Finally, the software process reaches the point where the scheduler can release the control and allow another process to run. The formal Definition 18 is given below.

18 Definition (Processor Scheduler Component)
We define the Processor Scheduler Component as $SC = (L_{SC}, X_{SC}, P_{SC}, \mathcal{T}_{SC})$ *with Acquire and Release scheduling transitions, where the behavior is illustrated in Figure 4.5:*

- $L_{SC} = \{\ell_1, \ell_2, \ell_3\}$, *the set of control locations,*

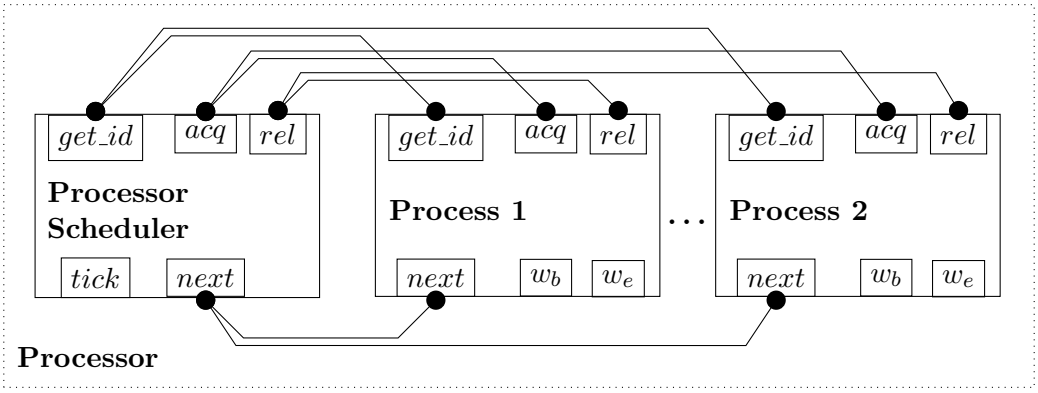


Figure 4.4: *Processor Abstract Model in BIP*

- $X_{SC} = \{count, delay\}$, *the set of variables,*
- $P_{SC} = \{acq, rel, get\_d, next, tick\}$, *the set of ports,*
- $\mathcal{T}_{SC} = \{$
  $\tau_{acq} = (\ell_1, acq, true, f_{acq}, \ell_2)$,
  $\tau_{rel} = (\ell_2, rel, true, skip, \ell_1)$,
  $\tau_{d} = (\ell_2, get\_d, true, skip, \ell_3)$,
  $\tau_{next} = (\ell_3, next, g_{next}, skip, \ell_2)$,
  $\tau_{tick} = (\ell_3, tick, g_t, f_t, \ell_3)$,
  $\tau'_{rel} = (\ell_3, rel, g_{rel}, skip, \ell_1)$
  $\}$.

, *with*
$f_{acq} : count = 0;$
$g_{rel}, g_{next} : [count == delay];$
$g_t : [count < delay], f_t : count + +;$

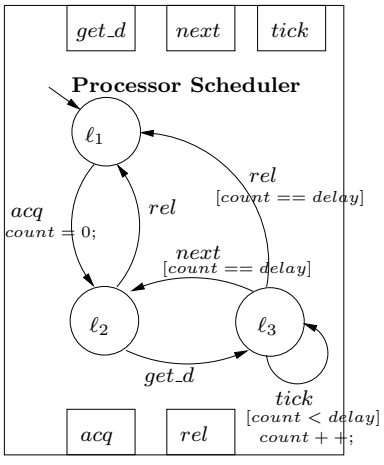


Figure 4.5: *Processor Scheduler Component in BIP*

The concrete implementation of the *Processor* Component is done during the generation of the system model. The use of a *Processor* in the HW platform is illustrated in Figures 4.10, 4.15. The above *Processor* model described in BIP, does not include pipeline

model, instruction and data caches or DMAs. We chose not to provide a detailed model of the *Processor*, so that we avoid explosion in the complexity, the size of the model and the simulation time.

4.2.2 HW Components for Communication Constraints

Crossbar Switch Bus and Memory

A *Crossbar Switch Bus* Component is concretely defined as a scheduled collection of communication path components, as it is shown in Figure 4.6. That is, for each write/read path going on an interconnect, we consider the path fragment defined by the *Bus Path* Component, which is responsible for the following operations:

- controlling the access of the communication path on the bus and initiates the write/read operation.
- modeling effectively the transfer of data over the bus, from the processor once it gets access to the bus, towards the memory.
- receiving data either from some software processes executing inside the processor or from the previous path segment, depending on its position on the path. It acts like a buffer and is needed to connect further either to the next path fragment or to the memory.

The component is a *timed* BIP component [BBS06]. It is equipped with a set of counters to measure the timing delays of all the operations. The delays counted are:

- the bus conflict, the time elapsed between the write/read request and the start of the data transfer,
- the bus delay, the data transfer delay over the bus
- the next operation conflict, the conflict time passed until the next path fragment or the memory becomes available.

There are used for observation purposes, as explained in Chapter 8.

For the transport of data the ports *req* (*Request*), *ack* (*Acknowledge*) are used to connect with upper components, and *op_begin* (*Operation-Begin*), *op_end* (*Operation-End*) to connect with lower components on the path. In addition, the ports *acq* (*Acquire*) and *rel* (*Release*) are used to interact with the *Bus Scheduler*. The component is predefined and belongs to the BIP hardware library. The formal definition is found below.

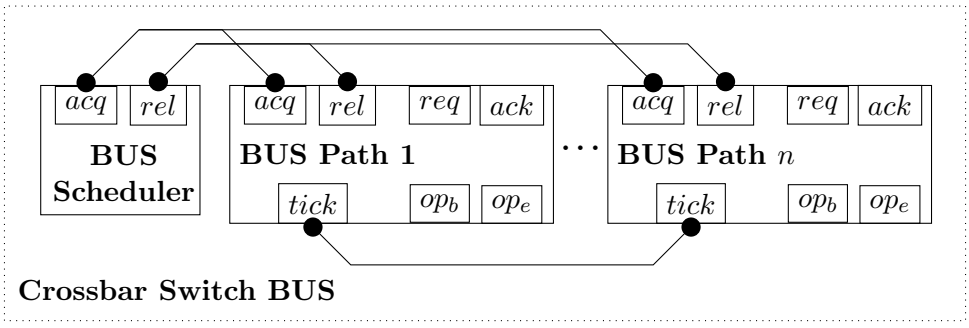


Figure 4.6: *Crossbar Switch BUS Model in BIP*

19 Definition (Bus Path Component)
We define the Bus Path Component as $BP = (L_{BP}, X_{BP}, P_{BP}, \mathcal{T}_{BP})$*, where:*

- $L_{BP} = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_7\}$,
- $X_{BP} = \{count, bus\_delay, bus\_conflict, op\_conflict, proc\_id\}$,
- $P_{BP} = \{req, ack, acq, rel, tick, op_b, op_e\}$, *where the port* op_e *is binded with the* $proc\_id$ *variable.*
- $\mathcal{T}_{BP} = \{$
  $\tau = (\ell_1, req, true, f_{req}, \ell_2)$ $,f_{req} : bus\_conflict = 0;$
  $\tau = (\ell_2, tick, true, f_{t_1}, \ell_2)$ $,f_{t_1} : bus\_conflict + +;$
  $\tau = (\ell_2, acq, true, f_b, \ell_3)$ $f_b : count = 0;$
  $\tau = (\ell_3, tick, g_{t_2}, f_{t_2}, \ell_3)$ $,g_{t_2} : [count < bus\_delay]$ $, f_{t_2} : count + +;$
  $\tau = (\ell_3, \beta, g_\beta, f_\beta, \ell_4)$ $,g_\beta : [count == bus\_delay]$ $, f_\beta : op\_conflict = 0;$
  $\tau = (\ell_4, tick, true, f_{t_3}, \ell_4)$ $, f_{t_3} : op\_conflict + +;$
  $\tau = (\ell_4, op_b, true, skip, \ell_5)$
  $\tau = (\ell_5, op_e, true, skip, \ell_6)$
  $\tau = (\ell_6, rel, true, skip, \ell_7)$
  $\tau = (\ell_7, ack, true, skip, \ell_1)$
  $\}$.

Each connection is realized using BIP connectors which strongly synchronize the corresponding ports. The behavior of the connector implements the transfer of data, its address and size between the successive components, corresponding to the *write* and *read* operations.

All the paths segments going over the same bus must share its transport capabilities according to some predefined bus policy. The scheduling can be of one of fixed-priority, round-robin or TDMA (Time Division Multiple Access). We model it explicitly by using a *Bus Scheduler* Component, which interacts with all the *Bus Path* Components and ensures exclusive access for transmission of data, according to the selected policy. The *Bus Scheduler* acts as an arbiter to resolve the bus access conflicts. A simplified *Bus Scheduler* Component is given below. It implements the basic mutual exclusion scheduling policy.

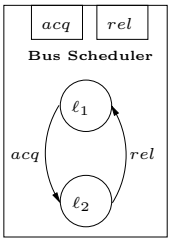


Figure 4.7: *Bus Scheduler Component in BIP*

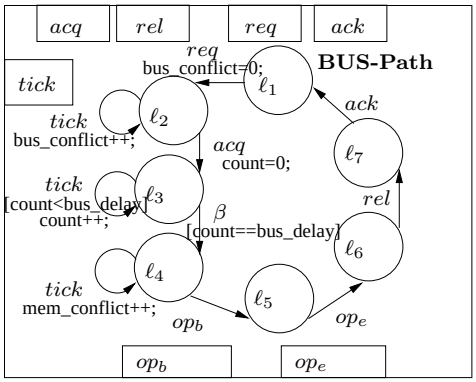


Figure 4.8: *Bus Path Component in BIP*

20 Definition (Bus Scheduler Component)
We define the Bus Scheduler Component as $BS = (L_{BS}, X_{BS}, P_{BS}, \mathcal{T}_{BS})$*, where:*

- $L_{BS} = \{\ell_1, \ell_2\}$,
- $X_{BS} = \emptyset$,
- $P_{BS} = \{acq, rel\}$,
- $\mathcal{T}_{BS} = \{\tau = (\ell_1, acq, true, skip, \ell_2), \tau = (\ell_2, rel, true, skip, \ell_1)\}$.

The *Memory* is a timed component used to model the write/read memory operations and measure the memory access delay. It is a component with ports op_b (*Operation-Begin*), op_e (*Operation-End*) corresponding respectively to the beginning and ending of the write/read. The use of *Memory* Component in the HW platform is shown in figure 4.9. Formally, the component is defined as:

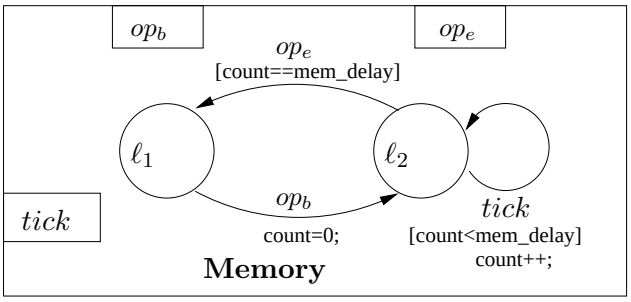


Figure 4.9: *Memory Component in BIP*

21 Definition (Memory Component)
We define the Memory Component as $M = (L_M, X_M, P_M, \mathcal{T}_M)$, *where:*

- $L_M = \{\ell_1, \ell_2\}$,
- $X_M = \{count, mem\_delay\}$,
- $P_M = \{op_b, tick, op_e\}$,
- $\mathcal{T}_M = \{$
  $\tau_{op_b} = (\ell_1, op_b, true, f_{op_b}, \ell_2)$,
  $\tau_{tick} = (\ell_2, tick, g_t, f_t, \ell_2)$,
  $\tau_{op_e} = (\ell_2, op_e, g_{op_e}, skip, \ell_1)$
  $\}$.

with,
$f_{op_b} : count = 0;$,
$g_t : [count < mem\_delay]$, $f_t : count + +;$,
$g_{op\_e} : [count == mem\_delay]$

We assume a platform where there are $k > 1$ processors, one *Crossbar Switch Bus* and a shared *Memory*.

22 Definition (Shared Memory Cluster Compound Component)
We define the Shared Memory Cluster Compound Component as $MC = \pi\gamma(BUS, M)$ *as the Communication Model of a Cluster, where:*
$BUS = \{BP_1, \ldots, BP_k, BS\}$, *where* k *the number of processors,* BP *the Bus Paths and*

BS the Bus Scheduler,
M the Memory Component. The set of interactions γ is defined as:
$\gamma = \{\alpha_b^{op}, \alpha_e^{op}, \alpha^{acq}, \alpha^{rel} | BP \in BUS\} \cup \{\alpha^{tick}\}$
with,
$\alpha^{acq} = (\{BP.acq, BS.acq\}, true, skip)$,
$\alpha^{rel} = (\{BP.rel, BS.rel\}, true, skip)$,
$\alpha_b^{op} = (\{BP.op_b, M.op_b\}, true, skip)$,
$\alpha_e^{op} = (\{M.op_e, BP.op_e\}, true, skip)$,
$\alpha^{tick} = (\{BS.tick, BP_1.tick, \ldots, BP_k.tick, M.tick\}, true, skip)$.
For each $\alpha \in \gamma$, the priority rule π implies that the tick interaction α^{tick} has lower priority than any other interaction α. We export the tick port of the α^{tick} interaction to dynamically enable synchronization with other tick interactions. We will further refer to the α^{tick} interaction as $MC.tick$.

13 Example
Figure 4.10 shows the BIP model of a 4-processor (P) HW Platform and a shared memory. Communication paths between the processors and the memory are implemented using the previously defined set of bus components.

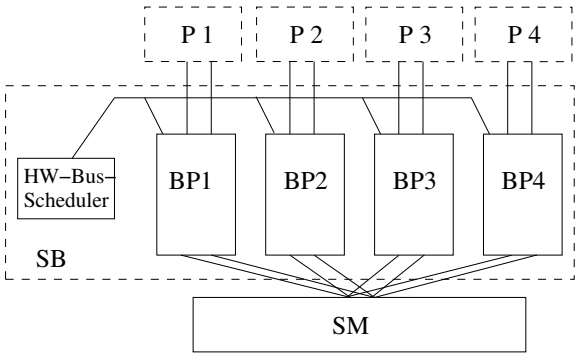


Figure 4.10: *BIP model of a HW platform with four processors and one shared memory*

Multiplexing Interconnect and MultiBank Memory

Another type of cluster bus is the *Multiplexing Interconnect* Component. The component is designed to connect processors to a multi-banked memory. Data are forwarded using routing and arbitration techniques. Based on address decoding, the requested address may correspond to the intra-cluster multi-banked memory or the off-cluster NoC environment. In case of simultaneous accesses in contiguous data, the number of the observed memory conflicts is reduced due to fine-grained address interleaving.

The *Multiplexing InterConnect* is a composite component synthesized by a set of *Bus Interface Components* and a set of connectors towards the multi-banked memory. An abstract illustration is provided in Figure 4.11. The number of the *Bus Interface* Components is the same as the number of processors used on the cluster. The *Bus Interface* Component is intended to receive the write/read requests to the memory and carry out the routing towards the corresponding *Memory Bank*. This is done with the use of address decoding, as mentioned above. In the formal definition found below, we use a random routing algorithm to forward the requests to the memory banks. The delay posed by the interconnect is taken into account by the memory bank components described next.

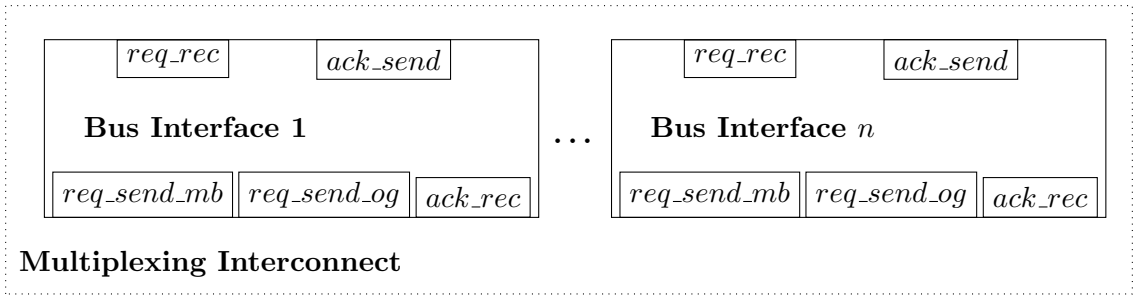


Figure 4.11: *Multiplexing Interconnect Model in BIP*

The *Bus Interface* Component has the following four ports: the *req_rec* (*Request-Receive*), *req_send* (*Request-Send*) to receive and forward the memory access requests to the memory, and the *ack_rec* (*Acknowledgment-Receive*), *ack_send* (*Acknowledgment-Send*) to forward the executed requests back to the processor.

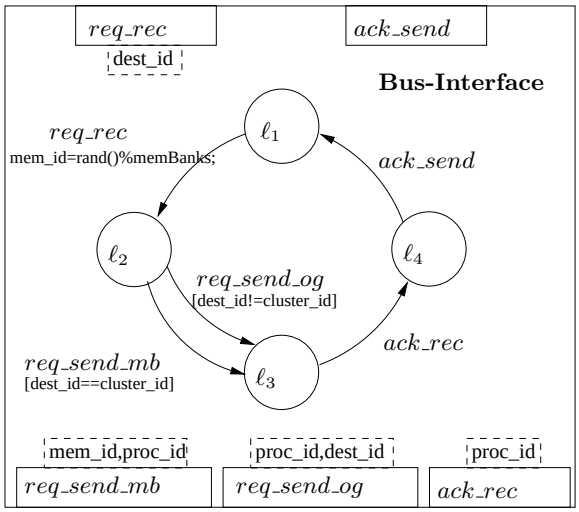


Figure 4.12: *Bus Interface Component in BIP*

23 Definition (Bus Interface Component)
We define the Bus Interface Component as $BI = (L_{BI}, X_{BI}, P_{BI}, \mathcal{T}_{BI})$*, where:*

- $L_{BI} = \{\ell_1, \ell_2, \ell_3, \ell_4\}$,
- $X_{BI} = \{proc\_id, mem\_id, memBanks, dest\_id, cluster\_id\}$,
- $P_{BI} = \{req\_rec, req\_send\_mb, req\_send\_og, ack\_rec, ack\_send\}$,
- $\mathcal{T}_{BI} = \{$
  $\tau = (\ell_1, req\_rec, true, f_{req}, \ell_2)$ $\quad, f_{req} : mem\_id = rand()\%(memBanks);$
  $\tau = (\ell_2, req\_send\_mb, g_{mb}, skip, \ell_3)$ $\quad, g_{mb} : dest\_id == cluster\_id$
  $\tau = (\ell_2, req\_send\_og, g_{og}, skip, \ell_3)$ $\quad, g_{og} : dest\_id! = cluster\_id$
  $\tau = (\ell_3, ack\_rec, true, skip, \ell_4)$ $\quad,$
  $\tau = (\ell_4, ack\_send, true, skip, \ell_1)$
  $\}$.

The *Memory Bank* is a timed component used to synthesize the *Multi-Bank Memory Component*, as illustrated in Figure 4.13. It is used to model the write/read memory operations and measure the interconnect and the memory access delay. It is a component with ports op_b, op_e corresponding respectively to the beginning and ending of the write/read and port *tick* for time synchronization. It can receive multiple requests from the *Multiplexing InterConnect*. The requests are stored to a buffer in case of conflict and upon execution operation end acknowledgment is sent back to the local bus or the off-cluster NoC.

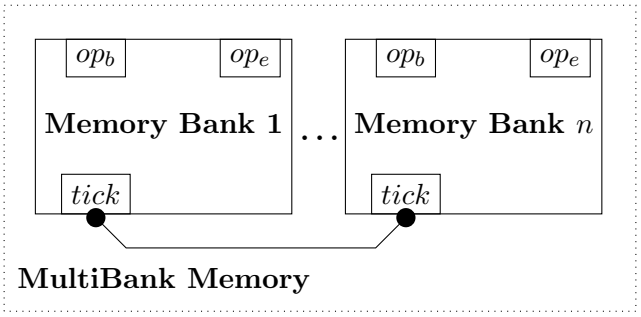


Figure 4.13: *MultiBank Memory Model in BIP*

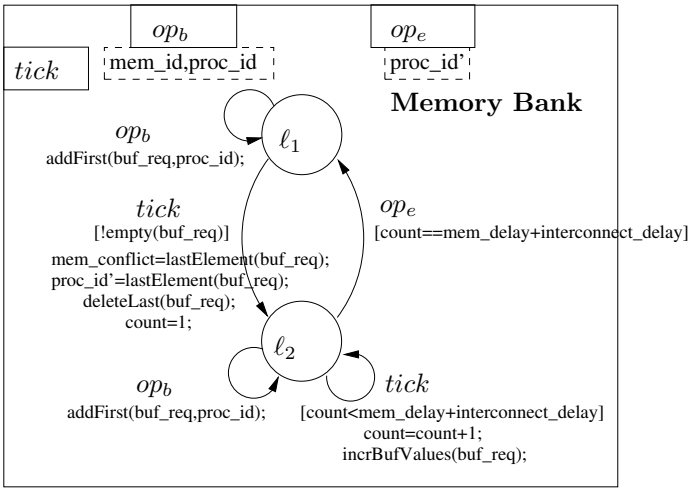


Figure 4.14: *Memory Bank Component in BIP*

24 Definition (Memory Bank Component)
We define the Memory Bank Component as $MB = (L_{MB}, X_{MB}, P_{MB}, \mathcal{T}_{MB})$*, where:*

- $L_{MB} = \{\ell_1, \ell_2\}$,
- $X_{MB} = \{buf\_req, mem\_conflict, count, mem\_delay, interconnect\_delay, proc\_id, proc\_id', mem\_id\}$,
- $P_{MB} = \{op_b, tick, op_e\}$,
- $\mathcal{T}_{MB} = \{$

$$\tau_{op_b} = (\ell_1, op_b, true, f_{op_b}, \ell_1),$$
$$\tau_{op_b} = (\ell_2, op_b, true, f_{op_b}, \ell_2),$$
$$\tau_{tick_1} = (\ell_1, tick, g_{t_1}, f_{t_1}, \ell_2),$$
$$\tau_{tick_2} = (\ell_2, tick, g_{t_2}, f_{t_2}, \ell_2),$$
$$\tau_{op_e} = (\ell_2, op_e, g_{op_e}, skip, \ell_1)$$
$$\}$$

, *with*

f_{op_b} : $addFirst(buf\_req, proc\_id);$,
g_{t_1} : $[!empty(buf\_req)]$,
f_{t_1} : $mem\_conflict = lastElement(buf\_req);$
$proc\_id' = lastElement(buf\_req);$
$deleteLast(buf\_req); count = 1;$,
g_{t_2} : $[count < mem\_delay + interconnect\_delay]$,
f_{t_2} : $count++; incrBufValues(buf\_req);$,
g_{op_e} : $[count == mem\_delay + interconnect\_delay]$

We assume a platform where there are k processors, one multiplexing interconnect and a multi-banked memory.

25 Definition (Multi-Banked Memory Cluster Compound Component) *We define the Multi-Banked Memory Cluster Compound Component as* $MC = \pi\gamma(BUS, M)$ *as the Communication Model of a Cluster, where:*
$BUS = \{BI_1, \ldots BI_k\}$, *where* k *the number of processors,* BI *the Bus Processor Interfaces*
$M = \{MB_1, \ldots, MB_l\}$ *the Multi-Banked Memory Component, where* l *the number of memory banks. The set of interactions* γ *is defined as:*
$\gamma = \{\gamma_{BI} | BI \in BUS\} \cup \{\alpha^{tick}\}$,
where,
$\gamma_{BI} = \{\alpha_b^m, \alpha_e^m | MB \in M\}$,
$\alpha_b^m =$ $(\{BI.req\_send\_mb, MB.op_b\}, g, f)$, *with*
$g : [BI.mem\_id == MB.mem\_id], f : MB.proc\_id = BI.proc\_id;)$
$\alpha_e^m =$ $(\{MB.op_e, BI.ack\_rec\}, g, skip)$, *with*
$g : [MB.proc\_id == BI.proc\_id]$
$\alpha^{tick} = (\{BI_1.tick, \ldots, BI_k.tick, MB_1.tick, \ldots, BI_l\}, true, skip)$.

For each $\alpha \in \gamma$, *the priority rule* π *implies that the tick interaction* α^{tick} *has lower priority than any other interaction* α. *We export the tick port of the* α^{tick} *interaction to dynamically enable synchronization with other tick interactions. We will further refer to the* α^{tick} *interaction as* $MC.tick$.

Network On Chip

We model the network-on-chip in BIP using a composition of network components. These components are network interface components, routers and the connections between them. A network interface component is responsible for establishing a connection between the intra-cluster bus and the off-cluster Network on Chip. It is modeled by two different components: the *Network Interface Outgoing Controller* and the *Network Interface Incoming Controller*, as shown in Figure 4.16.

The *Network Interface Outgoing Controller* Component receives write/read requests or acknowledgments from the cluster bus and encapsulates them into packages. It forwards them to the local router modeling the packaging and the forwarding delays. The ports

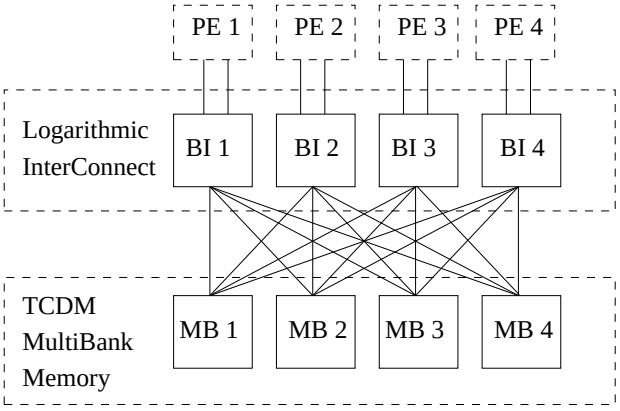


Figure 4.15: *BIP model of a HW platform with four processors and a multi-banked memory*

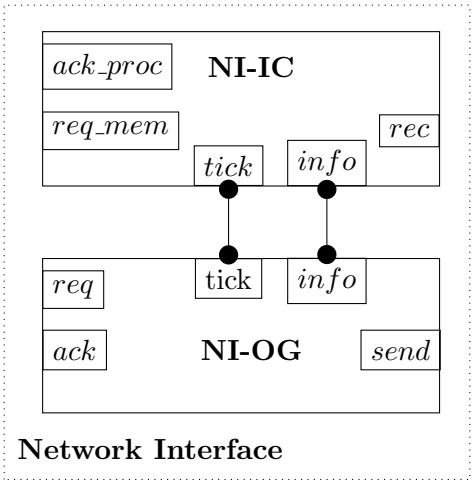


Figure 4.16: *Network Interface Model in BIP*

req (*Request*), *ack* (*Acknowledgment*) are used to interact with the cluster bus. The port *send* interacts with the local network router and the *info* port is used to receive packaging information from the *Network Interface Incoming Controller*.

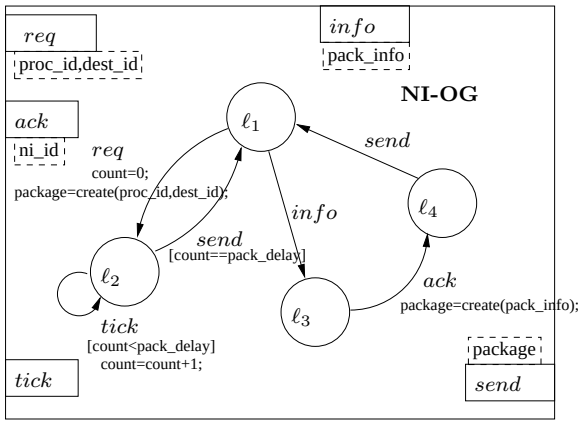


Figure 4.17: *Network Interface Outgoing Controller Component in BIP*

26 Definition (Network Interface Outgoing Controller Component)
We define the Network Interface Outgoing Controller Component as $OG = (L_{OG}, X_{OG}, P_{OG}, \mathcal{T}_{OG})$*, where:*

- $L_{OG} = \{\ell_1, \ell_2, \ell_3, \ell_4\}$,
- $X_{OG} = \{count, pack\_delay, ni\_id, proc\_id, package, dest\_id, pack\_info\}$,
- $P_{OG} = \{req, tick, send, info, ack\}$, *where the ports* $send, info$ *are binded with the package variable.*
- $\mathcal{T}_{OG} = \{$

$$\begin{array}{lll}
\tau = (\ell_1, req, true, f_{req}, \ell_2) & ,f_{req}: & count = 0; \\
 & & package = create(proc_id, dest_id);, \\
\tau = (\ell_2, tick, g_t, f_t, \ell_2) & ,g_t: & [count < pack_delay],\ f_t : count++;, \\
\tau = (\ell_2, send, g_s, skip, \ell_1) & ,g_s: & [count == pack_delay], \\
\tau = (\ell_1, info, true, skip, \ell_3) & , & \\
\tau = (\ell_3, ack, true, f_{ack}, \ell_4) & ,\ f_{ack}: & package = create(pack_info);, \\
\tau = (\ell_4, send, true, skip, \ell_1) & &
\end{array}$$

$\}$.

The *Network Interface Incoming Controller* Component receives packages from the external network, extracts the important information and forwards the write/read requests or acknowledgments to the cluster bus. The package reception from the NoC is realized through the *rec* (*Receive*) port. The ports *ack_proc* (Acknowledgment to Processor), *req_mem* (*Request to Memory*) forward the acknowledgment to the processor and respectively, the request to the memory. The *info* port provides the *Network Interface Outgoing Controller* with the packaging information needed to encapsulate the acknowledgments received from the memory.

27 Definition (Network Interface Incoming Controller Component)
We define the Network Interface Incoming Controller Component as $IC = (L_{IC}, X_{IC}, P_{IC}, \mathcal{T}_{IC})$*, where:*

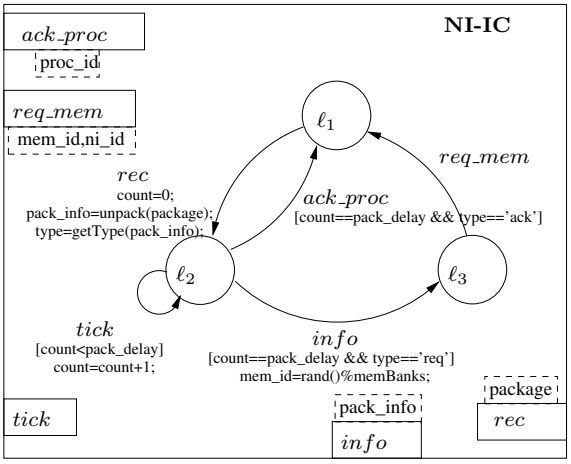


Figure 4.18: *Network Interface Incoming Controller Component in BIP*

- $L_{IC} = \{\ell_1, \ell_2, \ell_3\}$,
- $X_{IC} = \{count, pack\_delay, proc\_id, package, pack\_info, mem\_id, memBanks\}$,
- $P_{IC} = \{rec, tick, ack\_proc, req\_mem, info\}$, where the ports $rec\_pack, info$ are binded with the *package* variable.
- $\mathcal{T}_{IC} = \{$

$$\begin{array}{lll}
\tau = (\ell_1, rec, true, f_{rec}, \ell_2) & ,f_{rec}: & count = 0; \\
 & & pack_info = unpack(package); \\
 & & type = getType(pack_info); \\
\tau = (\ell_2, tick, g_t, f_t, \ell_2) & ,g_t: & [count < pack_delay],\ f_t : count++; \\
\tau = (\ell_2, ack_proc, g_{ack}, skip, \ell_1) & ,g_{ack}: & [count == pack_delay\ \&\&\ type ==' ack'], \\
\tau = (\ell_2, info, g_{info}, f_{info}, \ell_3) & ,g_{info}: & [count == pack_delay\ \&\&\ type ==' req'], \\
 & f_{info}: & mem_id = rand()\%memBanks;, \\
\tau = (\ell_3, req_mem, true, skip, \ell_1) & &
\end{array}$$

$\}$.

We independently define the *Cluster Components* below. At the end of the current section, we provide the complete definition of a NoC BIP model. We assume a HW platform where there are k processors, one crossbar switch, a shared memory and a network interface.

28 Definition (Shared Memory Cluster Compound Component with a Network Interface) *We define the Shared Memory Cluster Compound Component as* $CL^{SM} = \pi\gamma(BUS, M, NI)$ *as the Communication Model of a Cluster, where:*
$BUS = \{BP_1, \ldots BP_{k+k+1}, BS\}$, *where* k *the number of processors,* BP *the Bus Paths and* BS *the Bus Scheduler.*
$NI = \{OG, IC\}$.
M *the Memory Component. The set of interactions* γ *is defined as:*

$$\begin{array}{ll}
\gamma = & \{\alpha^{acq}, \alpha^{rel} | BP \in BUS\} \cup \{\alpha_b^m, \alpha_e^m | BP \in \{BP_1 \ldots BP_{k+1}\}\} \cup \\
 & \{\alpha_b^{out}, \alpha_e^{out} | BP \in \{BP_k \ldots BP_{k+k}\}\} \cup \{\alpha_b^{in}, \alpha_e^{in}, \alpha^{info}, \alpha^{tick}\},
\end{array}$$

with,

$\alpha^{acq} = (\{BP.acq, BS.acq\}, true, skip)$,
$\alpha^{rel} = (\{BP.rel, BS.rel\}, true, skip)$,
$\alpha_b^m = (\{BP.op_b, M.op_b\}, true, skip)$,
$\alpha_e^m = (\{M.op_e, BP.op_e\}, true, skip)$,
$\alpha_b^{out} = (\{BP.op_b, OG.req\}, true, skip)$,
$\alpha_e^{out} = (\{IC.ack\_proc, BP.op_e\}, [IC.proc\_id == BP.proc\_id], skip)$,
$\alpha_b^{in} = (\{IC.req\_mem, BP.req\}, true, skip)$,
$\alpha_e^{in} = (\{BP.ack, OG.ack\}, true, skip)$,
$\alpha^{info} = (\{IC.info, OG.info\}, true, skip)$,
$\alpha^{tick} = (\{BS.tick, BP_1.tick, \ldots, BP_{k+k+1}.tick, M.tick\}, true, skip)$.
For each $\alpha \in \gamma$, the priority rule π implies that the tick interaction α^{tick} has lower priority than any other interaction α. The Shared Memory Cluster Compound Component exports the ports OG.send as send, the IC.rec as rec and the α^{tick} interaction export the port tick.

We assume a HW platform where there are k processors, a multiplexing interconnect, a multi-banked memory and a network interface.

29 Definition (Multi-Banked Memory Cluster Compound Component with a Network Interface)
We define the Multi-Banked Memory Cluster Compound Component as $CL^{MB} = \pi\gamma(BUS, M, NI)$
as the Communication Model of a Cluster, where:
$BUS = \{BI_1, \ldots BI_k\}$, *where k the number of processors, BI the BUS Processor Interfaces,*
$M = \{MB_1, \ldots, MB_l\}$ *the Multi-Banked Memory Component, where l the number of memory banks,*
$NI = \{OG, IC\}$ *the network interface. The set of interactions γ is defined as:*
$\gamma = \{\gamma_{BI} | BI \in BUS\} \cup \{\alpha_b^{out}, \alpha_e^{out} | BI \in BUS\} \cup \{\alpha_b^{in}, \alpha_e^{in} | MB \in M\} \cup \{\alpha^{info}\} \cup \{\alpha^{tick}\}$,
where,
$\gamma_{BI} = \{\alpha_b^m, \alpha_e^m | MB \in M\}$,

$\alpha_b^m = (\{BI.req\_send\_mb, MB.op_b\}, g, f)$,
$g : [BI.mem\_id == MB.mem\_id], f : MB.proc\_id = BI.proc\_id;$,
$\alpha_e^m = (\{MB.op_e, BI.ack\_rec\}, g, skip)$,
$g : [MB.proc\_id == BI.proc\_id]$,
$\alpha_b^{out} = (\{BI.req\_send\_og, OG.req\}, true, f)$,
$f : OG.proc\_id = BI.proc\_id; OG.dest\_id = BI.dest\_id;$,
$\alpha_e^{out} = (\{IC.ack\_proc, BI.ack\_rec\}, g, skip)$,
$g : [IC.proc\_id == BI.proc\_id]$,
$\alpha_b^{in} = (\{IC.req\_mem, MB.op_b\}, g, f)$,
$g : [IC.mem\_id == MB.mem\_id], f : MB.proc\_id = IC.ni\_id;$,
$\alpha_e^{in} = (\{MB.op\_end, OG.ack\}, g, skip)$,
$g : [MB.proc\_id == OG.ni\_id]$,
$\alpha^{info} = (\{IC.info, OG.info\}, true, skip)$,
$\alpha^{tick} = (\{BI_1.tick, \ldots, BI_k.tick, MB_1.tick, \ldots, BI_l, IC.tick, OG.tick\}, true, skip)$.

