

Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP

Mohamad Jaber

▶ **To cite this version:**

Mohamad Jaber. Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP. Computer Science [cs]. Université Joseph-Fourier - Grenoble I, 2010. English. NNT : . tel-00531082v1

HAL Id: tel-00531082

https://theses.hal.science/tel-00531082v1

Submitted on 1 Nov 2010 (v1), last revised 18 Jan 2011 (v2)

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.

GRENOBLE UNIVERSITÉS

THESE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE
Spécialité : Informatique

Arrêté ministériel : 7 août 2006

Présentée et soutenue publiquement par

Mohamad JABER

le 28 Octobre 2010

Implémentations Centralisée et Répartie de Systèmes Corrects par construction à base des Composants par Transformations Source-à-source dans BIP

Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP

Directeur de Thèse	Joseph SIFAKIS
Co-Directeur de Thèse	Jean-Claude FERNANDEZ

JURY

M. Jean-Paul BODEVEIX	Professeur à l'Université Paul Sabatier	Examinateur
M. Jean-Claude FERNANDEZ	Professeur à l'Université de Grenoble	Co-Directeur de Thèse
M. Michel RAYNAL	Professeur à l'Université de Rennes I	Rapporteur
M. Joseph SIFAKIS	Directeur de Recherche au CNRS	Directeur de Thèse
M. Jean-Bernard STEFANI	Directeur de Recherche à l'INRIA	Président
M. Janos SZTIPANOVITS	Professeur à l'Université de Vanderbilt	Rapporteur

Thèse préparée au sein du Laboratoire VERIMAG dans l'Ecole Doctorale EDMSTII

Acknowledgements

I wish to express my thanks to Professor Jean-Paul Bodeveix, Professor Michel Raynal, Professor Jean-Bernard Stefani, Professor Janos Sztipanovits for accepting to be members of my jury and for their valuable remarks and comments on my thesis.

My greatest thank to my advisors, Professor Joseph Sifakis and Professor Jean-Claude Fernandez, for offering me technical and moral support during my Ph.D. program. They were always there, always patient to listen, to answer and to give advice. I have learnt a lot from their knowledge, their creative minds and their excellent guidance. They taught me not only the knowledge but also the different ways to think, to approach, to analyse and to solve a new problem. Thanks for being great advisors.

Special thanks to Doctor Marius Bozga, Doctor Borzoo Bonakdarpour and Jean Quilbeuf for helping me a lot on both theory and implementation. Their intelligence, their enthusiasm always made my difficult problems simple and interesting. It has been a great pleasure to work with them.

I'm pleased to have NGUYEN Thanh Hung as my colleague and as my friend. Although we did not work on the same topic but the discussions with him were always helpful and enjoyable. (Sint Chaw, khwe khon, khon khwe).
Thanks people in Verimag, especially in BIP group Andreas, Marc, Jacques, Simon, Sophie, Tesnim, Vaso, Matthieu, Yassin for useful exchanges, and all other colleagues for useful seminars. Thanks Imen, Artur, Olivier, Paraskevas, Emmanuel, Rajarshi for many memorable get-togethers.
I would like to thanks all the members of Verimag who created such a nice atmosphere in the lab. Special thank also for my friend and my colleague Ylies Falcone.

It has been a great pleasure to know my Lebanese friends in Grenoble. Thanks to them, especially Hassan Abbass, Jawad, Ftouni, Louay, Hussein Joumaa, Hassan Bazzi, Ozeir, Mohamad Ghassani, Ali Tahini, Doctor Jalal, Mo2nes, Ghadban, Ibrahim Safeydine, Bachir Aoun, Mohamad Jahjah, Mohamad/Moussa Barakat, ...

Special thanks to my wife to my family, my parents, my sisters, my brother, my aunts, who are always beside me, support me, help me in every possible way. Without them, nothing would have been possible.

To my parents & Roukaya ...

Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP

Abstract

The thesis studies theory and methods for generating automatically centralized and distributed implementations from a high-level model of an application software in BIP. BIP (Behavior, Interaction, Priority) is a component framework with formal operational semantics. Coordination between components is achieved by using multiparty interactions and dynamic priorities for scheduling interactions. A key idea is to use a set of correct source-to-source transformations preserving the functional properties of a given application software. By application of these transformations we can generate a full range of implementations from centralized to fully distributed.

Centralized Implementation: the implementation method transforms the interactions of an application software described in BIP and generates a functionally equivalent program. The method is based on the successive application of three types of source-to-source transformations: flattening of components, flattening of connectors and composition of atomic components. We shown that the system of the transformations is confluent and terminates. By exhaustive application of the transformations, any BIP component can be transformed into an equivalent monolithic component. From this component, efficient standalone C++ code can be generated.

Distributed Implementation: the implementation method transforms an application software described in BIP for a given partition of its interactions, into a Send/Receive BIP model. Send/Receive BIP models consist of components coordinated by using asynchronous message passing (Send/Receive primitives). The method leads to 3-layer architectures. The bottom layer includes the components of the application software where atomic strong synchronization is implemented by sequences of Send/Receive primitives. The second layer includes a set of interaction protocols. Each protocol handles the interactions of a class of the given partition. The third layer implements a conflict resolution protocol used to resolve conflicts between conflicting interactions of the second layer. Depending on the given partition, the execution of obtained Send/Receive BIP model range from centralized (all interactions in the same class) to fully distributed (each class has a single interaction). From Send/Receive BIP models and a given mapping of their components on a platform providing Send/Receive primitives, an implementation is automatically generated. For each class of the partition we generate C++ code implementing the global behavior of its components.

The transformations have been fully implemented and integrated into BIP tool-set. The experimental results on non trivial examples and case studies show the novelty and the efficiency of our approach.

Key words: Component-based modeling, source-to-source transformation, correct-by-construction, distributed systems, optimization for performance.

Associated Papers

Several Chapters in this thesis appeared in several papers in form of articles or of Verimag technical reports.

The symbolic implementation of BIP engine in Chapter 2 appeared in the paper [JBB09] in ICE 2009 (Structured Interactions CONCUR 2009 affiliated workshop), 31st August - Bologna, Italy. A journal version is under review in Mathematical Structures in Computer Science.

The architecture transformation for performance optimization method in Chapter 3 appeared in the paper [BJS09] in SIES 2009 (IEEE Symposium on Industrial Embedded Systems, Ecole Polytechnique Fédérale de Lausanne, Switzerland, July 8 - 10, 2009). A journal version is under review in IEEE Transactions on Industrial Informatics.

The BIP into Distributed BIP methods together with the results in Chapter 4 appeared in two papers: paper [BBJ+10a] has been accepted (best paper award) for SIES 2010 (IEEE Symposium on Industrial Embedded Systems, University of Trento, Italy, July 7 - 9, 2010), and paper [BBJ+10b] has been accepted for EMSOFT 2010 (International Conference on Embedded Software, Scottsdale, Arizona, USA , October 24 - 29, 2010).

Contents

1 Introduction **21**
- 1.1 Challenges in Building Correct Component-based Systems . . . 21
- 1.2 State-of-the-Art of Component-based Systems Design and Implementation . 24
  - 1.2.1 Existing component-based frameworks . . . 24
  - 1.2.2 Discussion . . . 26
- 1.3 Our Contribution . . . 27
- 1.4 Organization of the Thesis . . . 28

2 The BIP Component-based Framework **31**
- 2.1 General Overview . . . 32
- 2.2 The BIP Language . . . 33
  - 2.2.1 Ports and Interfaces . . . 34
  - 2.2.2 Atomic Component . . . 35
  - 2.2.3 Connectors and Interactions . . . 37
  - 2.2.4 Priorities . . . 41
  - 2.2.5 Composite Components . . . 42
- 2.3 Execution Platform . . . 44
  - 2.3.1 Enumerative Engine . . . 45
  - 2.3.2 Symbolic Engine . . . 45
- 2.4 The BIP Tool-Chain . . . 45
- 2.5 Summary . . . 47

3 Transformation for Generating Centralized Implementations **49**
- 3.1 Problem Statement . . . 49
- 3.2 Transformations . . . 50

3.2.1 Components Flattening . . . 51
3.2.2 Connectors Flattening . . . 52
3.2.3 Components Composition . . . 56
3.3 Efficient Sequential Implementation . . . 58
3.4 Experimental Validation . . . 58
3.4.1 MPEG Video Encoder . . . 58
3.4.2 Concurrent Sorting . . . 60
3.5 Summary . . . 65

4 Transformation for Generating Distributed Implementations 67
4.1 Problem Statement . . . 68
4.2 Original BIP Model . . . 69
4.3 Conflicting Interactions . . . 69
4.4 Proposed Solution . . . 73
4.5 The 3-Layer Architecture . . . 75
4.6 Transformations . . . 76
4.6.1 Transformation of Atomic Components . . . 76
4.6.2 The Interaction Protocol . . . 77
4.6.3 The Conflict Resolution Protocol . . . 81
4.6.3.1 Centralized Implementation . . . 82
4.6.3.2 Token Ring Implementation . . . 83
4.6.3.3 Implementation Based on Dining Philosophers . . . 83
4.6.4 Cross-Layer Interactions . . . 83
4.7 Correctness . . . 86
4.7.1 Compliance with Send/Receive Models . . . 86
4.7.2 Observational Equivalence between Original and Transformed BIP Models . . . 87
4.7.3 Interoperability of Conflict Resolution Protocol . . . 90
4.8 Transformation from Send/Receive BIP into C++ . . . 91
4.9 Component Composition . . . 93
4.10 Experimental Validation . . . 93
4.10.1 Diffusing Computation . . . 94
4.10.2 Utopar Transportation System . . . 99
4.10.3 Bitonic Sorting . . . 102
4.11 Summary . . . 105

5 Tool-Chain 107
5.1 BIP into Centralized Implementations Tool-Chain - BIP2BIP . . . 107
5.2 BIP into Distributed Implementations Tool-Chain - BIP2Dist . . . 109
5.3 Summary . . . 109

6 Conclusions and Perspectives **113**
6.1 Conclusions 113
6.2 Perspectives 115

List of Figures

1.1 High-level component-based design into implementation. 23

2.1 Components composition. 32
2.2 Incrementality of composition. 32
2.3 Compositionality of composition. 33
2.4 Composability of composition. 33
2.5 Layered component model. 34
2.6 Three-dimentional space construction. 34
2.7 An example of an atomic component in BIP. 36
2.8 Connectors and their interactions. 37
2.9 Graphic representation of connectors 38
2.10 An example of a connector containing one interaction in BIP. 39
2.11 γ_i dominates γ_j. 41
2.12 An example of compound component in BIP. 43
2.13 Component composition. 44
2.14 The BIP tool-chain. 46

3.1 Example. 51
3.2 Component flattening. 52
3.3 Component flattening for example in Figure 3.1. 53
3.4 Connector glueing. 53
3.5 Connector glueing for example in Figure 3.3. 55
3.6 Connector flattening for example in Figure 3.3. 55
3.7 Component composition. 57
3.8 Component composition for example in Figure 3.6. 57

3.9 MPEG4 encoder structure. . . . 58
3.10 Encode component structure. . . . 59
3.11 Execution time for the MPEG4 encoder. . . . 61
3.12 Concurrent sorting $n = 2$. . . . 62
3.13 Concurrent sorting $n = 4$. . . . 62
3.14 Execution time for concurrent sorting. . . . 64

4.1 A simple high-level BIP model. . . . 68
4.2 Centralized Engine. . . . 70
4.3 Violating of BIP semantics when badly distributed interactions. . . . 71
4.4 Conflict interactions. . . . 71
4.5 Drawbacks of conflict-free solution. . . . 72
4.6 Resolving conflict interactions using alpha-core protocol. . . . 73
4.7 General overview of the 3-layer architecture. . . . 74
4.8 3-layer Send/Receive BIP model of Figure 4.1. . . . 75
4.9 Transformation of atomic component. . . . 77
4.10 Component IP_1 in Figure 4.8. . . . 79
4.11 A centralized Conflict Resolution Protocol for Figure 4.8. . . . 82
4.12 Token-based Conflict Resolution Protocol for the BIP models in Figures 4.1 and 4.8. . . . 84
4.13 Dining philosophers-based Conflict Resolution Protocol for the BIP models in Figures 4.1 and 4.8. . . . 85
4.14 Proof of observational equivalence. . . . 89
4.15 The impact of merging components. . . . 93
4.16 Partial BIP model for diffusing computations. . . . 95
4.17 Different scenarios for diffusing computations. . . . 95
4.18 Performance of termination detection in diffusing computation in different scenarios (Torus 4×6). . . . 97
4.19 Performance of termination detection in diffusing computation in different scenarios (Torus 20×20). . . . 98
4.20 Utopar transportation system. . . . 99
4.21 High-level BIP model for Utopar system. . . . 100
4.22 Performance of responding 10 requests per calling unit ($25 = 5 \times 5$ calling units, and 4 cars). . . . 101
4.23 Performance of responding 10 request per calling unit ($49 = 7 \times 7$ calling units, and 4 cars). . . . 102
4.24 High-level BIP model for Bitonic Sorting Algorithm. . . . 103
4.25 3-layer Send/Receive BIP model for Bitonic Sorting Algorithm. . . . 104
4.26 Component composition components in Bitonic Sorting 3-layer Send/Receive BIP model. . . . 105

5.1 BIP2BIP toolset: General architecture 108
5.2 BIP2Dist toolset: General architecture 110
5.3 Design process to automatically generate distributed implementations . . . 111

List of Tables

3.1 Code size in lines-of-code (loc) for MPEG4 encoder. 60
3.2 Code size in loc for concurrent sorting. 63

4.1 Performance of Bitonic Sorting Algorithm. 104
4.2 The impact of component composition on Send/Receive models. 105

CHAPTER 1

Introduction

Contents

1.1 Challenges in Building Correct Component-based Systems . . 21
1.2 State-of-the-Art of Component-based Systems Design and Implementation 24
1.2.1 Existing component-based frameworks 24
1.2.2 Discussion 26
1.3 Our Contribution 27
1.4 Organization of the Thesis 28

1.1 Challenges in Building Correct Component-based Systems

Computer systems are ubiquitous today, they are touching all aspects of human life. We can find them in different types of applications from automobile, to mundane home appliances like washing machines and microwave ovens, to aeronautic, military, telecommunication, medical etc.

Systems become more and more complex and their adoption is increasing exponentially. Moreover, the need to prove and to ensure their correctness, efficiency and reliability is an essential task that is also becoming complex. Although different techniques in software engineering exist for ensuring correctness such as formal verification, simulation, and testing, building correct and reliable systems is still a time-consuming and hardly predictive task.

Errors still exist in up to 90% of production spreadsheets. Moreover, the time spent in debugging them is considerably high and sometimes with no satisfying results. As an example, the United States has attempted to replace its air traffic control system three times in the past twenty years, but despite the billions of US dollars spent, no such replacement has happened.

Component-based Approach The most basic technique to tackle complex and large problems is to decompose them into smaller ones. As complex systems can be obtained by assembling components (building blocks), the process design of these complex systems may be reduced to the study of smaller and simpler ones. Thus, using such frameworks (called *component-based*) that allow building systems from predefined given components would be a great interest.

Component-based system development requires methods and tools supporting different concepts of architecture which provide a characterization coordination between components. An architecture is the structure of a system. It involves components and their relationship between the externally visible properties. This means that the architecture describes how components are connected and how they can interact. Consequently, the global behavior of a system can in principle be inferred from the behavior of its components and its architecture. Component-based systems provide logical clear descriptions which make them a good candidate for a correct-by-construction process. In addition, they allow sub-systems to be reused as well as their incremental modification without requiring global changes, which may significantly simplify the verification process. Nonetheless, clarity and expressiveness of component-based systems may be at the detriment of efficiency. Indeed, naive compilation of component-based systems generally results in great inefficiency as a consequence of the interconnection of components [Lov77].

High-Level Design When designing systems, dealing with their complexity at a high-level is a great help for the designers. In fact, it would be easier for designers to start working on the system at a high-level, where they do not have to take into account the complexity of the system. This *modeling* phase is beneficial, as designers can abstract away implementation details and validate the model with respect to a set of intended requirements through different techniques such as formal verification, simulation, and testing.

Unfortunately, once the abstract model is validated, deriving *correct* and *efficient* implementation from it is always challenging, since adding implementation details involves many subtleties that can potentially introduce errors into the resulting system.

Implementation Level High-level design is used to prototype and validate new designs. After this phase, the design is typically translated by hand into code for a system implementation. However, this translation process is complex and it may reduce the contribution of the verification and testing done at the high-level model. Therefore, automatic code

generation is necessary and may provide an important time gain. Nevertheless, the main drawback of this automatic approach is its efficiency. Indeed, the efficiency of a generated code depends in general on the target architecture.

Given a high-level model, the implementation generated could be, in general, centralized or distributed. This depends both on the topology of the systems and the architecture on which the system will be deployed. For instance, if we consider a centralized system with a small amount of computation and a high-level of communications overhead, then a centralized implementation behaves better than a distributed one. In other cases, we may have to choose a distributed implementation because the system itself is geographically distributed, or because deriving more computational power by using multiple processors is necessary. These kind of systems where distributed implementation is needed, are mainly used for world-wide-web, network-file server, banking network, peer-to-peer networks, process control systems, sensor networks, grid computing, etc.
It is clear the complexity are amplified significantly in the case of deriving distributed implementation because of inherently concurrent, non-deterministic, and non-atomic structure of distributed systems, as well as the occurrence of unanticipated physical and computational events such as faults. Moreover, it is unclear how to transform an abstract model (where atomicity is assumed through global state semantics and distribution details are omitted via employing high-level synchronization primitives) into a real distributed implementation.

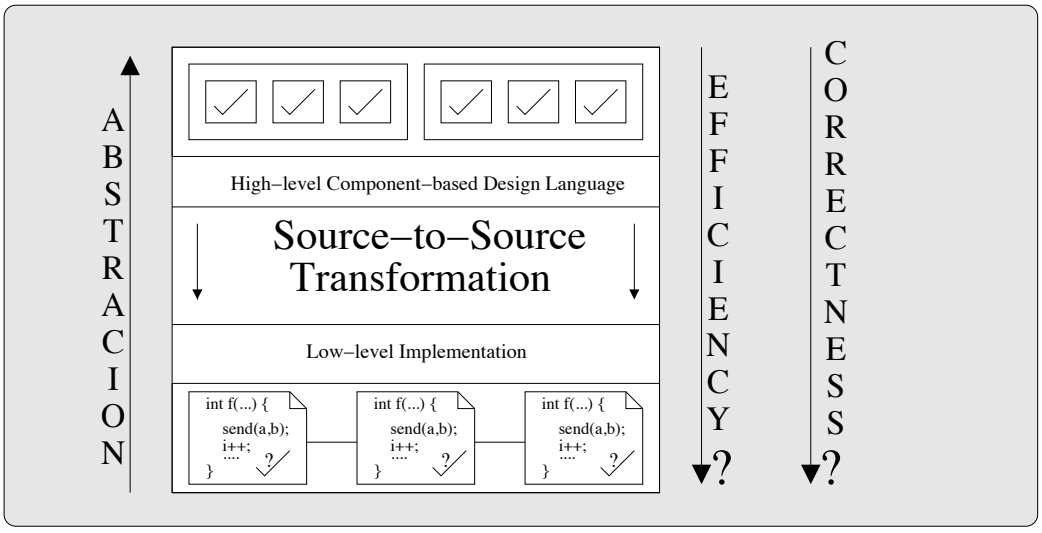


Figure 1.1: High-level component-based design into implementation.

In this thesis we propose a theory and tools for automatically derives *correct* and *efficient* centralized and distributed implementation in a systematic and ideally automated correct fashion from a high-level component-based framework. This is achieved, by using correct-by-construction source-to-source transformations techniques (see Figure 1.1).

The rest of the chapter is organised as follows. First in Section 1.2 we give a brief description of the current state of the art of component-based systems design and implementation. Then, in Section 1.3 we present the contribution of the thesis. Finally, we describe the outline of the thesis in Section 1.4.

1.2 State-of-the-Art of Component-based Systems Design and Implementation

In this section we summarize a brief description of the current state-of-the-art in component-based systems.

1.2.1 Existing component-based frameworks

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

- PtolemyII [DII+99, EJL+03] is a software framework, developed at Electrical Engineering and Computer Sciences (EECS) UC Berkeley University. PtolemyII focuses on component-based heterogeneous modeling. It advocates an actor-oriented view of a system, where the basic building block of a system is an actor. A model is a hierarchical interconnection of actors. Actors are software components that run concurrently and communicate through interfaces called ports. An actor can be atomic, in which case it must be at the bottom of the hierarchy. An actor can be composite, in which case it contains other actors. The semantics of a model is not determined by the framework, but rather than by a software component in the model called di-

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

- IF (Intermediate Format) [BGO+04] is a toolset, developed at Verimag. The IF toolset is a platform for component modeling and validation. It consists of set of timed automata communicating, either asynchronously by sending signals through FIFO, or synchronously with rendez on synchronized ports. The IF toolset uses techniques such as partial order reduction and on-the-fly model checking to explore the state space of the IF specification, giving access at the semantic level, to the corresponding labeled transition system (LTS). The latter can be analysed using the tool suite *CADP* [FGK+96], including the minimization and comparison tool *Aldebaran* based on bisimulation, and the alternating-free -calculus model-checker *Evaluator*.
- Fractal [BCS02, BCL+06] is a component model, developed at France Telecom R&D and INRIA France. Fractal is a general component model for implementing, deploying and managing complex software systems. It can be understood generally as being composed of a *membrane* which consists of a set of components (called sub-components) and one or more interfaces (similar to port in other component models). Interfaces can be of two kinds: server interfaces for incoming operation invocations, and client interfaces for outgoing operations invocations. Think [FSLM02, AHJ+09] is one of the Fractal implementation.
  However Fractal and THINK do not provide tools or analysis techniques, whether for simulation or verification.
- Metropolis [BWH+03] and its successor [DDM+07]. Metropolis is a component-based framework : atomic component containing behavior (code), composite component containing other sub-components. Interfaces of components consist of a set of ports used either for asynchronous communications (send events), or for rendez-vous (synchronization between components). Metro II provides a frontend which produces an internal representation from the meta-model. This representation can be used for, generation of C++ code for simulation, or generation model used with the SPIN model-checker, etc.

Other developments deal, one way or another, with issues related to component-based modeling:

- Software Design Description Languages [GS04, BFL+04], and Architecture Description Languages focusing on non-functional aspects [VPL99, AVCL02].
- Standardized system modeling languages such as UML [OMG] and associated tools.
- Languages and notations specific to system design tools such as SystemC [Pan01,

RHG+01], GME [BGK+06], Simulink/Stateow [Mat], Autofocus [HS02].
- Middleware standards such as Corba, Javabeans, .NET
- Software development environments such as PCTE, SWbus, Softbench, Eclipse.
- Coordination language extension of programming languages such as Linda, Javaspaces [FAH99], TSpaces [FLN+03], Concurrent Fortran, nesC [GLvB+03] and Polyphonic C♯ [BCF02].
- Theoretical frameworks based on process algebras e.g., the Pi-Calculus [Mil98] or based on automata e.g., [RC03].

1.2.2 Discussion

There are different requirements for building efficient and correct implementation for complex systems.

- Firstly, we need a component framework with the concept of component and associated composition operators for incremental description and correctness by construction.
- Secondly, this framework should be expressive enough to directly encompass all types of coordination, and hence, help designers to formulate their solutions in terms of tangible, well-founded and organized concepts instead of using dispersed coordination mechanisms such as semaphores, monitors, message passing, remote call, protocols etc.
- Thirdly, it also needs to be abstract enough by providing high-level primitives for modeling behaviors and communications. However, abstraction reduces expressiveness. Thus, the first challenge is to find the best compromise between a high level of abstraction and high expressiveness. All of these requirements lead to design complex systems in an easy and correct manner. Nonetheless, on top of abstraction and expressiveness, etc., other challenges appear, mainly how to automatically derive a correct and efficient implementation.
- For this reason, the fourth requirement, for such framework, is to provide a rigorous but not complex semantics, because complexity limitates abstraction. When having a rigorous semantics, we can define correct and automatic source-to-source transformation for deriving efficient low-level implementation from the high-level models in a correct manner. Moreover, a strong theoretical backing can be defined at the high-level models that allows formal verification of design properties.

Indeed, source-to-source transformations have been considered as a powerful means for optimizing programs [Lov77, HG06, BMFT07]. In contrast to conventional optimization techniques, they can be applied for deeper semantics-preserving transformations which are visible to programmers and subject to their direction and guidance.

In the context of component-based frameworks, we have not seen major work on source-to-source transformations, since component frameworks such as [BWH+03, DII+99] have well-defined *denotational semantics*. Nonetheless, it can be made only at the execution

level and not at source level. For example, it is not clear how to define component composition at source level from these semantics.
There also exist many component frameworks without rigorous semantics. This is particularly absent in the case of modeling, as well as for middleware and software development standards, like CORBA. They use ad-hoc mechanisms for building systems from components and offer syntax level concepts only. In this case, using ad-hoc transformations, may easily lead to inconsistencies e.g. transformations may not be confluent.
One the other hand, there are other techniques based on source-to-source transformations applied by language compilers. For example, compilation of synchronous languages [JHRC08], or optimizing communications in periodic reactive systems [CKL+05, CKL+02] use transformation techniques for optimization by flattening structure and composing Petri net behavior. Nonetheless, their underlying models are completely simple with respect to the communication primitives offered by these languages. For instance, the model considered in [CKL+05, CKL+02] is an extension of Kahn process networks allowing non-deterministic waiting on multiple input channels. The communication is binary, through point-to-point message passing on FIFO channels.

To this end, on exploring the current state of the art we have not seen a component-framework that meets the requirements above. Generally speaking, we can divide them into two categories. The first category provides high-level design and modeling, however it is still unclear how to derive correct and efficient implementation from the high-level models. In contrast, the second category provides efficient implementation, however the design process is either based on low-level primitives, or not expressive enough.

1.3 Our Contribution

We present, in this thesis, a methodology to provide automatically efficient and correct-by-construction centralized and distributed implementations starting from a high-level model of the software application in BIP. BIP (Behavior, Interaction, Priority) is a component-based framework with formal and rigorous semantics that rely on multiparty interactions for synchronizing components and dynamic priorities for scheduling between interactions.

A key idea of our methodology is to use a set of correct source-to-source transformations which preserve functional properties. Furthermore, they take into account extra-functional constraints. We propose several types of source-to-source transformations:

1. **Transformation for generating centralized implementations:** We define a set of transformations taking BIP models as input and transform them into functionally equivalent BIP models with different architectures. Such transformations allow in particular to generate from a hierarchical model an equivalent flat model or a single component by composing the behavior of the constituent components. From flat models monolithic C++ code can be generated. This code has been proven optimal

and much more efficient than the componentized code for implementation on a single-processor platforms;

2. **Transformation for generating distributed implementations:** Coordination in BIP is achieved through multiparty interactions and scheduling by using dynamic priorities. The associated semantics is defined on a global state model. This makes reasoning about systems easy. However, it is hard to obtain distributed implementations where the primitives available for communication and coordination are less powerful. For this reason, we propose automated transformation of high-level BIP models (where high atomicity is assumed and distributed coordination is sought by multi-party synchronization primitives) into distributed implementations in a systematic and correct fashion. In general, our methodology transforms arbitrary BIP models into Send/Receive BIP models, directly implementable on distributed execution platforms. The transformation consists of:
   – breaking atomicity of actions in atomic components by replacing strong synchronizations with asynchronous Send/Receive interactions;
   – inserting several distributed Engines that coordinate execution of interactions according to a user-defined partition;
   – augmenting the model with a distributed algorithm for handling conflicts between Engines.

   The obtained Send/Receive BIP models are proven observationally equivalent to the initial models. Hence, all the functional properties are preserved by construction in the implementation. Moreover, Send/Receive BIP models can be used to automatically derive distributed implementations. Currently, it is possible to generate stand-alone C++ implementations using either TCP sockets for conventional communication, or MPI implementation, for the deployment on multi-core platforms.

This approach has been fully implemented and integrated in the BIP framework.

1.4 Organization of the Thesis

The rest of the thesis consists of five chapters. In Chapter 2 we present an overview of the BIP framework. Then, in Chapter 3 we describe a set of source-to-source transformations for generating efficient centralized implementation for deployment on single-processor platforms. In Chapter 4 we present a method using source-to-source transformation for generating efficient distributed implementation for deployment on multi-core platforms. In Chapter 5 we present the tool implementing the techniques proposed in this thesis. Finally, Chapter 6 draws conclusion and future work. The details of all chapters are as follows:

– Chapter 2 presents the basic ideas about component-based methodology, the basic notions about components, their composition using glue operators, and the necessary properties for component-based construction of systems. It introduces the BIP component framework, describing its architecture, its semantics as well as its properties.

- Chapter 3 presents a method for generating efficient centralized implementation from high-level BIP model. The method is based on the successive application of three types of source-to-source transformations: flattening of components, flattening of connectors and composition of atomic components. We show that the system of the transformations is confluent and terminates. By exhaustive application of the transformations, any BIP component can be transformed into an equivalent monolithic component. From this component, efficient standalone C++ code can be generated. Applications of the method on two non trivial examples are also described in the chapter.
- Chapter 4 presents a method for generating efficient distributed implementations from high-level BIP model. First, in this chapter, we present the main subtitles of generation distributed implementations. Second, we define set of source-to-source transformations which transform a high-level BIP model (where atomicity is assumed through global state semantics and distribution details are omitted via employing high-level synchronization primitives) into a real distributed implementation that allows parallelism between components as well as parallel execution of interactions. Moreover, we prove that the defined transformations preserve observational equivalence. Applications of the method on three non trivial examples are also described in the chapter.
- In Chapter 5 we present a tool which implements the transformations defined in Chapter 3 and 4. Moreover, we give an overview of the integration of our tool in the design methodology of BIP for automatically deriving efficient centralized and distributed implementations from high-level BIP models.
- We conclude the thesis in Chapter 6, with an overview of the work and its future perspectives.

CHAPTER 2

The BIP Component-based Framework

Contents

2.1 General Overview 32
2.2 The BIP Language 33
2.2.1 Ports and Interfaces 34
2.2.2 Atomic Component 35
2.2.3 Connectors and Interactions 37
2.2.4 Priorities 41
2.2.5 Composite Components 42
2.3 Execution Platform 44
2.3.1 Enumerative Engine 45
2.3.2 Symbolic Engine 45
2.4 The BIP Tool-Chain 45
2.5 Summary 47

BIP (**B**ehavior, **I**nteraction, **P**riority) is a component framework for modeling heterogeneous real-time systems. In the first section, we give notions about component, their composition and the necessary properties for component-based construction of systems. Then, in Section 2.2 we present the BIP language. In Section 2.3 we present the execution platforms implementing the operational semantics of BIP. Then, Section 2.4 we give an overview of the BIP tool-chain. And we finish this chapter by giving conclusions in summary.

2.1 General Overview

BIP is a component framework with a formal operational semantics given in terms of Labeled Transition Systems and Structural Operational Semantics derivation rules. Indeed, A *component* is a behavioral entity, having a well defined interface. It denotes an executable specification whose runs can be modeled as sequences of discrete actions.
We distinguish two kinds of components: atomic and composite. Atomic components are the basic elements in the components hierarchy. Their behavior represented as labeled transition systems

Definition 2.1.1 (Labeled transition system.) *A labeled transition system is a triple* $B = (Q, \Sigma, \rightarrow)$*, where* Q *is a set of states,* Σ *is a set of labels, and* $\rightarrow \subseteq Q \times \Sigma \times Q$ *is a set of labeled transitions.*

For any pair of states $q, q' \in Q$ and label $a \in \Sigma$, we write $q \xrightarrow{a} q'$, iff $(q, a, q') \in T$. If such q' does not exist, we write $q \xrightarrow{a}\!\!\!\!\!\!\rightarrow$.

Composite components are obtained by composing together other components (atomic or composite) using a glue operator, then, a new component can be derived (see Figure 2.1). Their behavior is the product of behaviors of the inner components, with restriction implied by the glue.

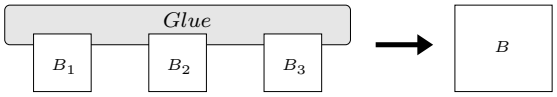


Figure 2.1: Components composition.

Our ultimate goal is to provide a methodology for component description and integration in a meaningful manner. The methodology must be incremental, i.e., components can be composed through a meaningful hierarchy of glues. Moreover, in order to ensure the correctness of composite components, it must provide support for compositinality and composability. In the following we give a description of these requirements:

– **Incrementally** a system can be considered as the composition of smaller components with the ability to the combination of decomposition and flattening (see Figure 2.2).

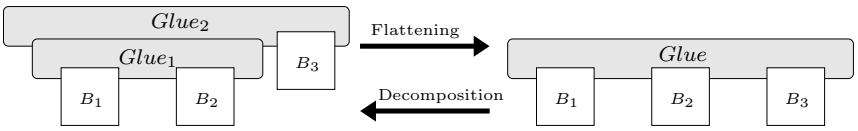


Figure 2.2: Incrementality of composition.

– **Compositionality** the possibility of inferring a global system properties from the local properties of sub-systems (e.g inferring global deadlock-freedom from the deadlock-freedom of the individual components) (see Figure 2.3).

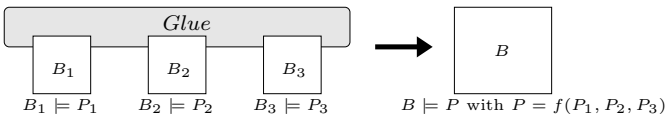


Figure 2.3: Compositionality of composition.

– **Composability** the preservation of the main properties of components during the construction of the system (see Figure 2.4).

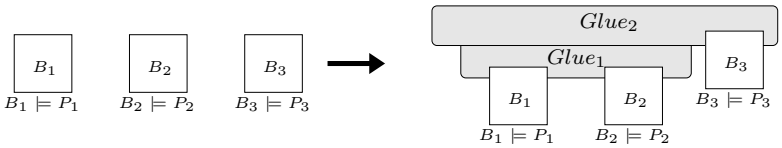


Figure 2.4: Composability of composition.

A detailed and fully formalized of these properties are presented in [Bas08]. The BIP component framework presents the composition of behaviors using two kinds of glue, *interactions* and *priorities*. It is shown in [BS08b] that these encompass the universal glue.

2.2 The BIP Language

BIP[Sif05, BBS06] is a component framework for constructing systems by superposing three layers of modeling (see Figure 2.5): Behavior, Interaction, and Priority. The lowest layer consists of a set of atomic components represented by transition systems. The second layer models Interaction between components. Interactions are sets of ports specified by connectors [BS08a]. Priority, given by a strict partial order on interactions, is used to enforce scheduling policies applied to interactions of the second layer. The BIP component framework has a formal operational semantics given in terms of Labeled Transition Systems and Structural Operational Semantics derivation rules. The BIP language offers primitives and constructs for modeling and composing complex behavior from atomic components. Atomic components are communicating Petri net extended with C functions and data. Transitions are labeled with sets of communication ports. Composite components are obtained from subcomponents by specifying connectors and priorities.

A component in BIP can also be viewed as a point in a three-dimensional space represented in Figure 2.6. The dimension *Behavior* characterizes component behavior and the

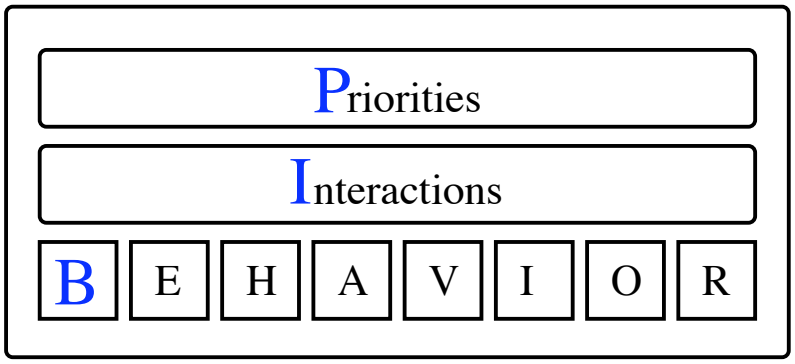


Figure 2.5: Layered component model.

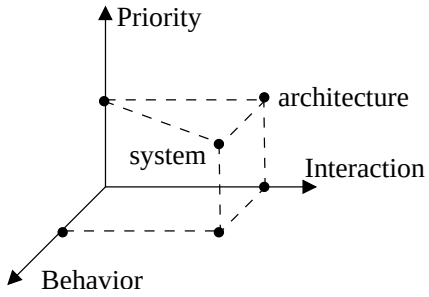


Figure 2.6: Three-dimentional space construction.

space $Interactions \times Priorities$ characterizes the overall structure of the system. In the following sections, we give a formal description of each of the layers, introduced here.