For each $\alpha \in \gamma$, the priority rule π implies that the tick interaction α^{tick} has lower priority than any other interaction α. The Multi-Banked Memory Cluster Compound Component exports the ports OG.send as send, the port IC.rec as rec and the α^{tick} interaction export the port tick.

In Figure 4.19, we illustrate an example of a BIP model of a HW platform with four processors, a multiplexing interconnect, a multi-banked memory and a network interface. The latter consists of the *NI-IN* (Network Interface Incoming) Component and the *NI-OG*

(Network Interface Outgoing) Component, which are both connected to the multiplexing interconnect.

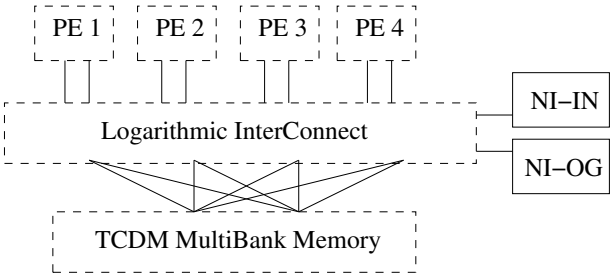


Figure 4.19: *BIP model of a HW platform with four processors, a multiplexing interconnect, a multi-banked memory and a network interface*

The key component which the NoC functionality is based on are the network routers. The routers read the information contained inside a package and they are charged with the task of routing and arbitrating the packages inside the NoC. We model the routers in BIP using a composite component called *Router*. The *Router* is composed by a set of coupled *Router Incoming* and *Router Outgoing Port* Components, as illustrated in Figure 4.20. There is one couple *Incoming* and *Outgoing Port* Component for each routing direction. The directions are North, South, East, West and local cluster. The port components are all connected with all the other directions. That is, each *Router Incoming Port* Component is connected with all the *Router Outgoing Port* Components of all the other directions. The connection is modeled with a connector controlled by a guard depending on the routing information of the package.

The *Router Incoming Port* Component uses the *rec* port to receive the incoming packages in the router. It simply forwards the packages to the guarded connectors sending explicitly the routing information to it along with all the package through the *fwd* port. The component is also timed, measuring the *Router Outgoing Port* conflict delays.

30 Definition (Router Incoming Port Component)
We define the Router Incoming Port Component as $IR = (L_{IR}, X_{IR}, P_{IR}, \mathcal{T}_{IR})$*, where:*

- $L_{IR} = \{\ell_1, \ell_2\}$,
- $X_{IR} = \{package, port\_id)\}$,
- $P_{IR} = \{rec, fwd\}$, *where port* rec *is binded with the package variable and* fwd *port is binded with the package and port_id variables.*
- $\mathcal{T}_{IR} = \{$
  $\tau = (\ell_1, rec, true, skip, \ell_2)\}, \quad f_{rec}: \quad port\_id = route(package)$
  $\tau = (\ell_2, fwd, true, skip, \ell_1)\}$
  $\}$.

The standard routing delay of a package is modeled by the *Router Outgoing Port* Component. Apart from the routing delay the link latency is also considered. The link latency derives from the links connecting the routers on the NoC. This latency is integrated in the *Router Outgoing Port* Component along with the routing delay. The ports *rec* and *send* interact with the *Incoming Port* and the next *Router* Component respectively.

31 Definition (Router Outgoing Port Component)
We define the Router Outgoing Port Component as $OR = (L_{OR}, X_{OR}, P_{OR}, \mathcal{T}_{OR})$*, where:*

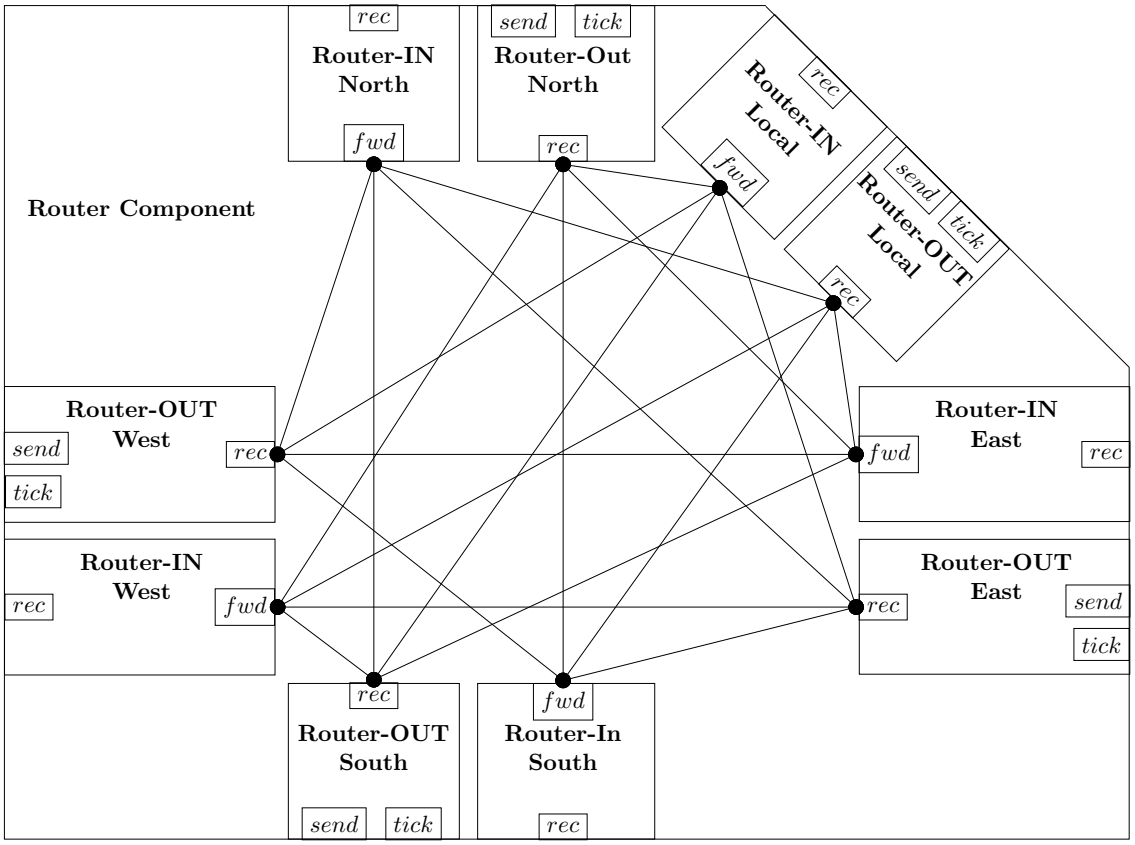


Figure 4.20: *Router Component in BIP*

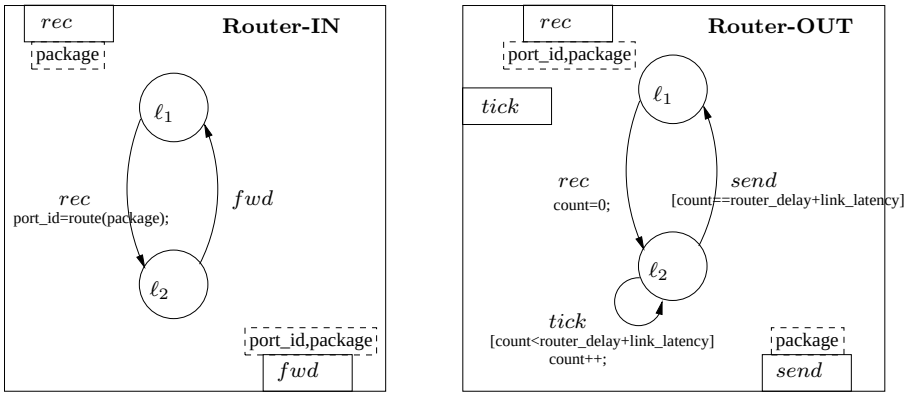


Figure 4.21: *Router Incoming and Router Outgoing Port Components in BIP*

- $L_{OR} = \{\ell_1, \ell_2\}$,
- $X_{OR} = \{package, count, routing\_delay, link\_latency, port\_id)\}$,
- $P_{OR} = \{rec, tick, send\}$, *where port rec is binded with the package and port_id variables and send port is binded with the package variable.*
- $\mathcal{T}_{OR} = \{$
  $\tau = (\ell_1, rec, true, f_r, \ell_2)\}$, f_r : $count = 0;$,
  $\tau = (\ell_2, tick, g_t, f_t, \ell_2)\}$, g_t : $[count < (routing\_delay + link\_latency)]$
  , f_t : $count++$,
  $\tau = (\ell_2, send, g_f, skip, \ell_1)\}$, g_f : $[count == (routing\_delay + link\_latency)]$
  $\}$.

32 Definition (Router Compound Component)
We define the Router Compound Component as $RT = \pi\gamma(IR_N, IR_S, IR_E, IR_W, IR_L, OR_N, OR_S, OR_E, OR_W, OR_L)$, *where abbreviations (N, S, E, W, L) stand for (North, South, East, West, Local) respectively. The set of interactions* γ^s *is defined as:*

$$\gamma = \{\alpha^{NS}, \alpha^{NE}, \alpha^{NW}, \alpha^{NL}\} \cup \{\alpha^{SN}, \alpha^{SE}, \alpha^{SN}, \alpha^{SL}\} \cup \{\alpha^{EN}, \alpha^{ES}, \alpha^{EN}, \alpha^{EL}\} \cup \{\alpha^{WN}, \alpha^{WS}, \alpha^{WE}, \alpha^{WL}\} \cup \{\alpha^{LN}, \alpha^{LS}, \alpha^{LE}, \alpha^{LW}\} \cup \{\alpha^{tick}\},$$

$$\alpha^{xy} = (\{IR_x.fwd, OR_y.rec\}, g, skip), \text{ with } g : [IR_x.port\_id == OR_y.port\_id] \text{ and } x, y \in \{N, S, E, W, L\},$$

$$\alpha^{tick} = (\{OR_N.tick, OR_S.tick, OR_E.tick, OR_W.tick, OR_L.tick\}, true, skip).$$

For each $\alpha \in \gamma$, *the priority rule* π *implies that* α^{tick} *has less priority that any other interaction* $\alpha \in \gamma$. *The Router Compound Component exports the ports* $IR_x.rec$ *as* rec_x, $OR_x.send$ *as* $send_x$

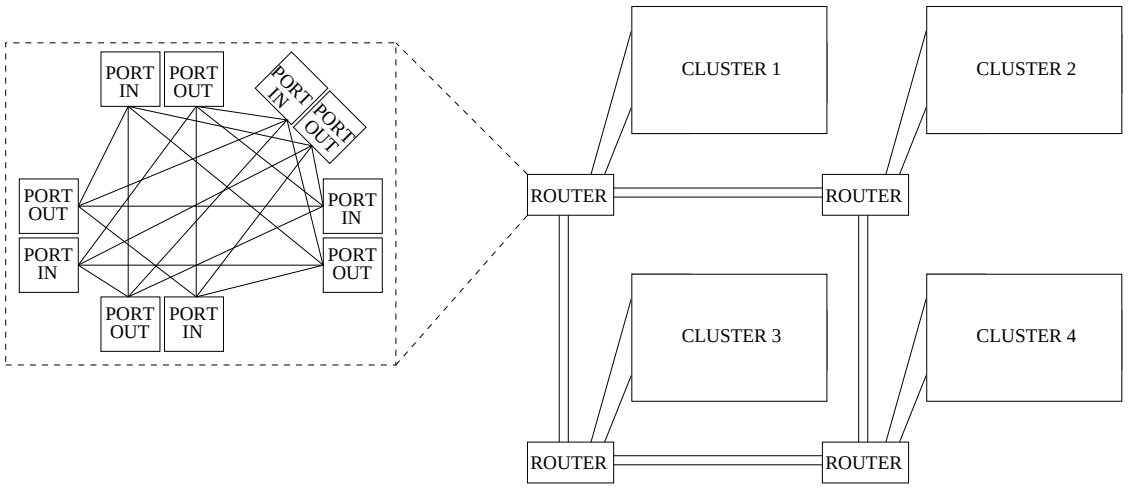


Figure 4.22: *NoC Component in BIP*

We assume a four-cluster NoC platform.

33 Definition (Four-Cluster NoC Compound Component)
We define the Four-Cluster NoC Compound Component as $NoC = \pi\gamma(CL_1, \ldots, CL_4, R_1, \ldots, R_4)$ *as the Communication Model of a NoC, where:*
CL *are the clusters and* R *are the routers. The set of interactions* γ *is defined as:*
$\gamma = \{\gamma_{R1}, \gamma_{R2}, \gamma_{R3}, \gamma_{R4}\} \cup \{\alpha^{tick}\}$,

where,
$\gamma_{R1} = \{\alpha^{in}_{CL1}, \alpha^{out}_{CL1}, \alpha^{in}_{R1}, \alpha^{out}_{R1}\}$,
$\gamma_{R2} = \{\alpha^{in}_{CL2}, \alpha^{out}_{CL2}, \alpha^{in}_{R2}, \alpha^{out}_{R2}\}$,
$\gamma_{R3} = \{\alpha^{in}_{CL3}, \alpha^{out}_{CL3}, \alpha^{in}_{R3}, \alpha^{out}_{R3}\}$,
$\gamma_{R4} = \{\alpha^{in}_{CL4}, \alpha^{out}_{CL4}, \alpha^{in}_{R4}, \alpha^{out}_{R4}\}$,
$\alpha^{in}_{CL} = \{R.send, CL.rec\}$,
$\alpha^{out}_{CL} = \{CL.send, R.rec\}$,
$\alpha^{in}_{R1} = \{R1.rec, R2.send\}$,
$\alpha^{out}_{R1} = \{R1.send, R2.rec\}$,
$\alpha^{in}_{R2} = \{R2.rec, R3.send\}$,
$\alpha^{out}_{R2} = \{R2.send, R3.rec\}$,
$\alpha^{in}_{R3} = \{R3.rec, R4.send\}$,
$\alpha^{out}_{R3} = \{R3.send, R4.rec\}$,
$\alpha^{in}_{R4} = \{R4.rec, R1.send\}$,
$\alpha^{out}_{R4} = \{R4.send, R1.rec\}$,
$\alpha^{tick} = (\{CL_1.tick, \ldots, CL_4.tick, R_1.tick, \ldots, R_4\}, true, skip)$.

For each $\alpha \in \gamma$, the priority rule π implies that the tick interaction α^{tick} has lower priority than any other interaction α. We export the tick port of the α^{tick} interaction to dynamically enable synchronization with other tick interactions. We will further refer to this interaction as $NoC.tick$.

4.3 Conclusion

In this chapter, we described the Abstract Model of HW Platform in BIP. A BIP Hardware Model should integrate both computation and communication constraints in a unified model. The computation constraints are added with the use of a Processor Scheduler Components and the profiling of the software processes with the corresponding timing delays as described in Chapter 7. The communication constraints are integrated with the use of *Cluster Compound* Components modeling the communication paths from PEs towards the platform memories and via versa. There are two types of Cluster used:

1. the *Shared Memory Cluster* using crossbar switch as a bus and shared memories.
2. the *Multi-banked Memory Cluster* using multiplexing interconnect as a bus and an efficient multi-banked memory.

The clusters include network interface to be capable to interact with off-cluster interconnect. The off-cluster interconnect modeled is a 4×4 network-on-chip. The NoC is implemented using *Router* Components modeling routing and arbitration policies. To compose the HW Platform models, we defined a library of atomic and compound components. These components can be parametrized, in order to model specific manycore platforms as we present in Sections 10.1 and 11.1. The correct-by-construction synthesis of the complete HW/SW model is analyzed in Chapter 6 considering the mapping of the SW application on the HW platform introduced in the next Chapter 5.

- Chapter 5 -

Binding BIP SW Model to HW Platforms

In the previous chapter, we specified the target HW platforms which we consider in our work. We defined the abstract HW platform models in BIP and all the components types needed for them. In this chapter, we introduce the mapping definition and the refinement of the software application model in BIP in order to conform with the mapping specification, thus accurately modeling the deployment on the HW platform. The refined software application model in BIP is obtained by a series of correct-by-construction transformations. We use a notion of trace equivalence, based on the principle of observational equivalence [Mil80], to prove the correctness of the refined model. The aim is to show the equivalence between the trace of the initial software application model and the final mixed SW/HW System model which we define in Chapter 6.

The chapter is structured in two sections. In the first section, we describe the mapping specification and in the second section, we define the refined software application model, the intermediate refinement steps and we prove the correctness of the transformed model.

5.1 Mapping Specification

Given an *Application-Software* and a *HW-Platform*, a *Mapping* associates the software processes to hardware processors and the software channels to hardware memories. Moreover, the mapping also defines the scheduling policies for processors, formally:

$$\begin{aligned} \textit{Mapping} &::= \textit{Mapping-Item}^{+} \ . \ \textit{Scheduling-Policy}^{+} \\ \textit{Mapping-Item} &::= \textit{SW-Process} \mapsto \textit{HW-Processor} \\ &\mid \textit{SW-Channel} \mapsto \textit{HW-Memory} \end{aligned}$$

As presented in Section 3.2.2 the *FIFO* Components contain behavior which controls the data buffer. In order to deploy the application software on the HW platform, we need a low level implementation model for the SW-Channels where the control and the data are dissociated and moreover, the *Write/Read* operations are no longer atomic, since they involve more than one component. As a result, we decompose the *FIFO* Component in BIP, which models the SW-Channels, into three new components: the *FIFO-Write*, the *FIFO-Read* and the *FIFO-Buffer* Component.

The *FIFO-Write* and *FIFO-Read* Components are used to model the *Write/Read* FIFO access routines which control the buffer indices, the buffer size and the data counters. Therefore, *the FIFO-Write* and *FIFO-Read* Components are binded with the *Process* connected to them, and will be referred to as *FIFO-Routine* Components. The *FIFO-Buffer* Component is used to model the store and load operations on the buffer. As a consequence

of the above decomposition, the *Write/Read* operations are no longer atomic, since they involve more than one component.

A more detailed *Mapping* associates a *FIFO-Routine* with the same hardware processor as the *Process* which is connected to and associates each *FIFO-Buffer* with a hardware memory.

5.2 Application-Software Model Refinement

Given an *Application-Software* model, the adjustment for connection with a hardware model is performed in a series of transformations:

1. Breaking the atomicity of *Write/Read* operations. In the BIP implementation of the *Application-Software*, the *Write/Read* are blocking operations. Since the next transformation decomposes the *SW-Channels* to three different components, we need to ensure the non-atomicity of the *Write/Read* operations, in order to preserve the blocking functionality.

2. Decomposing the *SW-Channels* into *Write/Read* FIFO access routines and data buffers to accurately map them to hardware processors and hardware memories.

3. Modifying *Process* and *FIFO-routines* Components to enable interaction with a hardware processor.

The above transformations refine the *Application-Software* model in a correct-by-construction manner and are described in the next Section 5.2.1.

5.2.1 Breaking Atomicity - Refinement

The first transformation consists in *breaking atomicity of write and read operations*. Every transition involving an input/output port x is split into two transitions, labeled by fresh ports, respectively x_b (i.e., *x-begin*) and x_e (i.e., *x-end*). This is obtained by adding new control locations for each read/write operation in the behavior of a process and a FIFO Component. The transformed components are called *Refined* and their formal definition is given below.

34 Definition (Refined Process Component)
Given a Process Component $G = (L, P, X, \mathcal{T})$ as defined in Definition 15, we define the Refined Process Component $G^R = (L^R, X, P^R, \mathcal{T}^R)$ with broken atomicity on the Read, Write transitions, where the behavior is described in Figure 5.1:

- $L^R = L \cup \{\ell'' | \tau \in T, port(\tau) \neq \beta\}$, *the set of control locations,*
- $P^R = \{w_b, w_e | w \in P_w\} \cup \{r_b, r_e | r \in P_r\}$, *the set of ports, where w_b, r_e are paired with wr_data and rd_data respectively,*
- $\mathcal{T}^R = \{\tau_b = (\ell, p_b, true, skip, \ell''), \tau_e = (\ell'', p_e, true, skip, \ell') | \tau = (\ell, p, true, skip, \ell') \in \mathcal{T}\} \cup \{\tau = (\ell, \beta, g, f, \ell') | \tau \in \mathcal{T}\}$, *the set of transitions.*

That is, we replace each transition τ labeled by a port p with two consecutive transitions τ_b, τ_e labeled respectively by ports p_b, p_e. The first initiates the communication (write or read) and is followed by the other that awaits for its completion. A transitive location ℓ'' is added between transitions τ_b and τ_e. Internal β transitions are kept unchanged.

14 Example (Refined Producer Component)
Given the Producer Component defined in Example 6, we illustrate in Figure 5.1 the Refined Producer Component based on the Definition 34 of the Refined Process Component.

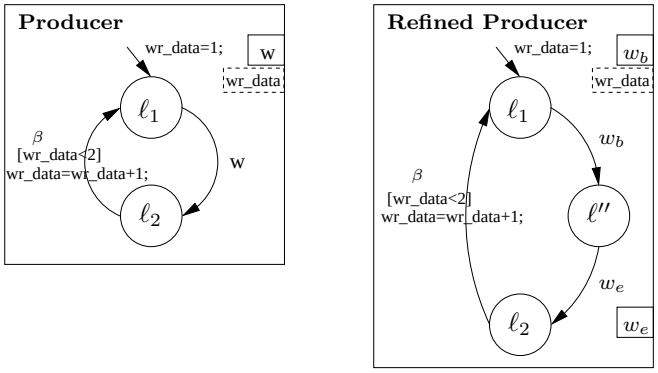


Figure 5.1: *Model of the Producer Component (left) and model of the Refined Producer Component in BIP (right).*

35 Definition (Refined FIFO Component)
Given a FIFO Component F as defined in Definition 16, we define a Refined FIFO Component as a BIP component $F^R = (L_{F^R}, X_F, P_{F^R}, \mathcal{T}_{F^R})$, where the behavior is described in Figure 5.2:

- $L_{F^R} = \{\ell, \ell_w, \ell_r\}$,
- $X_{F^R} = \{wr\_data, rd\_data, buff, i, j, k, count\}$,
- $P_{F^R} = \{w_b, w_e, r_b, r_e\}$, *where* w_b *and* r_b *paired with* $\{wr\_data\}$ *and* $\{rd\_data\}$ *respectively.*
- $\mathcal{T}_{F^R} = \{$
  $\tau_{w_b} = (\ell, w_b, g_w, f_{wm}, \ell_w)$,
  $\tau_{w_e} = (\ell_w, w_e, true, f_{wn}, \ell)$,
  $\tau_{r_b} = (\ell, r_b, g_r, f_{rm}, \ell_r)$,
  $\tau_{r_e} = (\ell_r, r_e, true, f_{rn}, \ell)$,
  $\}$.

We replace each transition τ labeled by a port p with two consecutive transitions τ_b, τ_e labeled respectively by ports p_b, p_e. After evaluating the guard, the first transition executes the Write or Read actions on the buffer. The second, increases or (resp. decreases) the FIFO counter and adjusts the buffer indices.

36 Definition (Refined Process Network Composition)
For a Process Network $N = \gamma(G_1, \ldots, G_n, F_1, \ldots, F_k)$ as defined in Definition 17, we define the Refined Process Network $N^R = \gamma^R(G_1^R, \ldots, G_n^R, F_1^R, \ldots, F_k^R)$, where G^R Components are defined in Definition 34, F^R Components are defined in Definition 35 and the set of interactions γ^R is defined as:
$\gamma^R = \{\alpha_b^w, \alpha_e^w | \alpha^w \in \gamma\} \cup \{\alpha_b^r, \alpha_e^r | \alpha^r \in \gamma\}$,

where, if $\alpha^w = (\{G.w, F.w\}, true, f_{\alpha^w})$ *then:*	$\alpha_b^w = (\{G^R.w_b, F^R.w_b\}, true, f_{\alpha^w})$
	$\alpha_e^w = (\{G^R.w_e, F^R.w_e\}, true, skip)$
and if $\alpha^r = (\{G.r, F.r\}, true, f_{\alpha^r})$ *then:*	$\alpha_b^r = (\{G^R.r_b, F^R.r_b\}, true, skip)$
	$\alpha_e^r = (\{G^R.r_e, F^R.r_e\}, true, f_{\alpha^r})$

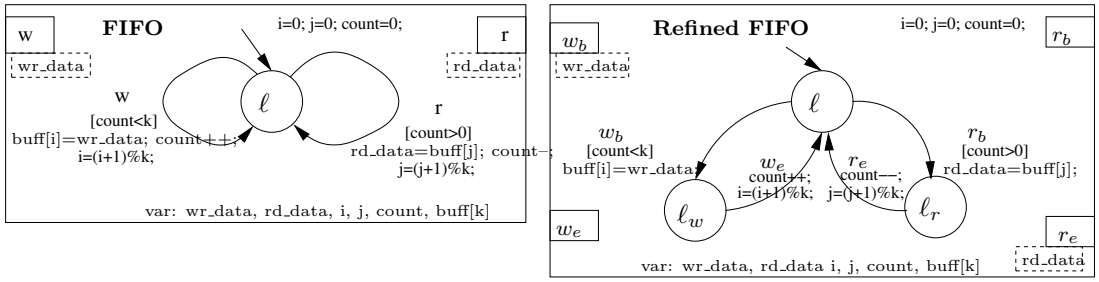


Figure 5.2: *Model of the FIFO channel (left) and the Refined FIFO channel in BIP (right).*

We split every interaction α into two consecutive interactions α_b, α_e, which depend on the initial interaction type. If it is a Write interaction α^w, the α_b^w will preserve the behavior of the initial α^w, and respectively for Read interactions α^r, the α_e^r transition will execute the data transfer.

15 Example (Producer-Consumer Refined Composition)
Given a Producer-Consumer Process Network PC as defined in Example 8, we break the atomicity of the interactions which belong to PC and we define the Refined Process Network $PC^R = \gamma^R(G_P^R, F^R, G_C^R)$, *where* $\gamma^R = \{\alpha_b^w, \alpha_b^r, \alpha_e^w, \alpha_e^r\}$,
$\alpha_b^w = (\{G_P^R.w_b, F^R.w_b\}, true, f_{\alpha^w})$, $\alpha_b^r = (\{F^R.r_b, G_P^R.r_b\}, true, f_{\alpha^r})$,
$\alpha_e^w = (\{G_P^R.w_e, F^R.w_e\}, true, skip)$, $\alpha_e^r = (\{F^R.r_e, G_P^R.r_e\}, true, skip)$
The Producer-Consumer Refined Composition in BIP is illustrated in Figure 5.3.

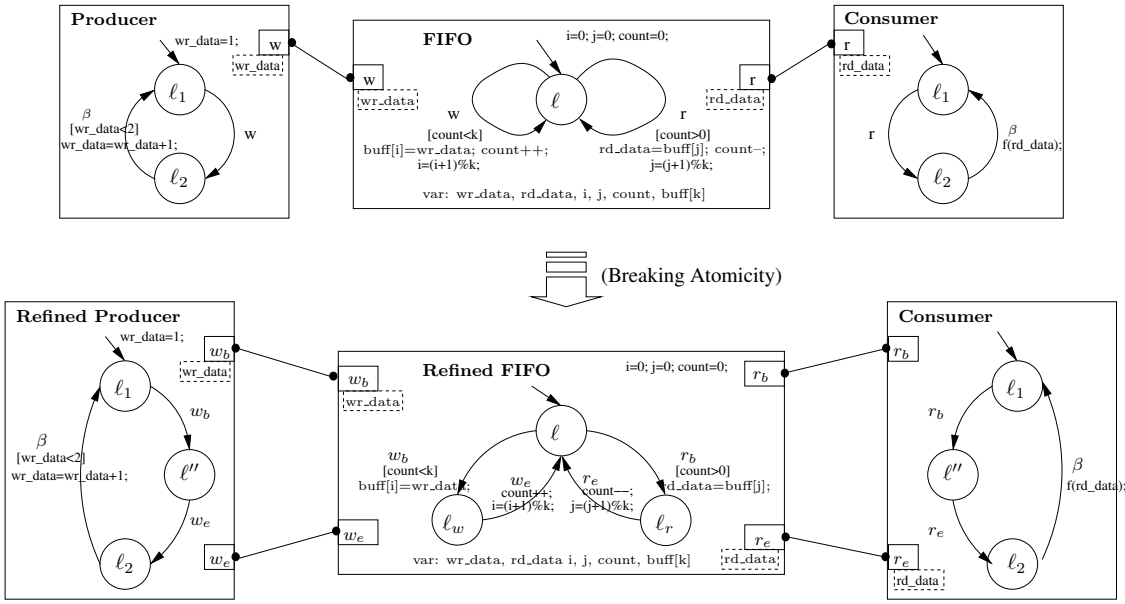


Figure 5.3: *Producer-Consumer Refined Composition in BIP*

The set of traces is represented as the interleavings in Figure 5.4.

Correctness

In this section, we prove trace equivalence between the above *Refined Process Network Composition* and the initial *Process Network*.

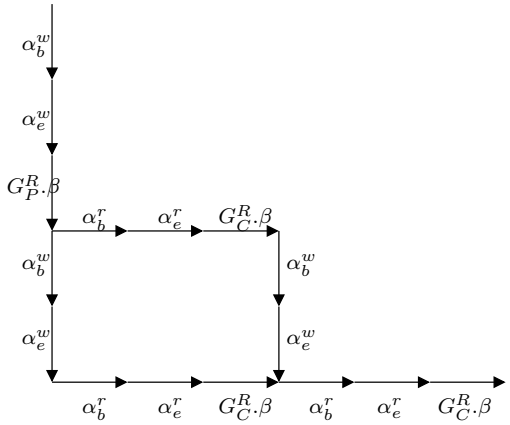


Figure 5.4: *Traces of the Refined Producer-Consumer Process Network.*

37 Definition (Complete Run)
For a given composition $C = \gamma(B_1, B_2, \ldots, B_N)$, a run θ is called ***complete****, if the number of begin events equals the end events.*

38 Definition (Trace Restriction of a Refined Process Network N^R)
For a Refined Process Network N^R:

- *The restriction of a run θ^R to end events is defined by:*

$$trace^R(\theta^R) = \begin{cases} \epsilon & ,\text{if } \theta^R : \epsilon \\ \beta.trace^R(\theta'^R) & ,\text{if } \theta^R : q \xrightarrow{\beta} q'.\theta'^R \\ \alpha.trace^R(\theta'^R) & ,\text{if } \theta^R : q \xrightarrow{\alpha_b} q' \xrightarrow{\alpha_e} q''.\theta'^R, \end{cases}$$

  where $\alpha \in \gamma$ and α_b, α_e use the same FIFO.

Let N^R be the Refined Process Network for N.

1 Theorem
For each run θ in the Process Network N:
$\theta : q_0 \xrightarrow{a_1}_\gamma q_1 \xrightarrow{a_2}_\gamma q_2 \xrightarrow{a_3}_\gamma \ldots \xrightarrow{a_n}_\gamma q_n$
There exists a complete run θ^R in the Refined Process Network N^R:
$\theta^R : q_0 \xrightarrow{a_1^R}_{\gamma^R} q_1^R \xrightarrow{a_2^R}_{\gamma^R} q_2^R \xrightarrow{a_3^R}_{\gamma^R} \ldots \xrightarrow{a_m^R}_{\gamma^R} q_m^R$
such that:

- $q_n = q_m^R$
- $trace(\theta) = trace^R(\theta^R)$

2 Theorem
For each complete run θ^R in the Refined Process Network N^R:
$\theta^R : q_0 \xrightarrow{a_1^R}_{\gamma^R} q_1^R \xrightarrow{a_2^R}_{\gamma^R} q_2^R \xrightarrow{a_3^R}_{\gamma^R} \ldots \xrightarrow{a_m^R}_{\gamma^R} q_m^R$
There exists a run θ in the Process Network N:
$\theta : q_0 \xrightarrow{a_1}_\gamma q_1 \xrightarrow{a_2}_\gamma q_2 \xrightarrow{a_3}_\gamma \ldots \xrightarrow{a_n}_\gamma q_n$
such that:

- $q_n = q_m^R$
- $trace^R(\theta^R) = trace(\theta)$

1 Lemma
For each complete run θ in the Refined Process Network N^R:
$\theta : q_0 \rightarrow \cdots \rightarrow q_n$
There exists a complete run θ' in the Refined Process Network N^R starting from q_0:
$\theta' : q_0 \rightarrow \cdots \rightarrow q_{n'}$
such that:

- $q_n = q_{n'}$
- *in θ' any begin interaction is followed immediately by the corresponding end interaction.*

1 Proof (Theorem 1)
We consider a run $\theta : q_0 \xrightarrow{\alpha^1}_{\gamma} q_1 \xrightarrow{\alpha^2}_{\gamma} ... \xrightarrow{\alpha^n}_{\gamma} q_n$ in a Process Network N. Based on the Definition 36 of the Refined Process Network, we break the atomicity of the Write and Read interactions and we replace them with begin/end-Write/Read interactions. Since we do not modify the β transitions we consider them hidden from the run. Thus, we obtain a run θ^R : $q_0 \xrightarrow{\alpha^1_b}_{\gamma^R} q_1^R \xrightarrow{\alpha^1_e}_{\gamma^R} q_2^R \xrightarrow{\alpha^2_b}_{\gamma^R} q_3^R \xrightarrow{\alpha^2_e}_{\gamma^R} ... \xrightarrow{\alpha^R_m}_{\gamma^R} q_m^R$, which belongs in the Refined Process Network N^R. According to the Definition 36 of the Refined Process Network Composition as it is illustrated in the Example 15 and the Figure 5.3, we conclude that $q_n = q_m^R$. In addition, we have $trace(\theta) = \alpha^1.\alpha^2 \ldots \alpha^n$ and $trace(\theta^R) = \alpha^1_b.\alpha^1_e.\alpha^2_b.\alpha^2_e \ldots \alpha^n_b.\alpha^n_e$. Based on the Definition 38, we restrict the trace θ^R such that $trace^R(\theta^R) = \alpha^1.\alpha^2 \ldots \alpha^n$. So, we conclude that $trace(\theta) = trace^R(\theta^R)$.

1 Proposition
Let a complete run $\theta^R : q_0 \xrightarrow{\alpha^1}_{\gamma^R} q_1 \xrightarrow{\alpha^2}_{\gamma^R} q_2.\theta'$, which belongs in the Refined Process Network N^R and $trace(\theta^R) = \alpha_1.\alpha_2.trace(\theta')$. Suppose that the interactions α^1, α^2 modify independent variables which do not impact the evaluation of any transition guard. Thus, we can reverse their execution order:
$\theta^R : q_0 \xrightarrow{\alpha^2}_{\gamma^R} q_1' \xrightarrow{\alpha^1}_{\gamma^R} q_2'.\theta'$ *and* $trace(\theta^R) = \alpha_2.\alpha_1.trace(\theta')$,
preserving the equality of the ending states $q_2' = q_2$. The proof is evident and thus not required.

2 Proof (Lemma 1)
Let a complete run $\theta^R : q_0 \xrightarrow{\alpha^1}_{\gamma^R} q_1 \xrightarrow{\alpha^2}_{\gamma^R} q_2.\theta'$, which belongs in the Refined Process Network N^R and $trace(\theta^R) = \alpha_1.\alpha_2.trace(\theta')$. Based on Proposition 1 we can reverse the execution order of interactions α^1, α^2: $\theta^R : q_0 \xrightarrow{\alpha^2}_{\gamma^R} q_1' \xrightarrow{\alpha^1}_{\gamma^R} q_2'.\theta'$.