2.2.1 Ports and Interfaces

Ports are particular names defining communication points for components. As we shall see later, they are used to establish interactions between components by using connectors.

In BIP, we assume that every port has an associated distinct data variable x. This variable is used to exchange data with other components, when interactions take place.

A set of ports is called an interface.

Definition 2.2.1 (Port.) *A port $p[x]$ is defined by*
- *p – the port identifier,*
- *x – the data variable associated with the port.*

2.2.2 Atomic Component

Atomic component is a unit of behavior with an interface consisting of ports, and behavior encapsulated as a set of transitions. It consists of :

- A set of *control states* L, denoting locations at which the components await for synchronization.
- A set of *ports* P used for synchronization with other components. Ports form the interface of atomic components. They are instances of predefined port-types, and may be associated with atom variables.
- A set of *variables* X used to store (local) data. Basic C types can be used for variables. Variables may be associated to one or more ports. A variable associated to a port can be modified as a result of an interaction involving that port.
- A *behavior* given by the set of transitions modeling atomic computation steps. A transition represents a step from a set of control states L_1 to L_2 labeled by a port p, guard g, function f. A transition is denoted as $L_1 \xrightarrow{p,g,f} L_2$.

Here p is a port through which an interaction is sought, g a pre-condition for interaction through p, and f is a computation step consisting of local state transformations. g, also know as the guard of the transition, is a boolean condition on X. The transition can be executed if the guard is true.

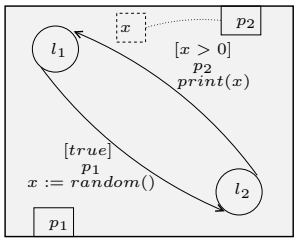
Example 1 *Figure 2.7(a) shows an example of an atomic component with two ports p_1, p_2, a variable x, and two control states l_1, l_2. At control state l_1, the transition labeled p_1 is enabled. When an interaction through p_1 takes place, a random value is assigned for the variable x. This value is exported through the port p_2. From the control state l_2, the transition labeled r_1 can occur (the guard is true by default), the variable a is eventually modified and the value of a is printed.*

Definition 2.2.2 (Atomic component.) *An atomic component B is defined by $B = (L, P, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$, where,*

- *(L, P, T) is a 1-safe Petri net, that is*
  - *$L = \{l_1, l_2, \ldots, l_k\}$ is a set of control states,*
  - *P is a set of ports,*
  - *$T \subseteq 2^L \times P \times 2^L$ is a set of transitions,*
- *$X = \{x_1, \ldots, x_n\}$ is a set of variables and for each transition $\tau \in T$, g_τ is a guard and f_τ is an update function that is state transformer defined on X, $(f_\tau(X))$.*

Hereafter, we use the dot notation to denote the parameters of atomic components. For example, $B.P$ means the set of ports of the atomic component B.

In order to define the operational semantics for atomic component, let us first introduce some notations. Given a Petri net $N = (L, P, T)$ we define the set of 1-safe markings $\mathcal{M}$



(a) Atomic component

```
port type IntPort(int x)
port type ePort()
atomic type Generator
    data int x = 0
    export port ePort p1() = p1
    export port IntPort p2(x) = p2
    place l1,l2
    initial to l1 do {}
        on p1 provided (true) from l1 to l2
            do { x=random(); }
        on p2 provided (x>0) from l2 to l1
            do { printf("The new value of x is:%d", x); }
end
```

(b) BIP code of the atomic component in Figure 2.7(a)

Figure 2.7: An example of an atomic component in BIP.

as the set of functions $m : L \to \{0, 1\}$. Given two markings m_1, m_2, we define inclusion $m_1 \leq m_2$ iff for all $l \in L$, $m_1(l) \leq m_2(l)$. Also, we define addition $m_1 + m_2$ as the marking m_{12} such that, for all $l \in L$, $m_{12}(l) = m_1(l) + m_2(l)$. Given a set of places $K \subset L$, we define its characteristic marking m_K by $m_K(l) = 1$ for all $l \in K$ and $m_K(l) = 0$ for all $l \in L \setminus K$. Moreover, when no confusion is possible from the context, we will simply use K to denote its characteristic marking m_K. Finally, for a given transition $\tau \in T$, we define its pre-set ${}^\bullet\tau$ (resp. post-set $\tau^\bullet$) as the set of the control states which are direct predecessors (resp. successors) of this transition.

Definition 2.2.3 (Atomic component semantics.) *The semantics of an atomic component* $B = (L, P, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ *is defined as the labeled transition system* $\mathcal{S}_B = (Q_B, \Sigma_B, \xrightarrow[B]{})$ *where*

- $Q_B = \mathcal{M} \times \mathcal{V}$ *is the set of states defined by:*
  - $\mathcal{M} = \{m : L \to \{0, 1\}\}$ *the set of* 1*-safe markings,*
  - $\mathcal{V} = \{v : X \mapsto \mathcal{D}\}$ *is the set of valuations of variables* X*,*
- $\Sigma_B = P \times \mathcal{D}^2$ *is the set of labels,*
- $\xrightarrow[B]{} = Q_B \times \Sigma_B \times Q_B$ *is the set of transitions defined by the following rule:*

$$\frac{\tau \in T \quad p \in P \quad {}^\bullet\tau \leq m \quad g_\tau(v) = true \quad v^{up} = v(x_p) \quad m' = m -^\bullet \tau + \tau^\bullet \quad v' = f_\tau(v[x_p \mapsto v^{dn}]) \quad v^{up}, v^{dn} \in \mathcal{D}}{(m, v) \xrightarrow[B]{p(v^{up}/v^{dn})} (m', v')}$$

This rule correspond to the firing of behavior transition. Indeed, a transition can be taken as soon as they are enabled by the marking and the guard, and update the data valuation and the marking, according to the net flow and annotations of the transition. Moreover, it perform an instantaneous data exchange through the port p: the current value v^{up} is sent and a new value v^{dn} is received for x_p, before the update.

2.2.3 Connectors and Interactions

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.

A connector is a set of ports of components which can be involved in an interaction. The number of interactions of a connector can grow exponentially to the number of ports. A connector is a macro notation for representing sets of related interactions in a compact manner.

Two types of port (*synchron*,*trigger*) are defined, in order to specify the feasible interactions of a connector. A trigger is an active port, and can initiate an interaction without synchronizing with other ports. It is represented graphically by a triangle. A synchron port is passive, hence needs synchronization with other ports, and is denoted by a circle. A feasible interaction of a connector is a set of its ports such that either it contains some trigger, or it is maximal, i.e., consisting of all the synchron ports. Example of sets of

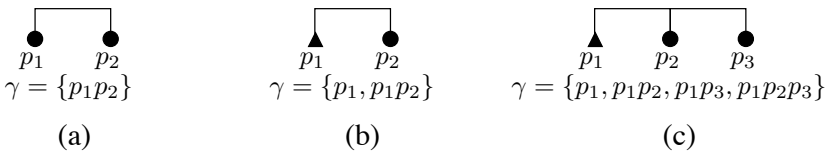


Figure 2.8: Connectors and their interactions.

connectors and their feasible interactions are shown in Figure 2.8. By convention, triangles represent trigger and circles represent synchron ports. In the partially ordered set of interactions, the shaded nodes denote feasible interactions. In (a), the connector consists of the ports $p1$ and $p2$, both are of type synchron. In this connector, the only feasible interaction is $p1p2$. It represents a rendezvous, meaning that both actions are necessary for the synchronization. In (b), the interaction between $p1$ and $p2$ is asymmetric as $p1$ is a trigger and can occur alone, even if $p2$ is not possible. Nevertheless, the occurrence of

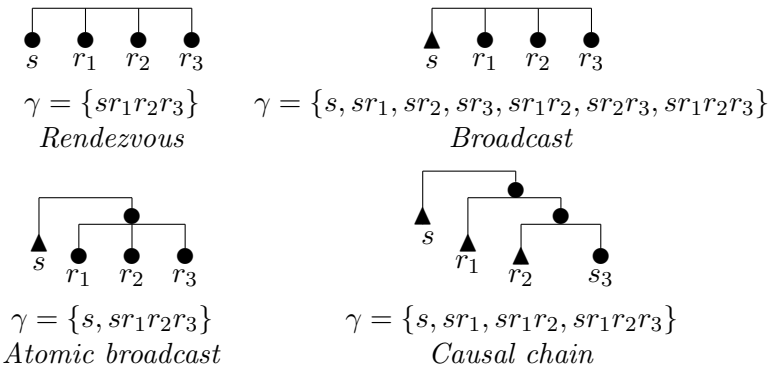


Figure 2.9: Graphic representation of connectors

$p2$ requires the occurrence of $p1$. The feasible interactions are $p1$ and $p1p2$. In (c), the interactions between $p1$, $p2$ and $p3$ are also asymmetric. The interactions $p1$ can occur alone or synchronize with either or both $p2$ and $p3$.

On the other hand, connectors sometimes need to be structured, i.e., having types associated to groups of ports. This is necessary to represent some interactions. The representation of structured connectors require connectors to be treated as expressions with typing and other operations on groups of connectors. This led to the formalization of the algebra of connectors defined in [BS07]. The Algebra of Connectors is defined to provide a compact notation for algebraic representation and manipulation of connectors. The Algebra of Connectors *AC(P)*, introduced in [BS07], formalizes the concept of connectors supported by the BIP component model. It extends the notion of connectors to terms built from a set of ports by using a n-ary fusion operator and a unary typing operator for *triggers* and *synchrons*.

Figure 2.9 shows four different coordination schemes:

- Rendezvous means strong synchronization between port s and all r_i. This is specified by a single interaction involving all the ports. This interaction can occur only if all the components are in states enabling transitions labeled respectively by s, r_1, r_2, r_3.
- Broadcast means weak synchronization, that is a synchronization involving s and any (possibly empty) subset of r_i. This is specified by the set of all interactions containing s.
- Atomic broadcast means that either a message is received by all r_i, or by none. Two interactions are possible: s, when at least one of the receiving ports is not active, and the interaction $s\, r_1\, r_2\, r_3$, corresponding to strong synchronization.
- Causal chain means that for a message to be received by r_i it has to be received at the same time by all r_j, for $j < i$.

For rendezvous, the priority model is empty. For all other coordination schemes, the

maximal progress priority model ensures that, whenever several interactions are possible, the interaction involving a maximal number of ports has higher priority.

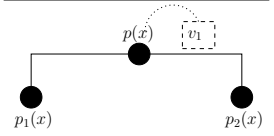
An interaction consists of one or more ports of the connector, a guard on the variables of the ports of the interaction, two methods `up{}` and `down{}` realize data transfer between the ports of the interaction. The method `up{}` updates the local variables of the connector based on the values of variables associated with the ports. The method `down{}` updates the variables associated with the ports based on the values of the interaction variables. This structure also allows data transfer in hierarchical connectors.

Example 2 *Figure 2.11 gives an example of hierarchical connectors allowing data transfer in hierarchical manner. Let consider an initial valuation* $p_1.x \mapsto 3$, $p_2.x \mapsto 5$, $p_3.x \mapsto 8$, $p_5.x \mapsto 9$ *for the bottom ports. During the upward transfer, the value* 9 *is propagated to* v_1 *and* v_2*, following the upward predicates* p_1p_2 *and then* $p_3p_4p_5$*. Then, during the downward transfer the value* 9 *gets propagated downwards to* $p1.x$, $p2.x$, $p3.x$ *and* $p_5.x$ *following* $p_3p_4p_5$ *and then* p_1p_2.

Definition 2.2.4 (Connector.) *A connector* $\gamma = (p[x], P, A)$ *is defined as follows*

- p *is the exported port of the connector* γ,
- $P = \{p_i[x_i]\}_{i \in I}$ *is the support set of* γ*, that is, the set of ports that* γ *synchronizes,*
- $A \subseteq 2^P$ *is a set of interactions* $a = \{p_i\}_{i \in I}$ *labeled by* G, U, D *where,*
  - G *is the guard of* γ*, an arbitrary predicate* $G(\{x_i\}_{i \in I})$,
  - U *is the upward update function of* γ *of the form,* $x := F^u(\{x_i\}_{i \in I})$,
  - D *is the downward update function of* γ *of the form,* $\cup_{p_i}\{x_i := F^d_{x_i}(x)\}$.

Hereafter, we use the dot notation to denote the parameters of connectors. For example, $\gamma.A$ means the set of interactions of the connector γ.



(a) Interaction

```
connector type Sync2Max(IntPort p1,IntPort p2)
   data int v1
   define [p1 p2]
   on p1 p2 provided (p1.x > 0) && (p2.x > 0)
      up {v1 = Max(p1.x,p2.x);}
      down {p1.x = p2.x = v1;}
   export port IntPort p(v1)
end
```

(b) BIP code of the interaction in Figure 2.10(a)

Figure 2.10: An example of a connector containing one interaction in BIP.

Example 3 *Figure 2.10(a) shows a connector with two ports p_1, p_2, and exported port p_3 (allows to define hierarchical connectors). There is one and only one interaction p_1p_2 feasible by this connector. Synchronization through this interaction involves two steps providing its guard $G : (p_1.x > 0)\&\&(p_2.x > 0)$ is true: 1) The computation of the upward update function U by assigning to v_1 the maximum of the values of $p_1.x$ and $p_2.x$; 2) The computation of the downward update function D by assigning the value of v to $p_1.x$ and $p_2.x$. Figure 2.10(b) presents the corresponding BIP code.*

Definition 2.2.5 *For a set of connectors $\Gamma = \{\gamma_j\}_{j\in J}$, we define the dominance relation $\rightarrow$ on Γ as follows:*

$$\gamma_i \rightarrow \gamma_j \equiv \gamma_j.p \in \gamma_i.P$$

That is, γ_i dominates γ_j means that the exported port of γ_j belongs to the support set of γ_i (see Figure 2.11). By definition, we assume that the dominance relation has no cycle.

Let $P(\Gamma) = \{p_0 \mid \gamma = (p_0, P, A) \in \Gamma\}$ be the set of their (distinct) exported ports..

Definition 2.2.6 (Interaction tree.) *Let $\Gamma = \{\gamma_j\}_{j\in J}$ a set of connectors, an interaction tree $a^t = (A_\Gamma, \rightarrow_\Gamma)$ of Γ is defined as follows:*

- *$A_\Gamma \subseteq \cup\{\gamma_j.A\}_{j\in J}$ is a set of interactions,*
- *Let $a_1, a_2 \in A_\Gamma$, we have $a_1 \rightarrow_\Gamma a_2$ iff $\exists\gamma_1, \gamma_2 \in \Gamma$ such that $a_1 \in \gamma_1$, $a_2 \in \gamma_2$ and $\gamma_2.p \in a_1$. Moreover, $\rightarrow_\Gamma$ should satisfy the following conditions:*
  - *a^t must contains a uniquely defined interaction $a_0 \in A_\Gamma$, called $top(a^t)$ from which all other interactions are recursively dependent, that is, $\exists a_0.\forall a_i \in A_\Gamma, a_0 \rightarrow^* a_i$, and a_0 is unique.*
  - *a^t must contains at most one interaction per connector. That is, if $\exists a_i \in A_\Gamma \cap \gamma.A$, then, $\forall a_k \in \gamma.A \setminus a_i,\ a_k \notin A_\Gamma$.*

Let Γ be a set of connectors such that $(\Gamma, \rightarrow)$ has no cycle and let $a^t = (A_\Gamma, \rightarrow_\Gamma)$ be an interaction tree of Γ. We denote by:

- *$A^t(\Gamma) = \{a_i^t \mid a_i^t$ is an interaction tree of $\Gamma\}$, the set of all interaction trees of Γ;*
- *$bottom(a^t) = \{a_i \mid a_i \in A_\Gamma, a_i \cap P(\Gamma) = \emptyset\}$, the leaf interactions of the tree a^t;*
- *$support(a^t) = \{p \mid p \in a_i, a_i \in bottom(a^t)\}$, the support set of ports for the leaf interactions of the tree a^t.*

Definition 2.2.7 (Hierarchical connectors semantics.) *Let $\Gamma = \{\gamma_j\}_{j\in J}$ a set of connectors. Executing of interactions in Γ implies an execution of an arbitrary interaction tree $a^t \in A^t(\Gamma)$. Moreover, it involves transfer of data between synchronizing ports. In particular, let σ_0 an initial valuation of $bottom(a^t)$, $\sigma_0 = \{p.x \rightarrow v_p \mid p \in support(a^t)\}$. The upward valuation $\mathcal{U}_{a^t}(\sigma_0)$ is obtained by propagating values from ports in the bottom interactions into the tree a^t according to upward update functions of the interactions of the tree, as long as the guard conditions allow them $\mathcal{G}_{a^t}$. In a dual manner, we define the downward*

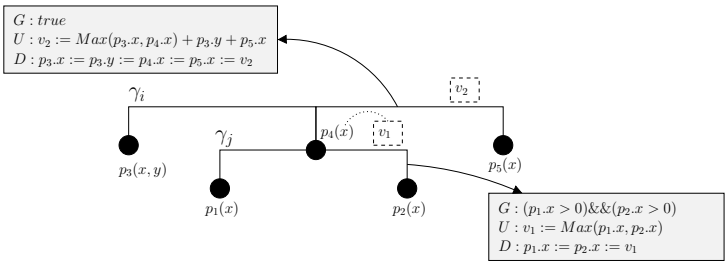


Figure 2.11: γ_i dominates γ_j.

valuation $\mathcal{D}_{a^t}(\sigma)$ obtained by transforming a given valuation σ on ports of interactions of a^t according to their downward update functions. More precisely, guards and update functions, $\mathcal{G}_{a^t}$, $\mathcal{U}_{a^t}$ and $\mathcal{D}_{a^t}$ are defined as follows:

$$\mathcal{U}_{a^t} = \begin{cases} U_a & a^t = \{a\}, \\ \mathcal{U}_{a^{t'}} \circ U_a & a^{t'} = a^t \setminus \{a\}, a \in bottom(a^t). \end{cases}$$

$$\mathcal{G}_{a^t} = \begin{cases} G_a & a^t = \{a\}, \\ \mathcal{G}_{a^{t'}} \wedge G_a & a^{t'} = a^t \setminus \{a\}, a \in bottom(a^t). \end{cases}$$

$$\mathcal{D}_{a_t} = \begin{cases} D_a & a^t = \{a\}, \\ D_a \circ \mathcal{D}_{a^{t'}} & a^{t'} = a^t \setminus \{a\}, a \in bottom(a^t). \end{cases}$$

Definition 2.2.8 (Flat connectors.) *Γ is a set of flat connectors, iff no connector dominates another, that is, $\forall \gamma_i, \gamma_j \in \Gamma$ we have $\gamma_i \not\rightarrow \gamma_j$.*

2.2.4 Priorities

Given a system of interacting components, priorities are used to filter the enabled interactions. They are given by a set of rules, each consisting of an ordered pair of interactions or connectors. When connectors are specified in a priority, the rules apply between all the respective interactions of the connectors. Dynamic priorities can be specified by providing guard condition, which are boolean expression in C on the variables of the components involved in the interactions. The maximal progress priority is enforced implicitly by the BIP Engine: if one interaction is contained in another one, the latter has higher priority. Below is an example of priority expressed in the BIP language.

```
priority p1 if(G) c1 < c2
```

This specifies the priority $p1$ that, when the boolean condition `G` is true, interactions of connector `c2` would be preferred to those of `c1`.