As we can see in Example 15 the traces of a Refined Process Network contain $\alpha^w_b, \alpha^w_e, \alpha^r_b, \alpha^r_e$ interactions. The order of execution of begin and end events is respected provided that the Write/Read operate on the same FIFO. If the FIFOs are different we can have interleavings of Write/Read and begin/end events. However, based on the Proposition 1 we can recursively relocate the execution order of begin events of the interactions which operate on different FIFOs, so that they are followed immediately by their corresponding end interaction. Thus, we can conclude that:

- *the ending state of the new run is equal to the ending state of the old run*
- *in $\theta^{R'}$ any begin interaction is followed immediately by the corresponding end interaction*

3 Proof (Theorem 2)
We consider a run $\theta^R : q_0 \xrightarrow{\alpha^1}_{\gamma^R} q_1 \xrightarrow{\alpha^2}_{\gamma^R} \dots \xrightarrow{\alpha^m}_{\gamma^R} q_m$ *in a Refined Process Network* N^R. *Based on Lemma 1, we consider* θ^R *in* N^R *such that any begin is immediately followed by the corresponding end interaction* $\theta'^R : q_0 \xrightarrow{\alpha^1_b}_{\gamma^R} q'_1 \xrightarrow{\alpha^1_e}_{\gamma^R} \dots \xrightarrow{\alpha^m_b}_{\gamma^R} q'_{m-1} \xrightarrow{\alpha^m_e}_{\gamma^R} q'_m$. *As a consequence of Lemma 1, we have* $q^m = q'^m$.

Additionally, based on the Definition 36 of the Refined Process Network as it is illustrated in the Example 15 and the Figure 5.3, we replace the consecutive begin/end interactions with a single Write or Read interaction. Thus, we obtain a run $\theta : q_0 \xrightarrow{\alpha^1}_{\gamma} q_2 \xrightarrow{\alpha^2}_{\gamma} \dots \xrightarrow{\alpha^m}_{\gamma} q_m$ *such that* θ *belongs in Process Network* N.

We have, $trace(\theta) = \alpha^1.\alpha^2 \dots \alpha^m$ *and* $trace(\theta^R) = \alpha^1_b.\alpha^1_e.\alpha^2_b.\alpha^2_e \dots \alpha^m_b.\alpha^m_e$. *Based on the restriction of traces in Definition 38, we also have* $trace^R(\theta^R) = \alpha^1.\alpha^2 \dots \alpha^m$. *So, we conclude that* $trace^R(\theta^R) = trace(\theta)$.

5.2.2 FIFO Decomposition - Refinement

Every *SW-Channel* in the application software is *replaced* by a composition of *FIFO-Write*, *FIFO-Read* and a *FIFO-Buffer* atomic components (see Figure 5.5). The two former components represent the control part of the software channel, that is, the hardware dependent software routines implementing the *read/write* operations. The latter component simply represents the buffer of data.

All the three components *FIFO-Read*, *FIFO-Write*, *FIFO-Buffer* are predefined BIP components and belong to the BIP hardware dependent software library. The *FIFO-Read* Component, illustrated in Figure 5.5, implements the *read* operation on channels. It has the ports r_b (*Read-Begin*), r_e (*Read-End*) for its interaction with a software process *read* operation, and ports mr_b (*Memory Read-Begin*), mr_e (*Memory Read-End*) for its interaction with the buffer. The *FIFO-Write* Component implements the *write* action in a similar manner.

Let us notice that the two routines, *FIFO-Write* and *FIFO-Read*, require extra synchronization with each other in order to maintain a coherent value for the used space within the buffer. This is realized by using strong synchronization between two control ports, w_u (*Write Update*)and r_u (*Read Update*) and the *Memory-End* interactions.

39 Definition (FIFO-Write Component)
We define the FIFO-Write Component as a BIP Component $FW = (L_{FW}, X_{FW}, P_{FW}, \mathcal{T}_{FW})$, *where the behavior is described in Figure 5.5.*

- $L_{FW} = \{\ell_1, \ell_2, \ell_3, \ell_4\}$,
- $X_{FW} = \{wr\_data, i, k, count, buff\}$,
- $P_{FW} = \{w_b, mw_b, mw_e, w_u, w_e\}$, *where* w_b *associated with* $\{wr\_data\}$,
- $\mathcal{T}_{FW} = \{$
  $\tau_{w_b} = (\ell_1, w_b, true, skip, \ell_2)$,
  $\tau_{mw_b} = (\ell_2, mw_b, g_w, skip, \ell_3)$, *with* g_w : $[count < N]$,
  $\tau_{mw_e} = (\ell_3, mw_e, true, f_{wn}, \ell_4)$, *with* f_{wn} : $count++;\ i = (i+1)\%N;\ buff[i] = data;$,
  $\tau_{w_e} = (\ell_4, w_e, true, skip, \ell_1)$,
  $\{\tau_u = (\ell, w_u, true, f_{rc}, \ell) | \ell \in L_{FW}\}$, *with* f_{rn} : $count--$;
  $\}$.

40 Definition (FIFO-Read Component)
We define the FIFO-Read Component as a BIP component $FR = (L_{FR}, X_{FR}, P_{FR}, \mathcal{T}_{FR})$, *where the behavior is described in Figure 5.5.*

- $L_{FR} = \{\ell_1, \ell_2, \ell_3, \ell_4\}$,
- $X_{FR} = \{rd\_data, j, k, count, buff\}$,
- $P_{FR} = \{r_b, mr_b, mr_e, r_u, r_e\}$, *where* r_e *associated with* $\{rd\_data\}$,
- $\mathcal{T}_{FR} = \{$
  $\tau_{r_b} = (\ell_1, r_b, true, skip, \ell_2)$,
  $\tau_{mr_b} = (\ell_2, mr_b, g_r, skip, \ell_3)$, *with* $g_r : [count > 0]$,
  $\tau_{mr_e} = (\ell_3, mr_e, true, f_{rn}, \ell_4)$, *with* $f_{rn} : count--; j = (j+1)\%N; data = buff[j];$,
  $\tau_{r_e} = (\ell_4, r_e, true, skip, \ell_1)$,
  $\{\tau_u = (\ell, r_u, true, f_{wc}, \ell) | \ell \in L_{FR}\}$, *with* $f_{rn} : count++$;
  $\}$.

The *FIFO-Buffer* represents a *passive* component modeling the data storage. It has ports op_b (*Operation Begin*)) and op_e *Operation End* for performing the write/read operations.

41 Definition (FIFO-Buffer Component)
We define the FIFO-Buffer Component as a BIP component $FB = (L_{FB}, X_{FB}, P_{FB}, \mathcal{T}_{FB})$, *where the behavior is described in Figure 5.5.*

- $L_{FB} = \{\ell_1, \ell_2, \}$,
- $X_{FB} = \emptyset$,
- $P_{FB} = \{op_b, op_e\}$,
- $\mathcal{T}_{FB} = \{$
  $\tau_{op_b} = (\ell_1, op_b, true, skip, \ell_2)$,
  $\tau_{op_e} = (\ell_2, op_e, true, skip, \ell_1)$.
  $\}$.

42 Definition (Split-FIFO Process Network Composition)
Given a Refined Process Network N^R *as defined in Definition 36, we define a Split-FIFO Process Network* $N^D = \gamma^D(G_1^R, \ldots, G_n^R, FW_1, FR_1, FB_1, \ldots, FW_k, FR_k, FB_k)$, *where the set of interactions* γ^D *is defined as:*
$\gamma^D = \{\alpha_b^w, \alpha_b^{mw}, \alpha_e^{mw}, \alpha_e^w | \alpha_b^w \in \gamma^R\} \cup \{\alpha_b^r, \alpha_b^{mr}, \alpha_e^{mr}, \alpha_e^r | \alpha_b^r \in \gamma^R\}$,
$\alpha_b^{mw} = (\{FW.mw_b, FB.w_b\}, true, skip)$,
$\alpha_e^{mw} = (\{FW.mw_e, FB.w_e, FR.r_u\}, true, skip)$,
$\alpha_b^{mr} = (\{FR.mr_b, FB.r_b\}, true, skip)$,
$\alpha_e^{mr} = (\{FR.mr_e, FB.r_e, FW.w_u\}, true, skip)$.
The write/read operations are executed in two steps. The interactions $\alpha_b^{mw}, \alpha_b^{mr}$ *Memory Write/Read Begin and* $\alpha_e^{mw}, \alpha_e^{mr}$ *Memory Write/Read End are added to synchronize the FIFO Buffer with the FIFO Routines.*

16 Example (Producer-Consumer Split-FIFO Composition)
Given a Refined Producer-Consumer Process Network PC^R, *we decompose the FIFO* F^R *Component into three new components FW, FR, FB such that we obtain* $PC^D =$

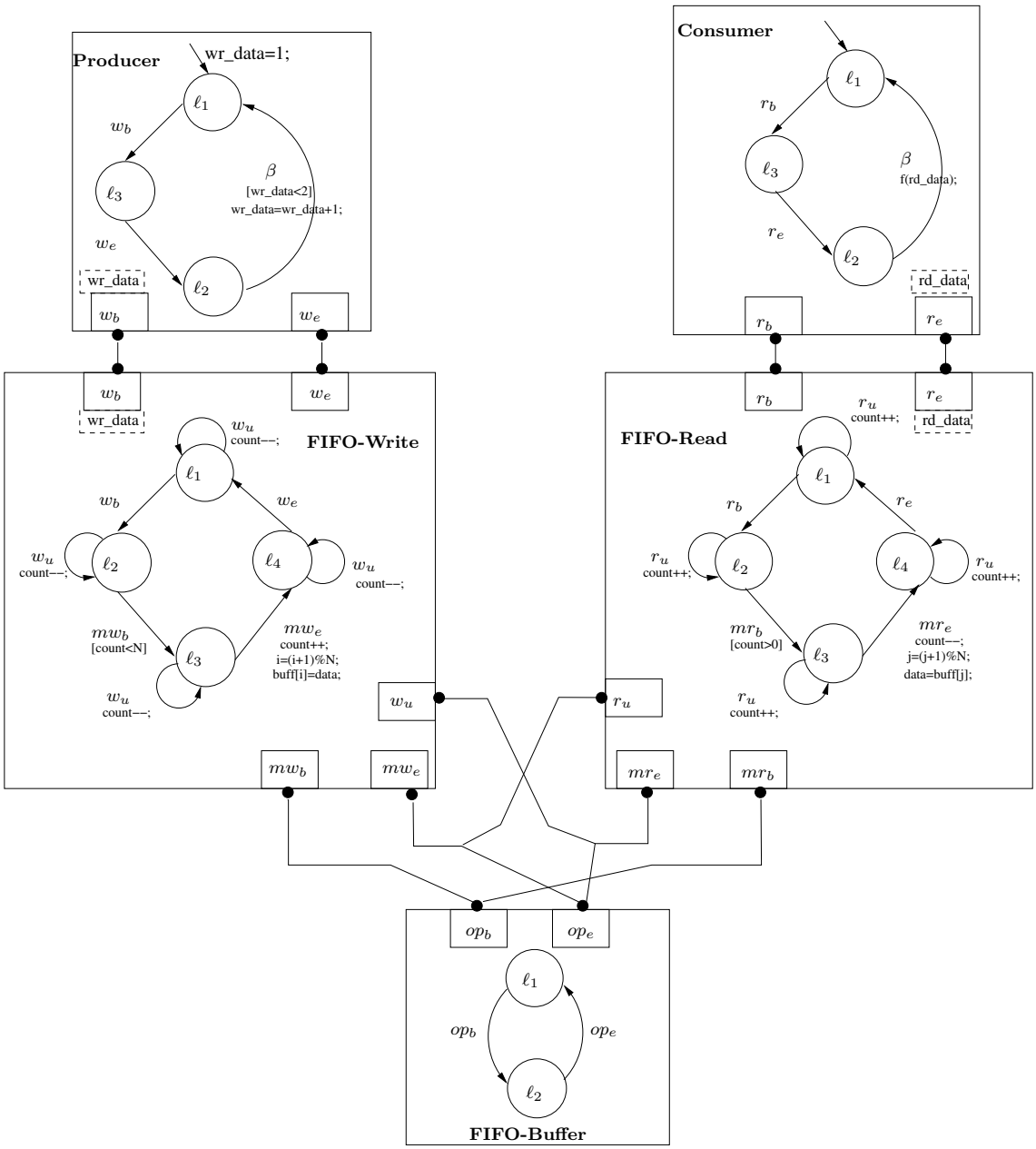


Figure 5.5: *Model of the Producer-Consumer Split-FIFO System Model in BIP*

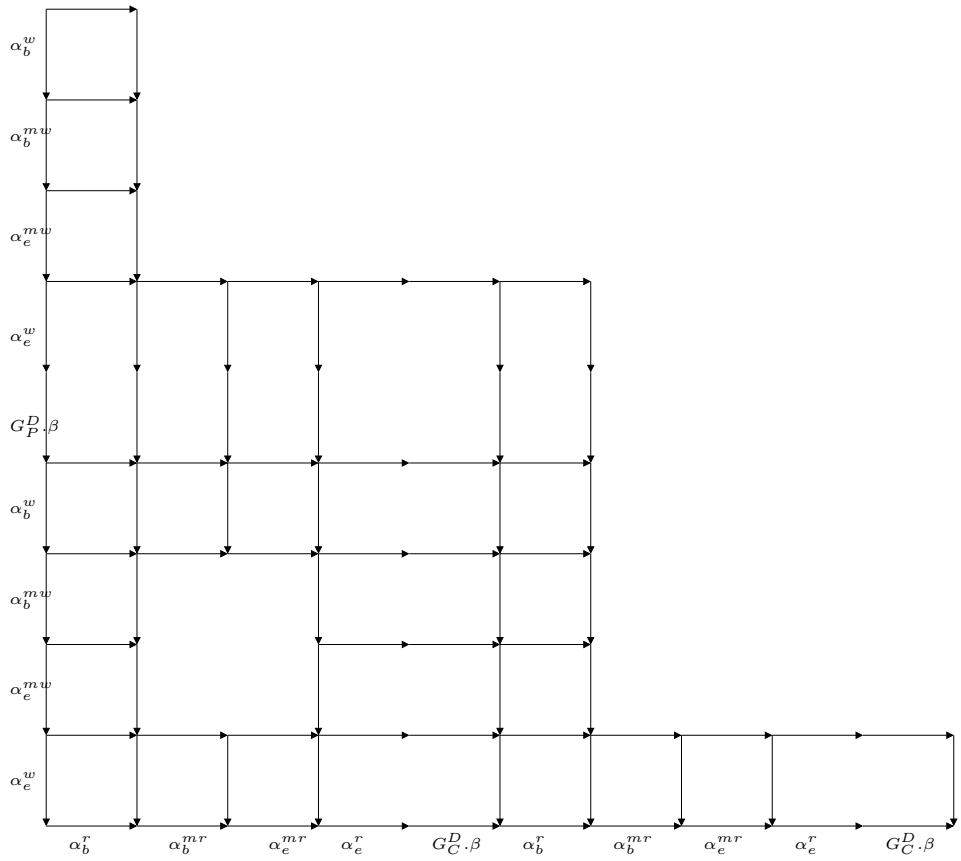


Figure 5.6: *Traces of Split-FIFO Producer-Consumer Process Network.*

$\gamma^D(G_P^R, FW, FR, FB, G_C^R)$, *where* $\gamma^D = \{\alpha_b^w, \alpha_b^r, \alpha^m w_b, \alpha^m w_e, \alpha_b^r, \alpha_e^w, \alpha_e^r, \alpha^m r_b, \alpha^m r_e\}$, *as illustrated in Figure 5.5, with*
$\alpha_b^{mw} = (\{FW.mw_b, FB.w_b\}, true, skip)$,
$\alpha_e^{mw} = (\{FW.mw_e, FB.w_e, FR.r_u\}, true, skip)$,
$\alpha_b^{mr} = (\{FR.mr_b, FB.r_b\}, true, skip)$,
$\alpha_e^{mr} = (\{FR.mr_e, FB.r_e, FW.w_u\}, true, skip)$.
The set of traces is represented as the interleavings in Figure 5.6.

Correctness

In the current section, we prove the trace equivalence of a Split-FIFO Process Network N^D with a Refined Process Network N^R. That is, the composition is a refined model of the *SW-Channel* which fully preserves the input/output behavior of the software channel.

43 Definition (Trace Restriction of a Split-FIFO Process Network N^D)
For a Split-FIFO Process Network N^D, the restriction of a trace of a run θ^D to non-memory events is defined by:

$$trace^D(\theta^D) = \begin{cases} \epsilon & ,\text{if } \theta^D : \epsilon \\ \beta.trace^D(\theta'^D) & ,\text{if } \theta^D : q \xrightarrow{\beta} q'.\theta'^D \\ trace^D(\theta_A).trace^D(\theta_B) & ,\text{if } \theta^D : \theta_A.q \xrightarrow{\alpha_b^m} q' \xrightarrow{\alpha_e^m} q''.\theta_B \end{cases}$$

Let N^D be the Split-FIFO Process Network for N^R.

3 Theorem
For each run θ^R in the Refined Process Network N^R:
$\theta^R : q_0 \xrightarrow{a_1^R}_{\gamma^R} q_1^R \xrightarrow{a_2^R}_{\gamma^R} q_2^R \xrightarrow{a_3^R}_{\gamma^R} \cdots \xrightarrow{a_n^R}_{\gamma^R} q_n^R$
There exists a complete run θ^D in the Split-FIFO Process Network N^D:
$\theta^D : q_0 \xrightarrow{a_1^D}_{\gamma^D} q_1^D \xrightarrow{a_2^D}_{\gamma^D} q_2^D \xrightarrow{a_3^D}_{\gamma^D} \cdots \xrightarrow{a_m^D}_{\gamma^D} q_m^D$
such that:

- $q_n^R = q_m^D$, *where q_m^D concerns the Process Components,*
- $trace(\theta^R) = trace^D(\theta^D)$

4 Theorem
For each complete run θ^D in the Split-FIFO Process Network N^D:
$\theta^D : q_0 \xrightarrow{a_1^D}_{\gamma^D} q_1^D \xrightarrow{a_2^D}_{\gamma^D} q_2^D \xrightarrow{a_3^D}_{\gamma^D} ... \xrightarrow{a_m^D}_{\gamma^D} q_m^D$
There exists a run θ^R in the Refined Process Network N^R:
$\theta : q_0 \xrightarrow{a_1^R}_{\gamma^R} q_1^R \xrightarrow{a_2^R}_{\gamma^R} q_2^R \xrightarrow{a_3^R}_{\gamma^R} \cdots \xrightarrow{a_n^R}_{\gamma^R} q_n^R$
such that:

- $q_n^R = q_m^D$, *where q_m^D concerns the Process Components,*
- $trace^D(\theta^D) = trace(\theta^R)$

2 Lemma
For each complete run θ in the Split-FIFO Process Network N^D:
$\theta : q_0 \rightarrow \cdots \rightarrow q_n$
There exists a complete run θ' in the Split-FIFO Process Network N^D starting from q_0:
$\theta' : q_0 \rightarrow \cdots \rightarrow q_{n'}$
such that:

- $q_n = q_{n'}$
- *in θ' any α_b^w or α_b^r interaction is followed immediately by the series of interactions α_b^{mw}, α_e^{mw}, α_e^w and α_b^{mr}, α_e^{mr}, α_e^r respectively.*

4 Proof (Theorem 3)
We consider a complete run θ^R in a Refined Process Network N^R. Based on Lemma 1, we re-order any begin interaction so that it is immediately followed by the corresponding end interaction. Since we do not consider the β transitions, we have $\theta^R : q_0 \xrightarrow{\alpha_b^1}_{\gamma^R} q_1^R \xrightarrow{\alpha_e^1}_{\gamma^R} q_2^R \xrightarrow{\alpha_b^2}_{\gamma^R} q_3^R \xrightarrow{\alpha_e^2}_{\gamma^R} ... \xrightarrow{\alpha_e^n}_{\gamma^R} q^R$. We refine the FIFO Component by replacing it with FIFO-Write, FIFO-Read and FIFO-Buffer Components. We obtain a run $\theta^D : q_0 \xrightarrow{\alpha_b^1}_{\gamma^D} q_1^D \xrightarrow{\alpha_b^{m1}}_{\gamma^D} q_2^D \xrightarrow{\alpha_e^{m1}}_{\gamma^D} q_3^D \xrightarrow{\alpha_e^1}_{\gamma^D} q_4^D \xrightarrow{\alpha_b^2}_{\gamma^D} q_5^D \xrightarrow{\alpha_b^{m2}}_{\gamma^D} q_6^D \xrightarrow{\alpha_e^{m2}}_{\gamma^D} q_7^D \xrightarrow{\alpha_e^2}_{\gamma^D} ... \xrightarrow{\alpha_e^n}_{\gamma^D} q^D$. According to the Definition 42 of the Split-FIFO Process Network as it is illustrated in the Example 16 and the Figure 5.5, we have $q^R = q^D$, where q^D concerns the Process Components.

In addition, we have

$$trace(\theta^R) = \alpha_b^1.\alpha_e^1.\alpha_b^2.\alpha_e^2 \ldots \alpha_b^n.\alpha_e^n$$

and

$$trace(\theta^D) = \alpha_b^1.\alpha_b^{m1}.\alpha_e^{m1}.\alpha_e^1.\alpha_b^2.\alpha_b^{m2}.\alpha_e^{m2}.\alpha_e^2 \ldots \alpha_e^n$$

Based on the Definition 43, we restrict the trace θ^D *such that*

$$trace^D(\theta^D) = \alpha_b^1.\alpha_e^1.\alpha_b^2.\alpha_e^2 \ldots \alpha_b^n.\alpha_e^n$$

So, we conclude that $trace(\theta^R) = trace^D(\theta^D)$.

5 Proof (Lemma 2)
Let a complete run $\theta^D : q_0 \xrightarrow{\alpha^1}_{\gamma^R} q_1 \xrightarrow{\alpha^2}_{\gamma^R} q_2.\theta'$, *which belongs in the Split-FIFO Process Network* N^R *and* $trace(\theta^D) = \alpha_1.\alpha_2.trace(\theta')$. *Based on Proposition 1 we can reverse the execution order of interactions* α^1, α^2*:*
$\theta^D : q_0 \xrightarrow{\alpha^2}_{\gamma^R} q_1' \xrightarrow{\alpha^1}_{\gamma^R} q_2'.\theta'$.

As we can see in Example 15 the traces of the Split-FIFO Process Network contain $\alpha_b^w, \alpha_b^{mw}, \alpha_e^{mw}, \alpha_e^w, \alpha_b^r, \alpha_b^{mr}, \alpha_e^{mr}, \alpha_e^r$ *interactions. If interleavings of the above interactions occur, we can recursively relocate the execution order according to Proposition 1. So, any* α_b^w *or* α_b^r *interaction is followed immediately by the series of interactions* α_b^{mw}, α_e^{mw}, α_e^w *and* α_b^{mr}, α_e^{mr}, α_e^r *respectively. Thus, we can conclude that:*

- *the ending state of the new run is equal to the ending state of the old run*
- *in* $\theta^{d'}$ *any* α_b *interaction is followed immediately by the series of* α_b^m, α_e^m, α_e *interactions.*

6 Proof (Theorem 4)
We consider a run $\theta^D : q_0 \xrightarrow{\alpha^D}_{\gamma^R} q_1 \xrightarrow{\alpha^2}_{\gamma^D} \ldots \xrightarrow{\alpha^k}_{\gamma^D} q_k$ *in a Split-FIFO Process Network* N^D. *Based on Lemma 2, we consider* θ'^D *in* N^D *such that any* α_b *interaction is followed immediately by the series of* $\alpha_b^m, \alpha_e^m, \alpha_e$ *interactions such that* $\theta'^D : q_0 \xrightarrow{\alpha_b^1}_{\gamma^D} q_1' \xrightarrow{\alpha_b^{m1}}_{\gamma^D} q_2' \xrightarrow{\alpha_e^{m1}}_{\gamma^D} q_3' \xrightarrow{\alpha_e^1}_{\gamma^D} \ldots \xrightarrow{\alpha_e^k} q'^k$. *As a consequence of Lemma 2, we have* $q^k = q'^k$.

Additionally, based on the Definition 42 of the Split-FIFO Process Network as it is illustrated in the Example 16 and the Figure 5.5, we replace the series of α_b, α_b^m, α_e^m, α_e *interactions with the couple of* α_b, α_e *interactions. Thus, we obtain a run* $\theta^R : q_0 \xrightarrow{\alpha_b^1}_{\gamma^D} q_1 \xrightarrow{\alpha_e^1}_{\gamma^D} \ldots \xrightarrow{\alpha_e^k} q^k$ *such that* θ^R *belongs in the Refined Process Network* N^R.

We have,

$$trace(\theta^R) = \alpha_b^1.\alpha_e^1.\alpha_b^2.\alpha_e^2 \ldots \alpha_b^k.\alpha_e^k$$

and

$$trace(\theta^D) = \alpha_b^1.\alpha_b^{m1}.\alpha_e^{m1}.\alpha_e^1.\alpha_b^2.\alpha_b^{m2}.\alpha_e^{m2}.\alpha_e^2 \ldots \alpha_e^{mk}.\alpha_e^k$$

Based on the Definition 43, we restrict the trace θ^D *such that*

$$trace^D(\theta^D) = \alpha_b^1.\alpha_e^1.\alpha_b^2.\alpha_e^2 \ldots \alpha_b^n.\alpha_e^n$$

So, we conclude that $trace^D(\theta^D) = trace(\theta^R)$.

5.2.3 Mutual Exclusion and Computation Time Refinement

Several processes, together with their associated FIFO access routines, are potentially mapped on the same hardware processor and must use it in mutual exclusion. The ports *acq* and *rel* are added for interaction with the processor scheduler. The port *acq* is used for acquiring and *rel* is for releasing the processor. The first time when a process acquires the processor is the start of its execution. It releases the processor on its termination.

Processes that are mapped on the same processor cannot simultaneously utilize the processor's computational resource. Thus, we use a mutual exclusion scheduler with *Acquire* and *Release* primitives. We add to the process components Acquire and Release scheduling primitives as ports associated respectively with a new transition to the initial ℓ_{ini} control location and with a new transition from each control location in L_{fin}. ℓ_{fin} states of the process component. The *Read* and *Write* operations of the processes can be blocked due to limited places of the FIFO buffer. Considering the above and the fact that the FIFO mechanism is managed by the FIFO *Read/ Write* Components, we add *Acquire* and *Release* scheduling primitives in these components too. The modified process and FIFO Components are defined below.

In addition, all processes contain computational hidden transitions β. Every β transition is profiled with a computational cost. In every transition β a *delay* variable is filled with the computational cost of the executed f_β function. The methods used to obtain the profiling value are explained in details in Chapter 7. This computational delay should be pushed to the processor scheduler, which is responsible for modeling the computational resource of the processor. We add the *send_d* (*Send Delay*) and *next* primitives to the process components. The *send_d* sends the delay to the processor and the *next* allows the process to continue to the next computational or communication step of the process. The *send_d* and *next* transitions are placed immediately after the β transitions. The Scheduled Process Component is depicted in Definition 44.

44 Definition (Scheduled Process Component)
Given a Process Component $G^R = (L^R, X^R, P^R, \mathcal{T}^R)$*, we define the Scheduled Process Component* $G^s = (L^s, P^s, X^s, \mathcal{T}^s)$ *with Acquire, Release scheduling transitions, where:*

- $L^s = L^R \cup \{\ell_0, \ell_n\} \cup \{\ell_p^\beta, \ell_q^\beta | \tau_\beta \in \mathcal{T}^R\}$*, where* $n-1$ *the number of states* $\in L^R$*,*
- $P^s = P^R \cup \{acq, rel, send\_d, next\}$*,*
- $X^s = X^R \cup \{delay\}$*,*
- $\mathcal{T}^s = \mathcal{T}^R \cup \{\tau_{acq} = (\ell_0, acq, true, skip, \ell_{ini}), \tau_{rel} = (\ell_{fin}, rel, g_{exit}, skip, \ell_n)\} \cup \{\tau = (\ell, \beta, g, f', \ell_d), \tau = (\ell_d, send\_d, true, skip, \ell_n), \tau = (\ell_n, next, true, skip, \ell') | \tau = (\ell, \beta, g, f, \ell')\}$

, with g_{exit} *the exiting condition and* $f' : f; delay = prof\_value();$*.*

Moreover, they also use the ports *rel* and *acq* for interaction with the processor scheduler. These ports are used to release (resp. acquire) the processor whenever the *Read/Write* operation is suspended (resp. resumed) due to lack (resp. presence) of available data (or available space) in the buffer.

The *FIFO-Write* and the *FIFO-Read* Components are also connected to the Processor Scheduler. At the point which a process initiates a *Write* or a *Read* call, the FIFO Components may be blocked due to the lack of availability of empty space in the buffer or due to the absence of data to read, the FIFO Components should release the control and re-acquire it when the buffer is in an utilizable state.

45 Definition (Scheduled FIFO-Write Component)
Given a FIFO-Write Component FW, we define the Scheduled FIFO-Write Component as
$FW^s = (L_{FW^s}, X_{FW^s}, P_{FW^s}, \mathcal{T}_{FW^s})$, *where the behavior is described in Figure 5.7.*

- $L_{FW^s} = L_{FW} \cup \{\ell_5\}$,
- $X_{FW^s} = X_{FW}$,
- $P_{FW^s} = P_{FW} \cup \{acq, rel\}$.
- $\mathcal{T}_{FW^s} = \{$
  $\tau_{w_b} = (\ell_1, w_b, true, skip, \ell_2)$,
  $\tau_{mw_b} = (\ell_2, mw_b, g_w, skip, \ell_3)$,
  $\tau_{mw_e} = (\ell_3, mw_e, true, f_{wn}, \ell_4)$,
  $\tau_{w_e} = (\ell_4, w_e, true, skip, \ell_1)$,
  $\tau_{acq} = (\ell_5, acq, g_w, skip, \ell_2)$,
  $\tau_{rel} = (\ell_2, rel, !g_w, skip, \ell_5)$,
  $\{\tau_u = (\ell, w_u, true, f_{rc}, \ell) | \ell \in L_{FW^s}\}$
  $\}$.

46 Definition (Scheduled FIFO-Read Component)
Given a FIFO-Read Component FR, we define the Scheduled FIFO-Read Component as $FR^s = (L_{FR^s}, X_{FR^s}, P_{FR^s}, \mathcal{T}_{FR^s})$, *where the behavior is described in Figure 5.7.*

- $L_{FR^s} = L_{FR} \cup \{\ell_5\}$,
- $X_{FR^s} = X_{FR}$,
- $P_{FR^s} = P_{FR} \cup \{acq, rel\}$.
- $\mathcal{T}_{FR^s} = \{$
  $\tau_{r_b} = (\ell_1, r_b, true, skip, \ell_2)$,
  $\tau_{mr_b} = (\ell_2, mr_b, g_r, skip, \ell_3)$,
  $\tau_{mr_e} = (\ell_3, mr_e, true, f_{rn}, \ell_4)$,
  $\tau_{r_e} = (\ell_4, r_e, true, skip, \ell_1)$,
  $\tau_{acq} = (\ell_5, acq, g_r, skip, \ell_2)$,
  $\tau_{rel} = (\ell_2, rel, !g_r, skip, \ell_5)$,
  $\{\tau_u = (\ell, r_u, true, f_{wc}, \ell) | \ell \in L_{FR^s}\}$
  $\}$.

Correctness

This is an intermediate model, specifically modified to be connected with a *Processor Scheduler* Component introduced in Section 4.2.1. The derived model is intended to integrate the scheduling and computation delay constraints of the HW platform. The formal definition and correctness are given in Section 6.1 in the next chapter.

5.3 Conclusion

In this chapter, we analyzed the mapping as it is used in the construction of the System Model. We defined the refinement steps of the initial software application model in order

to conform with the mapping, which models the accurate deployment on the HW platform. For each step, the intermediate models obtained are equivalent with the initial model. To prove the above, we used the notion of trace equivalence.

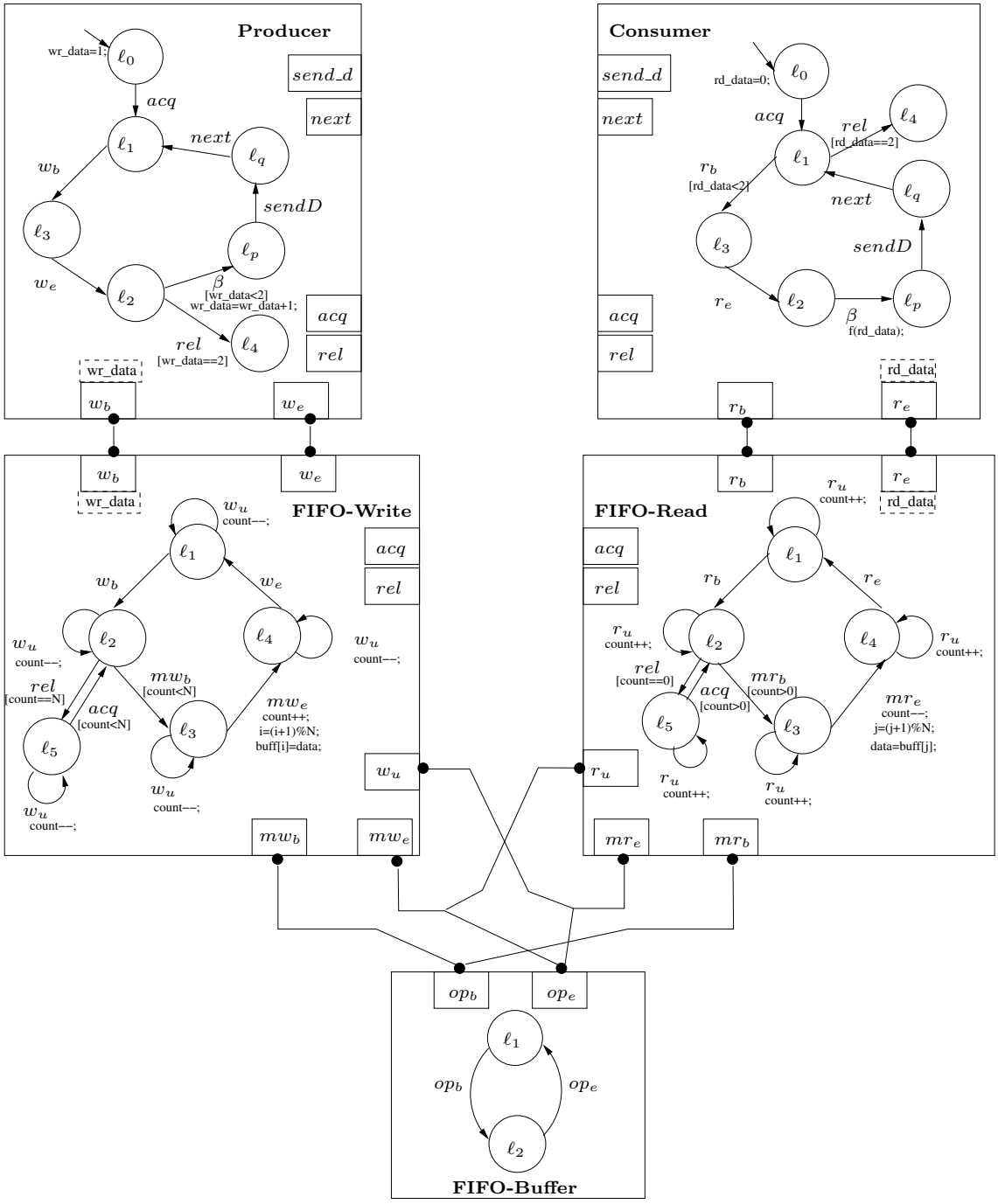


Figure 5.7: *Model of the Processor Scheduled System Model in BIP*

- Chapter 6 -

Integration of HW Constraints

In the previous chapter, we analyzed the refinement of the software application model in BIP according to a given mapping on a HW platform. The goal of the refinement is the accurate deployment on the HW platform. In this chapter, we focus on the integration of the HW constraints which arise upon the execution of a software application on a given platform. More specifically, we merge the refined software application model in BIP with the target abstract model of a HW platform in BIP.

6.1 HW Constraints For Computation

In Section 4.2, we listed the HW constraints which characterize a HW platform. In this section, we connect the HW computation model with the software application mode and we prove the correctness of the generated mixed HW/SW model.

47 Definition (Processor Scheduled Process Network Composition)
-Addition of Mutual Exclusion Scheduler and Computational delays-
Let a Split-FIFO Process Network N^D as defined in Definition 42 and a HW platform using m processors. The processor resource is modeled by the Processor Scheduler Component defined in Definition 18. We define a Scheduled Process Network $N^S = \gamma^S(G_1^S, \ldots, G_n^S, FW_1^S, FR_1^S, FB_1, \ldots, FW_k^S, FR_k^S, FB_k, SC_1, \ldots, SC_m)$, where the set of interactions γ^S is defined as:
$\gamma^S = \gamma^D \cup \{\alpha^{tick}\} \cup \{\alpha_G^{acq}, \alpha_G^{rel}, \alpha_G^{send\_d}, \alpha_G^{next} | G \in N^S\} \cup \{\alpha_{FW}^{acq}, \alpha_{FW}^{rel} | FW \in N^S\} \cup \{\alpha_{FR}^{acq}, \alpha_{FR}^{rel} | FR \in N^S\}$,
$\alpha_G^{acq} = (\{G.acq, SC.acq\}, true, skip)$,
$\alpha_G^{rel} = (\{G.rel, SC.rel\}, true, skip)$,
$\alpha_G^{send\_d} = (\{G.send\_d, SC.get\_d\}, f, skip)$, *with* $f : SC.delay = G.delay$;
$\alpha_G^{next} = (\{G.next, SC.next\}, true, skip)$,
$\alpha_{FW}^{acq} = (\{FW.acq, SC.acq\}, true, skip)$,
$\alpha_{FW}^{rel} = (\{FW.rel, SC.rel\}, true, skip)$,
$\alpha_{FR}^{acq} = (\{FR.acq, SC.acq\}, true, skip)$,
$\alpha_{FR}^{rel} = (\{FR.rel, SC.rel\}, true, skip)$,
$\alpha^{tick} = (\{SC_1, \ldots, SC_m\}, true, skip)$
We export the tick port of the α^{tick} interaction to dynamically enable synchronization with other tick interactions. We will further refer to this interaction as $N^S.tick$.

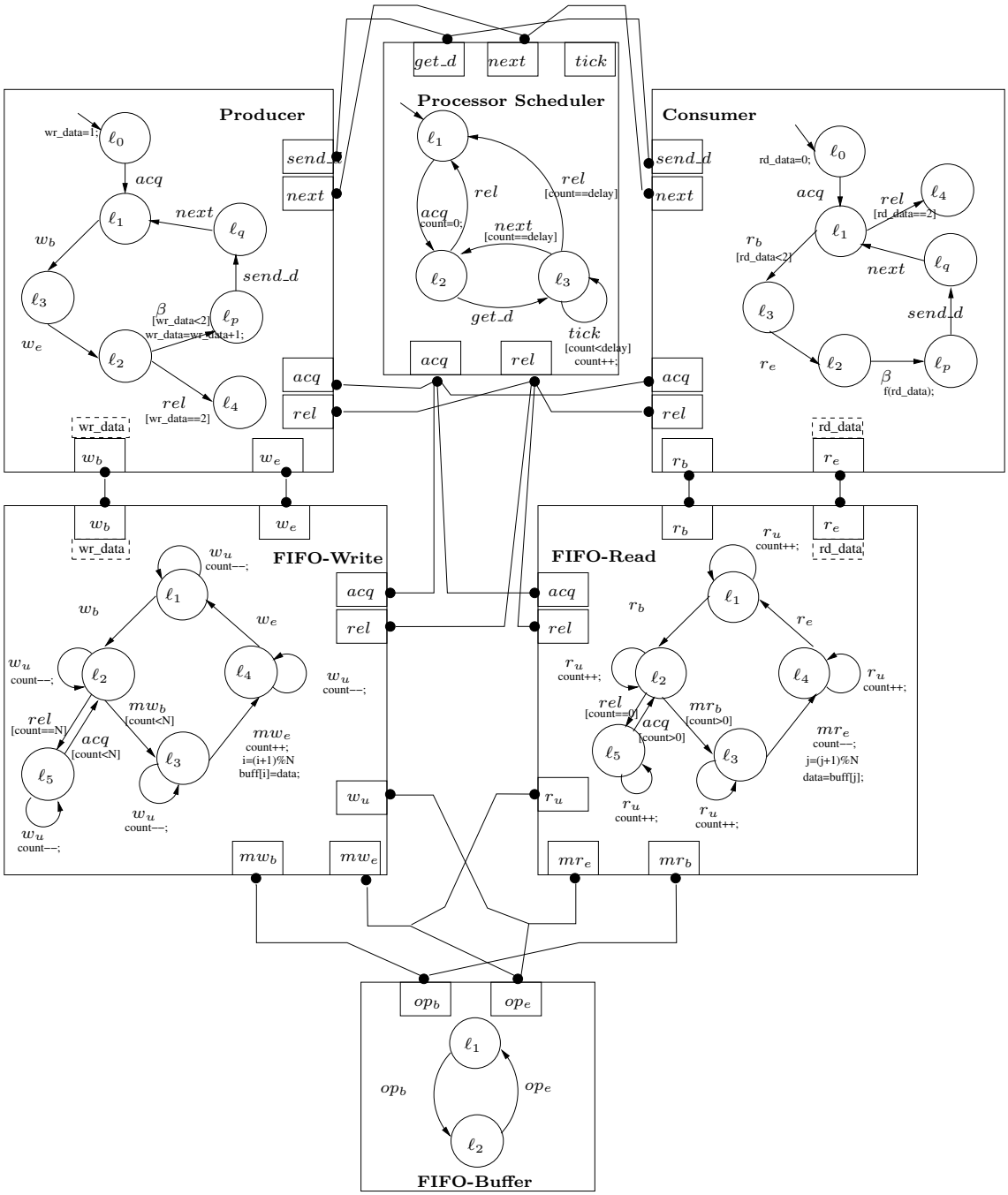


Figure 6.1: *Model of the Processor Scheduled System Model in BIP*

17 Example (Processor Scheduled Producer-Consumer Composition)
Given a Split-FIFO Producer-Consumer Composition $PC^D = (G_P^r, FW, FR, FB, G_C^r)$, *we add computational delays and processor scheduling. We map the Producer and the Consumer processes on a single processor. Consequently,* G_P *and* FW, *which is directly connected to* G_P, *are connected to the Processor Scheduler* SC_1. *Similarly,* G_C *and* FR *are connected to Processor Scheduler* SC_1, *such that we obtain:*
$PC^S = \gamma^S(G_P^S, FW^S, FR^S, FB, G_C^S, SC_1)$ *as illustrated in Figure 6.1, where*
$\gamma^S = \gamma^D \cup \{\alpha^{acq}, \alpha^{rel}, \alpha^{send\_d}, \alpha^{next} | G \in N^S\} \cup \{\alpha_{FW}^{acq}, \alpha_{FW}^{rel} | FW \in N^S\} \cup \{\alpha_{FR}^{acq}, \alpha_{FR}^{rel} | FR \in N^S\}$

The set of traces is represented as the interleavings in Figure 6.2.

Figure 6.2: *Traces of Processor Scheduled Producer-Consumer Composition*

Correctness

48 Definition (Trace Restriction of a Processor Scheduled Process Network Composition) *The restriction of a trace of a run θ^S to β events is defined by:*

$$trace^S(\theta^S) = \begin{cases} \epsilon & ,if\ \theta^S : \epsilon \\ \beta.trace^S(\theta'^S) & ,if\ \theta^S : q \xrightarrow{\beta} q_1 \xrightarrow{send\_d} q_2 \xrightarrow{tick_1} \cdots \xrightarrow{tick_n} q_k \xrightarrow{next} q'.\theta'^S \\ trace^S(\theta_A).trace^S(\theta_B) & ,if\ \theta^S : q_0 \xrightarrow{acq} q_1.\theta_A^S.q_k \xrightarrow{rel} q_{k+1}.\theta_B^S,\ with\ acq, rel \notin \theta_A^S \end{cases}$$

Note that, θ_B^S always starts with an acq and ends with a rel, so that we can recursively apply the above rule.

Let N^S be the Processor Scheduled Process Network Composition of N^D.

5 Theorem
For each complete run θ^D in the Split-FIFO Process Network N^D:
$\theta^D : q_0 \xrightarrow{a_1^D}_{\gamma^D} q_1^D \xrightarrow{a_2^D}_{\gamma^D} q_2^D \xrightarrow{a_3^D}_{\gamma^D} \cdots \xrightarrow{a_n^D}_{\gamma^D} q_n^D$
There exists a complete run θ^S in the Processor Scheduled Process Network N^S:
$\theta^S : q_0 \xrightarrow{a_1^S}_{\gamma^S} q_1^S \xrightarrow{a_2^S}_{\gamma^S} q_2^S \xrightarrow{a_3^S}_{\gamma^S} \cdots \xrightarrow{a_m^S}_{\gamma^S} q_m^S$
such that:

- $q_n^D = q_m^S$, *where* q_m^S *concerns the subsystem of the components which do not include the Processor Scheduler,*
- $trace(\theta^D) = trace^S(\theta^S)$

6 Theorem
For each complete run θ^S *in the Processor Scheduled Process Network* N^S*:*
$\theta^S : q_0 \xrightarrow{a_1^S}_{\gamma^S} q_1^S \xrightarrow{a_2^S}_{\gamma^S} q_2^S \xrightarrow{a_3^S}_{\gamma^S} ... \xrightarrow{a_m^S}_{\gamma^S} q_m^S$
There exists a complete run θ^D *in the Split-FIFO Process Network* N^D*:*
$\theta : q_0 \xrightarrow{a_1^D}_{\gamma^D} q_1^D \xrightarrow{a_2^D}_{\gamma^D} q_2^D \xrightarrow{a_3^D}_{\gamma^D} \cdots \xrightarrow{a_n^D}_{\gamma^D} q_n^D$
such that:

- $q_n^S = q_m^D$, *where* q_n^S *concerns the subsystem of the components which do not include the Processor Scheduler,*
- $trace^S(\theta^S) = trace(\theta^D)$

3 Lemma
For each complete run θ *in the Processor Scheduled Process Network* N^S*:*
$\theta : q_0 \to \cdots \to q_n$
There exists a complete run θ' *in the Processor Scheduled Process Network* N^S *starting from* q_0*:*
$\theta' : q_0 \to \cdots \to q_{n'}$
such that:

- $q_n = q_{n'}$
- *each trace that contains interactions which concern a single process are always enclosed between the* acq, rel *interactions,*
- *in* θ' *any* α^β *interaction is followed immediately by the series of interactions*

$$\alpha^{send\_d}, \alpha^{tick_1} \cdots \alpha^{tick_n}$$

7 Proof (Theorem 5)
We consider a complete run θ^D *in the Split FIFO Process Network* N^D*. Based on Lemma 1, we re-order any begin interaction so that it is immediately followed by the corresponding end interaction. We have* $\theta^D : q_0 \xrightarrow{\alpha^1}_{\gamma^D} q_1^D \xrightarrow{\alpha^1_\beta}_{\gamma^D} \cdots \xrightarrow{\alpha^2}_{\gamma^D} q_2^D \xrightarrow{\alpha^2_\beta}_{\gamma^D} q^D$*. We map each Process Component* G^D*, along with the FIFO Routines* FW^D, FR^D *connected to it, on a Processor Scheduler Component. We obtain a run* $\theta^D : q_0 \xrightarrow{acq}_{\gamma^S} q_1^S \xrightarrow{\alpha^1}_{\gamma^D} q_2^D \xrightarrow{\beta^1}_{\gamma^S} q_3^S \xrightarrow{send\_d}_{\gamma^S} q_4^S \xrightarrow{tick_1}_{\gamma^S} \cdots \xrightarrow{tick_n}_{\gamma^S} q_5^S \xrightarrow{next}_{\gamma^S} \cdots \xrightarrow{\alpha^2}_{\gamma^D} q_6^D \xrightarrow{\beta^2}_{\gamma^S} q_7^S \xrightarrow{send\_d}_{\gamma^S} q_8^S \xrightarrow{tick_1}_{\gamma^S} \cdots \xrightarrow{tick_n}_{\gamma^S} q_9^S \xrightarrow{next}_{\gamma^S} q_{10}^S \xrightarrow{rel}_{\gamma^S} q^S$.

According to the Definition 47 of the Processor Scheduled Process Network as it is illustrated in the Example 17 and the Figure 6.1, we have $q^D = q^S$*, where* q^S *concerns the subsystem of the components which do not include the Processor Scheduler.*

In addition, we have:

$$trace(\theta^D) = \alpha^1.\beta^1.\alpha^2.\beta^2$$

and

$$trace(\theta^S) = acq.\alpha^1.\beta^1.send\_d.tick_1 \ldots tick_n.next.\alpha^2.\beta^2 send\_d.tick_1 \ldots tick_n.next.rel$$

Based on the Definition 48, we restrict the trace θ^S *by removing the* $send\_d, tick$ *and* $next$ *interactions in the first step and by removing the* acq, rel *interactions in the second step, such that*

$$trace^S(\theta^S) = \alpha^1.\beta^1.\alpha^2.\beta^2$$

. So, we conclude that $trace(\theta^D) = trace^S(\theta^S)$.

8 Proof (Lemma 3)
Let a complete run $\theta^S : q_0 \xrightarrow{\alpha^1}_{\gamma^r} q_1 \xrightarrow{\alpha^2}_{\gamma^r} q_2.\theta'$, *which belongs in the Processor Scheduled Process Network* N^r *and* $trace(\theta^S) = \alpha_1.\alpha_2.trace(\theta')$. *Based on Proposition 1 we can reverse the execution order of interactions* α^1, α^2*:*
$\theta^S : q_0 \xrightarrow{\alpha^2}_{\gamma^r} q'_1 \xrightarrow{\alpha^1}_{\gamma^r} q'_2.\theta'$.

As we can see in Example 17 the traces of the Processor Scheduled Process Network contain:

$$acq.\alpha^1.\alpha_b^{mw}.\alpha_e^{mw}.\alpha_e^{w}.\beta.send\_d.tick_1 \ldots tick_n.next \ldots rel$$

Interleavings of the above interactions occur due to multiple Processor Schedulers that run in parallel. In this case, we can recursively relocate the execution order according to Proposition 1. So, any α_b^w *or* α_b^r *interaction is followed immediately by the series of interactions* α_b^{mw}, α_e^{mw}, α_e^w *and* α_b^{mr}, α_e^{mr}, α_e^r *respectively. In addition, any* α^β *interaction is followed immediately by the series of interactions* $send\_d, tick_1 \ldots tick_n, next$*. Also, the traces that concern a single process are enclosed between the* acq, rel *interactions, due to the Processor Scheduler resource demand. Thus, we can conclude that:*

- *the ending state of the new run is equal to the ending state of the old run,*
- *each trace that contains interactions which concern a single process are always enclosed between the* acq, rel *interactions,*
- *in* $\theta^{s'}$ *any* β *interaction is followed immediately by the series of* $send\_d, tick_1 \cdots tick_n, next$ *interactions.*

9 Proof (Theorem 6)
We consider a run $\theta'^S : q_0 \xrightarrow{\alpha'^S}_{\gamma^r} q'_1 \xrightarrow{\alpha'^2}_{\gamma^S} \ldots \xrightarrow{\alpha'^k}_{\gamma^S} q'_k$ *in a Processor Scheduled Process Network* N^S. *Based on Lemma 3, we consider* θ^S *in* N^S *such that:*

- *a trace that contains interactions which concern a single process are always enclosed between the* acq, rel *interactions,*
- *any* α_β *interaction is followed immediately by the series of* $send\_d, tick_1 \cdots tick_n, next$ *interactions such that* $\theta^S : q_0 \xrightarrow{\beta}_{\gamma^S} q_1 \xrightarrow{send\_d}_{\gamma^S} q_2 \xrightarrow{tick_1}_{\gamma^S} \ldots \xrightarrow{tick_n}_{\gamma^S} \xrightarrow{next}_{\gamma^S} \ldots \xrightarrow{\alpha_e^k} q$.

As a consequence of Lemma 3, we have $q'^k = q$.

Additionally, based on the Definition 47 of the Processor Scheduled Process Network as it is illustrated in the Example 17 and the Figure 6.1, we replace the series of $\beta, send\_d, tick_1 \cdots tick_n, next$ *interactions with only the* β *interaction and we omit the* acq, rel *interactions. Thus, we will obtain a run* $\theta^S : q_0 \xrightarrow{\alpha_b^1}_{\gamma^S} q_1 \xrightarrow{\alpha_b^{m1}}_{\gamma^S} q_2 \xrightarrow{\alpha_e^{m1}}_{\gamma^S} q_3 \xrightarrow{\alpha_e^1}_{\gamma^S} \ldots \xrightarrow{\alpha_e^k} q$ *such that* θ^S *belongs in the Split-FIFO Process Network* N^D.

Since we have,

$$trace(\theta^D) = \alpha_b^1.\alpha_b^{m1}.\alpha_e^{m1}.\alpha_e^1.\alpha_b^2.\alpha_b^{m2}.\alpha_e^{m2}.\alpha_e^2 \ldots \alpha_e^{mk}.\alpha_e^k$$

and

$$trace(\theta^S) = acq.\alpha_b^1.\alpha_b^{m1}.\alpha_e^{m1}.\alpha_e^1.\beta^1.send\_d.tick_1 \ldots tick_n.next.\alpha_b^2.\alpha_b^{m2}.\alpha_e^{m2}.\alpha_e^2.\beta^2\alpha_b^n.\alpha_e^n.rel$$

we conclude, considering the restriction of traces in Definition 48, that $trace^S(\theta^S) = trace(\theta^D)$.

6.2 HW Constraints For Communication

Given a Processor-Scheduled Process Network

$$N^S = \gamma^S(G_1^S, \ldots, G_n^S, FW_1^S, FR_1^S, FB_1, \ldots, FW_k^S, FR_k^S, FB_k, SC_1, \ldots, SC_m),$$

we replace the FB Components with a hardware template communication model and the $\alpha^m \in \gamma^S$ connectors with new ones depending on the mapping such as,

$$N^{S'} = \gamma^S(G_1^S, \ldots, G_n^S, FW_1^S, FR_1^S, \ldots, FW_k^S, FR_k^S, SC_1, \ldots, SC_m),$$

We assume a platform where there are k processors, with $k \geq 1$, one *Crossbar Switch Bus* and a shared *Memory*.

49 Definition (System Model Processor-Scheduled Process Network)
-NoC with four Shared Memory Clusters-
Given a Processor-Scheduled Process Network N^S, *we define* $N^M = \pi\gamma(N^{S'}, NOC)$ *as the System Model of a Process Network on a HW platform, where* $N^{S'}$ *is defined above and* $NOC = \pi^{NOC}\gamma^{NOC}(CL_1, \ldots, CL_4, R_1, \ldots, R_4)$ *is defined as the Communication Model of the system, where:*
$CL = \pi^{CL}\gamma^{CL}(BUS, M, NI)$
M *is the Shared Memory Component and*
$BUS = \pi^{bus}\gamma^{bus}(BP_1, \ldots, BP_k, BS)$, *where* k *the number of processors,* BP *the Bus Paths,* BS *the Bus Scheduler,*
$\gamma^{bus} = \{acq_1^{bus}, rel_1^{bus}, \ldots, acq_k^{bus}, rel_k^{bus}, tick^{bus}\}$, *with*
$acq_i^{bus} = (\{BS.acq, BP_i.acq\}, true, skip)$,
$rel_i^{bus} = (\{BS.rel, BP_i.rel\}, true, skip)$, *where* $i \in k$,
$tick^{bus} = (\{BP_1.tick, \ldots, BP_k.tick\}, true, skip)$.
and π^{bus} *giving priority on* acq^{bus}, rel^{bus} *interactions over* $tick^B$. *The FIFO Buffers are mapped to the shared Memory* M *of a cluster* CL. *Thus, we replace the FIFO Buffers with the Memory* M. *Since the FIFO Routines are mapped on a Processor, we connect them with the Bus Path Components* BP *of the* BUS *which connect the Processor* P *with the Memory* M *or with the Network Interface* NI *if the Memory is located in a different cluster.. The set of interactions* γ *is defined as:*
$\gamma = \gamma^{S'} \cup \{\alpha_b^m, \alpha_e^m | FIFO\ Routine \in N^{S'}\} \cup \{\alpha_b^\mu, \alpha_e^\mu | BP \in BUS\} \cup \{tick\}$,
with,
$\alpha_b^m = (\{FW.mw_b, BP.req\}, true, skip)$ *or* $(\{FR.mr_b, BP.req\}, true, skip)$,
$\alpha_e^m = (\{BP.ack, FW.mw_e, FR.r_u\}, true, skip)$ *or* $(\{BP.ack, FR.mr_e, FW.w_u\}, true, skip)$,
$\alpha_b^\mu = (\{BP.op_b, M.op_b\}, true, skip)$,
$\alpha_e^\mu = (\{BP.op_e, M.op_e\}, true, skip)$,
$tick = (\{N^{S'}.tick, BUS.tick, M.tick\}, true, skip)$.
For each interaction, $tick$ *has the lowest priority and the interactions* acq, rel *have the highest priority of all, in order to prevent infinite ticking.*

18 Example (Processor-Scheduled Producer-Consumer Composition) -Shared Memory Cluster-

Given a Processor Scheduled Producer-Consumer Composition $PC^S = \gamma^S(G_P^S, FW^S, FR^S, FB, G_C^S, SC_1)$, *we add communication delays. We map the FIFO-Buffer* FB *on the shared memory such that we obtain:* $PC^M = \gamma(G_P^{S'}, FW^S, FR^S, G_C^S, SC_1, BUS_1, M_1)$, *as illustrated in Figure 6.4, where:* $BUS_1 = \pi^{bus}\gamma^{bus}(BP_1, BP_2)$, *with* γ^{bus}*:*

$acq_1^{bus} = (\{BS.acq, BP_1.acq\}, true, skip)$,
$rel_1^{bus} = (\{BS.rel, BP_1.rel\}, true, skip)$,
$acq_2^{bus} = (\{BS.acq, BP_2.acq\}, true, skip)$,
$rel_2^{bus} = (\{BS.rel, BP_2.rel\}, true, skip)$,
$tick^{bus} = (\{BP_1.tick, BP_2.tick\}, true, skip)$.
and $\gamma = \gamma^S \cup \{\alpha^{rw}, \alpha^{aw}, \alpha^{rr}, \alpha^{ar}, \alpha_b^{mw}, \alpha_e^{mw}, \alpha_b^{mr}, \alpha_e^{mr}, tick\}$:
$\alpha_b^{mw} = (\{FW^S.mw_b, BP_1.req\}, true, skip)$,
$\alpha_e^{mw} = (\{FW^S.mw_e, BP_1.ack, FR^S.r_u\}, true, skip)$,
$\alpha_b^{mr} = (\{FR^S.mr_b, BP_2.req\}, true, skip)$,
$\alpha_e^{mr} = (\{FR^S.mr_e, BP_2.ack, FW^S.w_u\}, true, skip)$,
$\alpha_b^{\mu w} = (\{BP_1.op_b, M_1.op_b\}, true, skip)$,
$\alpha_e^{\mu w} = (\{M_1.op_e, BP_1.op_e\}, true, skip)$,
$\alpha_b^{\mu r} = (\{BP_2.op_b, M_1.op_b\}, true, skip)$,
$\alpha_e^{\mu r} = (\{M_1.op_e, BP_2.op_e\}, true, skip)$,
$tick = (\{SC_1.tick, BUS.tick, M_1.tick\}, true, skip)$

For each $\alpha \in \gamma$, *the tick interaction has the lowest priority and the interactions* $\alpha^{acq}, \alpha^{rel}$ *have the highest priority of all, in order to prevent infinite ticking. A figure illustrating the current example with the use of a Network-on-Chip is provided in Figure 6.5.*

Correctness

50 Definition (Trace Restriction of a System Model Process Network Composition) *The restriction of a trace of a run* θ^M *is defined by:*

$$trace^M(\theta^M) = \begin{cases} \epsilon & ,\text{if } \theta^M : \epsilon \\ \beta.trace^M(\theta'^M) & ,\text{if } \theta^M : q \xrightarrow{\beta} q'.\theta'^M \\ trace(\theta_A).\alpha_b^m.\alpha_e^m.trace^M(\theta_B) & ,\text{ if } \theta^M : \theta_A.q_0 \xrightarrow{\alpha_b^m} q_1.\theta_X.q_2 \xrightarrow{\alpha_e^m} q_3.\theta_B \end{cases}$$

where $\alpha_b^m\alpha_e^m \notin \theta_X$ *and all interactions in* θ_X *concern communication such as usage of BUS, Memory and NoC. The restriction means that all the interactions which concern communication and take place between* mw_b/mr_b *and* mw_e/mr_e *can be omitted.*

Let N^M be the System Model Composition Composition of N^S.

7 Theorem

For each complete run θ^S *in the Processor Scheduled Process Network* N^S*:*
$\theta^S : q_0 \xrightarrow{a_1^S}_{\gamma^S} q_1^S \xrightarrow{a_2^S}_{\gamma^S} q_2^S \xrightarrow{a_3^S}_{\gamma^S} \cdots \xrightarrow{a_n^S}_{\gamma^S} q_n^S$
There exists a complete run θ^M *in the System Model Process Network* N^M*:*
$\theta^M : q_0 \xrightarrow{a_1^M}_{\gamma^S} q_1^M \xrightarrow{a_2^M}_{\gamma^M} q_2^M \xrightarrow{a_3^M}_{\gamma^M} \cdots \xrightarrow{a_m^M}_{\gamma^M} q_m^M$
such that:

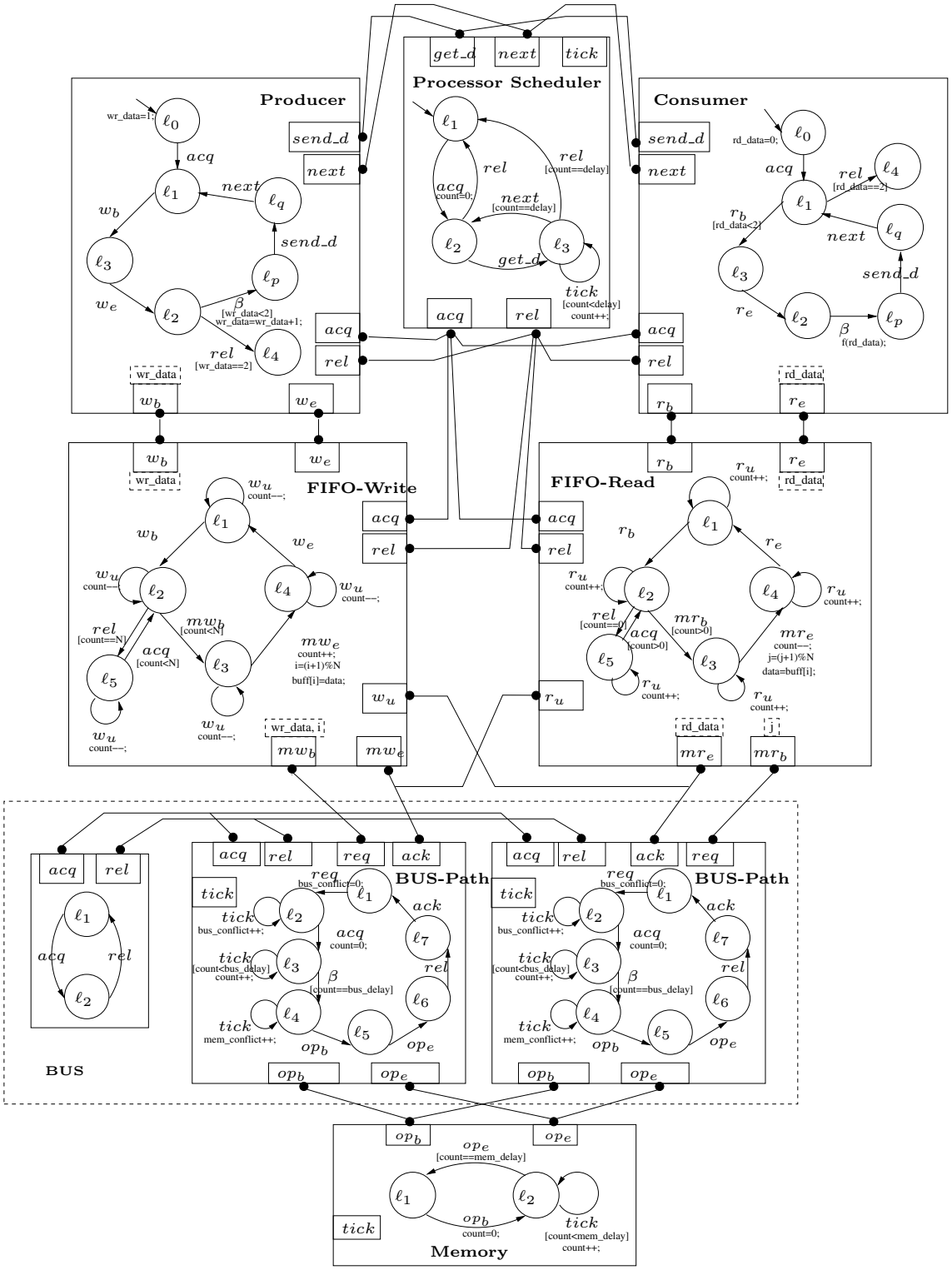


Figure 6.3: *Producer-Consumer System Model on a Shared Memory Cluster in BIP*

- $q_n^S = q_m^M$, *where* q_m^M *concerns the subsystem of the components which do not include the communication model,*
- $trace(\theta^S) = trace^M(\theta^M)$

Figure 6.4: *Traces of Processor Scheduled Producer-Consumer Composition on a Shared Memory Cluster*

8 Theorem
For each complete run θ^M *in the System Model Process Network* N^M*:*
$\theta^M : q_0 \xrightarrow{a_1^M}_{\gamma^S} q_1^M \xrightarrow{a_2^M}_{\gamma^M} q_2^M \xrightarrow{a_3^M}_{\gamma^M} ... \xrightarrow{a_m^M}_{\gamma^M} q_m^M$
There exists a complete run θ^S *in the Processor Scheduled Process Network* N^S*:*
$\theta : q_0 \xrightarrow{a_1^S}_{\gamma^S} q_1^S \xrightarrow{a_2^S}_{\gamma^S} q_2^S \xrightarrow{a_3^S}_{\gamma^S} \cdots \xrightarrow{a_n^S}_{\gamma^S} q_n^S$
such that:

- $q_n^M = q_m^S$, *where* q_m^M *concerns the subsystem of the components which do not include the communication model,*
- $trace^M(\theta^M) = trace(\theta^S)$

4 Lemma
For each complete run θ *in the System Model Process Network* N^M*:*
$\theta : q_0 \to \cdots \to q_n$
There exists a complete run θ' *in the System Model Process Network* N^M *starting from* q_0*:*
$\theta' : q_0 \to \cdots \to q_{n'}$
such that:

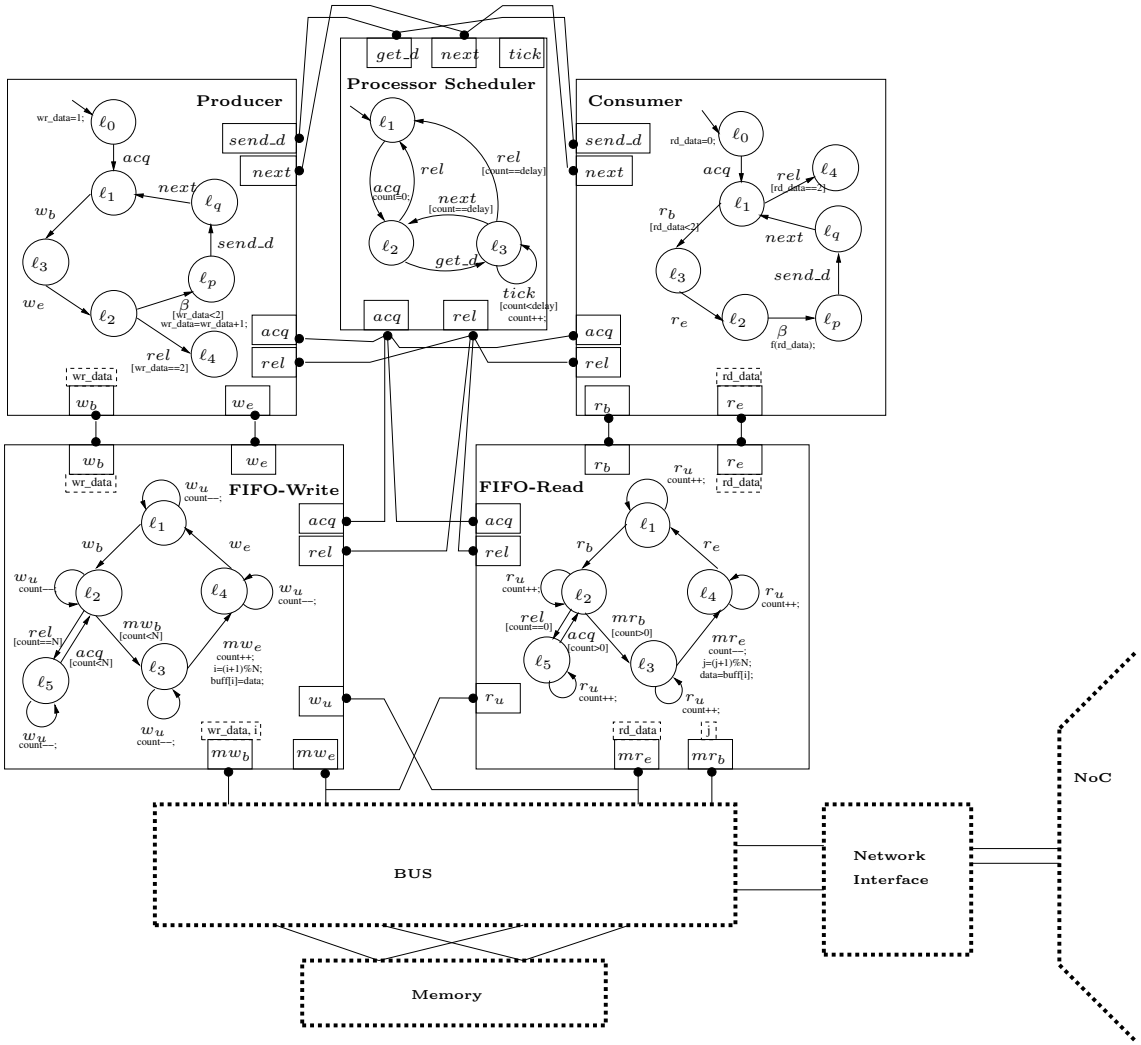


Figure 6.5: *Producer-Consumer System Model on a NoC in BIP*

- $q_n = q_{n'}$
- $\theta' : \theta_A.q_0 \xrightarrow{\alpha_b^m} q_1.\theta_X.q_2 \xrightarrow{\alpha_e^m} q_3.\theta_B$,*with* $\alpha_b^m\alpha_e^m \notin \theta_X$, *and all interactions in* θ_X *concern communication such as usage of BUS, Memory and NoC.*

10 Proof (Theorem 7)

We consider a complete run θ^S *in the Processor Scheduled Process Network Composition* N^S, *such that we have:*

$$\theta^S : q_0 \xrightarrow{acq}_{\gamma^S} q_1^S \xrightarrow{\alpha_b^1}_{\gamma^S} q_2^S \xrightarrow{\alpha_b^{m1}}_{\gamma^S} q_3^S \xrightarrow{\alpha_e^{m1}}_{\gamma^S} q_4^S \xrightarrow{\alpha_e^1}_{\gamma^S} \cdots \xrightarrow{rel}_{\gamma^S} q^S$$

We map each FIFO Buffer Component FB upon the shared memory. We obtain a run:

$$\theta^S : q_0 \xrightarrow{acq}_{\gamma} q_1 \xrightarrow{\alpha_b^1}_{\gamma} q_2 \xrightarrow{\alpha_b^{m1}}_{\gamma} q_3 \xrightarrow{\alpha_{tick_1}}_{\gamma} \cdots \xrightarrow{\alpha_{tick_n}}_{\gamma} q_4 \xrightarrow{acq^{bus}}_{\gamma} q_5 \xrightarrow{\alpha_{tick_1}}_{\gamma} \cdots \xrightarrow{\alpha_{tick_n}}_{\gamma}$$
$$q_6 \xrightarrow{\alpha_{BP.\beta}}_{\gamma} q_7 \xrightarrow{\alpha_{tick_1}}_{\gamma} \cdots \xrightarrow{\alpha_{tick_n}}_{\gamma} q_8 \xrightarrow{\alpha_b^{op}}_{\gamma} q_9 \xrightarrow{\alpha_{tick_1}}_{\gamma} \cdots \xrightarrow{\alpha_{tick_n}}_{\gamma} q_{10} \xrightarrow{\alpha_e^{op}}_{\gamma} q_{11} \xrightarrow{rel^{bus}}_{\gamma}$$
$$q_{12} \xrightarrow{\alpha_e^{m1}}_{\gamma} \cdots \xrightarrow{rel}_{\gamma} q^M.$$

We consider the Definition 49 of the System Model Processor-Scheduled Process Network on a Shared Memory Cluster, as illustrated in Example 18 and in Figure 6.3, At

the point where a FIFO routine Component does a mw_b or a mr_b to access a specific address in the hardware memory, no other FIFO Routine can access the same hardware memory address. The FIFO mechanism prevents this from happening. Only after the corresponding mw_e or a mr_e has been executed, should this particular memory address be accessed again. Thus, all the communications interactions taking place between mw_b/mr_b and mw_e/mr_e that concern interconnects, memories and NoC can be omitted, since the functional behavior of the initial process network and the FIFO functionality is always respected. Thus, we conclude that $q^M = q^S$, where q_m^M concerns the subsystem of the components which do not include the communication model. In addition we have,

$$trace(\theta^S) = acq.\alpha_b^1.\alpha_b^{m1}.\alpha_e^{m1}.\alpha_e^1 \ldots rel$$

and
$trace(\theta^M) = acq.\alpha_b^1.\alpha_b^{m1}.tick_1 \ldots tick_n.acq^{bus}.tick_1 \ldots tick_l.\alpha_{BP.\beta}.tick_1 \ldots tick_j.\alpha_b^{\mu}.$
$tick_1 \ldots tick_h.\alpha_e^{\mu}.rel^{bus}\alpha_e^{m1}.\alpha_e^1 \ldots rel$, *where* $n, l, j, h > 1$.