2.2.5 Composite Components

Composite components are defined recursively by composition from atomic components or other composite components using glue consisting of interaction and priority models. The interface of a composite component is defined by exporting ports of subcomponents and connectors.

Definition 2.2.9 (Component.) *A composite component (or simply component) C is defined by the following grammar:*

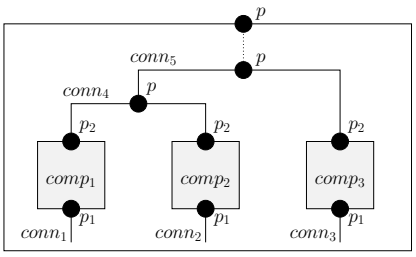
$$C ::= B|(\{C_i\}_{i\in I}, \Gamma, P)$$

where,

- *B is an atomic component,*
- *$\{C_i\}_{i\in I}$ is a set of constituent components,*
- *$P = (\cup_{i\in I} C_i.P) \cup (\cup_{j\in J}\{\gamma_j.p\})$, is the set of ports of the component, that is P contains the ports of the constituent components and the exported ports of the connectors,*
- *$\Gamma = \{\gamma_j\}_{j\in J}$ is a set of connectors, such that,*
  1. *$(\Gamma, \rightarrow)$ has no cycle,*
  2. *$\cup_{j\in J}\gamma_j.P \subseteq P$ (P is defined above),*
  3. *Each $\gamma \in \Gamma$ uses at most one port of every constituent component, that is, $\forall\gamma \in \Gamma, \forall i \in I$, $|C_i.P \cap \gamma.P| \leq 1$.*

Notice that a component is either an atomic component B or a composite component obtained as the composition of a set of constituent components $\{C_i\}_{i\in I}$ by using a set of connectors $\Gamma = \{\gamma_j\}_{j\in J}$. The restriction 3) is needed to prevent simultaneous firing of two or more transitions in the same atomic component, because they may affect the same variables.

Example 4 *Figure 2.12(a) shows a compound component consisting of three identical atomic components described in Figure 2.7, connected by using the connector described in Figure 2.10. Each atomic component generates an integer. Then it synchronizes with all the other atomic components. During synchronization the global maximal value is computed and each atomic component receives the maximum of the values generated. Figure 2.12(b) presents the corresponding BIP code.*



(a) Composite component

```
compound type CompoundExample
   component Generator comp1
   component Generator comp2
   component Generator comp3

   connector Singelton conn1(comp1.p1)
   connector Singelton conn2(comp2.p1)
   connector Singelton conn3(comp3.p1)
   connector Sync2Max conn4(comp1.p2,comp2.p2)
   connector Sync2Max conn5(conn4.p,comp3.p2)

   export port IntPort p is conn5.p
end
```

(b) BIP code of the compound in Figure 2.12(a)

Figure 2.12: An example of compound component in BIP.

Definition 2.2.10 (Flat component.) *Composite component C is flat, iff the set of constituent component $\{C_i\}_{i\in I}$ are atomic components.*

The operational semantics of composite components is recursively defined on the component structure. For atomic components, their semantics coincides with the semantics of the underlying behavior. For composition, the semantics is obtained by restricting the parallel behavior according to the interaction and priority models applied.

Definition 2.2.11 (Component Semantics.) *The semantics of component C is a labeled transition system $\mathcal{S}_C = (Q_C, \Sigma_C, \xrightarrow[C]{})$ defined inductively on the structure of C as follows:*

1. *C is an atomic component, defined by an atomic behavior $B = (L, P, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$. Then, $\mathcal{S}_C = \mathcal{S}_B$ (see Definition 2.2.3).*
2. *C is a composite component defined as the $(\{C_i\}_{i\in I}, \Gamma, P)$, where C is flat. Let $\mathcal{S}_{C_i} = (Q_C, \Sigma_{C_i}, \xrightarrow[C_i]{})$ be the semantics of its atomic components. The labeled transition system $\mathcal{S}_C = (Q_C, \Sigma_C, \xrightarrow[C]{})$ is defined as:*
   - *$Q_C = \bigotimes_{j\in J} Q_{C_j}$ is the of states, the Cartesian product of set of states of subcomponents,*
   - *$\Sigma_C = A^t(\Gamma)$ is the set of labels,*
   - *$\xrightarrow[C]{} \subseteq Q_C \times \Sigma_C \times Q_C$ is the transition relation, defined by the following rule:*

$$\frac{a^t = (A_\Gamma, \rightarrow) \in A^t(\Gamma) \qquad support(a^t) = \{p_j\}_{j\in J}}{\forall j \in J \qquad q_j \xrightarrow[C_j]{p_j(v_j/v_j')} q_j' \qquad \{v_j'\}_{j\in J} = \mathcal{D}_{a^t} \circ \mathcal{U}_{a^t}(\{v_j\}_{j\in J}) \qquad \mathcal{G}_{a^t}(\{v_j\}_{j\in J})}$$
$$\frac{v_j, v_j' \in \mathcal{D} \qquad \forall k \notin J.\ q_k = q_k'}{q \xrightarrow[C]{a^t} q'}$$

3. *if C is a composite component defined as the $(\{C_i\}_{i\in I}, \Gamma, P)$, and C is not flat. Then, the same principle as above can be defined.*

Example 5 *Figure 2.13 illustrates an abstract overview of the semantics of composite component. The composed behavior is shown in the right. It shows the product of the two behaviors, where the only allowed transitions are the ones with a solid arrow. The dotted transitions are not legal and shows the maximal behavior allowed by the interactions glue.*

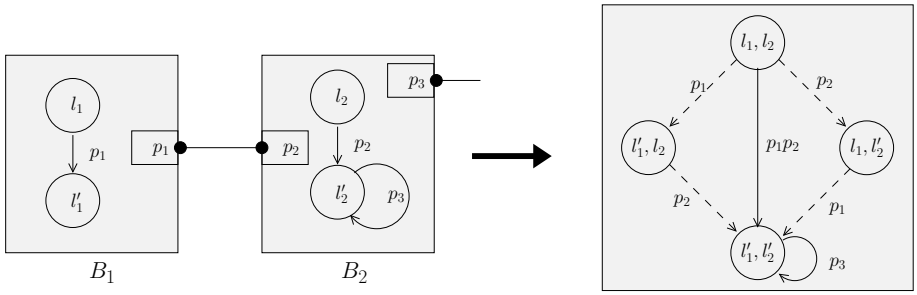


Figure 2.13: Component composition.

2.3 Execution Platform

The operational semantics is implemented by an Engine. In the basic implementation, the Engine computes the enabled interactions by enumerating over the complete list of interactions in the model (enumerative Engine). Another implementation is based on computing boolean representation for components and connectors by using an existing BDD package (symbolic Engine) [1].

1. The implementation of the boolean functions is made using the BDD package CUDD.

2.3.1 Enumerative Engine

During the execution, on each iteration of the Engine, the enabled interactions are selected from the complete list of interactions, based on the current state of the atomic components. Then, between the enabled interactions, priority rules are applied to eliminate the ones with low priority. The main loop of the Engine consists of the following steps:

1. Each atomic component sends to the Engine its current state.
2. The Engine enumerates on the list of interactions in the model, selects the enabled ones based on the current states of the atomic components and eliminates the ones with low priority.
3. Amongst the enabled interactions, the Engine selects any one and notifies the involved atoms the transition to take.

The time to compute the enabled interactions by Engine is proportional to the number of interactions in the model.

2.3.2 Symbolic Engine

In the enumerative BIP Engine [JBB09], for each connector, the Engine needs to compute all the possible interactions, check which ones are enabled in the current global state of the system, and select a maximal enabled one to be executed. As interactions are sets of ports, their number is potentially exponential in the number of ports in the connector. Hence, in the worst case, the performance of this Engine can be extremely poor.

The boolean BIP Engine leverages on representing component behavior, connector interactions, and priorities as boolean functions. For an atomic component, all ports and control states are represented by boolean variables. This allows to encode behavior as a boolean expression of these variables. Similarly, each connector is represented by the boolean expression on its ports. The global behavior is obtained as a boolean operation on the expressions representing atomic components, connectors, and priorities.

The choice of an interaction to be executed boils down to evaluating the control states, substituting their respective boolean variables, and picking a valuation of the port variables satisfying the boolean expression that represents the global behavior.

The boolean representation of connectors replaces the costly enumeration step by efficient BDD manipulations. In comparison to the exponential cost of the enumerative Engine, this renders a more efficient Engine with evaluation that, in practice, remains linear.

2.4 The BIP Tool-Chain

The BIP tool-chain provides a set of tools for the modeling, the execution, the verification and the static transformation of BIP models. The overview of the BIP tool-chain is

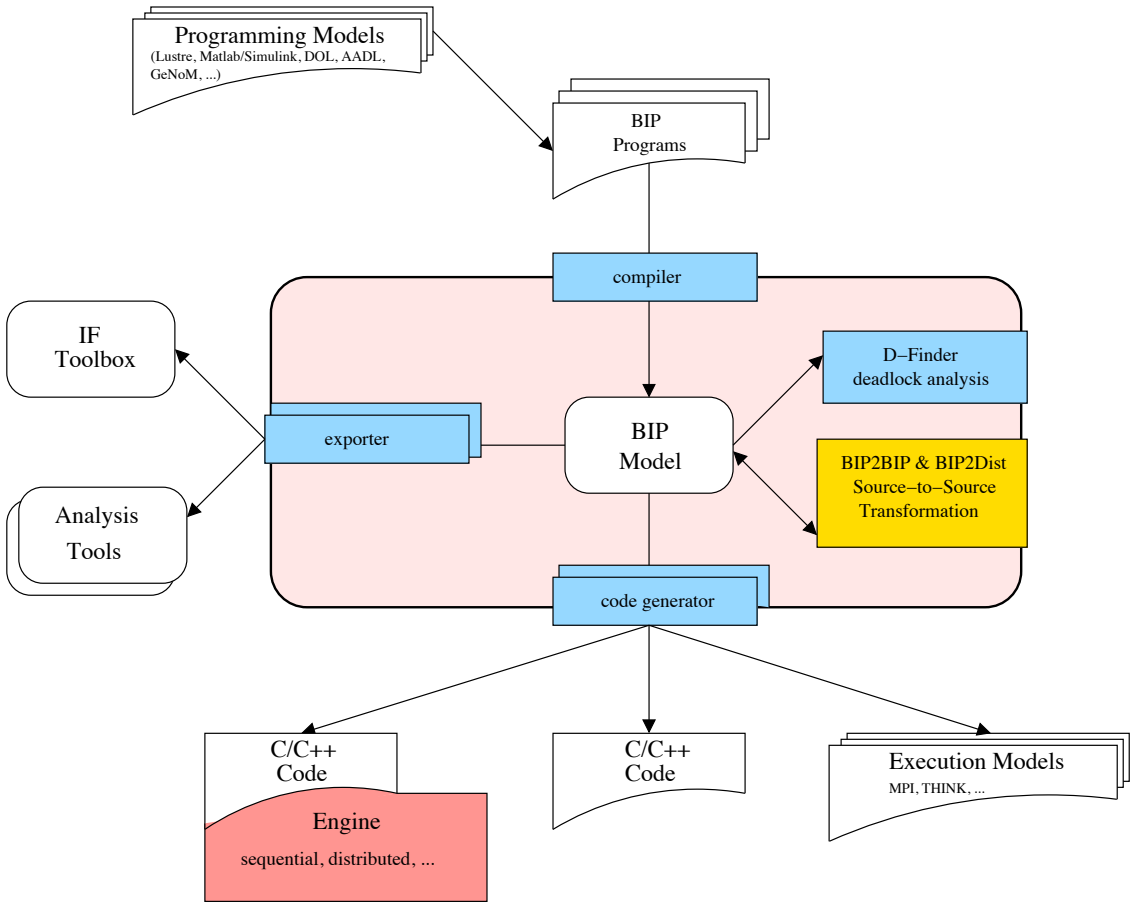


Figure 2.14: The BIP tool-chain.

shown in Figure 2.14. It includes the following tools:

- An *editor*, for describing textually a system in BIP language.
- A *compiler*, for generating a BIP model from BIP description source.
- A *code generator*, for generating, from a model, C++ code executable on the BIP Engine. The code-generator can also produce THINK specification [Pou10, PPRS06], from which the Think tool-chain can generate code to be executed over a choice of target platforms.
- *D-Finder*, is a compositional verification tool for component-based systems described in BIP language [BBNS09, BBSN08].
- *Source-to-Source transformations*, of which the method and the implementation are presented in this thesis, allow useful transformations which generate efficient central-

ized [BJS09] and distributed implementation from a composite component [BBJ+10b, BBJ+10a].
- An *exporter* to connect with external tools such as IF toolbox or analysis tools.
- A set of translators from other languages (Lustre, MATLAB/Simulink, AADL, etc.) to BIP. For example, a *Simulink-to-BIP* translation [STS+10] from Simulink models into BIP which allows the validation and implementation of Simulink models. In particular, compositional and incremental generation of invariants can be applied for complex Simulink models. These compilation paths are also becoming available for Simulink models. Moreover, Simulink models can be explored using the compilation paths of BIP. Other example of translations is *AADL-to-BIP* translation from Architecture Analysis & Design Language (AADL) into BIP [CRBS08], allows simulation of systems specified in AADL and application to these systems of formal verification techniques developed for BIP, e.g. deadlock detection.

2.5 Summary

Component-based approach is aimed to deal with the complexity of systems. It is based on the idea of building a complex system by assembling basic components (blocks). It provides important characteristics for system construction such as reuse, incrementality, compositionality, etc. It allows not only the reuse of components but also the reuse of known properties of constituent components.

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

BIP modeling framework allows dealing with complexity of systems by providing incremental composition of heterogeneous components. It also considers correctness-by-construction for a class of essential properties such as deadlock-freedom [GS05].

The BIP tool-chain has been developed providing automated support for component integration and generation of glue code meeting given requirements. Efficient model transformations, verification methods have also been studied and implemented in the BIP tool-chain.

We are now going to present a method for generating efficient centralized implementations from BIP models. We will also show applications of our method for MPEG4 encoder and network sorting algorithm described in the BIP language.

CHAPTER 3

Transformation for Generating Centralized Implementations

Contents

3.1 Problem Statement 49
3.2 Transformations 50
3.2.1 Components Flattening 51
3.2.2 Connectors Flattening 52
3.2.3 Components Composition 56
3.3 Efficient Sequential Implementation 58
3.4 Experimental Validation 58
3.4.1 MPEG Video Encoder 58
3.4.2 Concurrent Sorting 60
3.5 Summary 65

3.1 Problem Statement

Efficient implementation of component-based systems is a non-trivial task. Moreover, clarity of models may be at the detriment of efficiency. Indeed, naive compilation of component-based systems results in great inefficiency as a consequence of the interconnection of components [Lov77]. Nowadays, it is widely admitted that modularity in component-based development incurs an additional non-negligible overhead for implementation because of extensive use of interfaces, wrappers and other implementation artifacts. For instance, the generated BIP code is modular and can be executed on a dedicated platform consisting of an Engine which orchestrates the computation of atomic components

by executing their interactions (described in Section 2.3). Hierarchical description allows incremental reasoning and progressive design of complex systems. Nonetheless, it may lead to inefficient programs if structure is preserved at run time. Compared to functionally equivalent monolithic C programs, BIP programs may be more than two times slower. This overhead is due to the computation of interactions between components by the Engine.

The aim of this chapter is to show that it is possible to synthesize efficient monolithic code from component-based software described incrementally. We study source-to-source transformations for BIP allowing the composition of components and thus leading to more efficient code. These are based on the operational semantics of BIP which allows to compute the meaning of a composite component as a behaviorally equivalent atomic component. Thus, we show how by incremental composition of the components contained in a composite component, a behaviorally equivalent component can be computed. This composition operation has been implemented in the BIP2BIP tool, by using three types of source-to-source transformations. A set of interacting components is replaced by a functionally equivalent component. By successive application of transformations, an atomic component can be obtained, that is a component with no interactions.

The transformation from a composite component to an atomic one is fully automated and implemented through three steps:

1. *Component flattening* which replaces the hierarchy on components by a set of hierarchically structured connectors applied on atomic components;
2. *Connector flattening* which computes for each hierarchically structured connector an equivalent flat connector;
3. *Component composition* which composes atomic components to get an atomic component.

Using such a transformation allows to combine advantages of component-based descriptions such as clarity and reuse with efficient implementation. The generated code is readable and by-construction functionally equivalent to the component-based model. We show through non trivial examples the benefits of this approach.

The rest of the chapter is organised as follows. First in Section 3.2 we define the three source-to-source transformations. In section 3.4, we provide benchmarks for two examples: a MPEG encoder and a concurrent sorting program. In Section 3.5, we finish this chapter by giving conclusions in summary.

3.2 Transformations

In this section, we will define the transformations which successively transform a composite component into atomic components. That is, they eliminate component hierarchy and the hierarchical connectors by computing the product behavior. The transformation

from a composite component to an atomic one involves three steps: Component flattening, Connector flattening, Component composition. In this section, we describe the three transformations, and we illustrate them on the example shown in Figure 3.1. This example consists of two composite components (C_1 and C_2). Each one of these composite components consists of three identical atomic components described in Figure 3.1-c, connected by using the connector described in Figure 3.1-b. Each atomic component generates an integer. Then it synchronizes with all the other atomic components. During synchronization the global maximal value is computed and each atomic component receives the maximum of the values generated.