Based on the Definition 50, we restrict the trace θ^M such that:

$$trace^M(\theta^M) = acq.\alpha_b^1.\alpha_b^{m1}.\alpha_e^{m1}.\alpha_e^1 \ldots rel$$

So, we conclude that $trace(\theta^S) = trace(\theta^M)$.

11 Proof (Lemma 4)
Let a complete run $\theta : q_0 \xrightarrow{\alpha^1}_{\gamma^r} q_1 \xrightarrow{\alpha^2}_{\gamma^r} q_2.\theta'$, which belongs in the System Model Process Network N^r and $trace(\theta) = \alpha_1.\alpha_2.trace(\theta')$. Based on Proposition 1 we can reverse the execution order of interactions α^1, α^2:
$\theta : q_0 \xrightarrow{\alpha^2}_{\gamma^r} q_1' \xrightarrow{\alpha^1}_{\gamma^r} q_2'.\theta'$.

As we can see in Example 18 the traces of the System Model Process Network contain:
$acq.\alpha_b^w.\alpha_b^{mw}.tick_1 \ldots tick_n.acq^{bus}.tick_1 \ldots tick_l.\alpha_{BP.\beta}.tick_1 \ldots tick_j.\alpha_b^{\mu w}.$
$tick_1 \ldots tick_h.\alpha_e^{\mu w}.rel^{bus}\alpha_e^{mw}.\alpha_e^w \ldots rel$, *where* $n, l, j, h > 1$.

Interleavings of the above interactions occur due to multiple communication resources, such as interconnects, memories or NoC, that run in parallel. In this case, we can recursively relocate the execution order according to Proposition 1. So, any α_b^{mw} or α_b^{mr} interaction is followed immediately by the series of communication interactions such as $tick, acq^{bus}, \alpha_b^{\mu}, \alpha_b^{\mu}, rel^{bus}$, etc. and always ends with the α_e^{mw} or α_e^{mr} interaction respectively. Thus, we can conclude that:

- *the ending state of the new run is equal to the ending state of the old run,*
- *in θ' any communication interaction, such as $tick, acq^{bus}, \alpha_b^{\mu}, \alpha_b^{\mu}, rel^{bus}$, etc., is enclosed between $\alpha_b^{mw}, \alpha_e^{mw}$ or $\alpha_b^{mr}, \alpha_e^{mr}$ interactions.*

12 Proof (Theorem 7)
Proof evident based on Lemma 4 and Theorem 8.

6.3 Conclusion

In this chapter, we analyzed the method of integrating hardware platform components in a given software application model. The hardware platform components are assigned to model important functional and non-functional hardware constraints. Specifically, in

the current model we have focused on the timing delays imposed by the underlying hardware platform. These values vary in different implementations of the platform. They are strongly dependent on the characteristics of each individual component of the platform. Namely, these are the scheduling policies, interconnect throughput, routing protocols, processor frequency and computing speed. However, the model of the processor does not consider analysis of low-level assembly code and pipeline. Modeling the above would superlatively increase the system's complexity and considerably stall the performance analysis. Thus, in order to bypass the above problem, we propose an instrumentation technique of the software application process model. The instrumentation method is presented in the next chapter.

The overall performance outcome of the system is highly affected by the mapping specification. Currently, we consider static mapping of processes upon the platform processors. Important part of the future extension of this work would be to consider task migration and dynamic mapping. Task migration protocol should be aware of all critical performance metrics of the system leading to optimal mapping obtained on-the-fly. Except from timing constraints, thermal values, power consumption and dynamic scheduling policies should be considered. The latter properties of the hardware model should enrich the current models of hardware components attributing to an extensive performance analysis of our target system.

- Chapter 7 -

Integration of Runtime HW/SW Constraints (software dependent)

In the previous chapter, we mentioned that the proposed hardware platform model does not include a detailed processor model which implements instruction-level analysis and pipelining. However, we have developed techniques to obtain cycle-accurate execution delay values of the code included in the software processes running on a given hardware processor. These values are integrated in the generated system model in order to calibrate it so that it accurately models the computational delays derived from the use of a processor CPU. In this chapter, we describe the system model calibration and the associated methods we developed.

7.1 System Model Calibration

The generic procedure of the system model calibration includes the following steps. Firstly, each block of code of the BIP Processes, except the read/write communication transitions, is instrumented. The instrumentation is done by inserting function calls at the beginning and at the end of the executable block of code which is associated with a transition. Secondly, these calls are used by a tool, depending on the methods given below, to provide accurate execution times. We integrate the executions delay values to the system model by profiling each transition code block with them, as it is illustrated in Figure 7.1. This results to a faithful model of the HW/SW dependent run-time computational constraints. Finally, the calibrated BIP system model is used as such by the BIP tool-chain for performance analysis. The latter is achieved through compilation and execution using the BIP simulator. The various measurements (i.e. execution time, delays) are recorded by dedicated observers.

We enumerate the two different calibration methods below:

1. Instruction Weight Table.
2. Platform Dependent Code Generation.

7.1.1 Instruction Weight Table

The method and its supporting tool can provide simulation environment for performance estimation on a given BIP system model. We adopted a strategy to dynamically obtain accurate execution delay values based on fine-grained code analysis. The basic idea of this

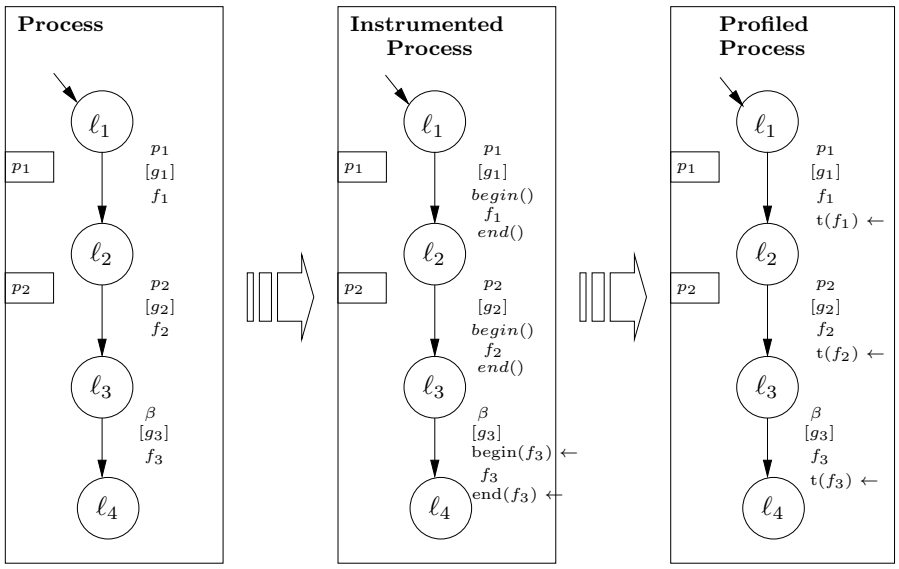


Figure 7.1: *Process Profiling steps*

strategy is to use gcov, which is a tool in conjunction with GCC to recover code coverage information using BIP simulation and analyze the executable code of each line considering the target platform. In order to get the execution result, the whole process is composed of four stages: 1) instrumentation stage to generate C codes with API function calls; 2) cross compilation with target platform; 3) GCC compilation with coverage parameters; 4) analysis stage which combines target platform information, code coverage and performance weight table to derive the execution delay values for each block of code in the initial System model. An overview of the above stages is presented in Figure 7.2.

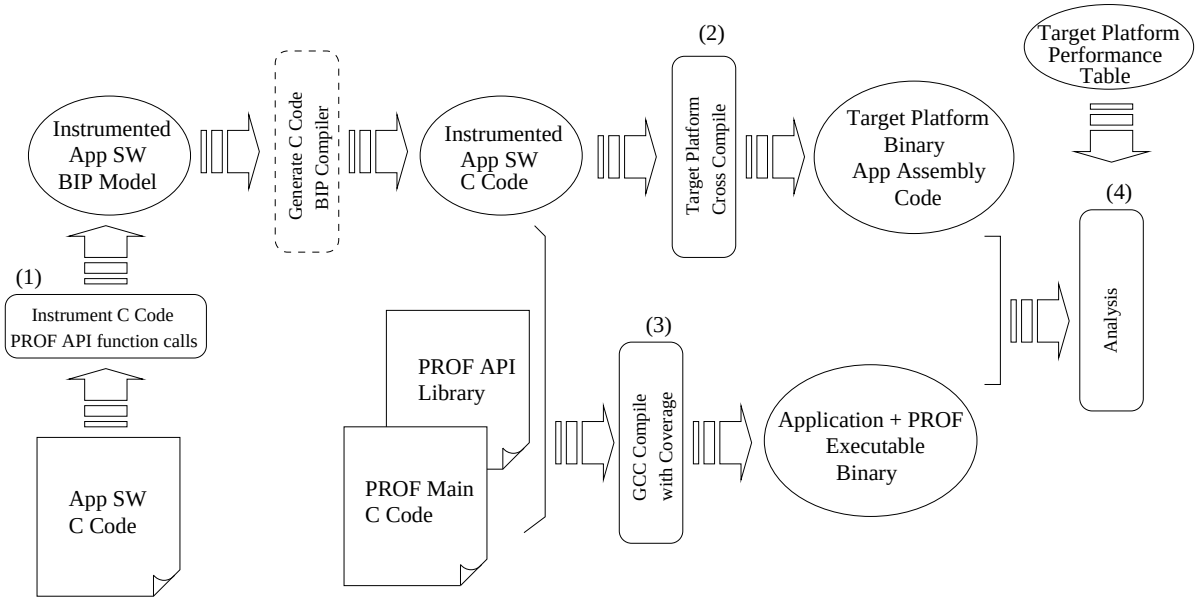


Figure 7.2: *Instruction Weight Table Profiling Flow*

Instrumentation The SW Application part of the BIP system model contains blocks of C code used to describe the behavior of each transition inside the BIP automata. API

functions calls are inserted in the beginning and at the end of each BIP transitions. These functions are critical for the analysis stage which we describe later. More specifically, there are four functions: *wpapi_init()*, *wpapi_begin()*, *wpapi_end()* and *wpapi_over()*. The analysis stage is triggered when function *wpapi_init()* is called. At the beginning some necessary environment valuables are configured and initialized. When the API function *wpapi_begin()* is called, we analyze the following block of C code to measure the execution delay on the target platform. The API function *wpapi_end()* signals that the calculated execution value is inserted back to the BIP System model to calibrate it. The API function *wpapi_over()* is used to terminate the process.

Cross compilation with target platform In the second stage, we cross compile the SW Application part of the BIP system model with the target platform compiler. The goal is to generate the corresponding low-level assembly codes which is necessary for the detection of the CPU arithmetic or load/store operations included in every line of C code.

GCC compilation with coverage As shown in Figure 7.2, the SW Application is compiled into an executable binary along with the API library with the use of GCC coverage parameters. There are also some extra files generated with suffix .gcov for line coverage statistics.

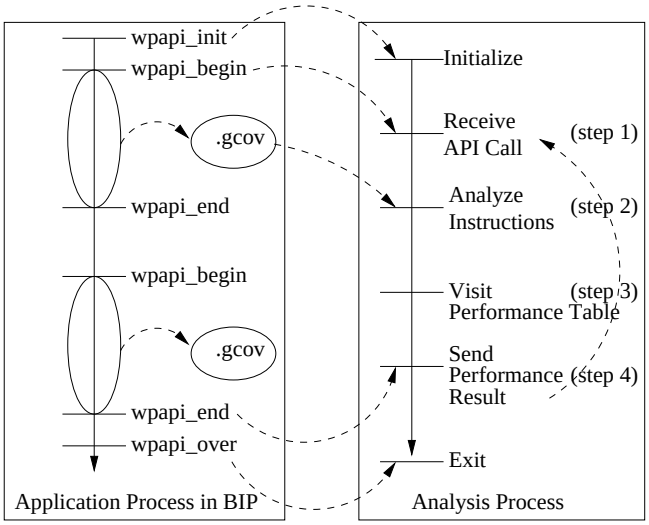


Figure 7.3: *Execution delay analysis*

Analysis For the analysis stage we instantiate a process which runs in parallel with the BIP simulation as shown in Figure 7.3. As soon as it receives the request from the application process, it starts to analyze performance on the given range C code. There are, mainly, four analysis steps in the analysis process. The first step is to receive the message of API call and get the information about which lines are about to be profiled. The second step is to analyze the C code with its coverage file (.gcov) and calculate its total instruction cost according to the assembly code generated by the target platform cross compiler. On the third step, we obtain the total cycle number of each given line of the C code. In this step, the total execution cycle number is analyzed by taking into account the instruction weight table. Other factors could also be considered such as the pipeline control, the data hazards and memory access conflicts. In the final step, the analysis process sends the profiling result back to the application process.

7.1.2 Platform Dependent Code Generation

The platform dependent code generation provides a method to obtain cycle-accurate execution times of SW-Processes by generating and deploying low level code on a virtual target platform. The method is based on an infrastructure developed in Verimag, for generating code from the system models in BIP. Seeking portability, the generated code targets a particular run-time that can be eventually deployed and run on different platforms, including P2012 (Section 11.1) and MPARM (Section 10.1). The run-time provides a generic API for thread management, memory allocation, communication and synchronization. The generated code is not bound to any particular platform and consists of the functional code and the glue code.

The functional code is generated from the application components consisting of processes and FIFOs. Processes are implemented as threads, and FIFOs are implemented as shared *queue* objects provided by the Native Programming Layer (NPL) library. The implementation in C contains the thread local data, queue handles and the routine implementing the specific thread functionality. The latter is a sequential program consisting of plain C computation statements and communication calls (e.g., *queue* API) provided by the run-time. A *read* transition is substituted by a *pop* API call on the respective queue handle. Similarly, a *write* transition is substituted by a *push* API call on its respective queue handle.

The glue code implements the deployment of the application to the platform, i.e., allocation of threads to cores and the allocation of data to memories. The glue code is essentially obtained from the mapping. Threads are created and allocated to cores according to the process mapping. Data allocation deals with allocation of the thread stacks and allocation of FIFO queues for communication. In particular, for MPARM deployment, every thread stack is allocated into the $L1$ memory of the core to which the thread is deployed. Queue handles and queue objects are allocated from the cluster shared $L2$ memory. All these operations are implemented by using the API provided by the run-time.

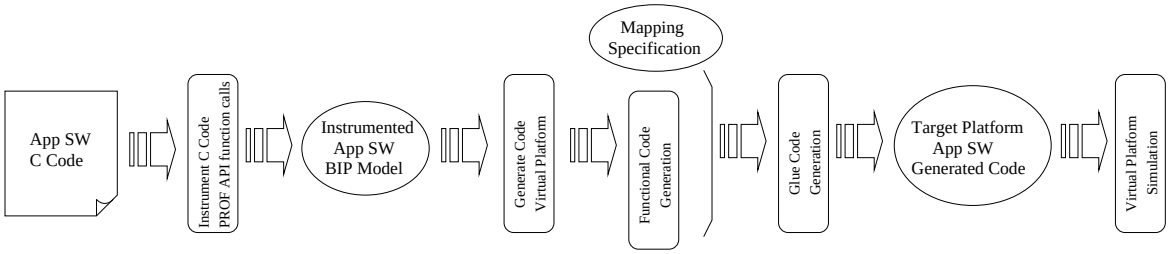


Figure 7.4: *Platform Dependent Code Generation Profiling Flow*

The code generation flow is illustrated in Figure 7.4. The code generator, as a tool, has been fully integrated into a tool-chain and connected to the BIP system model generation flow. The code is linked with the run-time, hardware dependent library, to produce the binary executable(s) for execution on the platform. The execution results concerning the software applications computational results are used to calibrate the System Model in BIP. However, the overall performance results obtained by the execution on the virtual platform are also used as a means of comparison the performance results obtained by the simulation of the corresponding System model in BIP.

For our experiments, we have used the Native Programming Layer (NPL), a common run-time implemented for both P2012 and MPARM platforms. For P2012, the generated code has been run on virtual platforms available in the P2012 SDK 2011.1, namely GEPOP

- the P2012 POSIX-based simulator - and the P2012 TLM simulator. For MPARM, the generated code is compiled by the `arm-gcc` compiler. The compiled code is linked with the run-time library to produce the binary image for execution on the MPARM virtual simulator.

7.2 Discussion

In this chapter, we presented the two methods we use to obtain accurate performance estimation of all C code in the application software. These methods are namely, the *Instruction Weight Table* and the *Platform Dependent Code Generation*.

The second method is considered more accurate than the first one, since it utilizes a dedicated virtual platform developed to simulate the functionality of a target hardware platform. The performance analysis results obtained from both methods are given in Chapter 10 and Chapter 11 with the experimentation upon different use cases.

In the next chapter, we present the performance analysis techniques we use to obtain performance results on the complete mixed hardware/software system model.

- Chapter 8 -

Performance Analysis

In this chapter, we focus on the performance analysis model incorporated in our system model generation dedicated to monitor the behavior of the target properties. In the next two sections, we describe the performance model developed in this work and more specifically the timed model in BIP extended with observers and we provide a discussion about performance models in general and the contribution of our own in the domain.

8.1 Performance Model

In our BIP design flow, system models are used to integrate the (extra-functional) hardware constraints into the software model according to some chosen deployment mapping. The system model is constructed through a series of transformations from the BIP models of respectively the application software and hardware platform. These two models are *composed* according to the mapping. The construction has been presented in the previous chapters. The transformations preserve functional properties of the application software model.

The system model is then calibrated by including timing constraints for execution on the chosen platform. These constraints define execution times for elementary functional blocks, that is, BIP transitions within the application software model. More precisely, the system model calibration is done by instrumenting the generated code with API function calls. The API provides cycle accurate estimates for executing a block of code in each processor. As analyzed in the previous chapter, the execution times are measured by two methods: the Instruction Weight Table and the Platform Dependent Code Generation method presented in the previous chapter. The Platform Dependent Code Generation is more accurate than the Instruction Weight Table method, since it utilizes a virtual platform to obtain the results.

The calibrated system model provides analysis of the non-functional properties such as contention for buses and memory accesses, transfer latencies, contention for processors, etc. All system properties are evaluated by simulation of the system model extended with observers. Observers are regular BIP components that sense the state of the system model and collect pertinent information with respect to the properties of interest i.e., the delay for particular data transfers, the blocking time on buses, etc. Actually, we provide a collection of predefined observers allowing to monitor and record specific information for most common non-functional properties. In the current work, we focus on observing the non-functional time properties. For this purpose, the generated system model in BIP

is developed to incorporate a performance model equipped with accurate mechanisms to capture time.

The performance model is specifically a timed model in BIP extended with observers. We provide below the formal definition of the timed model which we consider.

51 Definition (Timed Composition)
-Timing Model in BIP-
We define a Timed Composition T composed by k number of B Components as $T = \gamma^H(B_1, \ldots, B_k)$, where the set of interactions γ^H is defined as an hierarchical connector:
$\gamma^H = \gamma_1 \cup \ldots \cup \gamma_{k-1}$ *with,*
the initial connector γ_1 connected to the two first components such that:
$\gamma_1 = (\{B_1.tick, B_2.tick\}, \alpha_1, tick)$ *with*
$\alpha_1 = (\{B_1.tick, B_2.tick\}, true, skip)$
and the rest of the connectors recursively connected with the first one such that:
For all $i \in [2, k-1]$ we define:
$\gamma_i = (\{\gamma_{i-1}.tick, B_{i+1}.tick\}, \alpha_i, tick)$ *with*
$\alpha_i = (\{\gamma_{i-1}.tick, B_{i+1}.tick\}, true, skip)$
For all connectors γ_i, with $i \in [1, k-1]$ composing the hierarchical connector γ^H, the corresponding description in the BIP language is given below.

connector *type BroadcastTick(TickPort p_1, TickPort p_2)*
define p_1' p_2'
on p_1
on p_2
on p_1 p_2
export *port TickPort tick*
end

The connector of type BroadcastTick can be triggered even if only one component is available for interaction.

Considering the γ^H as a composite hierarchical connector, we assume that the exporting port of the connector is the exported port of the top-most connector of the hierarchy. Meaning, that the export $\gamma_{k-1}.tick = \gamma^H.tick$.

19 Example
In Figure 8.1, we illustrate an example of a timed model in BIP. It is a composition of four BIP atomic components B and an hierarchical connector γ^H as defined in the Composition Definition 51.

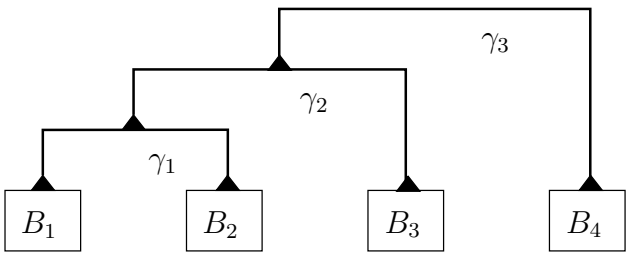


Figure 8.1: *Timed Composition in BIP with hierarchical connectors.*

Let us assume the Timed Composition as defined in Definition 51. We measure the maximum number of ticks that happened on the system by adding an Observer Component connected to the hierarchical connector.

52 Definition (Timed Composition with Observer)
Given a Timed Composition as defined in Definition 51, we define the Timed Composition with an Observer Component B_{ob} *as* $T_{ob} = \gamma(B_1, \ldots, B_k, B_{ob})$. $\gamma = \gamma^H \cup \gamma^{ob}$ *with,*
$\gamma^{ob} = (\{\gamma^H.tick, B_{ob}.tick\}, \alpha_{ob}, tick)$
, $\alpha^{ob} = (\{\gamma^H.tick, B_{ob}.tick\}, true, skip)$
The description in BIP language of the observation connector γ^{ob} *used to incorporate the Observer Component to the Timed model is given below.*

connector *type ObserverTick(TickPort* $tick_1$*, TickPort* $tick_2$*)*
define $tick_1$ $tick_2$
on $tick_1$ $tick_2$
export *port TickPort tick*
end

The connector of type ObserverTick implies strong synchronization between the Timed Model and the Observer Component enabling the latter to sense all ticks taking place. The hierarchy of the initial γ^H *connector is extended by the added connector* γ^{ob}*. The latter can be connected further on with other Timed models synthesizing a larger Timed Composition.*

20 Example
In Figure 8.2, we illustrate an example of a timed model with observer in BIP. We extend the timed model given in Example 19 by adding an observation connector (thick connector) on top of the hierarchical one. The observation connector strongly synchronized the timed model with an Observer Component which senses all ticks taking place.

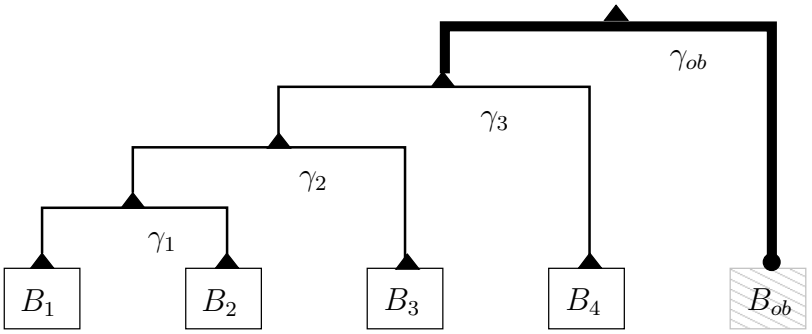


Figure 8.2: *Timed Composition in BIP with hierarchical connector and an Observer Component.*

The sub-model used to capture the computational constraints of the system model is given in Figure 8.3. Assuming a hardware platform with two processors, we connect each processor with a Processor Computational Observer via observation connectors. The latter are synchronized together with *BroadcastTick* type of connector, which eventually is synchronized with a Global Computational Observer capturing the total computational delay of the system.

A generic Observer Component in BIP is illustrated in Figure 8.4. We provide below the formal definition.

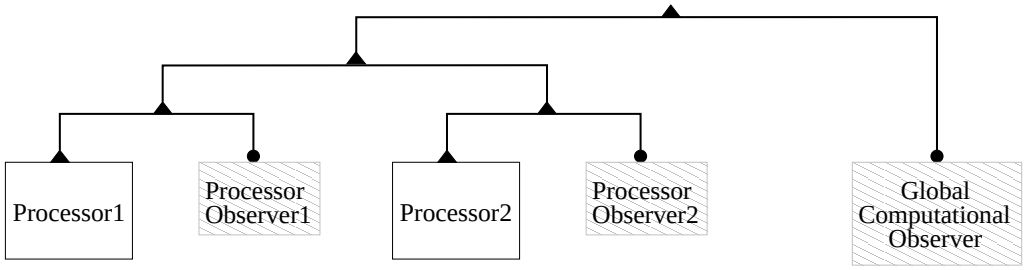


Figure 8.3: *Computational Observers in BIP System Model.*

53 Definition
We define the Observer Component as $OB = (L_{ob}, X_{ob}, P_{ob}, \mathcal{T}_{ob})$*, where:*

- $L_{ob} = \{\ell_1, \ell_2, \ell_3\}$,
- $X_{ob} = \{count, LOG\_FILE\}$,
- $P_{ob} = \{tick, begin, end\}$,
- $\mathcal{T}_{ob} = \{$
  $\tau = (\ell_1, begin, true, f_1, \ell_2)$ $,f_1 : count = 0;$
  $\tau = (\ell_2, tick, true, f_2, \ell_2)$ $,f_2 : count = count + 1;$
  $\tau = (\ell_2, end, true, f_3, \ell_3)$ $,f_3 : write(LOG\_FILE, count);$
  $\}$.

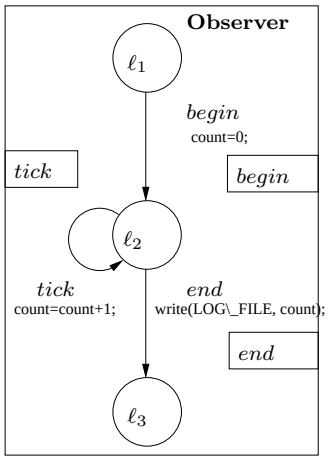


Figure 8.4: *Observer Component in BIP System Model*

The Observer Component has three control locations: the initial ℓ_1, the intermediate observing location ℓ_2 and the final ℓ_3 location. Transitions *begin* and *end* signal the starting and ending observation points of the component. The variable *count* is responsible for capturing time and is stored in a given *LOG_FILE* at the observation ending point. Transitions *begin* and *end* are usually connected to software application components to capture non-functional delay numbers concerning specific software application cycles.

Simulation is performed by using the native BIP simulation tool [bip]. The BIP system model extended with observers is used to produce simulation code that runs on top of the BIP engine, that is, the middleware for execution/simulation of BIP models. The outcome of the simulation with the BIP engine is twofold. First, the information recorded by

observers can be used as such to gain insight about the properties of interest. Second, the same information can be used to build much simpler, abstract stochastic models. These models can be further used to compute probabilistic guarantees on properties by using statistical-model checking. This two-phase approach combining simulation and statistical model-checking has been successfully experimented in a different context [BBB+10]. It is fully scalable and allows (at least partially) overcoming the drawbacks related to simulation-based approaches, that is, the long simulation times and the lack of confidence in the results obtained.

8.2 Discussion

In this chapter, we described the performance analysis model developed in the this work. The System Model in BIP is specifically a timed model extended with observers. All system properties are evaluated by simulation and can be further used to evaluate probabilistic guarantees with the help of statistical model-checking. The BIP design flow bridges the gap between simulation based and formal methods by presenting a complete framework based on BIP, which is a formal, rigorous and expressive language, and can be easily used both for native simulation and code generation for simulation on a virtual platform.

Related work focusing in performance analysis include methods based on simulation, trace-based co-simulation of different models, analytical models, analytical models with timed automata and methods that combine simulation and analytical approach. Simulation based methods use ad hoc executable system models such as [KDVvdW97] or tools based on SystemC [MGN03]. The latter provide cycle-accurate results, but in general, they have long simulation time as a major drawback. As such, these tools are not adequate for thorough exploration of hardware platform dynamics, neither for estimating effects on real-life software execution. Alternatives include trace-based co-simulation methods as used in Spade [LSvdWD01], Sesame [EPTP07] or Daedalus [NTS+08]. Additionally, there exist fast techniques that work on abstract system models such as DOL [TBHH07], which is based on Real Time Calculus [TCN02], and SymTA/S [Hea05]. They use formal analytical models representing a system as a network of nodes exchanging streams. They often oversimplify the dynamics of the execution characterized by execution times. Moreover, they allow only estimation of pessimistic worst-case quantities (delays, buffer sizes, etc) and require adequate abstract models of the application software. Building such models entails an additional significant modeling effort. Similar difficulties arise in performance analysis techniques based on Timed-Automata [AAM06, SBM09]. These can be used for modeling and solving scheduling problems. An approach combining simulation and analytic models is presented in [KPBT06], where simulation results can be propagated to analytic models and vice versa through adequate interfaces.

In the forthcoming chapters of the document, we present the implementation and experimentation part which describes the whole tool-chain supporting the design flow and provides experimental results in two case studies targeting two different HW platforms.

Part

Implementation and Experimentation

- Chapter 9 -

Tool

In this chapter, we provide the description of the tool-flow and all the individual tools contributing in the System modeling and the performance analysis. The associated toolbox is available in the website[1] of Verimag.

In Figure 9.1 we illustrate the whole tool-flow. The flow mainly contains four different sectors: Input Specification in DOL, System Model Generation, System Model Calibration and Performance Analysis. We developed four different tools which traverse the above sectors and automatize the tool-flow. The tools are given below:

- **DOL2BIP**, which generates the BIP SW Model.
- **BIPWeaver**, which constructs the BIP System Model.
- **Weight Table Profiler**, which provides a technique for System Model Calibration, calculating execution delays of computational blocks of code.
- **Code Generator**, which generates code to be simulated on a Virtual HW Platform, providing an alternative and more accurate method for calculating execution delays of computational blocks of code.

Except the **Weight Table Profiler**, which is implemented in C Language and Perl Scripts, all the other tools are implemented in **Java**. Details concerning the complexity of the tools are given in the next sections.

9.1 DOL2BIP Tool

In Algorithm 3, we describe the algorithm we developed to generate a BIP SW Model out of an input specification in DOL. The tool requires as an input the XML specification in DOL of the Process Network and the corresponding source code in C of each process. The Process Network XML file should conform with the process network XML Schema as it is defined in DOL. Further details and information about the DOL input specification is available in the following website[2]. In Figure 3.6 (Chapter 3), a process network example is found. As described in Section 3.3, the process network consists of three XML entities: *process*, *sw_channel* and *connection*. Each *process* has input, output ports and behavior written in C code. Each *sw_channel* has a single input port and a single output port,

[1] http://www-verimag.imag.fr/BIP-System-Designer.html
[2] http://www.tik.ee.ethz.ch/ shapes/dol.html

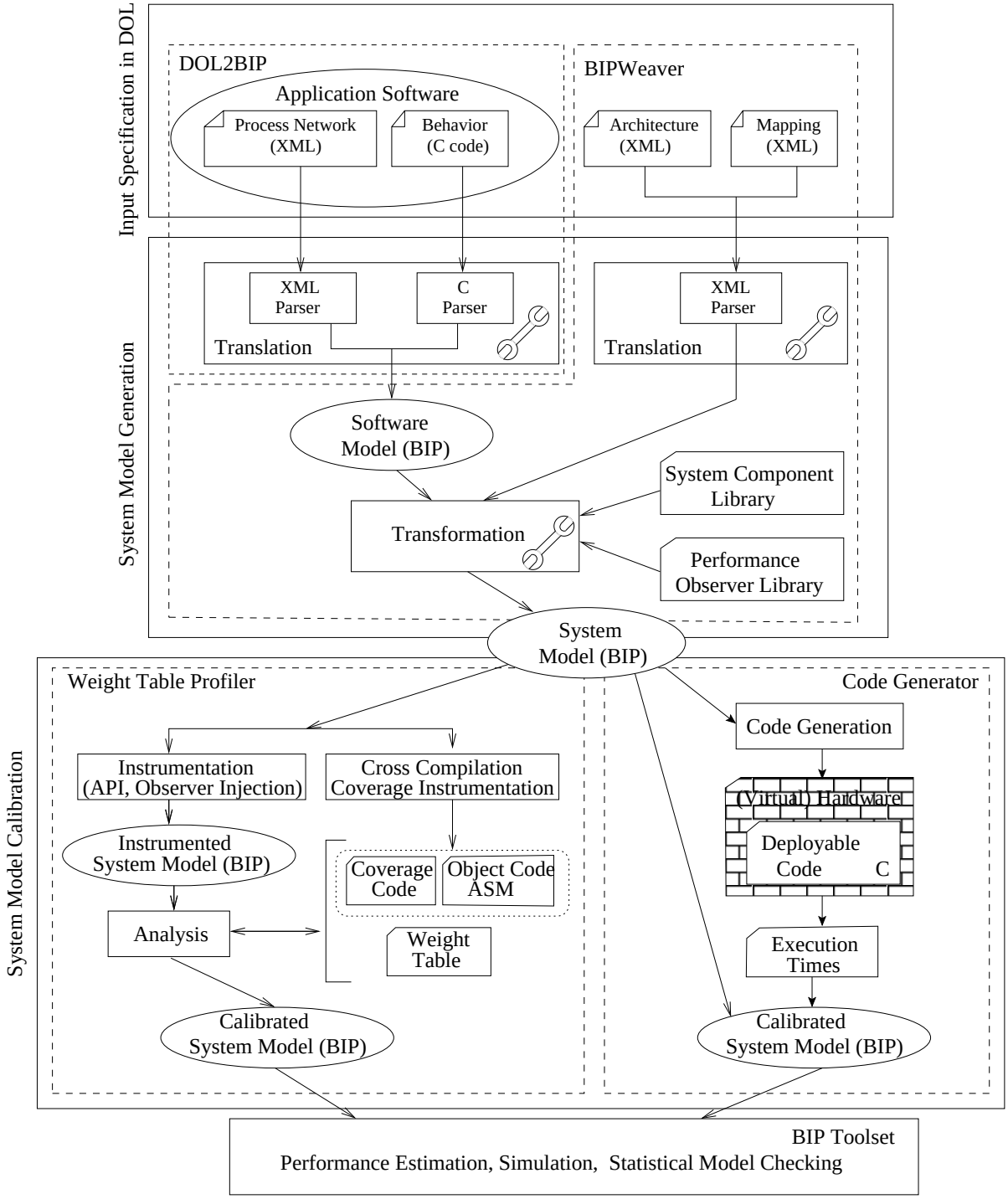


Figure 9.1: *System Model Tool Flow*

```
<variable value="3" name="N"/>

<iterator variable="i" range="N">
  <process name="square">
    <append function="i"/>
    <port type="input" name="0"/>
    <port type="output" name="1"/>
    <source type="c" location="square.c"/>
  </process>
</iterator>

<iterator variable="i" range="N + 1">
  <sw_channel type="fifo" size="10" name="F">
    <append function="i"/>
    <port type="input" name="0"/>
    <port type="output" name="1"/>
  </sw_channel>
</iterator>
```

Figure 9.2: *Fragment of the XML specification of the process network of Figure 9.3 using an iterator.*

uniquely associated with the ports of processes. Each *connection* defines the association between the *processes* and the *sw_channels*. A special element in the XML specification is the *iterator*. Its purpose is to iterate on the number of the above XML entities, namely, *processes*, *sw_channels* and *connections*. An example of an XML code using *iterators* is given in Figure 9.2. The variable N is used to bound the iteration number. The corresponding process network is shown in Figure 9.3.

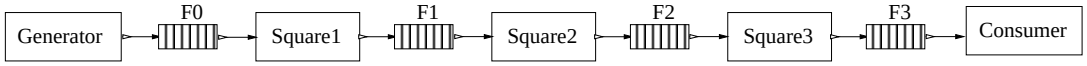


Figure 9.3: *Multiple Square application in DOL*

Initially, the *DOL2BIP* tool flattens the Process Network XML description. The result is an XML file where the *iterator* element is omitted and all children entities of *iterator* are instantiated by evaluating the corresponding *append* elements. The flattened XML file is parsed and the elements are loaded to Java Classes. The tool also parses the Process C Source Code and builds up an AST (Abstract Syntax Tree).

After all the inputs are processed, the tool visits the C code of each process. It detects the function calls and sets the interaction points of each BIP process atomic component. These points are the *DOL_Write*, *DOL_Read* function call. For each one of them, a BIP port is defined. Since the BIP ports are defined the tool completes the BIP process atomic component.

At the next step, according to the process network specification, the tool instantiates each BIP process component, the BIP SW_channel components according to the process network specification and creates the connectors between the above. The BIP SW model is now generated.

As we depict in Table 9.1, the *DOL2BIP* tool along with the *C2BIP* part consists of 12 Java files and a sum of approximately 5920 lines of code.

Algorithm 3 DOL2BIP Tool Algorithm

Require: *Process Network XML*, *Process C Source Code*
Ensure: *BIP SW Model*

```
load(Process Network XML)
load(Process C Source Code)
//Create atomic components
for all process (p) ∈ processList do
  Cc = get_source_code(p)
  create_C_Model(Cc)
  fc = find_function_call(DOL_Write, DOL_Read)
  inp = set_interaction_Points(fc)
  create_DOL_Write_Read_Ports(inp)
  construct_BIP_component_transition_system()
end for
//Create compound component
for all process (p) ∈ processList do
  create_component(p)
end for
for all SW_channel (sw) ∈ SW_channelList do
  create_component(sw)
end for
for all connection (cn) ∈ connectionList do
  or = get_origin_component(cn)
  tr = get_targer_component(cn)
  if or is process then
    create_connectors(DOL_write);
  else
    create_connectors(DOL_read);
  end if
end for
```

9.2 BIPWeaver

We present below the algorithm we use in our tool to generate the BIP System Model. The algorithm is divided in five sections.

Firstly, we parse and load the input BIP SW Model into Java classes, the Library of BIP System components, the Architecture XML and the Mapping XML specification. The corresponding XML Schemata are found in the DOL website [3].

Secondly, we iterate on the Process Atomic Types breaking the atomicity of the *DOL_Read* and *DOL_Write* actions.

Thirdly, we instantiate the Software components. In order to do this, we iterate on each process creating the Process Components. For each port of the Process Components we create the corresponding *FIFO_Write* or *FIFO_Read* Component and we connect the process with them by creating BIP connectors. We complete this step by creating the control connectors between each pair of *FIFO_Write* and *FIFO_Read* Components.

Fourthly, we create the Hardware components based on the Architecture specification. For each cluster specified in the Hardware Architecture, we do the following: For each processor, we create a HW Processor Scheduler Component. Respectively, for each cluster memory we create a Memory Component and then, we create the Network Interface Components. We iterate on the cluster interconnects specified on the architecture XML and we instantiate the cluster interconnect models in BIP in two steps. In the first step, we detect the Memory Components connected to the cluster interconnect. In the second step, we iterate on the processors and for each one of them, we create the corresponding Cluster Interconnect Components, after having checked if the processor uses the current cluster interconnect or not. To complete this step, we create the connectors between the Cluster Interconnect Components, the target Memory Component and the Network Interface Components. After having completed the Cluster Interconnect model, we create a Router Component for each cluster and we connect the Cluster Component via the Network Interface Components with the Router Component. To complete the NoC interconnect model, we create the connectors between the routers based on the specified network topology.

Finally, based on the Mapping XML specification, we create the necessary connectors between the instantiated software and hardware BIP components. Each Process Component is connected to the Processor Component (Processor Scheduler) which is mapped to. Along with the process, the fifo_rd and fifo_wr associated with the process are also connected to the same Processor Component. To complete the System model, we connect the fifo_rd and fifo_wr components with the Cluster Interconnect Components which lead to the Memory (local cluster or external cluster memory) which the initial FIFO Components are mapped on.

As depicted in Table 9.1, the *BIPWeaver* tool consists of 25 Java files summing up 8873 lines of code. The BIP component Library which specifies the System Model Components, the Software Model Components (i.e. FIFO Components) and the Performance Observer Library is described by 4 files in 7200 lines of code. There are also 2 different files of 887 LoC, used for the XML description of HW Platform Architectures.

9.3 Weight Table Profiler Tool

In Section 7.1.1, we described the Weight Table Profiling method in details. In this section, we briefly provide the **Weight Table Profiler Tool** algorithm and the tool's complexity.

[3] http://www.tik.ee.ethz.ch/ shapes/dol.html

Algorithm 4 BIPWeaver Tool Algorithm

Require: *BIP SW Model*, *Architecture XML*, *Mapping XML*, *System Component BIP Library*, *Performance Observer Library*
Ensure: *BIP System Model*

```
load(BIP SW Model, System Component BIP Library)
load(Architecture XML, Mapping XML)
for all process (p) ∈ processList do
  break_atomicity(DOL_Read, DOL_Write methods)
end for
//Create software components
for all process (p) ∈ processList do
  create_component(p)
  for all ports of process(p) do
    fc = create_components(fifo_wr, fifo_rd)
    create_connectors(fc, p)
  end for
end for
//Create hardware components
for all cluster cl ∈ clusterList do
  for all processor pr ∈ processorList do
    create_component(pr)
  end for
  //Create Cluster Communication Part
  icn=create_components(interconnect)
  m=create_components(memory)
  ni=create_components(network_interface)
  create_connectors(icn, m, ni)
  rt=create_component(router)
  create_connectors(ni,rt)
end for
for all router rt ∈ routerList do
  rt'=get_neighbor_router(rt)
  create_connectors(rt,rt')
end for
//Create connectors between software and hardware components
for all process (p) ∈ processList do
  pr = get_processor(p)
  create_connectors(p, pr)
  for all fifo_wr/fifo_rd fc used by p do
    create_connectors(fc, pr)
    mc=get_memory_mapped_to(fc)
    for all interconnect component icn_c used by pr do
      if interconnect component leads to mc then
        create_connectors(fc, icn_c)
      end if
    end for
  end for
end for
```

Table 9.1

	#Files	#Lines of Code
C2BIP	7	2892
DOL2BIP	5	3028
BIPWeaver	25	8873
BIP Library	4	7200
Architecture XML	2	887
Weight Table Profiler	22	4000
Code Generator	12	1744

As given in Algorithm 5, we initially load the BIP System Model in the tool. All the blocks of C code of the BIP transitions are instrumented with profiling functions. Then, we configure the HW Platform Cross Compiler and we generate the assembly code. Next, we compile the instrumented BIP model with the profiling library. Finally, we simulate the BIP model and we analyze the target blocks of code based on the low-level instructions of the corresponding assembly code and the pre-defined operation weight table.

The tool is implemented both in C language and Perl Scripts. It consists of approximately 22 files and 4000 lines of code, as seen in Table 9.1.

Algorithm 5 Weight Table Profiling Tool Algorithm

Require: *BIP System Model*, *HW Platform Cross Compiler*, *Operation Weight Table*, *Profiling Library*
Ensure: *Calibrated System Model*
load(*BIP System Model*)
sw_i = instrument(*BIP SW Application Components*)
hwpcc = configure(*HW Platform Cross Compiler*)
assembly = cross_compile(*sw_i*, *hwpcc*)
compile(*sw_i*, *Profiling Library*)
while simulate(*sw_i*) is running **do**
  analyze(*sw_i*, *assembly*, *Operation Weight Table*)
end while

9.4 Code Generator Tool

As presented in Section 7.1.2, we developed a method for generating code for simulation on a virtual hardware platform. In this section, we describe the tool algorithm we used to implement the tool as given in Algorithm 6. Firstly, we load the SW part of the BIP System Model, the Architecture details and the Mapping information into Java classes. Secondly, we implement every process as a thread and every SW_channel as shared queue object of the NPL Library. Finally, the final step, before the generated code is completed, is to allocate the threads to cores and the shared queues to memories.

The generated code is described in C language. The tool is implemented in Java and it consists of approximately 12 files and 1744 lines of code, as seen in Table 9.1.

Algorithm 6 Platform Dependent Code Generator Tool Algorithm

Require: *BIP System Model*, *Virtual Target Platform*, *Architecture XML*, *Mapping XML*
Ensure: *Calibrated System Model*
load(*BIP System Model*)
sys_spec = load(*Architecture XML*, *Mapping XML*)
for all process (p) $\in$ processList **do**
thread = implement_as_thread(p)
end for
for all SW_channel (sw) $\in$ SW_channelList **do**
sh_q_ob = implement_as_shared_queue_object(sw)
end for
for all threads **do**
allocate_to_cores(*thread*, *sys_spec*)
end for
for all data **do**
allocate_to_memories(data, *sys_spec*)
end for

9.5 Conclusion

In this chapter, we presented the complete tool-flow developed to generate a complete BIP System Model. The flow is completely automated and supported by a set of tools. These tools are the **DOL2BIP**, the **BIPWeaver**, the **Weight Table Profiler** and the **Code Generator**. **DOL2BIP** and **BIPWeaver** construct the BIP SW Model and the initial BIP System Model, respectively. **Weight Table Profiler** and **Code Generator** calibrate the System Model with accurate execution delay values of all the C code included in the BIP transitions. Although, these two tools were developed independently, **Code Generator** and specifically, simulation on the virtual hardware platform, has been proven more accurate in providing execution delay values for the transition blocks of code.

In the next chapters, we present the experimental results obtain with the use of the above tools. We applied the method in two different case studies: the MPARM and the P2012 Platform using a various set of running applications.

- Chapter 10 -

Case Study on MPARM Hardware Platform

In the previous chapter, we described the tools developed to support the complete design flow from the high level software and hardware specifications to fine-grain performance analysis.

In this chapter, we present the MPARM Hardware Platform, the corresponding hardware model in BIP, and a set of software applications considered as case studies to run on the MPARM Platform.

10.1 MPARM Hardware Platform

The MPARM [BBB+05, MPS] platform is a virtual ARM-based multi-cluster manycore platform. It is configured by the number of clusters, the number of ARM cores per cluster, and the interconnect between the clusters. The MPARM simulator allows experimentation with at most four clusters, each with eight ARM7-TDMI processors. The clusters are connected through a 2×2 NoC interconnect. The architecture is shown in Figure 10.1.

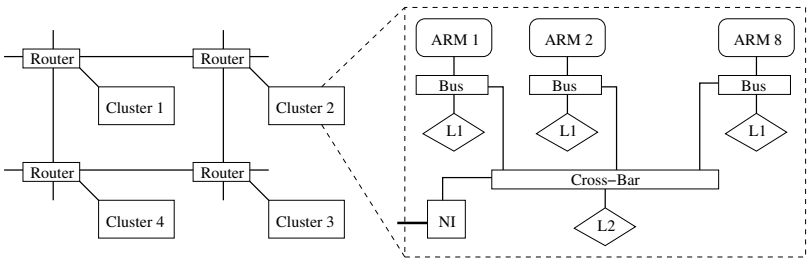


Figure 10.1: *An MPARM architecture with four clusters*

Inside a cluster, each ARM core is connected with its private DRAM ($L1$) memory through a local bus. There is also a shared cluster memory SRAM ($L2$) which is connected with the cores through a AMBA-AHB cross-bar interconnect. Although the MPARM is highly parametrized, we present in Table 10.1, some typical MPARM settings. These features characterize the MRARM processor speed, the memory access delays and the cross-bar interconnect. A part of the description of an MPARM cluster in XML is given in Figure 10.2.

A NoC-based infrastructure is used for inter-cluster communication, which consists of a router, a link, and the network interface (NI) of the individual clusters. In Table 10.2, we describe the characteristics of the NoC which we considered. The data transfer is

```
<cluster name="C1" type="MPARM">
    <processor name="P1" type="ARMv7">
       <memory name="Private" type="L1">
            <configuration name="cycles" value="1"/>
       </memory>
       <hw_channel name="local" type="Bus">  </hw_channel>
    </processor>
    . . .
    <processor name="P8" type="ARMv7">
       <memory name="Private" type="L1">
            <configuration name="cycles" value="1"/>
       </memory>
       <hw_channel name="local" type="Bus">  </hw_channel>
    </processor>
    <hw_channel name="X-bar" type="CrossBar">
       <configuration name="cyclesperbyte" value="1"/>
    </hw_channel>
    <memory name="Shared" type="L2">
       <configuration name="cyclesperbyte" value="2"/>
    </memory>
</cluster>
```

Figure 10.2: *Fragment of the DOL description of an MPARM cluster*

implemented using packages containing a set of flits in a specific order. There is a header flit with all the routing information, internal flits which carry all the data, and the tail flit which signals the end of a package. Consequently, important features are the packaging delay, the flit size, the router latency, the link latency and the router throughput.

Table 10.1: *MPARM Cluster Features*

	MPARM Cluster Features
CPU frequency	200MHz
DRAM Access Delay (conflict free) (L1)	0.5 cycles/byte
SRAM Access Delay (conflict free) (L2)	5.5 cycles/byte
AMBA-AHB	1 cycles/access (4 bytes)

The MPARM simulator provides cycle-accurate measurements for the execution of the applications on the virtual platform. This is achieved via generation of low level code as described in Section 7.1.2. The execution times of the SW-Processes are used to calibrate BIP System Models to accurately model computational delays. Henceforth, we will use the term MPARM execution to denote execution on the MPARM virtual simulator.

Another technique, used to obtain cycle-accurate measurements for ARM platforms, is the Weight Table Profiling, described in Section 7.1.1. In order to model ARM7 processors, we utilize the *arm-rtems-g++* cross compiler for generation of instruction-level assembly code and the ARM7 data sheet as an operation weight-table guideline.

10.2 MPARM Hardware Template Model in BIP

For the hardware model in BIP, we assumed all the local memories as DRAM with an access time of 2 cycles. The shared memory is a SRAM with an access time of 22 cycles. All

Table 10.2: *NoC Features*

	NoC Features
Packaging Delay (header creation)	2 cycles
Flit size	72-76 bits
Router Latency	1.2 ns/flit
Link Latency	200 ps/flit
Router throughput	800 Mflit/sec

CPU frequencies are assumed to be 200MHz. Communication paths are defined between all five processors (cores) using shared and local memories.

It is configured using five identical cores and a shared memory, connected via a shared AMBA-AHB cross-bar interconnect.

In Table 10.3, we catalog the MPARM Cluster features used to parametrize the MPARM virtual platform simulator and incorporated in the MPARM System Models in BIP. These features are the CPU frequency, the memory and interconnect access delay and the number of cores per cluster.

Table 10.3: *MPARM Cluster Parameters*

	MPARM Cluster Parameters
CPU Frequency	200MHz
DRAM Access Delay (L1) (conflict free)	0.5 cycles/byte
SRAM Access Delay (L2) (conflict free)	6 cycles/byte
AMBA-AHB	1 cycles/byte
Cores/Cluster	8

For the sake of simplicity, we present below a subset of the complete MPARM cluster model.

21 Example (MPARM Cluster Model in BIP)
Considering the parameters given in Table 10.3, we present an example of an MPARM architecture model of a single MPARM Cluster, configured with eight cores, one shared bus interconnect and one shared memory. Therefore, we firstly instantiate eight abstract components of the processors. These components will be filled with software processes, fifo routines and the processor scheduler, according to the given software application and the mapping specification. The software processes are calibrated, in advance, with accurate execution times with the instrumentation methods presented in Chapter 7. Secondly, we use one Bus Scheduler Component and eight Bus Path Components parametrized with the bus delay to model the Cross-bar interconnect. Lastly, the Bus Path Components are connected to the Shared Memory Component which is parametrized with the memory access delay. We illustrate the above in Figure 10.3.

The NoC parameters considered in both MPARM and P2012 platform models in BIP are given in Table 10.4. They characterize the packaging delay, the router and link delay, and the number of clusters and routers (network nodes).

In Figure 10.4, we illustrate the *Router Component* in BIP. It is composed using the *Router IN* and *Router OUT* Components for each direction: the North, the South, the East, the West and the Local Cluster. The parameters specifying the component are the router delay and the link latency which is incorporated into the Router Components.

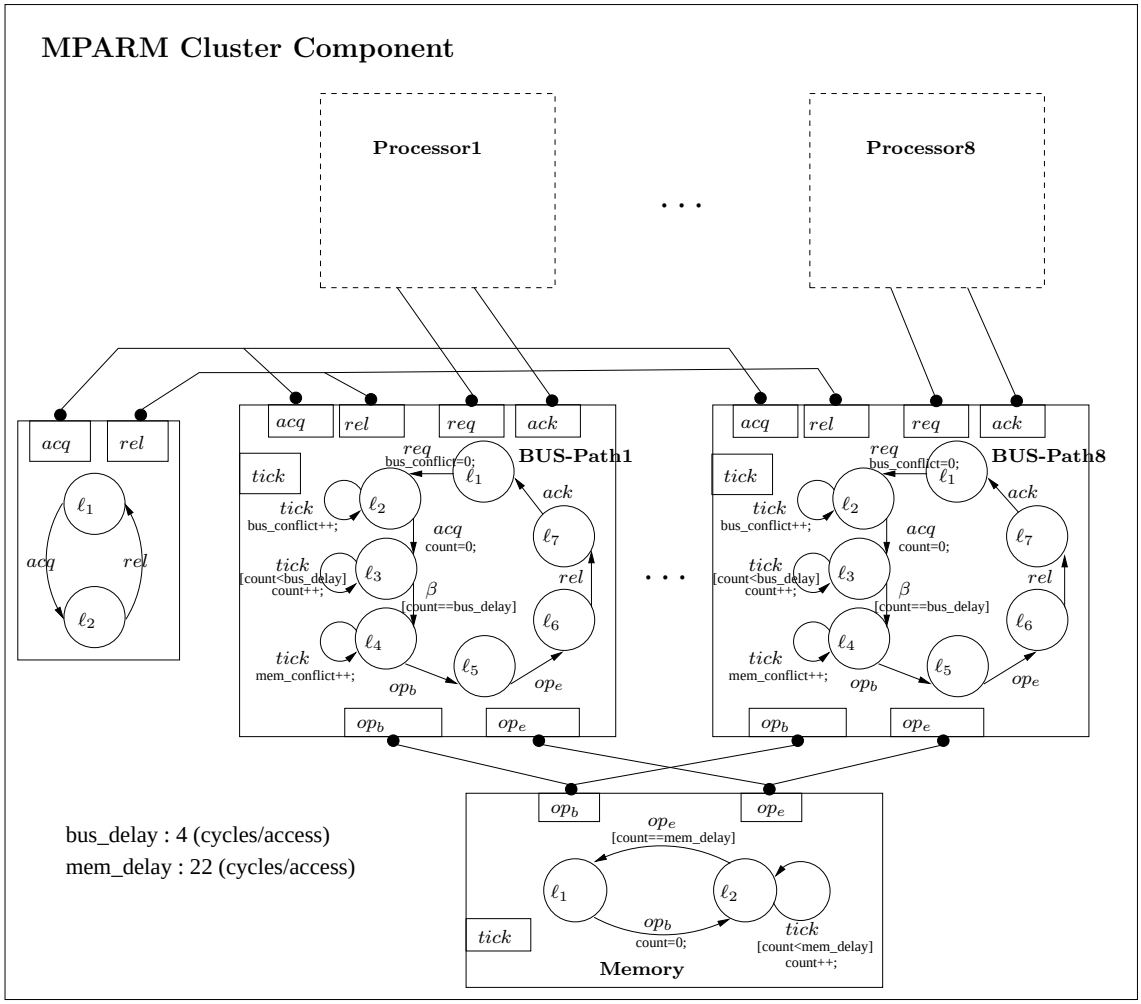


Figure 10.3: *MPARM Cluster Model in BIP*

Table 10.4: *NoC Parameters*

	NoC Parameters
Packaging Delay (header creation)	2 cycles/package
Router Delay	2 cycles/package
Link Latency	1 cycle/package
Clusters	4
Routers	4

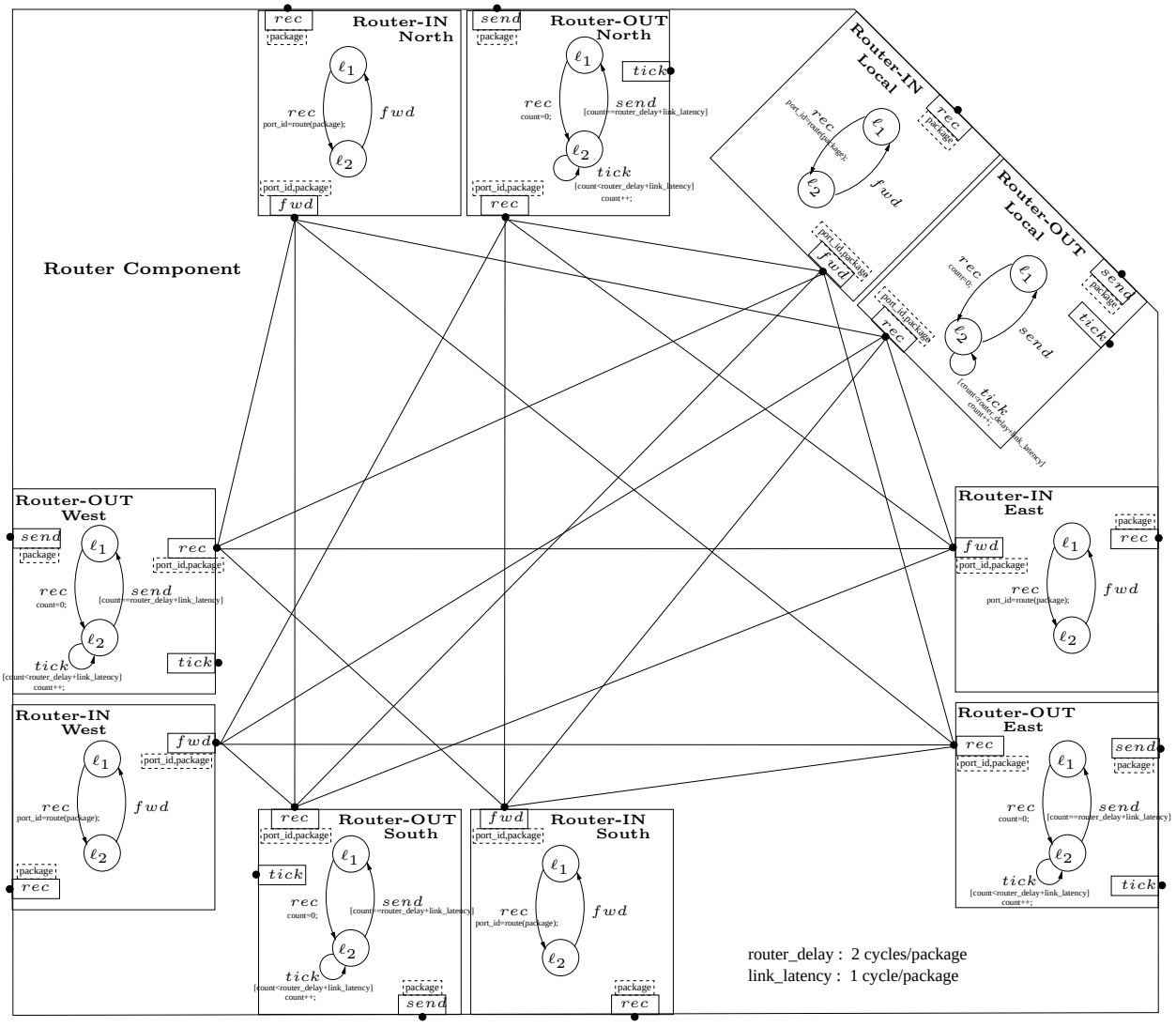


Figure 10.4: *Router Component in BIP*

In Table 10.5, we quote the complexity of the abstract model of MPARM considered. The number of BIP types and BIP type instances needed to model the MPARM platform in BIP are given in Table 10.6.

Table 10.5: *MPARM architecture*

Hardware Component	Number
Cores	8
Local Memories	8
Local Buses	8
Shared Memories	1
Shared Buses	1
Network Interface	1
Clusters	4
Routers	4

Table 10.6: *BIP MPARM Library*

	BIP types	BIP type instances
Ports	8	268
Connectors	12	167
Atomic Components	53	78
Compound Components	4	8

Based on the MPARM Platform model in BIP described in the current section, we generate various mixed hardware/software System models of a series of application case studies. The experimentations conducted are presented in the forthcoming sections.

10.3 MPEG-2 Application on MPARM

The MPEG2 decoder application software decodes a set of moving pictures and associated audio information. The corresponding process network is depicted in Figure 10.5. We used a case study where there are seven processes *DispatchGops* (*DG*), *DispatchMb* (*DM*), *DispatchBlocks* (*DB*), *TransformBlock* (*TB*), *CollectBlocks* (*CB*), *CollectMb* (*CM*) and *CollectGops* (*CG*) and six software FIFO channels *C1*, ..., *C6*. The behavior of the decoder is described in approximately 7000 lines of C code. The process and the FIFO mappings are illustrated on Table 10.7.

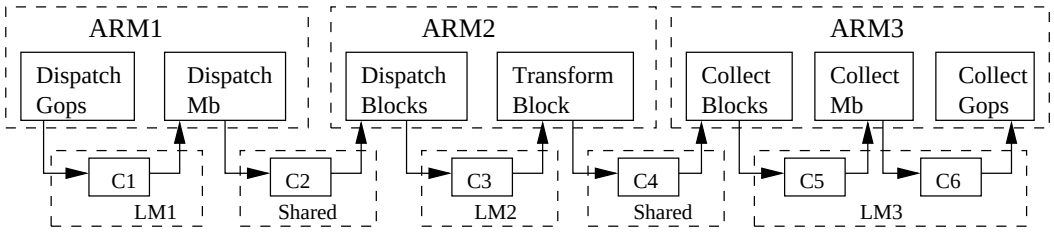


Figure 10.5: *MPEG-2 Decoder application software and a mapping*

For the MPEG-2 case study the BIP System Model contains about 90 BIP atomic components, 340 BIP interactions and 30K lines of BIP code generating approximately

Table 10.7: *Mapping Description of the processes and the FIFOs*

	$ARM1$	$ARM2$	$ARM3$	$ARM4$	$ARM5$	
1	*all*					
2	DG, DM, DB, TB	CB, CM, CG				
3	DG, DM	DB, TB	CB, CM, CG			
4	DG	DM, DB	TB	CB, CM, CG		
5	DG	DM, DB	TB	CB, CM	CG	
6	DG, DM	DB	TB	CB	CM, CG	
7	DG	DM, DB	TB	CB, CM	CG	

	Shared	$LM1$	$LM2$	$LM3$	$LM4$	$LM5$
1		all				
2	C4	C1, C2, C3	C5, C6			
3	C2, C4	C1	C3	C5, C6		
4	C1, C3, C4		C2		C5, C6	
5	C1, C3, C4, C6		C2		C5	
6	C2, C3, C4, C5	C1				C5
7		C1	C2, C3	C4	C5, C6	

100K lines of C code. The method used for calibrating the System model was the Weight Table Profiling, described in Section 7.1.1. The total computation and communication delays for decoding 5 frames for different mappings are shown in Figure 10.6. The MPEG-2 process network is characterized as computationally intensive. The more we distribute the computational load to different CPUs, the smaller is the computational delay. Since the FIFOs are few, there is small difference in the communication delays between the different mappings, except for mapping (1) where all processes and FIFOs are mapped on a single core. However, as we distribute the processes into more cores, the communication delay increases and more bus conflicts occur.

Regarding the overall performance, we observe that Mapping (2) results into the worst frame throughput due to high computational and communication delays. Although in mapping (4) we use more CPUs than mapping (3), the overall performance decreased due to higher occurrence of communication conflicts, as illustrated in Figure 10.7. The best throughput is achieved in Mapping (7) due to the usage of five CPUs and their local memories.

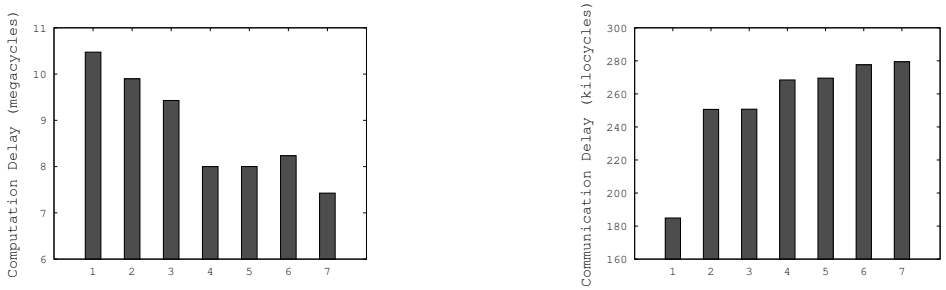


Figure 10.6: *Mpeg-2 Performance Analysis Results*

10.4 MJPEG Application on MPARM

The MJPEG decoder application software reads a sequence of JPEG frames and displays the decompressed video frames. The process network of the application software is shown

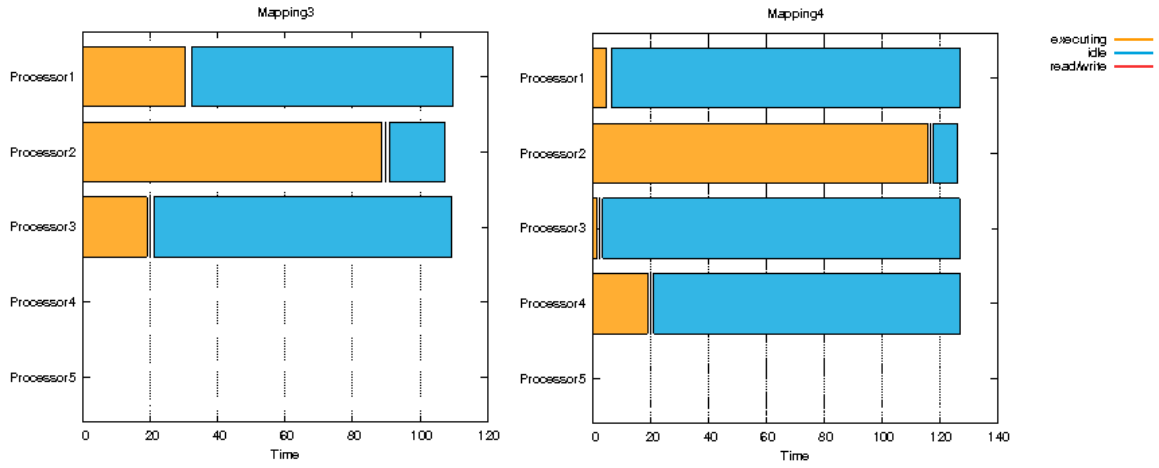


Figure 10.7: *Mpeg-2 Processor Performance Analysis Results*

in Figure 10.8. It contains five processes *SplitStream* (*SS*), *SplitFrame* (*SF*), *IqzigzagIDCT* (*IDCT*), *MergeFrame* (*MF*) and *MergeStream* (*MS*). The DOL description of the application processes contains approximately 1600 lines of C code.

The system model in BIP contains 42 atomic components and 198 interactions, and consists of approximately 7325 lines of BIP code.

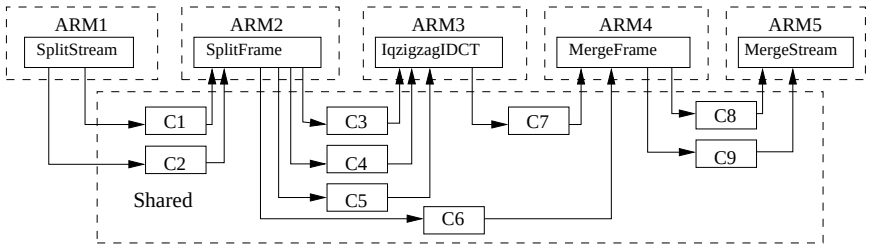


Figure 10.8: *Process Network of the MJPEG Decoder Application*

To experiment on this case study we calibrated the BIP System model using both profiling methods described in Chapter 7. We quote the results below.

Weight Table Profiling We analyzed the effect of eight different mappings on the total computation and communication delay for decoding a frame. The process and the FIFO mappings are illustrated on Table 10.8.

The total computation and communication delays for decoding a frame for different mappings are shown in Figure 10.9. Mapping (1) produces the worst computation delay as all processes are mapped to a single processor. Mapping (2) uses two processors, but still the performance does not improve much. Mapping (3) drastically improves performance as the computation load is balanced. The other mappings cannot further enhance performance as the load cannot be further balanced, even if more processors are used. The communication overhead is reduced if we map more FIFOs to the local memories of the processors. The bus and memory access conflicts are shown in Figure 10.9. As more FIFOs are mapped to the local memory, the shared bus contention is reduced. However, this might increase the local memory contention, as shown for (8).