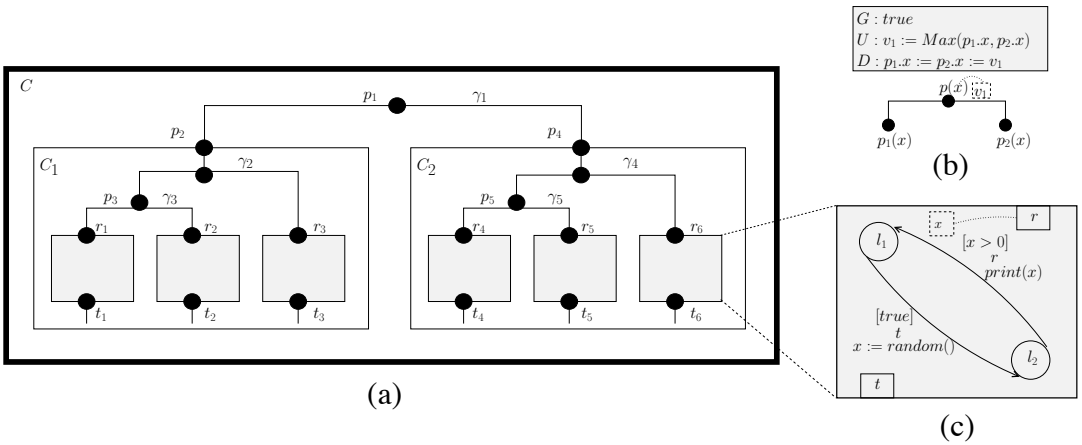


Figure 3.1: Example.

3.2.1 Components Flattening

This transformation replaces the hierarchy on components by a set of hierarchically structured connectors applied on atomic components. Consider a composite component C, obtained as the composition of a set of components $\{C_i\}_{i\in I}$. The purpose of this transformation is to replace each non atomic component C_j of C by its description. By successive applications of this transformation, the component C can be modelled as the set of its atomic components and their hierarchically structured connectors (see Figure 3.2).

Definition 3.2.1 (Component flattening.) *Consider a non atomic component* $C = (\{C_i\}_{i\in I}, \Gamma, P)$ *such that there exists a non atomic component* $C_j \in \{C_i\}_{i\in I}$ *with* $C_j = (\{C_{jk}\}_{k\in K}, \Gamma_j\ , P_j)$. *We define* $C[C_j \mapsto \Gamma_j]$ *as the component* $C = (\{C_i\}_{i\in I} \cup \{C_{jk}\}_{k\in K} \setminus \{C_j\}, \Gamma \cup \Gamma_j, P)$. *Component flattening is defined by the following function:*

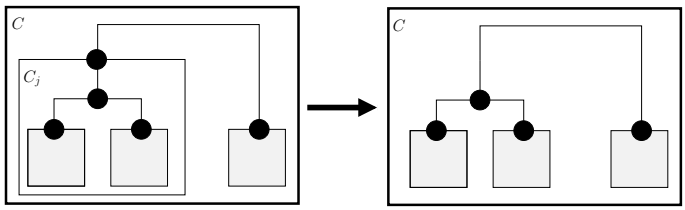


Figure 3.2: Component flattening.

$$\mathcal{F}_c(C) = \begin{cases} C & \text{if } C \text{ is flat} \\ \mathcal{F}_c(C[C_j \mapsto \Gamma_j]) & \text{if } C \text{ is not flat} \end{cases}$$

Proposition 1 *Component flattening is well-defined i.e., $\mathcal{F}_c$ is a function which produces a unique result on every input component, and terminates in a finite number of steps.*

Proof Regarding unicity of the result, we can show that, if two constituent components respectively C_j and C_k can be replaced inside the composite component C, then the replacement can be done in any order and the final result is the same. That is, formally we have $C[C_j \mapsto \Gamma_j][C_k \mapsto \Gamma_k] = C[C_k \mapsto \Gamma_k][C_j \mapsto \Gamma_j]$. The result follows immediately from the definition and elementary properties of union on sets.

Regarding termination, every transformation step decreases the overall number of composite components by one, so component flattening eventually terminates when all the components are atomic.

By applying to Example in Figure 3.1 the transformation $C[C_1 \mapsto \{\gamma_2, \gamma_3\}]$ then $C[C_2 \mapsto \{\gamma_4, \gamma_5\}]$, we obtain the new component in Figure 3.3.
Finally, notice that this transformation never increases the structural complexity of the transformed component. The transformation does not change the set of atomic components as well as the set of the hierarchical connectors. Hence, it preserves the operational semantics of the original model.

3.2.2 Connectors Flattening

This transformation flattens hierarchical connectors. It takes two connectors γ_i and γ_j with $\gamma_i \rightarrow \gamma_j$ (recall that the dominance relation $\rightarrow$ is given in Definition 2.2.5 in Chapter 2) and produces an equivalent connector.
We show in Figure 3.4 the composition of two connectors γ_i and γ_j. It consists in "glueing" them together on the exported port p_j. For the composite connector, the update functions

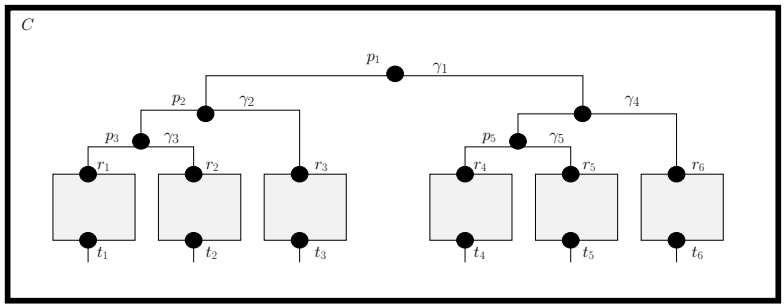


Figure 3.3: Component flattening for example in Figure 3.1.

are respectively, the bottom-up composition of the upward update functions, and the top-down composition of the downward update functions. This implements a general two-phase protocol for executing hierarchical connectors. First, data is synthesized in a bottom up fashion by executing upward update functions, as long as guards are true. Second, data is propagated downwards through downward update functions, from the top to the support set of the connector.

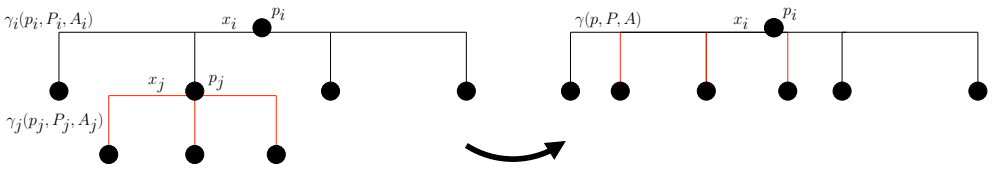


Figure 3.4: Connector glueing.

Definition 3.2.2 (Connector glueing.) *Given connectors* $\gamma_i = (p_i[x_i], P_i, A_i)$ *and* $\gamma_j = (p_j[x_j], P_j, A_j)$ *such that* $\gamma_i \rightarrow \gamma_j(p_j \in P_i)$ *we define the composition* $\gamma_i[p_j \mapsto \gamma_j]$ *as a connector* $\gamma = (p, P, A)$ *where*

- $p = p_i$,
- $P = P_j \cup P_i \setminus \{p_j\}$,
- $A = A_1 \cup A_2$, *where,*
  - $A_1 = \{a_i \mid a_i \in A_i, p_j \notin a_i\}$,
  - $A_2 = \{a_i \setminus \{p_j\} \cup a_j \mid a_i \in A_i, p_j \in a_i, a_j \in A_j\}$

  If $a \in A_1$ *the guards and data transfer are inherited as such from* γ_i. *In the second case, if* $a \in A_2$ *the guard and the transfer are defined as follows:*
  - $G_a = G_{a_j} \wedge G_{a_i}[U_{a_j}/x_j]$,
  - $U_a = x_i := U_{a_i}[U_{a_j}/x_j]$,

– $D_a = (\cup_{p_k \in a_j} x_k := D_{a_j,x_k}[D_{a_i,x_i}/x_i]) \cup (\cup_{p_k \in a_i \setminus \{p_j\}} x_k := D_{a_i,x_k})$.

Intuitively, by composition, two linked connectors are glued together into a single connector. Their guards, respectively the upward and downward transfer functions are composed. Consequently, any port valuation obtained by the successive application of the upward (resp. downward) transfer predicates of the two connectors is equally obtained by the application of the upward (resp. downward) transfer predicate of the composed connector.

Let us introduce some notations. Let $\Gamma = \{\gamma_i = (p_i[x], P_i, A_i)\}_{i \in I}$ a set of connectors, and let $P = \{\{p_i\} \cup P_i\}_{i \in I}$ the set of all used ports. We call a port $p_j \in P$ *transient* in Γ if it is both exported by some connector γ_j from Γ and used by another connector γ_i from Γ. Obviously, transient ports can be eliminated through connector glueing.
For a transient port p_j exported by a connector γ_j, we will use the notation $\Gamma[p_j \mapsto \gamma_j]$ to denote the new set of connectors obtained by replacing thoroughly p_j by its exporting connector γ_j, formally: $\Gamma[p_j \mapsto \gamma_j] = \{\gamma \mid \gamma \in \Gamma, p_j \notin \gamma.ports, \gamma \neq \gamma_j\} \cup \{\gamma[p_j \mapsto \gamma_j] \mid \gamma \in \Gamma, p_j \in \gamma.ports\}$. That is, all connectors (except γ_j) without p_j in their support set are kept unchanged, while the others are transformed according to definition 3.2.2.

Definition 3.2.3 (Connector flattening.) *Connector flattening is defined by the following function:*

$$\mathcal{F}_\gamma(\Gamma) = \begin{cases} \Gamma & \text{if } \Gamma \text{ is a set of flat connectors} \\ \mathcal{F}_\gamma(\Gamma[p_j \mapsto \gamma_j]) & \text{if } \Gamma \text{ is not a set of flat connectors, } p_j \text{ is a transient port of } \Gamma \end{cases}$$

Proposition 2 *Connector flattening is well-defined i.e., $\mathcal{F}_\gamma$ produces a unique result for any set of connectors, and terminates in a finite number of steps.*

Proof Regarding unicity of the result, if p_j and p_k are two transient ports of Γ defined respectively by connectors γ_j and γ_k, then flattening in any order gives the same result, formally $\Gamma[p_j \mapsto \gamma_j][p_k \mapsto \gamma_k] = \Gamma[p_k \mapsto \gamma_k][p_j \mapsto \gamma_j]$.

To show this result it is sufficient to show that any connector γ of Γ, different from γ_j and γ_k gets transformed in the same way, independently of the order of application of the two transformations. This can be shown, case by case, depending on the occurrence of ports p_j and p_k in the supports of γ, γ_j and γ_k following definition 3.2.2.

Regarding termination, flattening of connectors is applicable as long as there are transient ports. Moreover, it can be shown that, every flattening step reduces the number of transient ports by one - the one that is replaced by its definition. Hence, flattening eventually terminates when no more transient ports exist, that is, Γ is a set of flat connectors.

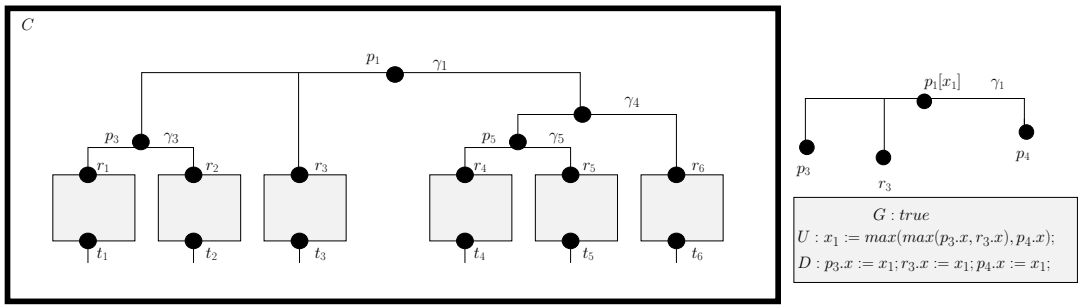


Figure 3.5: Connector glueing for example in Figure 3.3.

By application of the transformation $\gamma_1[p_2 \mapsto \gamma_2]$ to Example 2 in Figure 3.3, we obtain the new composite component given in Figure Figure 3.5. If we apply successively, $\gamma_1[p_3 \mapsto \gamma_3], \gamma_1[p_4 \mapsto \gamma_4], \gamma_1[p_5 \mapsto \gamma_5]$ we obtain the new composite component given in Figure 3.6.

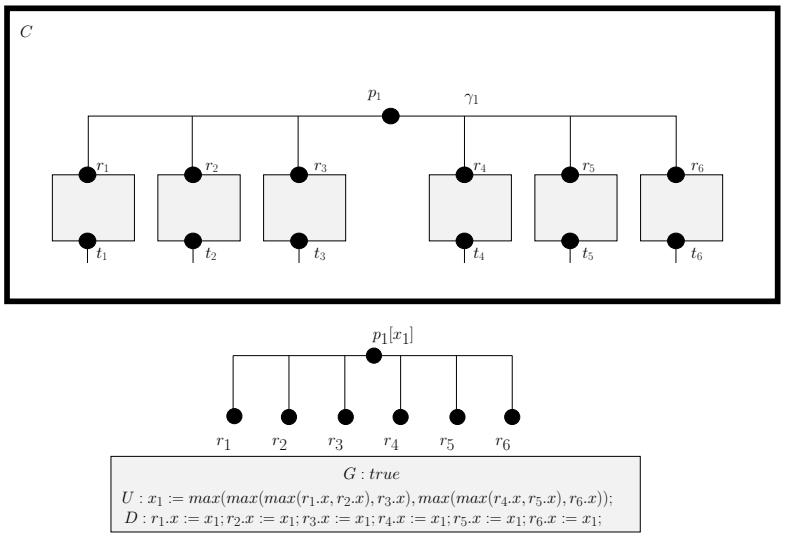


Figure 3.6: Connector flattening for example in Figure 3.3.

In a similar way to component flattening, this second transformation does not increase the structural complexity of the transformed components. The set of atomic components is preserved as such, whereas, the overall set of connectors is decreasing. However, the remaining connectors have an increased computational complexity, because they integrate the guards and the data transfer of the eliminated ones. The operational semantics is also

preserved. The effect of the eliminated connectors is "in-lined" in the remaining according to definition 2.2.7.

3.2.3 Components Composition

We present the third transformation which allows to obtain a single atomic component from a set of atomic components and a set of flat connectors. This transformation defines the composition of behaviors.
Intuitively, as shown in Figure 3.7, the composition operation consists in "glueing" together transitions from atomic components that are synchronized through the interaction of some connector (interaction p_1p_2 for this example). Guards of synchronized transitions are obtained by conjuncting individual guards and the guard of the connector. Similarly, actions of synchronized transitions are obtained as the sequential composition of the upward update function followed by the downward update function of the connector, followed by the actions of the components in an arbitrary order.

Definition 3.2.4 (Component composition.) *Consider a component* $C = (\{B_i\}_{i\in I}, \Gamma, P)$ *such that* $\forall i \in I$ B_i *is an atomic component and* Γ *is a set of flat connectors. We define the composition* $\Gamma(\{B_i\}_{i\in I})$ *as component* $B = (L, P, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$ *defined as follows:*

- *the set of control states* $L = \cup_{i\in I} B_i.L$,
- *the set of ports* $P = \cup_{\gamma\in\Gamma}\{\gamma.p\}$,
- *the set of variables* $X = (\cup_{i\in I} B_i.X) \cup (\cup_{\gamma\in\Gamma}\gamma.p.x)$,
- *each transition in* T *corresponds to a set of interacting transitions* $\{\tau_1, \ldots, \tau_k\} \subseteq \cup_{i\in I} T_i$ *such that* $\cup_{i=1}^{k} \tau_i.p = a$ $(\gamma \in \Gamma, a \in \gamma.A)$. *We define the transition* $\tau = (l, \gamma.p, l')$ *where,*
  - $l =^\bullet \tau_1 \cup \ldots \cup^\bullet \tau_k$,
  - $l' = \tau_1^\bullet \cup \ldots \cup \tau_k^\bullet$,
  - *the guard* $g_\tau = \wedge_{i=1}^{k} g_{\tau_i} \wedge a.G$,
  - *the action* $X := f_\tau(X)$ *with* $f_\tau = a.U; a.D; (\cup_{i=1}^{k} f_{\tau_i})$.

Figure 3.8 shows the Petri net obtained by composition of the atomic components of Figure 3.6 through the interaction $r_1r_2r_3r_4r_5r_6$.

In contrast to previous transformations, component composition may lead to an exponential blowup of the number of transitions in the resulting Petri net. This situation may happen if the same interaction can be realized by combining different transitions from each one of the involved components. For instance, the interaction p_1p_2 can give rise to four transitions in the resulting Petri net if there are two transitions labeled by p_1 and p_2 in the synchronizing components. Nevertheless, in practice exponential explosion seldom occurs, as in atomic components each port labels at most one transition (as in the examples shown hereafter). In this case, the resulting Petri net has as many transitions as connectors in Γ.

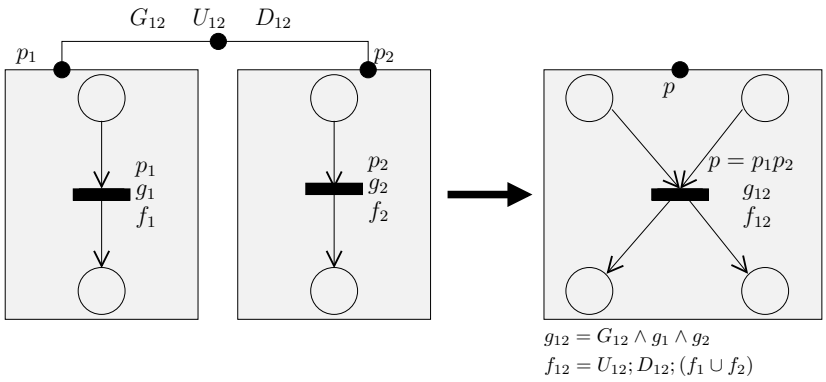
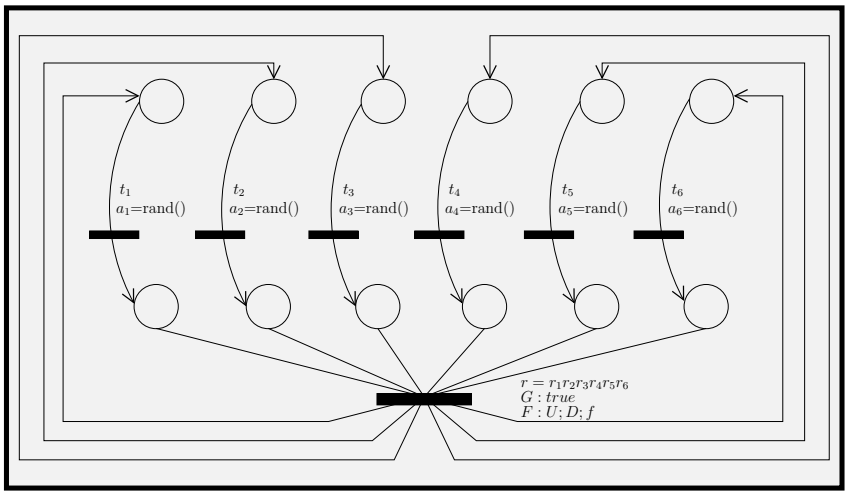


Figure 3.7: Component composition.