Table 10.8: *Mapping Description of the processes and the FIFOs*

	*ARM*1	*ARM*2	*ARM*3	*ARM*4	*ARM*5
1	*all*				
2	*SS*, *SF*, *IQ*	*MF*, *MS*			
3	*SS*, *SF*	*IQ*, *MF*, *MS*			
4	*SS*, *SF*	*IQ*	*MF*, *MS*		
5	*SS*, *MS*	*SF*	*IQ*	*MF*	
6	*SS*	*SF*	*IQ*	*MF*	*MS*
7	*SS*, *SF*	*IQ*	*MF*, *MS*		
8	*SS*	*SF*	*IQ*	*MF*	*MS*

	Shared	*LM*1	*LM*2	*LM*3	*LM*4
1		all			
2	C6, C7	C1, C2, C3, C4, C5	C8, C9		
3	C3, C4, C5, C6	C1, C2	C7, C8, C9		
4	C3, C4, C5, C6, C7	C1, C2		C8, C9	
5	all				
6	all				
7	C6, C7	C1, C2, C3, C4, C5		C8, C9	
8		C1, C2	C3, C4, C5, C6	C7	C8, C9

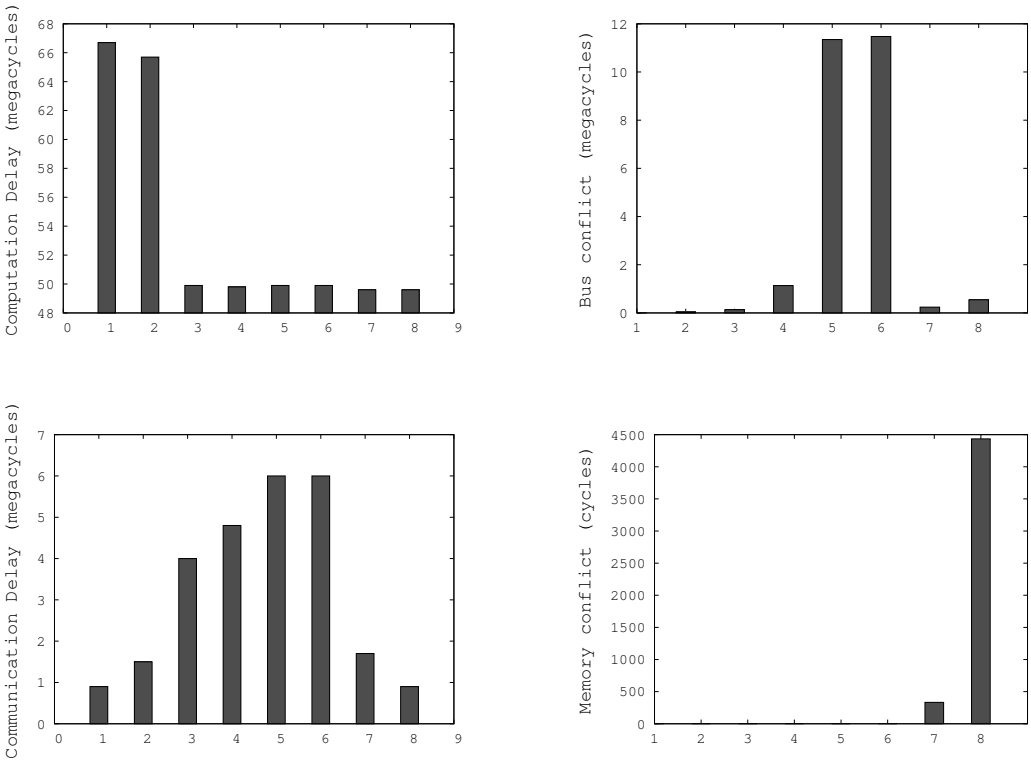


Figure 10.9: *Mjpeg Performance Analysis Results*

Platform-Dependent Code Generation Profiling The implementation generated for MPARM execution is approximately 3174 lines of code.

For the experiments, we mapped the application on a single MPARM cluster. Each computational process is deployed into an ARM processor and all the FIFO buffers are allocated to the $L2$ shared memory. The performance results per process obtained by simulation of the system model are depicted in Figure 10.10. We remark that process

IqzigzagIDCT is the heaviest in terms of computation, while process *MergeStream* stays idle most of the time. The low values of memory conflicts highlights the restricted parallelism within the application.

At system level, we measured the total execution time needed for the decompression of a single frame. Using BIP system model simulation, this time is estimated at 472.88 Mcycles. This result is very near to the cycle-accurate value obtained by measuring the MPARM execution, which is 468.83 Mcycles. The relative error of our estimation is therefore less than 0.87%. Regarding analysis time, BIP system model simulation outperforms execution on (virtual) MPARM. The former completes in $9'46''$ and is approximately 5.2 times faster than the second, which completes in $50'48''$. The above numbers are given in Table 10.9.

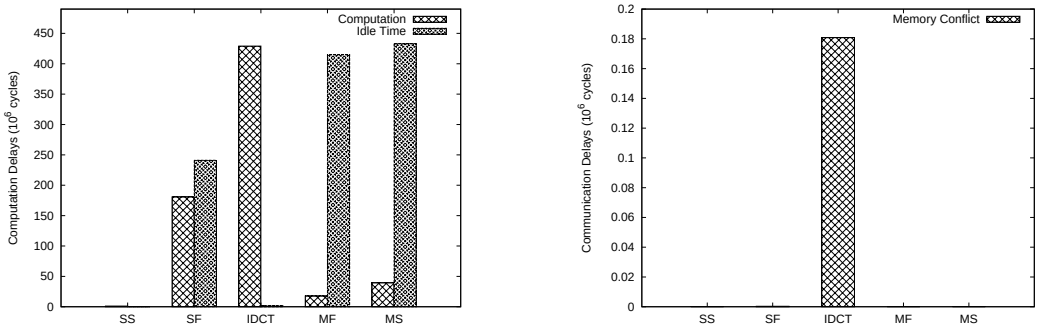


Figure 10.10: *Performance Results of Computational Processes in MJPEG Decoder*

Table 10.9: *Simulation Comparison in MPARM & BIP System Model (*10^6 *cycles)*

Cycle Count	MPARM	468.83
	BIP System Model	472.88
	Accuracy	0.87%
Simulation Time	MPARM	50' 48"
	BIP System Model	9' 46"
	Speed-up	5.20

Notably, BIP System Model calibration using Platform-Dependent Code Generation is time consuming, since it based on the MPARM execution on the virtual platform to obtain accurate execution time delays of the SW-processes. However, since the derived execution time delays involve pure computation parts of the processes and considering that the virtual platform is configured with each process running on an independent CPU, we assume that the calibration numbers remain the same for each mapping. Based on these numbers, we apply a set of various mappings to analyze, through BIP simulation, the performance of the System nodes on each case.

10.5 Fast Fourier Transform (FFT) Application on MPARM

A Fast Fourier transform (FFT) is an efficient algorithm to compute the Discrete Fourier transform (DFT) and its inverse. A DFT decomposes a sequence of values into components of different frequencies. This operation is useful in many fields but computing it directly

from the definition is often too slow to be practical. An FFT is a way to compute the same result more quickly.

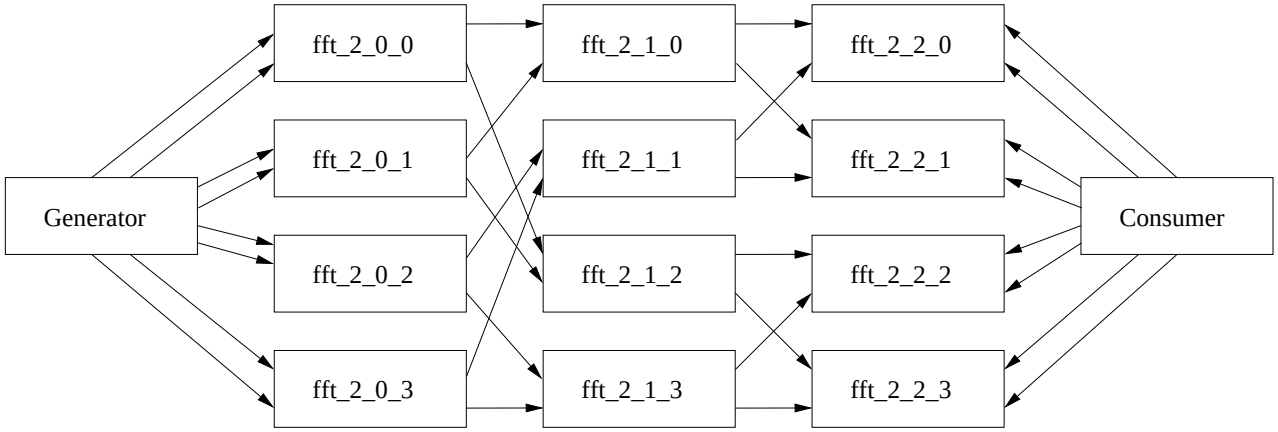


Figure 10.11: *FFT application*

The process network of the application software is shown in Figure 10.11. It contains a *Generator* and a *Consumer* process and twelve intermediate *FFT* processes aligned in a 3×4 matrix. The number of FIFO channels are 32. The process network is described in DOL using approximately 1616 lines of code.

As target platform we considered the complete MPARM platform, configured with four cluster and a NoC infrastructure. The generated System Model in BIP contains 168 atomic components, 706 interactions, and consists of approximately 9641 lines of BIP code. The above implementation characteristics are depicted in Table 10.10.

Table 10.10: *DOL, BIP Models and MPARM Implementation Characteristics*

FFT		
DOL Process Network	# processes	14
	# FIFOs	32
	# lines of code	1616
BIP System Model	# components	168
	# interactions	706
	# lines of code	9641
MPARM implementation	# lines of code	3986

The calibration method used for the System Model was the Platform Dependent Code Generation. We catalog the execution times of the SW-processes in Table 10.11. The code generated for the MPARM execution on the virtual platform is around 3986 LoC.

We analyzed the effect of five different mappings illustrated in Figure 10.12. Each process is mapped on an individual CPU located in different clusters and each FIFO is mapped on a L2 cluster memory. More specifically, each FIFO is mapped on the same cluster which the process reading from the FIFO is mapped on.

The performance results per process obtained by simulation of the system model are depicted in Figure 10.13. The high communication and idle time delays observed at the *fft_2_2_0*, *fft_2_2_1*, *fft_2_2_0* and *fft_2_2_2* processes are due to the blocking read operations, which particularly for these processes are the most time-consuming since data are not immediately available. Figure 10.14 illustrates the total communication delays per mapping,

Table 10.11: *Execution times for computational routines on FFT processes (in* 10^6 *cycles)*

FFT	
FFT2_0_0	2.52
FFT2_0_1	2.51
FFT2_0_2	2.44
FFT2_0_3	0.13
FFT2_1_0	2.33
FFT2_1_1	32.35
FFT2_1_2	2.43
FFT2_1_3	32.32
FFT2_2_0	2.36
FFT2_2_1	32.41
FFT2_2_2	32.32
FFT2_2_3	34.09

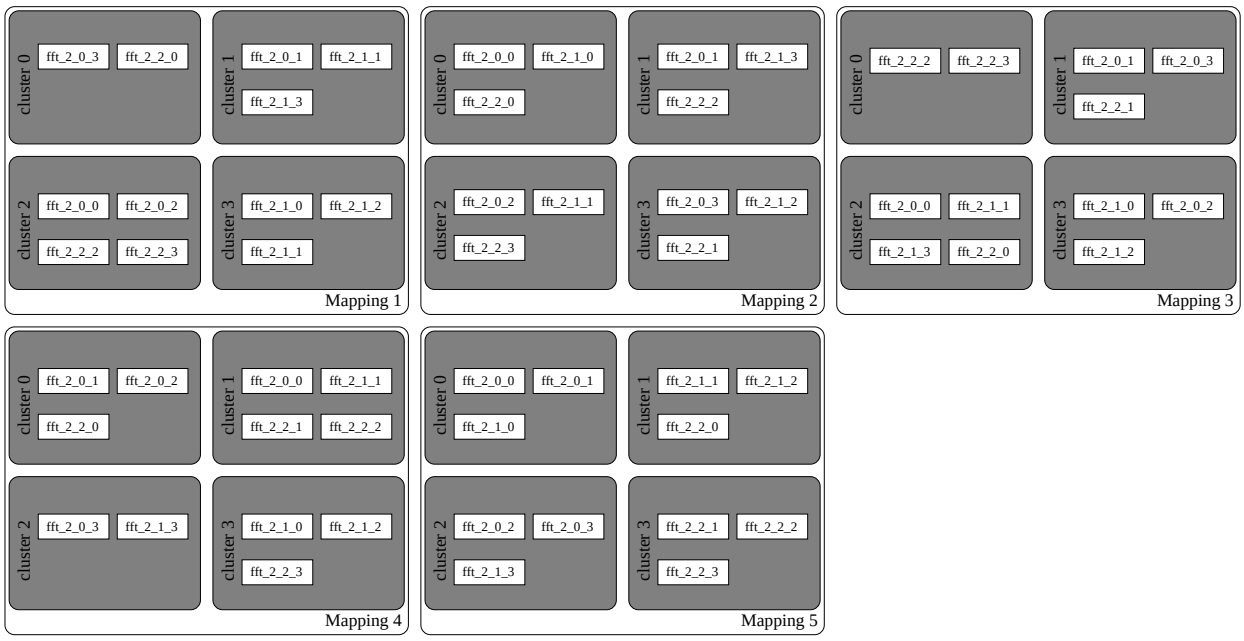


Figure 10.12: *FFT Mappings on 4-Cluster MPARM Platform*

highlighting mapping 2 as the one with the lowest communication overhead. In terms of total execution time, all mappings result at almost same value due to significantly higher computation delay of the *Generator* and *Consumer* compared to the FFT processes.

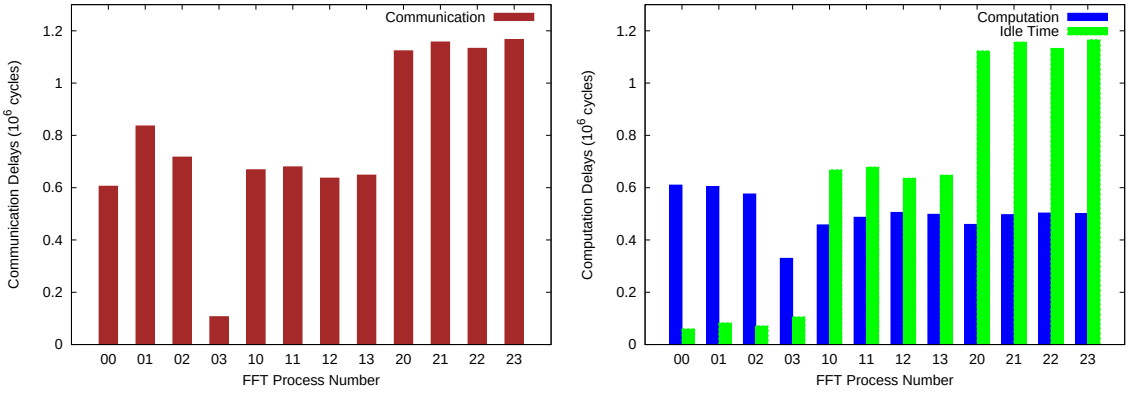


Figure 10.13: *FFT Performance Analysis Results per process*

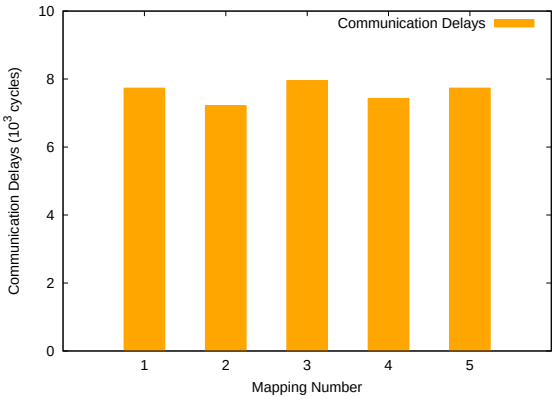


Figure 10.14: *FFT Performance Analysis Results - Mapping1*

10.6 Demosaicing Algorithm Application on MPARM

Demosaicing algorithm is a digital image processing algorithm used to reconstruct a full color image from the incomplete color samples output from an image sensor, overlaid with a color filter array (CFA). More specifically, the algorithm generates YUV components by transformations on the initial raw image.

Demosaicing application works on 5×5 matrices.The resulting pixels are the resulting averages of centered points of each matrix, which results to the loss of four lines and four columns of the initial image.

The process network of the application software is shown in Figure 10.15. It contains a *Splitter* and a *Joiner* process, a pre-demosaicing (*Demo_pre*) and a post-demosaicing

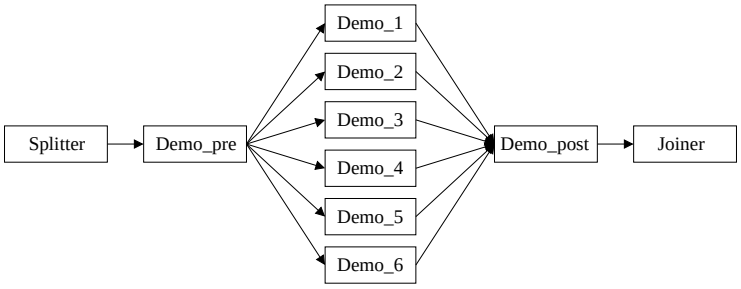


Figure 10.15: *Demosaicing application*

(*Demo_post*) process and six internal demosaicing *Demo* processes that run in parallel. The number of FIFO channels are 28. The process network is described in DOL using approximately 1672 lines of code.

As target platform we considered the complete MPARM platform, configured with four cluster and a NoC infrastructure. The generated System Model in BIP contains 152 atomic components, 614 interactions, and consists of approximately 15366 lines of BIP code. The above implementation characteristics are depicted in Table 10.12.

Table 10.12: *DOL, BIP Models and MPARM Implementation Characteristics*

Demosaicing		$Model(1 \times 6)$
DOL Process Network	# processes	10
	# FIFOs	28
	# lines of code	1672
BIP System Model	# components	152
	# interactions	614
	# lines of code	15366
MPARM implementation	# lines of code	10124

The calibration method used for the System Model was the Platform Dependent Code Generation. We catalog the execution times of the SW-processes in Table 10.13. The code generated for the MPARM execution on the virtual platform is around 3986 LoC.

Table 10.13: *Execution times for computational routines on demosaicing processes (in* 10^6 *cycles)*

Demosaicing	$Model(1 \times 6)$
Demo 1	14.23
Demo 2	27.03
Demo 3	26.86
Demo 4	26.53
Demo 5	23.87
Demo 6	14.66

We analyzed the effect of three different mappings illustrated in Figure 10.16. Each process is mapped on an individual CPU located in different clusters and each FIFO is mapped on a L2 cluster memory. More specifically, each FIFO is mapped on the same cluster which the process reading from the FIFO is mapped on.

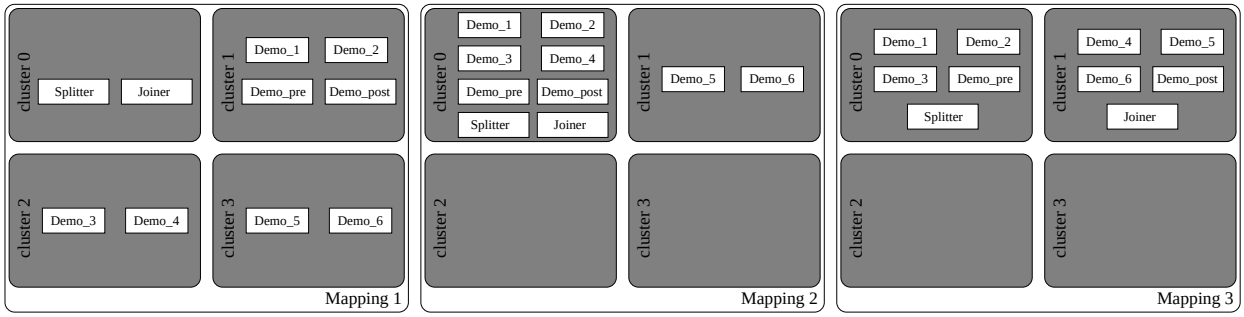


Figure 10.16: *Demosaicing Mappings on 4-Cluster MPARM Platform*

The performance results per process obtained by simulation of the system model are depicted in Figure 10.17. The high communication and idle time delays observed at the *Demo_00* and *Demo_05* processes are due to the blocking write operations, which particularly for these processes are the most time-consuming. This is observed because the *Demo_00* and *Demo_05* processes complete their computations quicker than the other processes (see Table 10.13) and by the time they start writing the results to the *Demo_post* process they are conflicting with the read operations of the rest processes These read operations are longer because they involve greater amount of data. Figure 10.18 illustrates the total communication delays per mapping, highlighting mapping 1 as the one with the lowest communication overhead. Mapping 1 is the only one with all the processes spread onto all four available cluster resulting to greater communication delays. In terms of total execution time, all mappings result at almost same value due to the significantly higher computation delay of the *Splitter*, *Joiner* and the pre and post demosaicing processes compared to the six internal demosaicing processes.

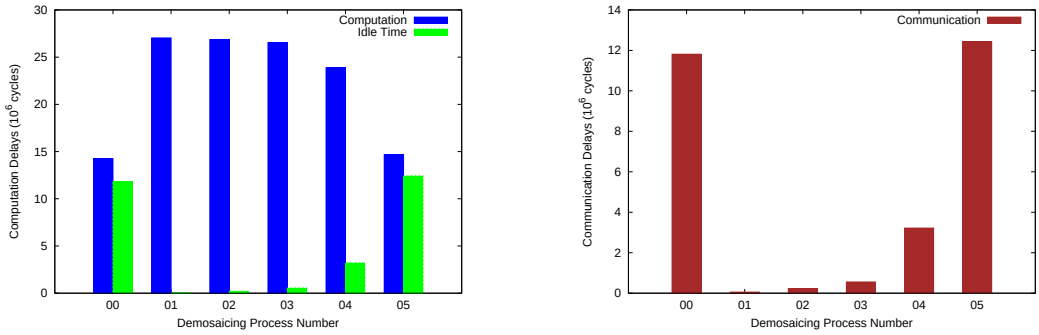


Figure 10.17: *Demosaicing Performance Analysis Results per process*

10.7 Cholesky Decomposition Application on MPARM

Cholesky Factorization decomposes a Hermitian positively-defined real-valued matrix A into the product $L \cdot L^T$ of a lower triangular real-valued matrix L and its conjugate transpose L^T. The Cholesky decomposition is used for solving numerically linear equations $Ax = b$. If A is symmetric and positive definite, then we can solve $Ax = b$ by first computing the Cholesky decomposition $A = L \cdot L^T$, then solving $Ly = b$ for y, and finally

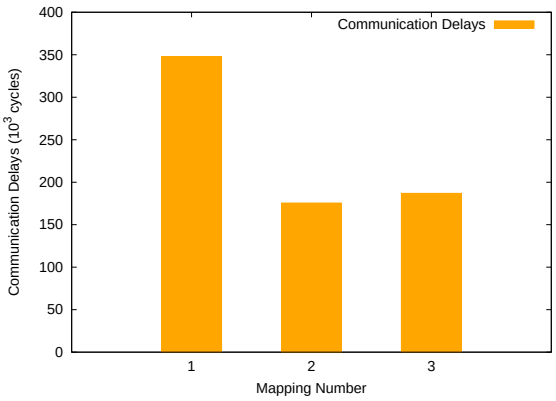


Figure 10.18: *Demosaicing Performance Analysis Results - Mapping3*

Algorithm 7 Right-Looking Block-Based Cholesky Factorization

Require: *A Hermitian, positive definite matrix*
Ensure: $A = L \cdot L^T$, *L lower triangular*

```
for k = 1 to B do
    L_kk := seq-cholesky(A_kk)
    L_kk^{-T} := invert(transpose(L_kk))
    for i = k + 1 to B do
        L_ik := A_ik · L_kk^{-T}
    end for
    for j = k + 1 to B do
        L_jk^T := transpose(L_jk)
        for i = j to B do
            A_ij := A_ij − L_ik · L_jk^T
        end for
    end for
end for
```

solving $L^T x = y$ for x.

The sequential Cholesky factorization algorithm has computational complexity $\mathcal{O}(N^3)$ for matrices of size $N \times N$. In this paper, our starting point is the sequential right-looking block-based version [OS85] provided as algorithm 7 which provides immediate support for parallelization. In this algorithm, B denotes the number of blocks composing the original matrix A, that is $A = (A_{ij})_{1 \leq j \leq i \leq B}$ and every A_{ij} is a block matrix of size $K = N/B$. The algorithm computes the matrix L, block by block, such that $A = L \cdot L^T$. Algorithm 7 is easily parallelizable by separating computations related to different ij-blocks on different processes P_{ij}. Nevertheless, interactions between these processes are highly non-trivial. There are complex patterns for data dependencies, as illustrated in Figure 10.19 for the cases $B = 2, 3, 4$. Moreover, the amount of computation carried by each process is different. That is, as factorization proceeds, processes with higher indexes (i, j) become computationally more intensive. Furthermore, both data dependencies and the local amount of computation are tightly related to the decomposition size B as well as to the block size K. Altogether, finding optimal implementation on multi-processor platforms with fixed communication and computation resources is a non-trivial problem.

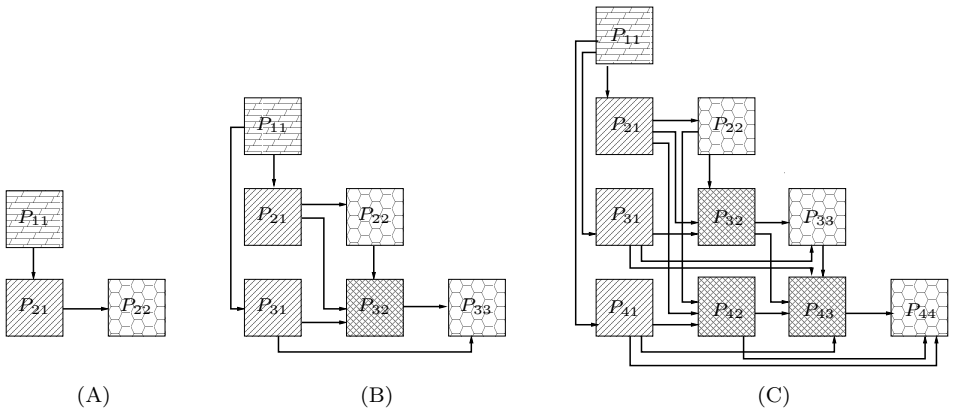


Figure 10.19: *Data dependencies for $2 \times 2(A), 3 \times 3(B)$ and $4 \times 4(C)$ process decomposition. Identical patterns indicate respectively a similar amount of local computation (processes) or potential for parallel communication (data dependencies).*

For every B, we denote by *Cholesky(B)* the Cholesky factorization using a $B \times B$ block decomposition. For our experiments, we implemented three versions in DOL, for respectively $B = 2, 3, 4$. In all cases, the process networks contain a *Splitter* process, a *Joiner* process and the computational processes for each block $(P_{ij})_{1 \leq j \leq i \leq B}$. Process *Splitter* splits the initial A matrix into blocks and dispatches them to computational processes. Every process P_{ij} implements the computation required on its corresponding matrix blocks A_{ij} and L_{ij}. As an example, the computational processes for *Cholesky(4)* are $P_{11}, P_{21}, P_{22}, P_{31} \ldots P_{44}$ as shown in Figure 10.19. The final L matrix is re-constructed by the *Joiner* process. Explicit communication between P_{ij} processes is used to enforce data dependencies. In these models, a dedicated FIFO is used for every pair of dependent processes to transfer the result block from the source to the target process. In the MPARM implementation, each computational process is deployed into an ARM processor and all the FIFO buffers are allocated to the $L2$ shared memory. It is to be noted that for $B = 2, 3$ the implementation fits into a single cluster, and for $B = 4$, two clusters have been used. The magnitude of the different representations produced along the BIP design flow (number of processes, FIFOs, components, interactions, lines of code) is depicted in Table 10.14. The calibration method used for the System Model was the Platform

Dependent Code Generation.

Table 10.14: *DOL, BIP Models and MPARM Implementation Characteristics*

		$B = 2$	$B = 3$	$B = 4$
DOL Process Network	# processes	5	8	12
	# FIFOs	8	20	40
	# lines of code	864	1400	2171
BIP System Model	# components	40	120	181
	# interactions	182	445	882
	# lines of code	5207	7491	13648
MPARM implementation	# lines of code	1977	3163	4923

For every $B = 2, 3, 4$, we evaluate *Cholesky(B)* on 60×60 input matrices of double precision floating point numbers. Therefore, computational processes operate on matrix blocks of size 30×30, 20×20 and 15×15 for respectively $B = 2, 3, 4$. During the calibration phase, each computational routine on matrix blocks is characterized by the number of cycles required to execute it on an ARM processor. This is done by running the generated application code on MPARM and by accurate measurement of the number of cycles, for each routine. Table 10.15 reports the worst case execution times for different size of matrix blocks.

Table 10.16 presents an overview of the system-level performance analysis results obtained using two methods, respectively simulation of the system model *vs.* implementation and measurement of code execution on the MPARM platform. For both methods, we report the *total execution time* taken by the application to run on the platform and the *analysis time*, that is, the time taken by the methods to produce the results. We point out that simulation of BIP system models produces fairly accurate results (max 20.95% relative error with respect to the cycle-accurate MPARM execution) while significantly reducing the analysis time (up to 19 times, in some situations). Note that for $B = 4$, the MPARM simulation did not terminate in 72 hours and the simulation data is unavailable. However, an estimate is obtained from the BIP system model simulation. A higher cycle count reflects the communication overhead due to the presence of two clusters with the NoC interconnect.

Finally, Figure 10.20 presents a detailed view of execution times and communication delays for computational processes for *Cholesky(4)*. For each process, the idle time denotes the waiting time spent before it gets access to read or write on FIFO channels. The communication time denotes the time effectively spent on reading or writing. The computation time denotes the total execution time without the idle and the communication

Table 10.15: *Execution times for computational routines on matrix blocks (in* 10^6 *cycles)*

	$B = 2$	$B = 3$	$B = 4$
	$K = 30$	$K = 20$	$K = 15$
seq-cholesky	33.82	15.47	14.94
invert	34.85	16.06	15.47
transpose	0.13	0.08	0.08
multiply	115.64	53.23	47.16
tmultiply	104.80	45.01	34.89
subtract	1.66	1.05	1.05

Table 10.16: *Performance Analysis: MPARM Execution* vs *BIP System Model Simulation*

		$B=2$	$B=3$	$B=4$
Total Execution Time (in 10^6 cycles)	MPARM Execution	317.70	229.58	-
	BIP System Model Simulation	325.23	277.69	356.00
	Accuracy	2.37%	20.95%	-
Analysis Time (in minutes)	MPARM Execution	$69'49''$	$34'25''$	-
	BIP System Model Simulation	$3'43''$	$7'54''$	$26'5''$
	Speed-up	18.78	4.35	-

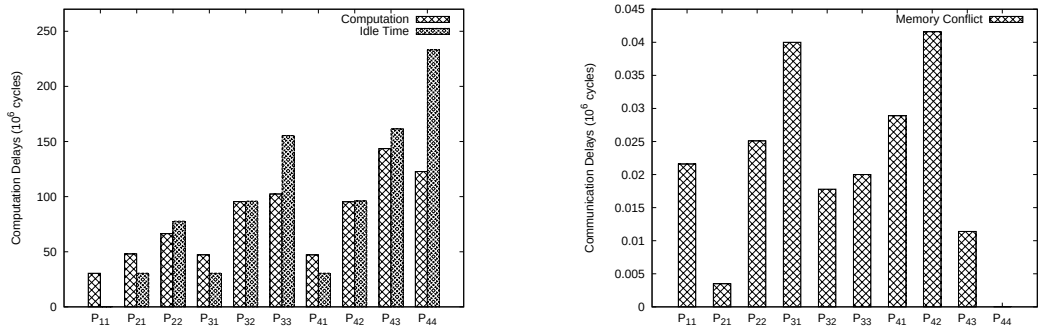


Figure 10.20: *Performance Results of Computational Processes in Cholesky(4)*

time. The Figure 10.20 (left) confirms that processes with higher indexes (i, j) are indeed computationally more intensive than the others. Additionally, the same processes are also idle for longer time than the others. This happens because of an increased number of data dependencies from processes with lower indexes (i, j). Communication time is impacted by memory conflicts. Memory conflicts occur when two different processes try to access simultaneously FIFO buffers located in the same shared memory. Figure 10.20 (right) depicts the delays due to memory conflicts for each process.

10.8 Discussion

In this chapter, we presented the MPARM Hardware Platform, the corresponding hardware model in BIP, and a set of software applications considered as case studies to run on the MPARM Platform. The software applications were the MPEG-2 and the MJPEG decoders, the Fast Fourier Transform (FFT), the Demosaicing Algorithm and the Cholesky Decomposition. For each case study, we created mixed hardware/software System Models of the MPARM platform and the corresponding software application, based on different mappings. Both profiling methods were used to calibrate the System Models, as presented in Chapter 7. For the MPEG2, we used the Weight Table Profiling method. For MJPEG, we used both the Weight Table Profiling and the Platform Dependent Code Generation. For the rest, FFT, Demosaicing and Cholesky we calibrated the System Models using only the Platform Dependent Code Generation , since it is considered to provide more accurate execution times of the software processes. Both profiling methods are time consuming. However, they are applied once for every case study and the results are re-used for all

mapping configurations.

The experiments show the capability of the BIP design flow for fine grain performance analysis on manycore platforms. It also shows the speedup compared to simulation based techniques, without adversely affecting the accuracy of the measurements.

- Chapter 11 -

Case Study on P2012 Hardware Platform

In the previous chapter, we experimented with the MPARM Hardware platform and a series of software applications as case studies.

In this chapter, we present the Platform 2012 Hardware Platform, the corresponding hardware model in BIP, and a software applications considered as case study to run on the Platform 2012 Hardware Platform.

11.1 P2012 Hardware Platform

Platform 2012 (P2012) [SC10] is an area and power efficient manycore computing fabric, jointly developed by STMicroelectronics and CEA. The P2012 computing fabric is highly modular, as it is based on multiple clusters implemented with independent power and clock domains, enabling aggressive fine-grained power, reliability and variability management. Clusters are connected via a high-performance fully-asynchronous network-on-chip (NoC), which provides scalable bandwidth, power efficiency and robust communication across different power and clock domains. Each cluster features up to 16 tightly-coupled processors sharing multi-banked level-1 instruction and data memories, a multi-channel advanced DMA engine, and specialized hardware for synchronization and scheduling acceleration. P2012 achieve extreme area and energy efficiency by aggressive exploitation of domain-specific acceleration at the processor and cluster level. In the scope of this case study, each processor has been specialized with modular extensions dedicated to floating-point unit computation. Other extension such as vector units or other special-purpose instructions may also be chosen at design-time.

P2012 is based on a modular infrastructure as depicted in Figure 11.1. Fabric-level communication is based on an asynchronous NoC organized in a 2D mesh structure. The routers of this NoC are implemented in a Quasi-Delay-Insensitive (QDI) asynchronous (clock-less) logic. They provide a natural Globally Asynchronous Locally Synchronous (GALS) scheme isolating the clusters logically and electrically. The number of clusters is a parameter of the fabric. A configuration up to 32 clusters is supported in the current implementation. The exact type of NoC is used for the MPARM Platform, described in Chapter 10, where a set of important NoC features are given in Figure 10.2.

One significant characteristic of the fabric is that all local storage at the cluster level is visible in a global memory map, which also includes memory-mapped peripherals. In this non-uniform memory architecture (NUMA), remote memories (off-cluster or off-fabric) are expensive to access. For this reason, DMA engines are available for hardware-accelerated

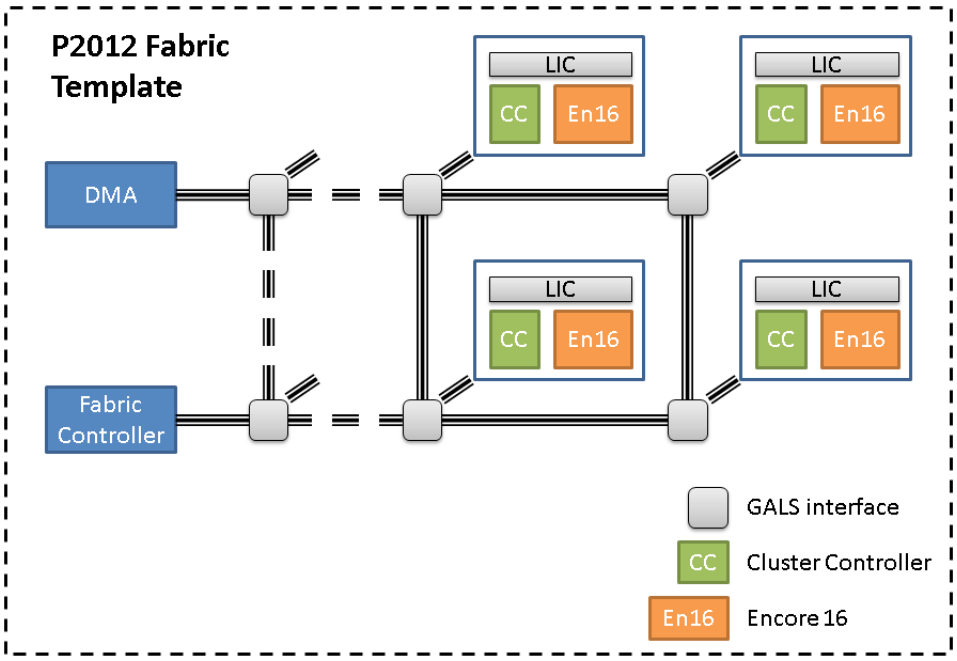


Figure 11.1: *P2012 Fabric Template*

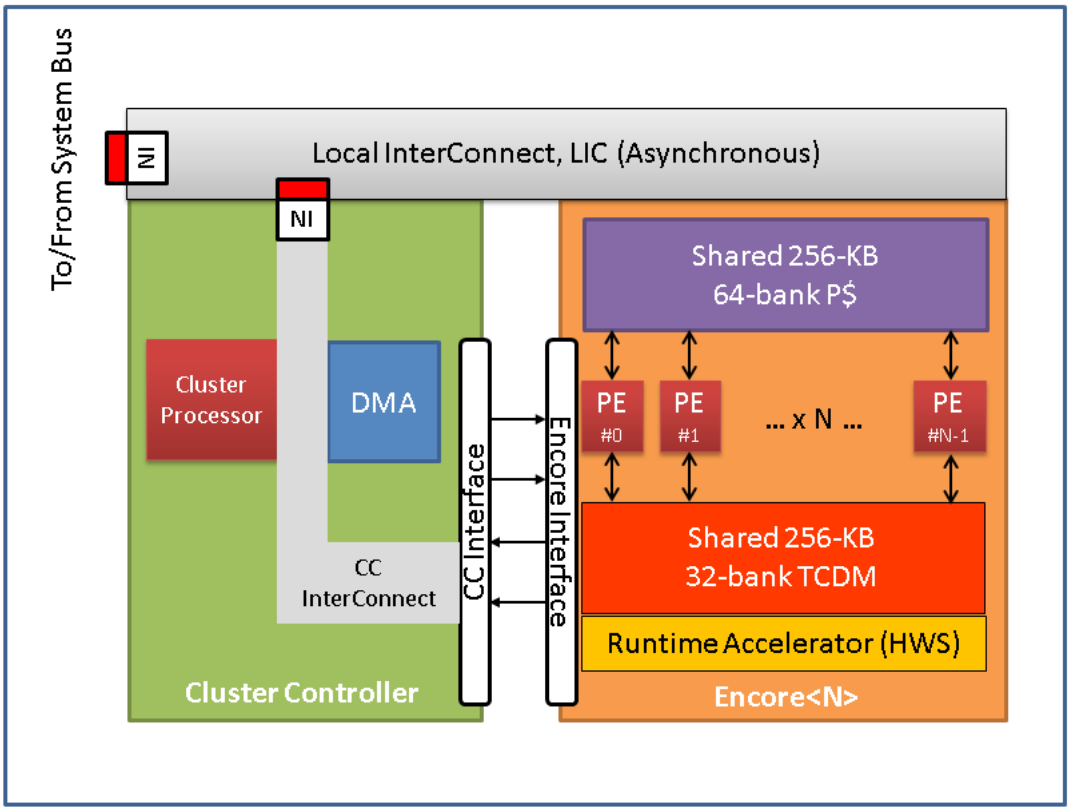


Figure 11.2: *P2012 Cluster*

global memory transfers. At the fabric level, a configurable number of I/O channels, implemented via multiple DMAs, can be used for connecting the fabric to the rest of the SoC. Finally, a fabric controller serves as the control interface between the SoC and the fabric.

A P2012 cluster (Figure 11.2) aggregates a multicore computing engine, called Encore and a cluster controller. The Encore cluster includes a number of processing elements (PEs) varying from 1 to 16. Each PE is built with a highly configurable and extensible processor called STxP70-v4. It is a cost effective and customizable 32-bit RISC core supported by comprehensive state-of-art toolset. The Encore 16 PEs do not have private data caches or memories, therefore avoiding memory coherency overhead. Instead, the PEs can directly access a L1-shared program cache (P$) and a L1-shared Tightly Coupled Data Memory (TCDM). Each core therefore has two 64 bit-ports to the shared memories, a read-only instruction port and a read/write data port. The P$ cache is a 256 KB, 64-bank, direct mapped cache memory while the TCDM is a 256-KB, 32-bank memory. The P$ and the TCDM have been architected with a banking factor of 4 and 2, respectively. The P$ can therefore support a throughput of one instruction fetch per PE on each clock cycle. Encore provides run-time acceleration by the means of the Hardware Synchronizer (HWS). Various synchronization primitives such as semaphores, mutexes, barriers, joins, etc. can be implemented using accelerated support of the HWS.

A logarithmic interconnect is used to access the TCDM memory. Data are forwarded using routing and arbitration techniques. Based on address decoding, the requested address may correspond to the intra-cluster, multi-banked memory or the off-cluster NoC environment. In case of simultaneous accesses in contiguous data, reduced memory conflicts are observed due to fine-grained address interleaving. The crossing latency of the interconnect consists of one clock cycle. In case of multiple conflicting requests, for fair access to memory banks, a roundrobin scheduler arbitrates access and a higher number of cycles is needed depending on the number of conflicting requests, with no latency in between. In case of no banking conflicts data routing is done in parallel for each core, thus enabling a sustainable full bandwidth for processors-memories communication. To reduce memory access time and increase shared memory throughput, read broadcast has been implemented and no extra cycles are needed when broadcast occurs.

A multi-ported, multi-banked, Tightly Coupled Data Memory (TCDM) is directly connected to the logarithmic interconnect. The number of memory ports is equal to the number of banks to have concurrent access to different memory locations. Once a read or write requests is brought to the memory interface, the data is available on the negative edge of the same clock cycle, leading to two clock cycles latency for conflict-free TCDM access. If conflicts occur there is no extra latency between pending requests, once a given bank is active, it responds with no wait cycles.

The cluster controller (CC) consists of a cluster processor subsystem, a DMA subsystem, a CC interconnect and several interfaces: one to the Encore 16 PEs and one to the asynchronous NoC. The cluster processor is designed around a STxP70-V4 dual-issue core without extension and with 16-KB P$ and 16-KB of local memory.

The Platform 2012 Development Kit provides support for several platform programming models (PPM). Standards-based programming models are based on industry standards that can be implemented effectively on P2012. The supported standards are OpenMP and OpenCL. Another supported PPM is called Native Programming Layer (NPL). NPL is an API which is closely coupled with the platform capabilities. It allows the highest level of control on application to resource mapping at the expense of abstraction and platform independence.

The P2012 SDK also features platform models for the execution and the simulation of applications running on the P2012 platform. For the scope of this paper, we used a mono-cluster simulator of the fabric and an Encore engine featuring 16 PEs. We targeted the NPL for fine-tuned control of the deployment of the application on the platform, and to achieve better performance.

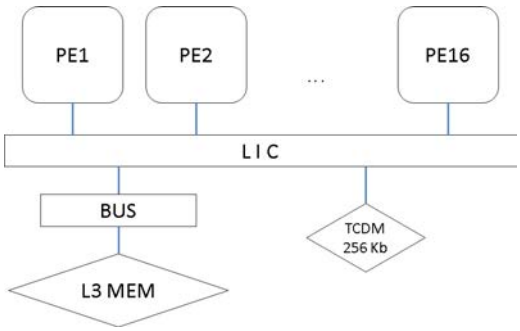


Figure 11.3: *Abstract model of a P2012 Cluster*

For the scope of this thesis, we target a simplified, preliminary version of the P2012 platform. This version consists of a mono-cluster version of the P2012 fabric and an Encore engine featuring 16 PEs. Figure 11.3 presents the abstract model of this platform.

11.2 Platform 2012 Hardware Template Model in BIP

In this section, we provide the P2012 architecture model in BIP, along with all the parameters which characterize the model.

The NoC model parameters are the same as those used for the MPARM NoC model presented in Table 10.4. The P2012 Cluster parameters, given in Table 11.1, specify the Logarithmic Interconnect Access Delay and the TCDM access delay, considering zero conflicts. Notably, the above parameters characterize the communication part of the P2012 Cluster. As for the computation part, the execution times which profile each software process of the given application is obtained via simulation on the P2012 Platform simulator based on the method described in Section 7.1.2.

Table 11.1: *P2012 Cluster Parameters*

	P2012 Cluster Parameters
Logarithmic Interconnect Delay (conflict free)	1 cycle/byte
TCDM Access Delay (conflict free)	1 cycle/byte

The complete abstract model of the P2012 architecture we considered consists of four clusters, each of them enumerating sixteen processing elements, one 32-banked TCDM memory and one Logarithmic Interconnect, as shown in Table 11.2. To concretely model this abstract model in BIP, we need various BIP types and BIP type instances as depicted below in Table 11.3.

In Figure 11.4, we illustrate the complete P2012 Cluster model in BIP. There are eight cores (PEs), given as template components, which will be filled with the *Processor Scheduler* Components, the software processes and the FIFO routines, based on the given mapping. For the modeling of the logarithmic interconnect, there are eight *Bus Interface*

Table 11.2: *P2012 architecture*

Hardware Component	Number
Clusters	4
Processing Elements/Cluster	16
Logarithmic Interconnect/Cluster	1
TCDM/Cluster	1
Memory Banks/TCDM	32
Routers	4

Table 11.3: *BIP P2012 Library*

	BIP types	BIP type instances
Ports	5	122
Connectors	7	1094
Atomic Components	11	69
Compound Components	3	2

Components, each one per core, parametrized by the processor ID and the number of memory banks. They are connected via a set of guarded connectors with all the *Memory Bank* Components of the TCDM memory and the *Network Interface* Components. The *Memory Bank* Components are parametrized by the memory bank ID and the communication access delays, namely, the interconnect access delay and the multi-banked memory access delay. Moreover, the *Network Interface* Components are parametrized by the packaging delay, the number of memory banks and the cluster ID.

11.3 HMAX application on P2012

HMAX is a powerful computational model of object recognition [RP99] which attempts to model the rapid object recognition of human brain. Hierarchical approaches to generic object recognition have become increasingly popular over the years [SWP05, ML08], they indeed have been shown to consistently outperform flat single-template (holistic) object recognition systems on a variety of object recognition task. Recognition typically involves the computation of a set of target features at one step, and their combination in the next step. A combination of target features at one step is called a layer, and can be modeled by a 3D array of units which collectively represent the activity of set of features (F) at a given location in a 2D input grid.

HMAX starts with an image layer of gray scale pixels (a single feature layer) and successively computes higher layers, alternating (S) and (C) layers:

- Simple (S) layers apply local filters that compute higher-order features by combining different types of units in the previous layer.
- Complex (C) layers increase invariance by pooling units of the same type in the previous layer over limited ranges. At the same time, the number of units is reduced by subsampling.

In our case study experiment, we only considered the two first layers of the HMAX model algorithm. In a pre-processing phase, the input raw image is converted to gray

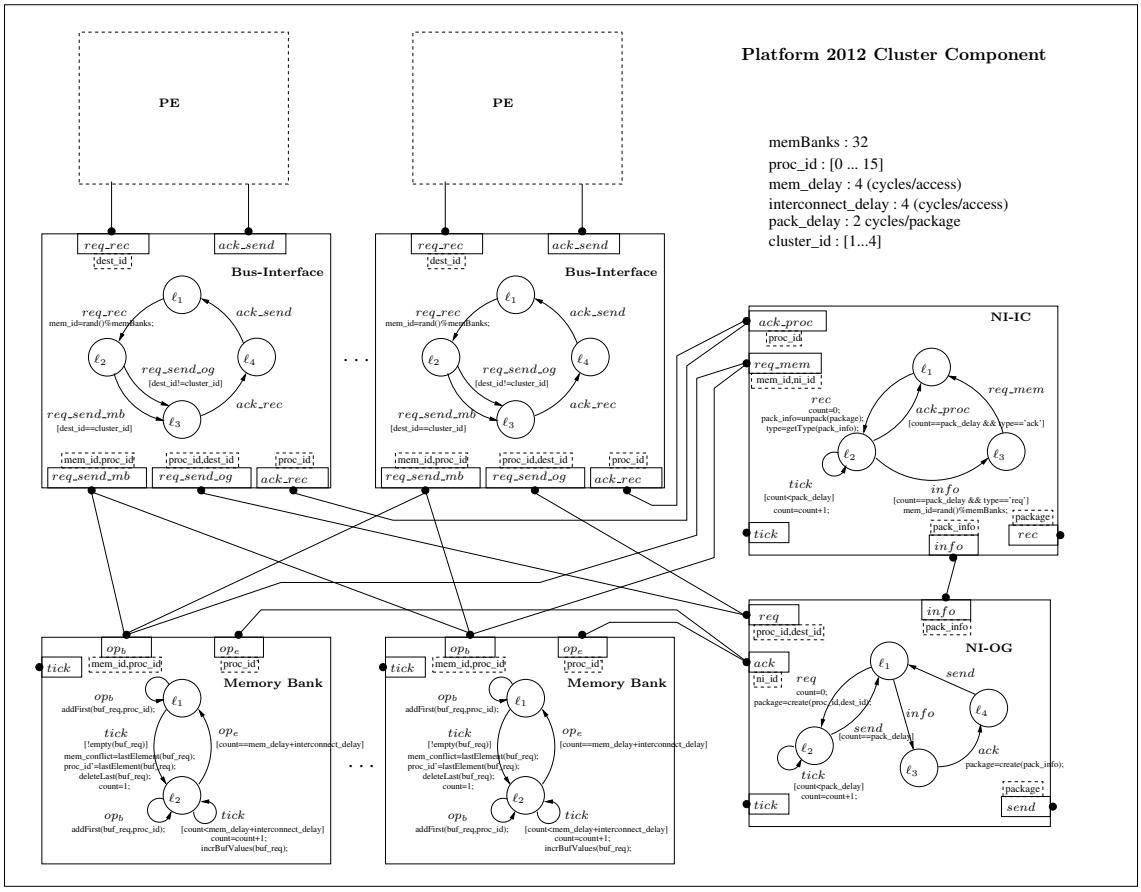


Figure 11.4: *Platform 2012 Cluster Model in BIP*

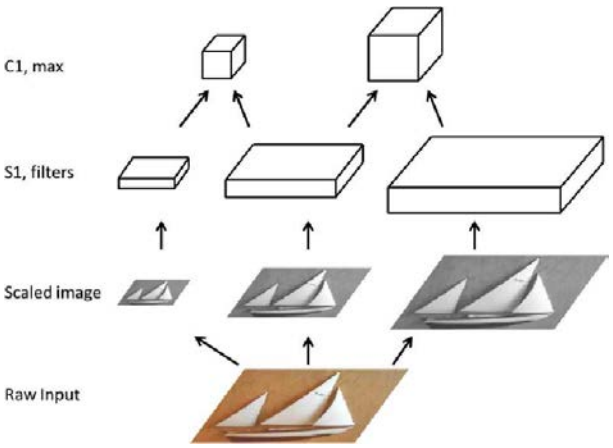


Figure 11.5: *HMAX Computation Layers*

scale input (only one input feature: intensity at pixel level) and the image is then sub-sampled at several resolutions. For the S1 layer, a battery of three filters is applied to the sub-sampled images (three features) and finally for C1 layer we take the spatial max of computed filters across two successive scales. The process is illustrated in Figure 11.5.

In this application model, parallelism can be exploited at several levels. First at the layer level, independent features can be computed simultaneously. Second, at the pixel level, the atomic computation of contribution to a feature may be distributed among computing resources. In the scope of this paper, we will consider parallelism at the layer level.

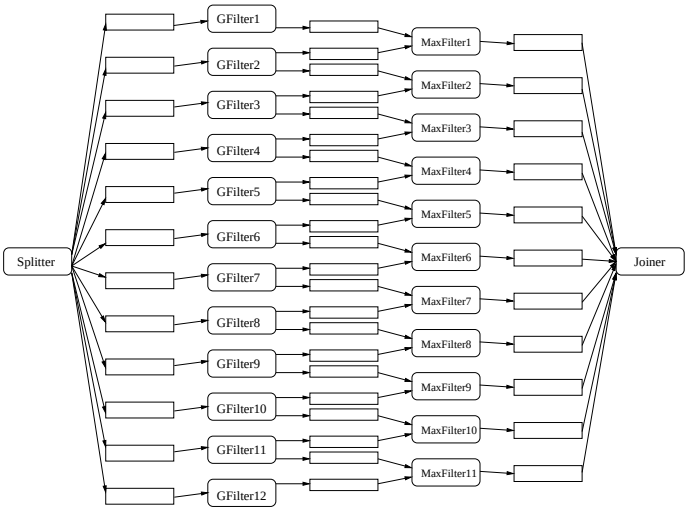


Figure 11.6: *KPN model of the HMAX S1-C1 layers in DOL*

Figure 11.6 presents the process network model constructed from the S1 layer of the HMAX models algorithm. It contains processes Splitter, GFilter1 ... GFilter12, MaxFilter1 ... MaxFilter11 and Joiner. The Splitter builds the 12 scales of the input image and dispatches them to Filters. Each GFilter1 ... GFilter12 implements a 2D-Gabor filter with different orientation. Their results are then sent, feature by feature, to MaxFilters. Each MaxFilter convolves outputs produced by two adjacent GFilters. The results are finally gathered by the Joiner.

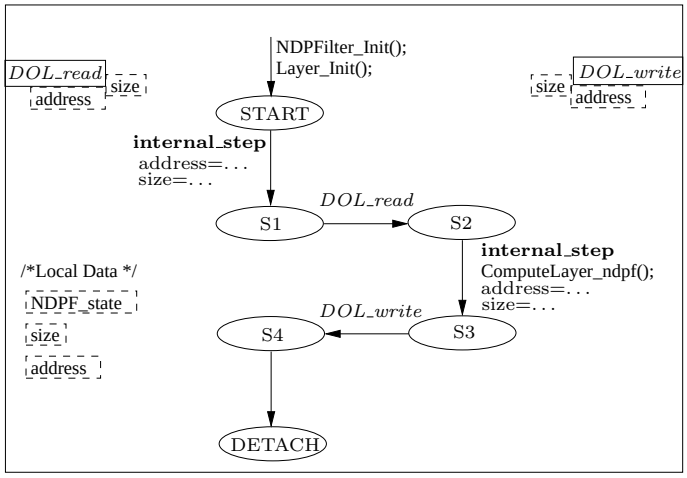


Figure 11.7: *BIP Model of a 2D-Gabor Filter*

Figure 11.7 presents the model of a 2D-Gabor filter as an atomic component in BIP. This component consists of 6 control locations (START, S1, ..., S4, DETACH) and two ports, DOL_read and DOL_write. NDPF_state, size and address are local data (variables) of the component. The variables address and size are associated with the ports. The transitions are either internal transitions (internal_step) where local computation and updates are made, or port interactions, where the component exchanges data and synchronizes with the other BIP components.

We restrict ourselves to the S1 layer of the HMAX models algorithm. The process network in DOL consists of 14 processes and 24 FIFO channels. This DOL model is about 700 lines of XML (defining the process network structure) and 1500 lines of C (defining the process behavior). The software model in BIP is constructed automatically from the DOL model. It consists of 38 atomic components interconnected using 48 connectors. The BIP software model is about 2000 lines of BIP code. The system model obtained by deploying the S1 layer on a single P2012 cluster consists of 125 atomic components interconnected using about 1500 connectors. The total BIP description totalizes about 13000 lines of BIP code. This description is compiled into about 50000 lines of C++ code, used for simulation and performance analysis.

The execution time of 2D-Gabor filters on P2012 PEs ranges from $220 \cdot 10^6$ to $0.68 \cdot 10^6$ cycles, depending on the size of the input image (ranging from 100×100 to 15×15 pixels). By using these values in the system model, the total execution time of the S1 layer is estimated as $225 \cdot 10^6$ cycles. This overall execution time is negatively impacted by the long access time (i.e., about 100 cycles) to the L3 memory (where all FIFOs are mapped) as well as by the bus contention. A slightly better result is obtained if the FIFOs are all mapped into the TCDM memory. In this case, the memory access time is about 1 cycle and there is no more contention. The total execution time reduces to about $220 \cdot 10^6$ cycles. However, such a mapping is not feasible due to memory size constraints, that is, FIFOs cannot fit all simultaneously within the TCDM memory.

11.4 Conclusion

In this chapter, we presented the P2012 Hardware Platform, the corresponding hardware model in BIP, and the HMAX software application to run on the MPARM Platform. We created mixed hardware/software System Model of the P2012 platform and the HMAX software application, based on different mappings. The Platform Dependent Code Generation method was used to calibrate the System Models, as presented in Chapter 7.

Part

Conclusion

- Chapter 12 -

Conclusion and Perspectives

Introductive We developed a rigorous and automated design flow for generating a model which faithfully represents the behavior of a mixed hardware/software system from a model of its application software and a model of its underlying hardware architecture. The presented method allows generation of a correct-by-construction system model for manycore platforms from an application software and a mapping. The method is based on source-to-source correct-by-construction transformation of BIP models. It is completely automated and supported by the BIP toolset.

Method Firstly, we generate the *application software model in BIP*. This is achieved by an automatic translation of the input application software model which should be described in a process network with a well defined structure. The translation preserves intact the behavior and the characteristics of the initial application software.

Secondly, we model the *hardware architecture in BIP*. A library of BIP atomic components that characterize multi-processor architectures is defined. Combining the hardware architecture specifications and the BIP library components, we synthesize the hardware architecture model in BIP.

Thirdly, we construct the mixed software/hardware *system model*. This model represents the behavior of the application software running on the hardware architecture according to the mapping, but without taking into account execution times for the software actions. This step consists in progressively enriching the application software model by doing:

- Application of a sequence of source-to-source transformations to synthesize *hardware dependent software routines* implementing communication by using the hardware components.

- Integration of hardware components used in the system model.

The transformations are proved correct-by-construction, that is, they preserve functional properties of the application software.

Finally, the (bounds for) execution times are obtained by analysis or simulation of the run of every software process in isolation on the platform. These bounds are injected into the *system model* and lead to the *calibrated system model*.

Method Advantages and Comments The above method sticks to the general principles of *rigorous design* as described in Chapter 1. Namely, it is *model-based*, *component-based*, *correct-by-construction* and *tool-supported*.

The construction of the system model is incremental and structure-preserving. This ensures scalability as the complexity of system models increases polynomially with the size of the application software and of the target hardware architecture. Mastering system model complexity is achieved thanks to the expressiveness of the BIP modeling framework. All properties established for the initial model will hold for all the models obtained by transformations. The transformations are correct-by-construction, that is, they are proven to preserve a trace equivalence between the initial and the transformed model. The complexity of the transformations is linear with the size of the transformed models. So, correctness is ensured at minimal cost and by construction, thus overcoming obstacles of design flows involving different and not semantically related languages and models.

The method clearly separates software and hardware design issues. It is also parametrized and allows flexible integration of design choices related to resource management such as scheduling policies, memory size and execution times, etc. This allows estimation of the impact of each parameter on system behavior.

We have defined a library of BIP atomic components that characterize manycore architectures, including models for hardware components (e.g., processor, memory) and for hardware-dependent software components (e.g., FIFO channel read/write, bus controllers, schedulers).

Using BIP as a unifying modeling formalism for both hardware and software confers multiple advantages, in particular rigorousness. The obtained system models are correct-by-construction. This is a main difference from other ad hoc model construction techniques. The use of a single modeling framework allows to maintain the overall coherency of the design flow by comparing different architectural solutions and their properties. This is a significant advantage of our approach. Semantically related models are used for verification, simulation and performance evaluation. Designers use many different languages e.g. programming languages, UML, SystemC, SES/Workbench. Code generation and deployment is often independent from validation and evaluation.

Toolset The method has been implemented and integrated in the BIP toolset. We used the DOL framework [TBHH07] as a frontend to describe the application software, hardware architectures and mapping specifications. The backend of the tool produces the system model in BIP, which can be analyzed by the BIP tool chain for:

- Code generation for simulation/validation on a Linux PC.
- Code generation for simulation/validation on two Virtual Platforms, the MPARM and the P2012.
- Functional correctness using the D-Finder tool, checking for deadlocks.
- Performance analysis (e.g. delay computation) and design space exploration, based on simulation and statistical model checking.

Experimentation and Case Studies Experimental results show the feasibility of the system model for fine granular analysis of the effects of architecture and mapping constraints on the system behavior. The final model allows accurate estimation through simulation of real-time characteristics (response times, delays, latencies, throughput, etc.) and

indicators regarding resource usage (bus conflicts, memory conflicts, etc.). The method is tractable and allows design space exploration to determine optimal solutions.

We have experimented on two case studies, the MPARM and the P2012/STHORM Hardware Platform. We presented the MPARM hardware model in BIP, and a set of software applications to run on the MPARM Platform. The software applications were the MPEG-2 and the MJPEG decoders, the Fast Fourier Transform (FFT), the Demosaicing Algorithm and the Cholesky Decomposition. For each software application, we created mixed hardware/software System Models of the MPARM platform and the corresponding software application, based on different mappings. Both profiling methods were used to calibrate the System Models, as presented in Chapter 7.

The experiments show the capability of the BIP design flow for fine grain performance analysis on manycore platforms. It also shows the speedup compared to simulation based techniques, without adversely affecting the accuracy of the measurements.

Perspectives

Future work includes more experiments based on the P2012 hardware platform. In addition, formal verification using D-Finder could be applied at the level of software application model in BIP described as KPN process networks.

An other extension should be to explore different programming models for the application software and richer hardware architecture models that include DMA (Direct Memory Access) Controller, Bus Bridge, Network on Chip communication and three dimensional architectures.

Except from timing constraints, thermal values, power consumption and dynamic scheduling policies should be considered. The latter properties of the hardware model should enrich the current models of hardware components attributing to an extensive performance analysis of our target system.

Moreover, we plan to include statistical model checking on the generated system models consisting of multiple applications running on complex multicore architectures for performance analysis, as in [BBB$^+$10].

Finally, important part of the future extension of this work would be to consider task migration and dynamic mapping. Task migration protocol should be aware of all critical performance metrics of the system leading to optimal mapping obtained on-the-fly.

List of Figures

List of Tables

Bibliography

[AAM06] Yasmina Abdeddaim, Eugene Asarin, and Oded Maler. Scheduling with Timed Automata. *Theoretical Computer Science*, 354:272–300, 2006.

[ACF+98] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *INTERNATIONAL JOURNAL OF ROBOTICS RESEARCH*, 17:315–337, 1998.

[Aea07] Davare Abhijit et al. A next-generation design framework for platform-based design. In *DVCon 2007*, February 2007.

[BB91] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, pages 1270–1282, 1991.

[BBB+05] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *Journal of VLSI Signal Processing Systems*, 41:169–182, 2005.

[BBB+08] Ananda Basu, Philippe Bidinger, Marius Bozga, Joseph Sifakis, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *FORTE*, pages 116–133, 2008.

[BBB+10] Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Caillaud, Benoît Delahaye, and Axel Legay. Statistical abstraction and model-checking of large heterogeneous systems. In *Proceedings of FMOODS/FORTE'10*, volume 6117 of *LNCS*, pages 32–46. Springer, 2010.

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

[BBB+12] Ananda Basu, Saddek Bensalem, Marius Bozga, Benot Delahaye, Axel Legay, and Axel Legay. Statistical abstraction and model-checking of large heterogeneous systems. pages 53–72, 2012.

[BBGLG85] Albert Benveniste, Patricia Bournai, Thierry Gautier, and Paul Le Guernic. SIGNAL : a data flow oriented language for signal processing. Rapport de recherche RR-0378, INRIA, 1985.

[BBJ+10] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. In Luca P. Carloni and Stavros Tripakis, editors, *EMSOFT*, pages 209–218. ACM, 2010.

[BBL+10] Saddek Bensalem, Marius Bozga, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. Incremental Component-based Construction and Verification using Invariants. In *Proceedings of FMCAD'10*, pages 257–266. IEEE, 2010.

[BBN+09] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, Joseph Sifakis, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *CAV*, pages 614–619, 2009.

[BBNS08] S. Bensalem, M. Bozga, T-H. Nguyen., and J. Sifakis. Compositional Verification for Component-based Systems and Application. In *Proceedings of ATVA'08*, volume 5311 of *LNCS*, pages 64–79. Springer, 2008.

[BBNS09] S. Bensalem, M. Bozga, T-H. Nguyen, and J. Sifakis. D-Finder: A Tool for Compositional Deadlock Detection and Verification. In *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 614–619. Springer, 2009.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[BCG+97] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara, editors. *Hardware-software co-design of embedded systems: the POLIS approach.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

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

[BGL+08] Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Félix Ingrand, and Joseph Sifakis. Incremental component-based construction and verification of a robotic system. In *Proceedings of the 2008 conference on ECAI 2008: 18th European Conference on Artificial Intelligence*, pages 631–635, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[BGL+11] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, Rongjie Yan, and Rongjie Yan. D-finder 2: Towards efficient correctness of incremental design. In *NASA Formal Methods*, pages 453–458, 2011.

[bip] Bip tools. `http://www-verimag.imag.fr/BIP-Tools,93.html/`.

[BJS09] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-source architecture transformation for performance optimization in bip. In *SIES*, pages 152–160. IEEE, 2009.

[BLP+02] Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto L. Sangiovanni-Vincentelli, Marco Sgroi, and Yosinori Watanabe. Modeling and designing heterogeneous systems. In Jordi Cortadella, Alexandre Yakovlev, and Grzegorz Rozenberg, editors, *Concurrency and Hardware Design*, volume 2549 of *Lecture Notes in Computer Science*, pages 228–273. Springer, 2002.

[BMP+07] Ananda Basu, Laurent Mounier, Marc Poulhiès, Jacques Pulou, and Joseph Sifakis. Using bip for modeling and verification of networked systems – a case study on tinyos-based networks. In *NCA*, pages 257–260, 2007.

[BS08a] Simon Bliudze and Joseph Sifakis. The algebra of connectors - structuring interaction in bip. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.

[BS08b] Simon Bliudze and Joseph Sifakis. A notion of glue expressiveness for component-based systems. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 508–522. Springer, 2008.

[BSS09] Marius Bozga, Vassiliki Sfyrla, and Joseph Sifakis. Modeling synchronous systems in bip. In *EMSOFT*, pages 77–86, 2009.

[BW01] A. Burns and A. Welling. *Real-Time Systems and Programming Languages*. Addison-Wesley, 2001. 3rd edition.

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto L. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003.

[CHEP71] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *J. Comput. Syst. Sci.*, 5(5):511–523, October 1971.

[CMLS11] Scott Cotton, Oded Maler, Julien Legriel, and Selma Saidi. Multi-criteria optimization for mapping programs to multi-processors. In *SIES*, pages 9–17. IEEE, 2011.

[CoF] Cofluent. `http://www.cofluentdesign.com`.

[CRBS08] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. Translating aadl into bip - application to the verification of real-time systems. In *MoDELS Workshops*, pages 5–19, 2008.

[dal] Laboratoire d'analyses et d'architecture des systèmes. `http://www.laas.fr`.

[EJL+03] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity: The Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[EPTP07] Cagkan Erbas, Andy D. Pimentel, Mark Thompson, and Simon Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP Journal on Embedded Systems*, 2007, 2007.

[ET05] Stephen A. Edwards and Olivier Tardieu. Shim: a deterministic model for heterogeneous embedded systems. In *EMSOFT*, pages 264–272, 2005.

[FHC97] Sara Fleury, Matthieu Herrb, and Raja Chatila. Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture. In *In International Conference on Intelligent Robots and Systems*, pages 842–848, 1997.

[Gro02] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[Hea05] Rafik Henia et al. System-level performance analysis - the SymTA/S approach. In *IEEE Proceedings Computers and Digital Techniques*, volume 152, pages 148–166, 2005.

[HS06] T. Henzinger and J. Sifakis. The Embedded Systems Design Challenge. In *Formal Methods FM'06 Proceedings*, volume 4085 of *LNCS*, pages 1–15. Springer, 2006.

[HSKM08] Christian Haubelt, Thomas Schlichter, Joachim Keinert, and Mike Meredith. Systemcodesigner: automatic design space exploration and rapid prototyping from behavioral models. In *DAC*, pages 580–585, 2008.

[Kah74] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[KDVvdW97] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proceedings of ASAP'97*, pages 338–349. IEEE Computer Society, 1997.

[KPBT06] Simon Künzli, Francesco Poletti, Luca Benini, and Lothar Thiele. Combining simulation and formal methods for system-level performance analysis. In *DATE*, pages 236–241, 2006.

[Lee09] Edward A. Lee. Finite state machines and modal models in ptolemy ii. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow: Describing signal processing algorithm for parallel computation. In *COMPCON*, pages 310–315, 1987.

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow process networks. pages 773–801, 1995.

[LSvdWD01] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System level design with SPADE: an M-JPEG case study. *ICCAD*, pages 31–38, 2001.

[Mat] http://www.mathworks.com/products/simulink/index.html. Accessed: 19/08/2012.

[MGN03] Imed Moussa, Thierry Grellier, and Giang Nguyen. Exploring SW Performance Using SoC Transaction-Level Modeling. In *Proceedings of DATE'03*, pages 20120–20125, 2003.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[ML08] Jim Mutch and David G. Lowe. Object class recognition and localization using sparse features with limited receptive fields. *International Journal of Computer Vision*, 80(1):45–57, 2008.

[MMMcMc] Matthieu Moy, Florence Maraninchi, Laurent Maillet-contoz, and Laurent Maillet-contoz. Lussy: an open tool for the analysis of systems-on-a-chip at the transaction level. design automation for embedded systems.

[MPS] http://www-micrel.deis.unibo.it/sitonew/research/mparm.html.

[NSD06] Hristo Nikolov, Todor Stefanov, and Ed F. Deprettere. Multi-processor system design with espam. In Reinaldo A. Bergamaschi and Kiyoung Choi, editors, *CODES+ISSS*, pages 211–216. ACM, 2006.

[NSD08] Hristo Nikolov, Todor Stefanov, and Ed F. Deprettere. Automated integration of dedicated hardwired ip cores in heterogeneous mpsocs designed with espam. *EURASIP J. Emb. Sys.*, 2008, 2008.

[NTS+08] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *Proceedings of DAC'08*, pages 574–579. ACM, 2008.

[OS85] Dianne P. O'Leary and G. W. Stewart. Data-flow algorithms for parallel matrix computation. *Commun. ACM*, 28(8):840–853, August 1985.

[PHL+01] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring embedded-systems architectures with artemis. *IEEE Computer*, 34(11):57–63, 2001.

[Pim08] Andy D. Pimentel. The artemis workbench for system-level performance evaluation of embedded systems. *IJES*, 3(3):181–196, 2008.

[RE02] Kai Richter and Rolf Ernst. Event model interfaces for heterogeneous system analysis. In *DATE*, pages 506–513. IEEE Computer Society, 2002.

[RP99] Maximilian Riesenhuber and Tomaso Poggio. Hierarchical models of object recognition in cortex. 1999.

[RZJE02] Kai Richter, Dirk Ziegenbein, Marek Jersak, and Rolf Ernst. Model composition for scheduling analysis in platform design. In *DAC*, pages 287–292. ACM, 2002.

[SAE09] SAE. Architecture analysis and design language (aadl) (standard sae as5506), 2009.

[SBM09] Ramzi Ben Salah, Marius Bozga, and Oded Maler. Compositional Timing Analysis. In *Proceedings of EMSOFT'09*, pages 39–48, 2009.

[SC10] STMicroeletronics and CEA. Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology, 2010.

[Sta] http://www.mathworks.com/products/stateflow/. Accessed: 13/12/2012.

[STS+10] 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.

[SWP05] Thomas Serre, Lior Wolf, and Tomaso Poggio. Object recognition with features inspired by visual cortex. In *CVPR (2)*, pages 994–1000, 2005.

[TBHH07] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, ACSD '07, pages 29–40, Washington, DC, USA, 2007. IEEE Computer Society.

[TCN02] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *ISCAS*, volume 4, pages 101–104. IEEE, March 2002.

[Tea10] Basten Twan et al. Model-driven design-space exploration for embedded systems: The octopus toolset. In *ISoLA (1)*, pages 90–105, 2010.

[Tin] www.tinyos.net/.