$U : x_1 := max(max(max(a_1, a_2), a_3), max(max(a_4, a_5), a_6));$
$D : a_1 := x_1; a_2 := x_1; a_3 := x_1; a_4 := x_1; a_5 := x_1; a_6 := x_1;$
$f : print(a_1) \cup print(a_2) \cup print(a_3) \cup print(a_4) \cup print(a_5) \cup print(a_6)$

Figure 3.8: Component composition for example in Figure 3.6.

3.3 Efficient Sequential Implementation

By exhaustive application of these transformations, an atomic component can be obtained. From the latter, the *code-generator* can generate efficient standalone C++ code, which can be run directly without the Engine. In particular, all the remaining non-determinism in the final atomic component is eliminated at code generation by applying an implicit priority between transitions.
It should be noted that the transformations also can be applied independently, to obtain models that respond to a particular user needs. For example, one may decide to eliminate only partially the hierarchy of components, or to compose only some components.

3.4 Experimental Validation

These transformations have been implemented in the BIP2BIP tool which is currently integrated in the BIP toolset [BIP]. A detailed description of the BIP2BIP tool is given in Chapter 5.
For two examples, we compare the execution times of BIP programs before and after flattening. These examples show that it is possible to generate efficient standalone C++ code from component-based descriptions in BIP.

3.4.1 MPEG Video Encoder

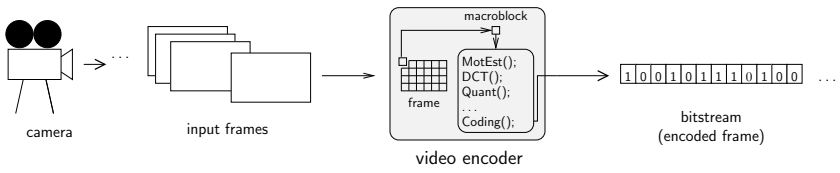


Figure 3.9: MPEG4 encoder structure.

In the framework of an industrial project, we have componentized in BIP an MPEG4 encoder written in C by an industrial partner. The aim of this work was to evaluate gains in scheduling and quality control of the componentized program. The results were quite positive regarding quality control [CFLS05a, CFLS05b, CFSS07] but the componentized program was almost two times slower than the handwritten C program. We have used BIP2BIP to generate automatically standalone C++ code from the BIP program as explained below (see Figure 3.10).

The BIP program consists of 11 atomic components, and 14 connectors. It uses the data and the functions of the initial handwritten C program. It is composed of two atomic

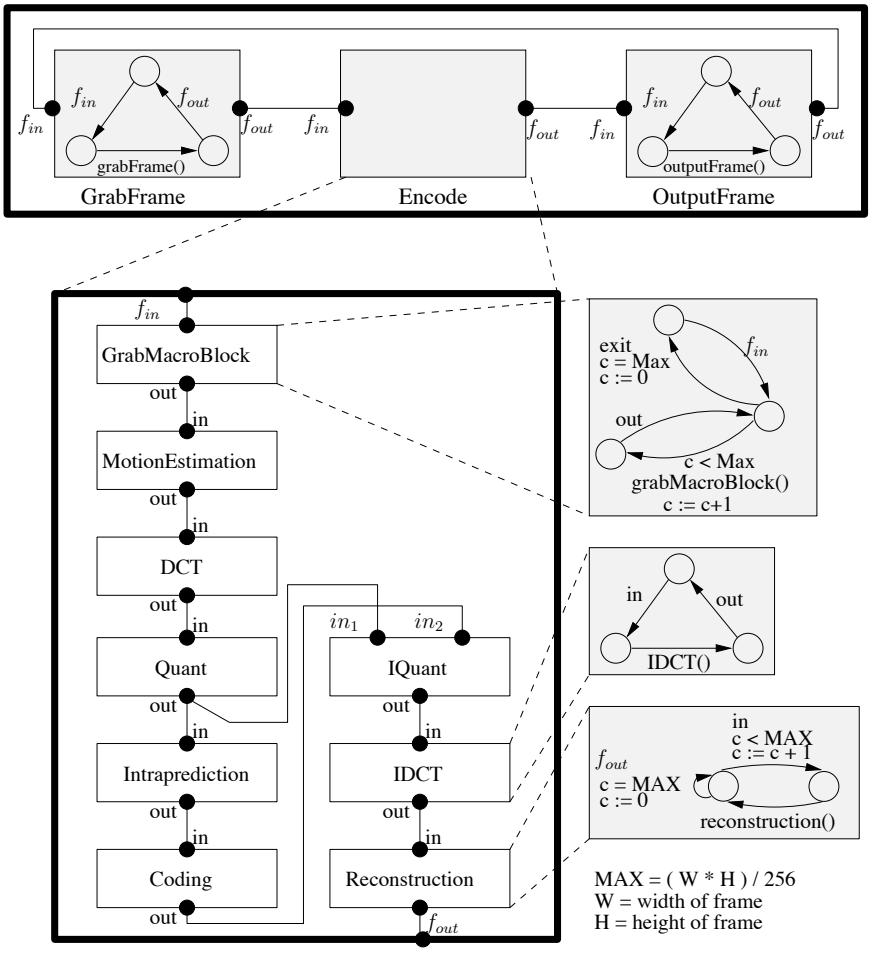


Figure 3.10: Encode component structure.

components and one composite component. The atomic component **GrabFrame** gets a frame and produces macroblocks (each frame is split into N macroblocks of 256 pixels). The atomic component **OutputFrame** produces an encoded frame. The composite component **Encode** consists of 9 atomic components and the corresponding connectors. It encodes macroblocks produced by the component **GrabFrame**.

Figure 3.11 shows execution times for the initial handwritten C code, for the BIP program and the corresponding standalone C++ code generated automatically by using the presented technique. Notice that the automatically generated C++ code and the handwritten C code have almost the same execution times. The advantages from the componentization of the handwritten code are multiple. The BIP program has been rescheduled as shown in [CFLS05a] so as to meet given timing requirements.
Table 3.1 gives the size of the handwritten C code, the BIP model, as well of the generated C++ code from the BIP model $C^{(1)}$ and the generated C++ code from the BIP model after flattening $C^{(2)}$. The time taken by the BIP2BIP tool to generate automatically $C^{(2)}$ is less than 1sec.

	Handwritten	BIP	$C^{(1)}$	$C^{(2)}$
loc	600	350	1800	800

Table 3.1: Code size in lines-of-code (loc) for MPEG4 encoder.

3.4.2 Concurrent Sorting

This example is inspired from a network sorting algorithm [AKS83]. We consider 2^n atomic components, each of them containing an array of N values. We want to sort all the values, so that the elements of the first component are smaller than those of the second component and so on. We solve the problem by using incremental hierarchical composition of components with particular connectors.

In Figure 3.12, we give a model for sorting the elements of 4 atomic components. The components C_1 and C_2 are identical. The pair (B_1, B_2) is composed by using two connectors γ_1 and γ_2 to form the composite component C_1. Each atomic component computes the minimum and the maximum of the values in its array. These values are then exported on port p. The connector γ_1 is used to compare the maximum value of B_1 with the minimum value of B_2, and to permute them if the maximum is bigger than the minimum value.

When the maximum value of B_1 is smaller than the minimum value of B_2, that is the components are correctly sorted, then the second connector γ_2 is triggered. It is used to export the minimum value of B_1 and the maximum value of B_2 to the upper level. At this level the same principle is applied to sort the values of the composite components C_1 and C_2. This pattern can be repeated to obtain arbitrary higher hierarchies (see Figure 3.13).

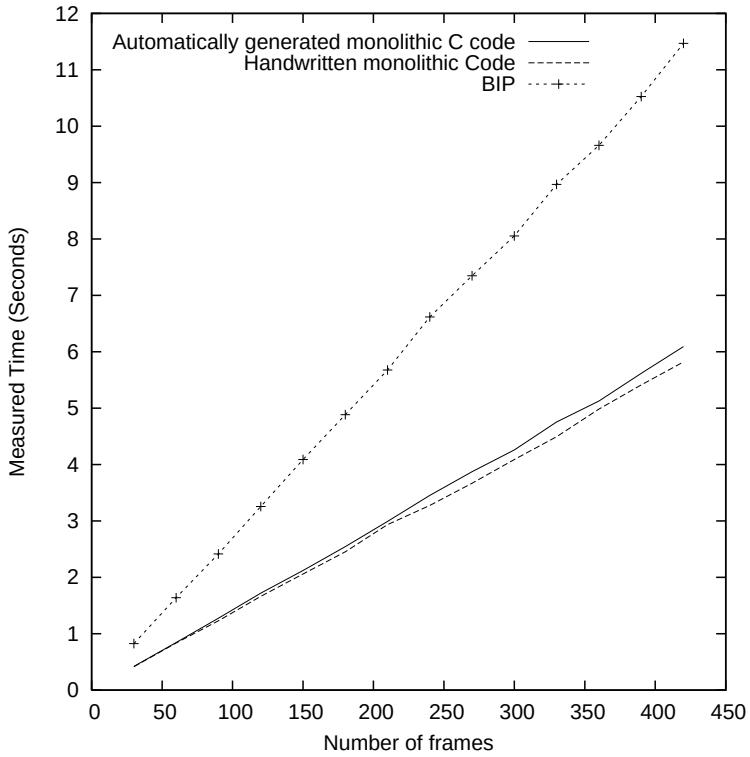


Figure 3.11: Execution time for the MPEG4 encoder.

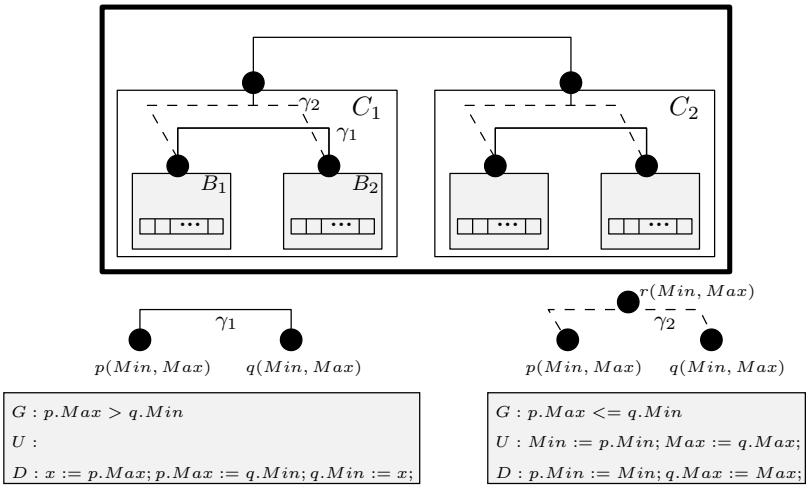


Figure 3.12: Concurrent sorting $n = 2$.

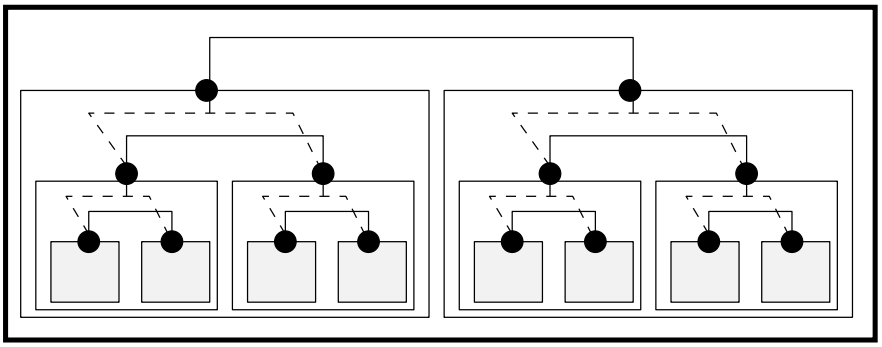


Figure 3.13: Concurrent sorting $n = 4$.

n		BIP	$C^{(1)}$	$C^{(2)}$
2	loc	112	360	400
3	loc	120	400	620
4	loc	128	440	1100
5	loc	136	480	1850
6	loc	144	520	2850

Table 3.2: Code size in loc for concurrent sorting.

Figure 3.14 shows the execution times for the hierarchically structured BIP program and for the corresponding standalone C++ code generated automatically by using the presented technique. Notice the exponentially increasing difference between the execution time of the component-based BIP program and the corresponding C++ code. In particular, component flattening and connector flattening do not provide much better performance, because the hierarchical structure is actually exploited by the BIP Engine to compute enabled interactions in an efficient manner. However, these transformations are mandatory for applying the static composition.
Notice that the overhead is due to many reasons when using the BIP Engine. First, each atomic components sends to the Engine its current state and the list of enabled ports. Second, the Engine enumerates on the list of interactions in the model, identifies *all* enabled ones based on the current state of the atomic components, then among them it selects one for execution and, finally, notifies atoms to take the corresponding transition. This overhead is partially eliminated in the standalone C++ code generated automatically. Indeed, the call function between components and the Engine is omitted. The time needed to select an enabled interaction is drastically reduced. Moreover, control and code optimization such as guard combination, removal of unnecessary assignments, etc., are applied

Table 3.2 shows the size in lines of code of the BIP model, as well of the generated C++ generated from the BIP model $C^{(1)}$ and the generated C++ code from the BIP model after flattening $C^{(2)}$, for 4, 8, 16, 32 and 64 atomic components. The size of the BIP model changes only linearly with n. However, we notice that for this example, the size of the generated C++ code from the BIP model is much smaller than the generated C++ code from the BIP model after flattening. This is due to the use of component types and component types instantiation. In particular, for this example, the initial BIP model contains just one component type instantiated, respectively $4, 8, 16, 32, 64$ times for $n = 2, 3, 4, 5, 6$. However, the BIP model after flattening, contains one component types with one instance each. The size of the generated code is directly dependent on the number of component types and not on the number of component types instance.

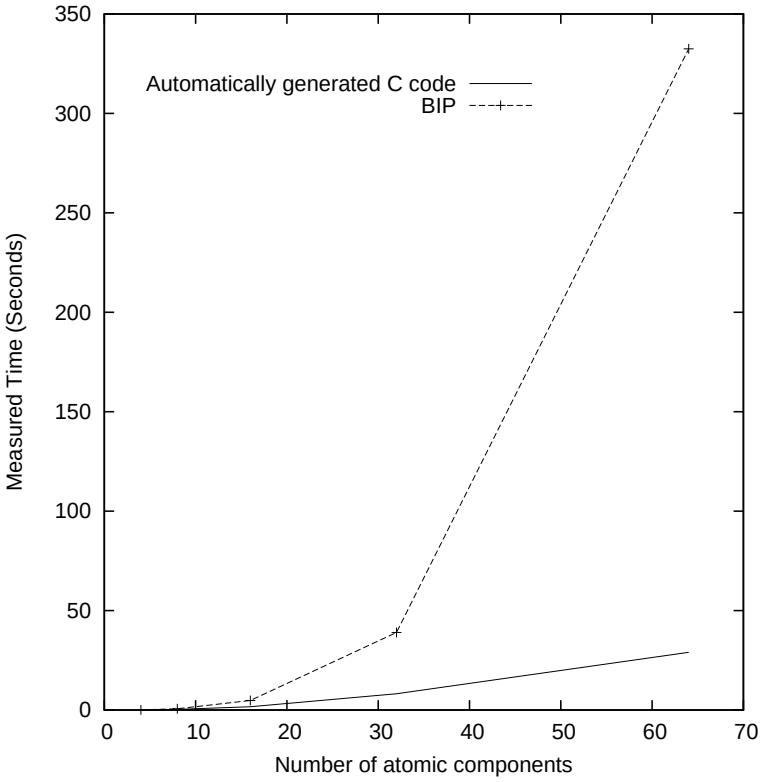


Figure 3.14: Execution time for concurrent sorting.

3.5 Summary

In this chapter we presented a technique for generating efficient centralized implementation running on a single-core platforms. The technique is based on source-to-source transformations. We defined three transformations: Component flattening, Connector flattening, Component composition. The aim of these transformation is to transform a composite component into a single atomic component. From the latter an efficient code can be generated.

Moreover, we shown that it is possible to reconcile component-based incremental design and efficient code generation by applying a paradigm based on the combined use of:

1. a high level modelling notation based on well-defined operational semantics and supporting powerful mechanisms for expressing structured coordination between components;
2. semantics-preserving source-to-source transformations that progressively transform architectural constraints between components into internal computation of product components.

This paradigm opens the way to the synthesis of efficient monolithic software which is correct-by-construction by using the design methodology supported by BIP.

We are now going to present a method for generating efficient distributed implementations from BIP models. We will also show applications of our method on non trivial example described in the BIP language.

CHAPTER 4

Transformation for Generating Distributed Implementations

Contents

4.1 Problem Statement . . . 68
4.2 Original BIP Model . . . 69
4.3 Conflicting Interactions . . . 69
4.4 Proposed Solution . . . 73
4.5 The 3-Layer Architecture . . . 75
4.6 Transformations . . . 76
4.6.1 Transformation of Atomic Components . . . 76
4.6.2 The Interaction Protocol . . . 77
4.6.3 The Conflict Resolution Protocol . . . 81
4.6.4 Cross-Layer Interactions . . . 83
4.7 Correctness . . . 86
4.7.1 Compliance with Send/Receive Models . . . 86
4.7.2 Observational Equivalence between Original and Transformed BIP Models . . . 87
4.7.3 Interoperability of Conflict Resolution Protocol . . . 90
4.8 Transformation from Send/Receive BIP into C++ . . . 91
4.9 Component Composition . . . 93
4.10 Experimental Validation . . . 93
4.10.1 Diffusing Computation . . . 94
4.10.2 Utopar Transportation System . . . 99
4.10.3 Bitonic Sorting . . . 102
4.11 Summary . . . 105

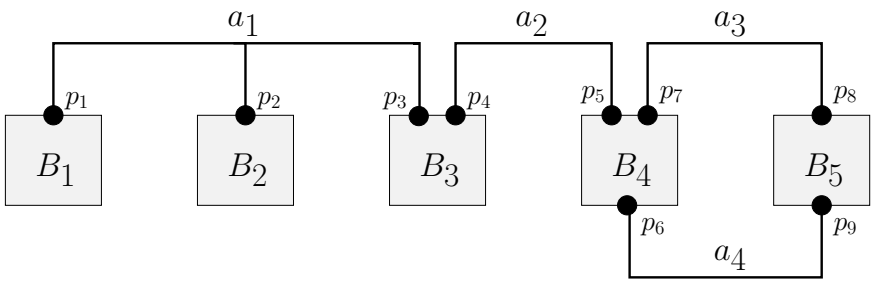


Figure 4.1: A simple high-level BIP model.

In the previous chapter, we present a method to generate an efficient centralized implementation from a high-level BIP model. Centralized implementations can be useful when the target platform consists of a single-processor. However, nowadays, most applications themselves they can be geographically distributed, on the other hand, most of target platforms are multi-processor boards. Thus, a distributed and parallel implementation should be derived from the high-level BIP model. In this chapter, we propose a methodology for producing automatically efficient and correct-by-construction distributed implementations from a high-level BIP models.

4.1 Problem Statement

Deriving from a high-level model a correct and efficient distributed implementation, that allows parallelism between components as well as parallel execution between interactions, is a challenging problem. As adding implementation details involves many subtleties (e.g., inherently concurrent, non-deterministic, and non-atomic structure of distributed systems) that can potentially introduce errors to the resulting system.

In order to understand the subtleties of transformation from high-level BIP model to distributed implementations, consider the BIP model in Figure 4.1. In this model, atomic components $B_1 \cdots B_5$ synchronize through four rendezvous interactions $a_1 \cdots a_4$. In sequential implementation, interactions are executed atomically by a centralized Engine. On the contrary, introducing concurrency and distribution (and possibly multiple Engines) to this model requires dealing with more complex issues:

– (*Partial observability*) Suppose interaction a_1 (and, hence, components $B_1 \cdots B_3$) is being executed. If component B_3 completes its computation before B_1 and B_2, and, ports p_4, p_5 are enabled, then interaction a_2 is enabled. In such a case, distributed Engines must be designed so that concurrent execution of interactions does not intro-

duce behaviors that were not allowed by the high-level model. We address the issue of partial observability by breaking the atomicity of execution of interactions, so that a component can execute unobservable actions once a corresponding interaction is being executed [BBBS08].
- (*Resolving conflicts*) Suppose interactions a_1 and a_2 are enabled simultaneously. Since these interactions share component B_3, they cannot be executed concurrently. We call such interactions *conflicting*. Obviously, distributed Engines must ensure that conflicting interactions are mutually exclusive.
- (*Performance*) On top of correctness issues, a real challenge is to ensure that a transformation does not add considerable overhead to the implementation. After all, one crucial goal of developing distributed and parallel systems is to exploit their computing power.

4.2 Original BIP Model

In this chapter, we consider that the original BIP component consists only of atomic component and flat connectors. Moreover, each connector defines one and only one interaction (see Figure 4.1). Indeed, these assumptions does not impose any restrictions on the original model, since we can apply the first two transformation *Component Flattening* and *Connector Flattening* described in Section 3.2. Hence, we obtain a model which meets these assumptions. Under these assumptions, a composite component C can be denoted as $C = \gamma(B_1, B_2, \ldots, B_n)$ where B_1, B_2, $\ldots$, B_n is a set of atomic components and γ is a set of interactions.

Recall that an interaction is defined as follows:

Definition 4.2.1 (Interaction.) *An interaction a is given by a tuple (P, G, F) such that:*
- *P is a set of ports $P = \{p_i[x_i]\}_{i \in I}$,*
- *G is a guard (a predicate on the variables x_i),*
- *F is an update function defined on the variables x_i .*

Furthermore, let $B = (L, P, T, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ be an atomic component. Hereafter, we assume that $T \subseteq L \times P \times L$. That is, the behavior of an atomic component is described as labeled transition system. For any pair of control states $l, l' \in L$ and a port $p \in P$, we write $l \xrightarrow{p} l'$, iff $(l, p, l') \in T$. When the communication port is irrelevant, we simply write $l \rightarrow l'$. Similarly, $l \xrightarrow{p}$ means that there exists $l' \in L$ such that $l \xrightarrow{p} l'$.

4.3 Conflicting Interactions

In order to understand what is a conflicting interaction, let us first recall how the execution of interactions and local code of components in a model are orchestrated by the

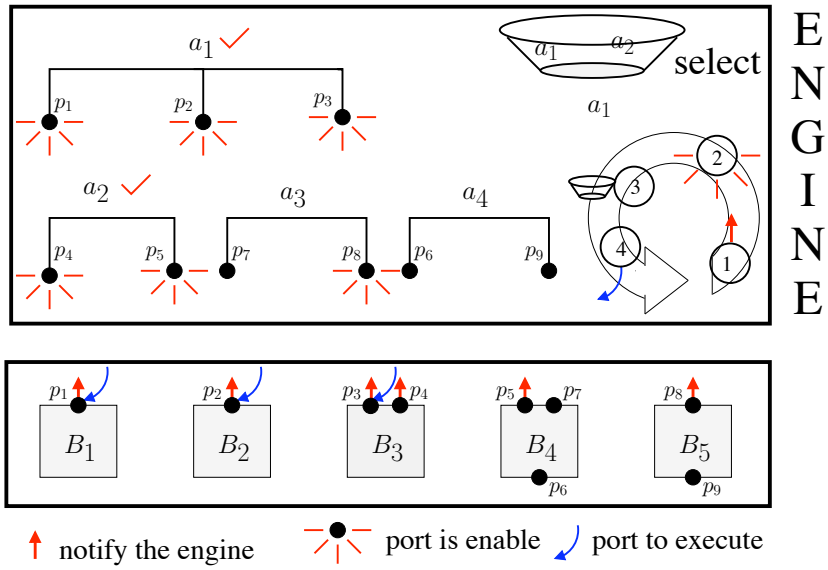


Figure 4.2: Centralized Engine.

centralized Engine (see Figure 4.2):

1. Each atomic component sends to the Engine the set of enabled ports based on the current state of the atomic components.
2. The Engine enumerates the list of interactions in the model, selects the enabled ones.
3. The Engine selects any one from the enabled ones after eliminating the ones with low priority.
4. Finally, the Engine notifies the involved atomic components the transition to take and wait for completion.

The operational semantics of the BIP framework is handled by the centralized Engine. Nonetheless, introducing concurrency and distribution between interactions by adding multiple Engines requires dealing with more complex issues. For instance, the system must respect the global state semantics although it works in a distributed setting where components do not have a global view of the system. Moreover, suppose that interaction a_1 and a_2, in the example in Figure 4.1 are enabled simultaneously. Since these interactions share the same component B_3, they cannot be executed concurrently such as the example in Figure 4.3. We call such interactions conflicting. Obviously, distributed Engines must ensure that conflicting interactions are mutually exclusive.

Definition 4.3.1 (Conflict interaction.) *Let $\gamma(B_1, \ldots, B_n)$ be a BIP component. We say that two interactions $a_1, a_2 \in \gamma$ are* conflicting *iff:*

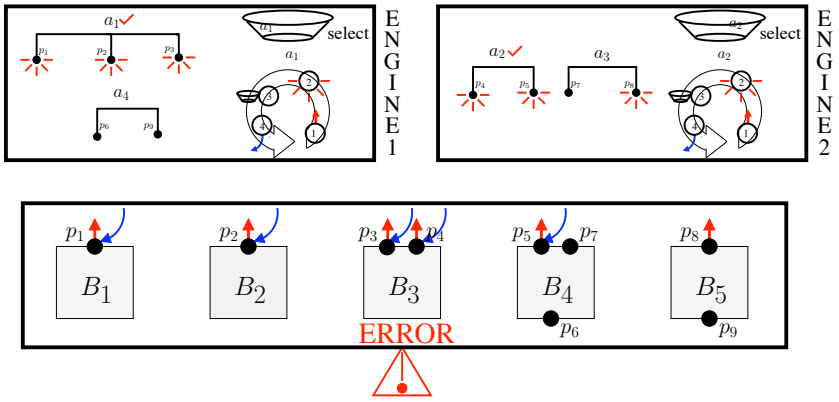


Figure 4.3: Violating of BIP semantics when badly distributed interactions.

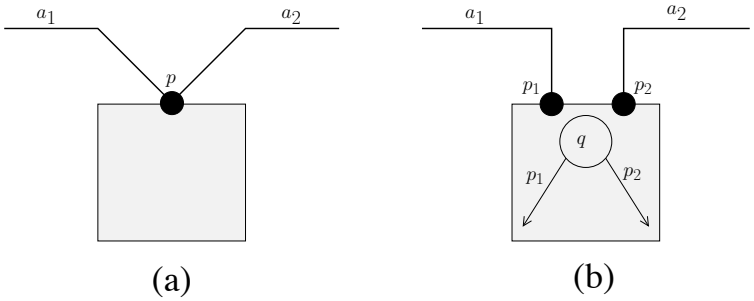


Figure 4.4: Conflict interactions.

- *either, they share a common port p; i.e.,* $p \in a_1 \cap a_2$ *(see Figure 4.4-a),*
- *or, there exist an atomic component* $B_i = (L_i, P_i, T_i, \{g_{\tau_i}\}_{\tau_i \in T_i}, \{f_{\tau_i}\}_{\tau_i \in T_i})$*, a control state* $l \in L_i$*, and two ports* $p_1, p_2 \in P_i$ *such that (1)* $p_1 \in a_1$*, (2)* $p_2 \in a_2$*, and (3)* $l \xrightarrow{p_1} \wedge\, l \xrightarrow{p_2}$ *(see Figure 4.4-b).*

Therefore, introducing concurrency by adding multiple Engines requires to resolve conflict between conflicting interactions.

The first straightforward solution consists of generation distributed Engines which is conflict-free by construction. This is done by grouping interactions according to the transitive closure of the conflict relation in the same Engine. In this case, we do not need communications in order to safely execute interactions of the high-level model.

However, this solution has a drawback, because grouping conflicting interactions according to the transitive closure reduces drastically parallelism between interactions. In

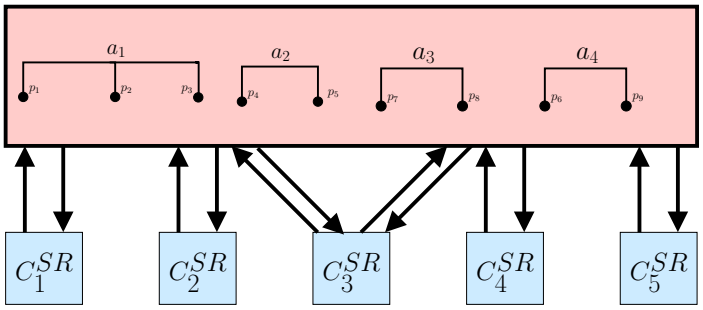


Figure 4.5: Drawbacks of conflict-free solution.

other words, two interactions which are not in direct conflict cannot run in parallel. For instance, let us consider the example in the Figure 4.1. Assuming that a_1 is conflict only with a_2, and, interaction a_2, a_3, a_4 are pairwise conflict. This situation leads to create only one centralized Engine handling all the interactions (see Figure 4.5), and then, it is not possible to run the interaction a_1 and a_3 in parallel, despite that it is possible without violating the operational semantics of BIP.

For this reason, we need a solution which solve conflict dynamically by using some protocols. Indeed, resolving conflicts leads us to solving the *committee coordination problem* [CM88], where a set of professors organize themselves in different committees and two committees that have a professor in common cannot meet simultaneously. The original distributed solution to the committee coordination problem assigns one *manager* to each interaction [CM88]. Conflicts between interactions are resolved by reducing the problem to the dining or drinking philosophers problems [CM84], where each manager is mapped onto a philosopher. Bagrodia [Bag87] proposes an algorithm where message counts are used to solve synchronization and exclusion is ensured by using a circulating token. In a follow-up paper [Bag89], Bagrodia modifies the solution in [Bag87] by combining the use of message counts to ensure synchronization and reducing the conflict resolution problem to dining or drinking philosophers problems. Also, Perez et al [PCT04] propose an approach, Alpha-Core Protocol, that essentially implements the same idea using a lock-based synchronization mechanism. For instance, if we consider the algorithm proposed by Perez et al, this leads to embedding the protocol into the high-level BIP model as it is presented in the Figure 4.6. Notice that, using such methodology the modification of protocol requires to do a modification in the different layers.

Thus, several distributed algorithms exist in the literature for conflict resolution, moreover the performance of each of them depends on the type of the application. For this reason, we need to design our framework, so that it provides appropriate interfaces with minimal restrictions.

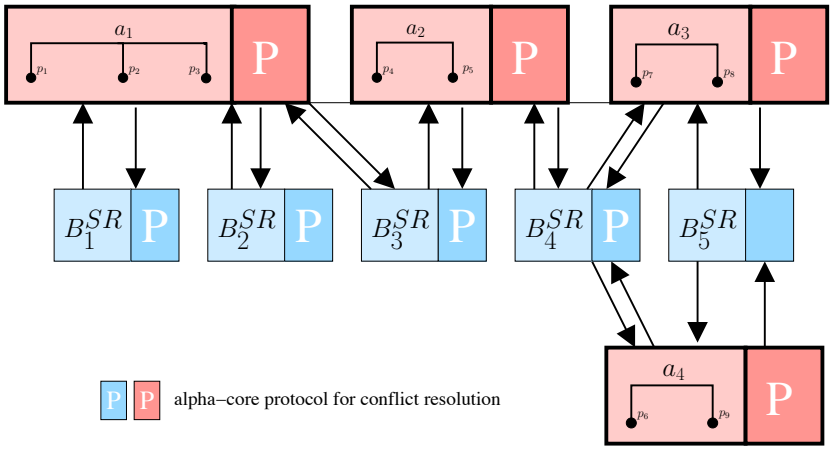


Figure 4.6: Resolving conflict interactions using alpha-core protocol.

4.4 Proposed Solution

With this motivation we propose a generic framework for transforming high-level BIP models into a distributed implementation that allow parallelism between components as well as parallel execution of non-conflicting interactions by embedding a solution to the committee coordination problem. To the best of our knowledge, this is the first instance of such a transformation (the related work mentioned above only focus on impossibility results, abstract algorithms, and in one instance [Bag89] simulation of an algorithm). Our method utilizes the following sequence of transformations preserving *observational equivalence*:

1. First, we transform the given BIP model into another BIP model that (1) operates in partial-state semantics, and (2) expresses multi-party interactions in terms of asynchronous message passing (Send/Receive primitives). Moreover, the target BIP model is structured in three layers:
   (a) The *Atomic Components Layer* consists of a transformation of behavioral components in the original model.
   (b) The *Interaction Protocol Layer* detects enabledness of interactions of the original model and executes them after resolving conflicts either locally or by the help of the third layer. This layer consists of a set of components, each hosting a user-defined subset of interactions from the original BIP model.
   (c) The *Conflict Resolution Protocol Layer* resolves conflicts requested by the Interaction Protocol. The Conflict Resolution Protocol implements a committee coordination algorithm and our design allows employing any such algorithm.

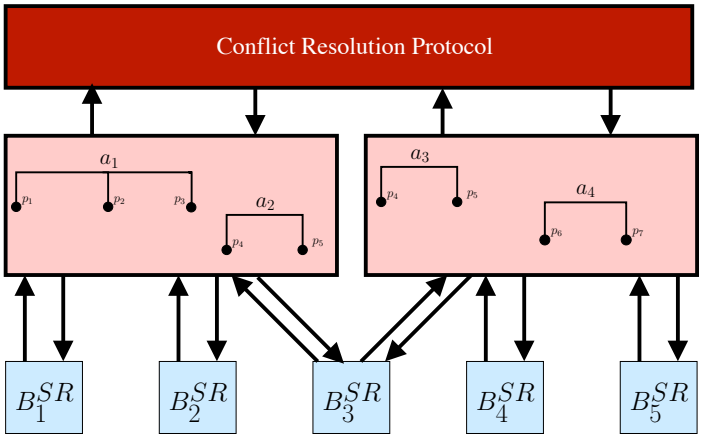


Figure 4.7: General overview of the 3-layer architecture.

We, in particular, consider three committee coordination algorithms:

i. a fully centralized algorithm,

ii. a token-based distributed algorithm,

iii. an algorithm based on reduction to distributed dining philosophers.

2. Then, we transform the 3-layer BIP model into C++ code that employs TCP sockets or MPI for communications.

The BIP composite component generated from the first phase is called *Send/Receive* BIP, and it is defined as the following:

Definition 4.4.1 (Send/Receive BIP.) *We say that* $B^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR})$ *is a* Send/Receive *BIP composite component iff we can partition the set of ports in* B^{SR} *into three sets* P_s, P_r, P_u *that are respectively the set of* send-ports, receive-ports, *and* unary interaction ports, *such that:*

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

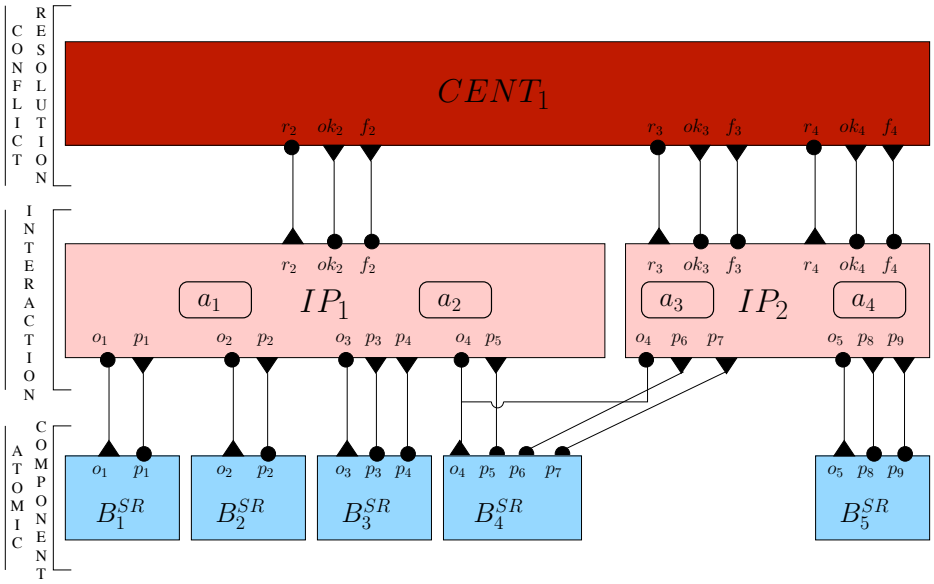


Figure 4.8: 3-layer Send/Receive BIP model of Figure 4.1.

Notice that the second condition requires that only one component can receive a "message" sent by another component. The last condition ensures that every Send/Receive interaction can take place as soon as the sender is enabled, i.e., the sender can send the message immediately.

4.5 The 3-Layer Architecture

We design our target BIP model based on the three tasks identified above, where we incorporate one layer for each task. We use the high-level BIP model in Figure 4.1 as a running example throughout this section to describe the concepts of our transformation. We assume that interaction a_1 is in conflict with only interaction a_2, and, interactions a_2, a_3, and a_4 are in pairwise conflict. Our 3-layer architecture consists of the following layers.

Atomic Components Layer. Atomic components in the high-level model are placed in this layer with the following additional ports per component. The send-port o that shares the list of enabled ports in the component with the upper layer. Also, for each port p in the original component, we include a receive-port p through which the component is notified to execute the transition labeled by p once the upper layers resolve conflicts and decide on which components can execute on what port. The bottom layer in Figure 4.8 includes components illustrated in Figure 4.1.

Interaction Protocol Layer. This layer consists of a set of components each hosting a set of interactions in the high-level model. Conflicts between interactions included in the same component are resolved by that component locally. For instance, interactions a_1 and a_2 (resp. a_3 and a_4) of Figure 4.1 are grouped into component IP_1 (resp. component IP_2) in Figure 4.8. Thus, the conflict between a_1 and a_2 (resp. a_3 and a_4) is handled locally in IP_1 (resp. IP_2). On the contrary, the conflicts between a_2 and either a_3 or a_4 must be resolved using an external algorithm that solves the committee coordination problem. Such an algorithm forms the top layer of our model. The Interaction Protocol also evaluates the guard of each interaction and executes the code associated with an interaction that is selected locally or by the upper layer. The interface between this layer and the component layer provides ports for receiving enabled ports from each component (i.e., port o) and notifying the components on permitted port for execution.

Conflict Resolution Protocol Layer. This layer accommodates an algorithm that solves the committee coordination problem. For instance, the external conflicts between interactions a_2 and a_3, and, interactions a_2 and a_4 are resolved by the central component RP_1 in Figure 4.8. We emphasize that the structure of components in this layer solely depends upon the augmented conflict resolution algorithm. Incorporating a centralized algorithm results in one component $CENT_1$ as illustrated in Figure 4.8. Other algorithms (as will be discussed in 4.6.3), such as ones that use a circulating token [Bag87] or dining philosophers [CM84, Bag89] result in different structures. The interface between this layer and the Interaction Protocol involves ports for receiving request to reserve an interaction (labeled r) and responding by either success (labeled ok) or failure (labeled f).

4.6 Transformations

In this section, we describe our technique for transforming a BIP model into a 3-layer distributed BIP model in detail. Construction of the three layers are described in Subsections 4.6.1, 4.6.2, and 4.6.3 respectively. Finally, we describe cross-layer interactions in Subsection 4.6.4.

4.6.1 Transformation of Atomic Components

We now present how we transform an atomic component B from a given BIP model into a Send/Receive atomic component B^{SR} that is capable of communicating with the Interaction Protocol in the 3-layer model. As mentioned in Section 4.5, B^{SR} sends *offers* to the Interaction Protocol that are acknowledged by a *response*. An offer includes the set of enabled ports of B^{SR} at the current state through which the component is ready to interact. Enabled ports are specified by a set of Boolean variables. These variables are

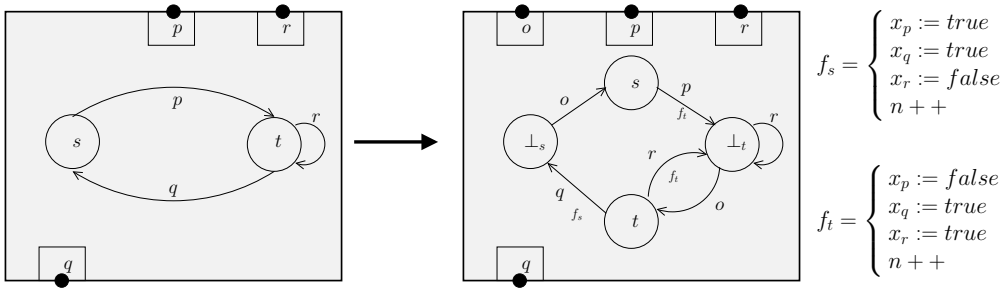


Figure 4.9: Transformation of atomic component.

modified by a *port update* function. The function evaluates each variable when reaching a new state. When the upper layers select an interaction involving B^{SR} for execution, B^{SR} is notified by a response sent on the port chosen. We also include a *participation number* variable n in B^{SR}, which counts the number of interactions B^{SR} has participated in.

Since each response triggers an internal computation, following [BBBS08], we split each state s into two states, namely, s itself and a *busy state* $\perp_s$. Intuitively, reaching $\perp_s$ marks the beginning of an unobservable internal computation. We are now ready to define the transformation from B into B^{SR}.

Definition 4.6.1 (Transformation of atomic component.) *Let* $B = (L, P, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ *be an atomic component. The corresponding Send/Receive atomic component is* $B^{SR} = (L^{SR}, P^{SR}, T^{SR}, X^{SR}, \{g^{SR}_\tau\}_{\tau \in T^{SR}}, \{f^{SR}_\tau\}_{\tau \in T})$ *such that:*

- $L^{SR} = L \cup L^{\perp}$, *where* $L^{\perp} = \{\perp_s \mid s \in L\}$.
- $X^{SR} = X \cup \{n\} \cup \{x_p\}_{p \in P}$, *where* x_p *it is a new boolean variable associated to the port* p, *and* n *is the number of interactions* B^{SR} *has participated in.*
- $P^{SR} = P \cup \{o\}$, *where the set of variables* X^{SR} *are associated to* offer *port* o.
- *For each transition* $\tau = (s, p, t) \in T$, *we include the following two transitions in* T^{SR}: $\tau_1 = (\perp_s, o, s)$ *and* $\tau_2 = (s, p, \perp_t)$. *The guards of the transition* τ_1 *and* τ_2 *are true. Moreover, the transition* τ_2 *triggers the functions* f_τ *followed by the* f_t. *Where the function* f_t *modifies* X^{SR} *as follows: it sets* x_p *to true if* $(t \xrightarrow{p}) \wedge (g_\tau = true)$, *it sets it to false otherwise, and increments* n.

Figure 4.9 illustrates transformation of the component into its corresponding Send/Receive component.

4.6.2 The Interaction Protocol

Given a high-level BIP model $B = \gamma(B_1 \cdots B_n)$, one parameter to our transformation is a partition of interactions $\gamma_1, \ldots, \gamma_m$. Partitioning of interactions is a means for the designer

to enforce load-balancing and improving the performance of the given model when running in a distributed fashion. It also determines whether or not a conflict between interactions can be resolved locally. We associate each class γ_j of interactions to an Interaction Protocol component IP_j that is responsible for (1) detecting enabledness by collecting *offers* from the Components Layer, (2) selecting a set of non-conflicting interactions (either locally or by the help of the Conflict Resolution Protocol), and (3) executing the selected interactions in γ_i by notifying the corresponding atomic components. For instance, in Figure 4.8, we have two classes: $\gamma_1 = \{a_1, a_2\}$ (hosted by component IP_1) and $\gamma_2 = \{a_3, a_4\}$ (hosted by component IP_2).

Since components of the Interaction Protocol deal with interactions of the original model, they need to be aware of conflicts in the original model as defined in Definition 4.3.1. We distinguish two types of conflicting interactions according to a given partition:

- *External:* two interactions are externally conflicting if they conflict and they belong to different classes of the partition. External conflicts are resolved by the Conflict Resolution Protocol. For instance, in Figure 4.8, interaction a_2 is in external conflict with interactions a_3 and a_4.
- *Internal:* two interactions are internally conflicting if they conflict, but they belong to the same class of the partition. Internal conflicts are resolved by the Interaction Protocol within the component that hosts them. For instance, in Figure 4.8, interaction a_1 is in internal conflict with interaction a_2. If component IP_1 chooses interaction a_1 over a_2, no further action is required. Note, however, that if IP_1 chooses a_2, then it has to request its reservation from $CENT_1$, as it is in conflict with a_3 and a_4 externally.

For each Interaction Protocol we create the corresponding atomic component. The behavior of an Interaction Protocol component IP_j handling a class γ_j of interactions is constructed as follows. We refer to Figure 4.10 as a concrete example for construction of the atomic component of IP_1 in Figure 4.8.

Definition 4.6.2 (Interaction protocol.) *Let $B = \gamma(B_1 \cdots B_n)$ be a composite component, and $\gamma_j = \{a_1 = (P_1, F_1, G_1), \ldots, a_k = (P_k, F_k, G_k)\}$ a set of interactions. The corresponding atomic component $IP_j = (L^{IP_j}, P^{IP_j}, T^{IP_j}, X^{IP_j}, \{g_\tau^{IP_j}\}_{\tau \in T^{IP_j}}, \{f_\tau^{IP_j}\}_{\tau \in T})$ handling γ_j is defined as follows:*

- ***Control states L^{IP_j}.*** *We include three types of places:*
  - *For each component B_i involved in interactions of γ_j, we include* waiting *and* received *places w_i and rcv_i, respectively. IP_j waits in a waiting place until it receives an offer from the corresponding component. When an offer from component B_i is received (along with the fresh values of the Boolean variables associated to the ports of the sender), IP_j moves from w_i to rcv_i. In Figure 4.10, since components $B_1 \cdots B_4$ are involved in interactions hosted by IP_1 (i.e., a_1 and a_2), we include waiting places $w_1 \cdots w_4$ and received places $rcv_1 \cdots rcv_4$.*

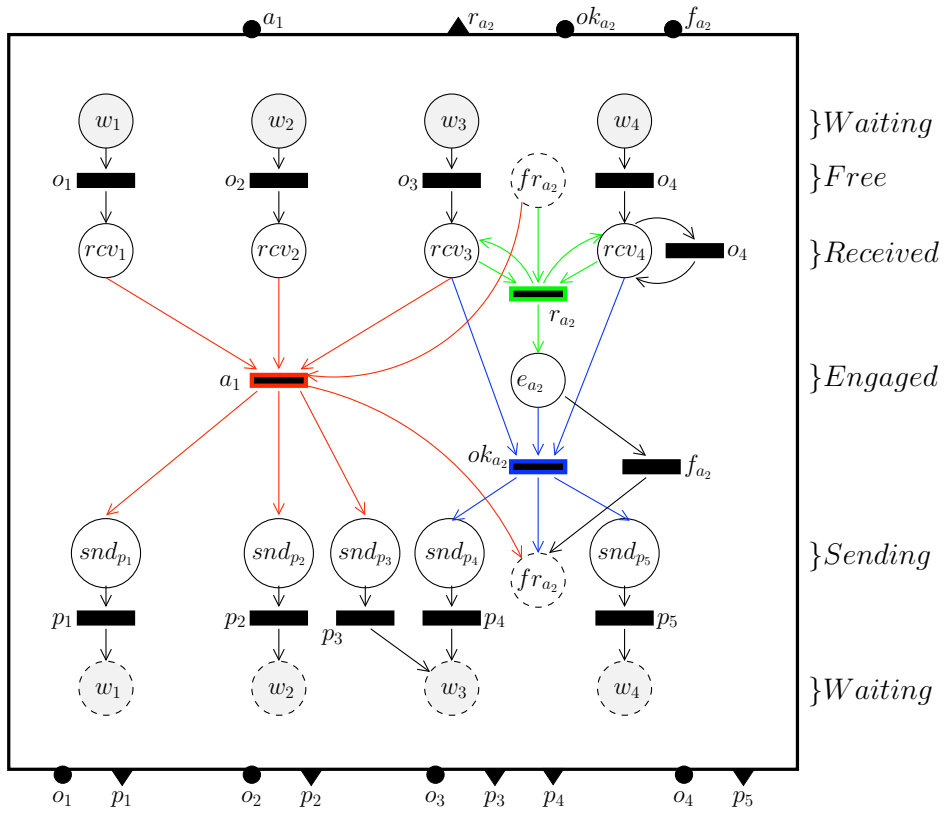


Figure 4.10: Component IP_1 in Figure 4.8.

  - *For each port p involved in interactions of γ_j, we include a* sending *place snd_p. The response to an offer with $x_p = true$ is sent from this place to port p of the component that has made the offer. In Figure 4.10, places $snd_{p_1} \cdots snd_{p_5}$ correspond to ports $p_1 \cdots p_5$ respectively, as they form interactions hosted by IP_1 (i.e., a_1 and a_2).*
  - *For each interaction $a \in \gamma_j$ that is in external conflict with another interaction, we include an* engaged *place e_a and a* free *place fr_a. In Figure 4.10, only interaction a_2 is in external conflict, for which we add places e_{a_2} and fr_{a_2}.*
- ***Variables*** X^{IP_j}**.** *For each port p involved in interactions of γ_j, we include a Boolean variable x_p. The value of this variable is equal to the value of the same variable in the most recent offer received from the corresponding component. Also, for each component B_i involved in interactions of γ_j, we include an integer n_i that stores participation number of B_i, and its own variables.*
- ***Ports*** P^{IP_j}**.** *The set of ports of IP_j is the following:*
  - *For each component B_i involved in interactions of γ_j, we include an offer port o_i. Each port o_i updates the values of variables n_i, x_p and the variable associated for each port p exported by B_i. In Figure 4.10, ports $o_1 \cdots o_4$ represent offer ports for components $B_1 \cdots B_4$.*
  - *For each port p involved in interactions of γ_j, we include a response port p. In Figure 4.10, ports $p_1 \cdots p_5$ correspond to the ports that form interactions a_1 and a_2.*
  - *For each interaction $a = (P, G, F) \in \gamma_j$ that is in external conflict, we include reservation ports r_a, ok_a, and f_a. If $P = \{p_i\}_{i \in I}$, the port r_a is associated to the variables $\{n_i\}_{i \in I}$, where I is the set of components involved in interaction a. In Figure 4.10, ports r_{a_2}, ok_{a_2}, and f_{a_2} represent the external conflict of a_2 with interactions a_3 and a_4.*
  - *For each interaction $a \in \gamma_j$ that is not in external conflict, we include a unary port a. In Figure 4.10, we include unary port a_1, as a_1 is only in internal conflict with a_2.*
- ***Transitions*** T^{IP_j}**.** *IP_j performs two tasks: (1) receiving offers from components in the lower layer and responding to them, and (2) requesting reservation of an interaction from the Conflict Resolution Protocol in case of an external conflict. The following set of transitions of IP_j performs these two tasks:*
  - *In order to receive offers from a component B_i, we include transition (w_i, o_i, rcv_i). If B_i participates in an interaction not handled by IP_j, we also include transition (rcv_i, o_i, rcv_i) to receive new offers when B_i takes part in such an interaction. Transitions labeled by $o_1 \cdots o_4$ in Figure 4.10 are of this type.*
  - *Requesting reservation of an interaction $a = (P, G, F) \in \gamma_j$ that is in external conflict is accomplished by transition $(\{rcv_i\}_{i \in I} \cup \{fr_a\}, r_a, \{rcv_i\}_{i \in I} \cup \{e_a\})$, where I is the set of components involved in interaction a. This transition is guarded by the predicate $\bigwedge_{i \in I} x_{p_i} \wedge G$ which ensures enabledness of a. Notice that this transition*

is enabled when the token for each participating component is in its corresponding receive place rcv_i and the guard G of the interaction is true. Execution of this transition results in moving the token from a free place to an engaged place. In Figure 4.10, transition r_{a_2} is of this type, and is guarded by $x_{p_4} \wedge x_{p_5}$.

- *For the case where the Conflict Resolution Protocol responds positively, we include the transition $(\{rcv_i\}_{i\in I} \cup \{e_a\}, ok_a, \{snd_{p_i}\}_{i\in I} \cup \{fr_a\})$. The execution of this transition triggers the function F of the interaction a, and then, the token from the engaged place moves to the free place and the tokens from received move to sending places for informing the corresponding components. Transition ok_{a_2} in Figure 4.10 occurs when interaction a_2 is successfully reserved by the Conflict Resolution Protocol.*
- *For the case where the Conflict Resolution Protocol responds negatively, we include the transition (e_a, f_a, fr_a). Upon execution of this transition, the token moves from the engaged place to the free place. Transition f_{a_2} in Figure 4.10 occurs when the Conflict Resolution Protocol fails to reserve interaction a_2 for component IP_1.*
- *For each interaction $a = \{p_i\}_{i\in I}$ in γ_j that has only internal conflicts, let A be the set of interactions that are in internal conflict with a, but are externally conflicting with other interactions. We include the transition $(\{rcv_i\}_{i\in I} \cup \{fr_{a'}\}_{a'\in A}, a, \{snd_{p_i}\}_{i\in I} \cup \{fr_{a'}\}_{a'\in A})$. This transition is guarded by the predicate $\bigwedge_{i\in I} x_{p_i}$ and moves the tokens from receiving to sending places. Tokens from $fr_{a'}$ places ensure that no internally conflicting interaction requested a reservation. The transition labeled by a_1 in Figure 4.10 falls in this category.*
- *Finally, for each component B_i exporting p, we include the transitions (snd_p, p, w_i). This transition notifies component B_i to execute the transition labeled by port p. These are transitions labeled by $p_1 \cdots p_5$ in Figure 4.10.*

4.6.3 The Conflict Resolution Protocol

As discussed earlier, the main task of the Conflict Resolution Protocol is to ensure that externally conflicting interactions are executed mutually exclusive. The Conflict Resolution Protocol can be implemented using any algorithm that solves the committee coordination problem. Our design of Conflict Resolution Protocol allows employing any such algorithm with minimal restrictions.

We adapt a variation of the idea of the message-count technique from [Bag89] as a minimal restriction to ensure that our design makes *progress* (see Lemma 2) and it does not interfere with exclusion algorithms. This technique is based on counting the number of times that a component interacts. Each component keeps a counter n which indicates the current number of participations of the component in interactions. The Conflict Resolution Protocol ensures that each participation number is used only once. That is, each component takes part in only one interaction per transition. To this end, in the Conflict Resolution Protocol, for each component B_i, we keep a variable N_i which stores the latest

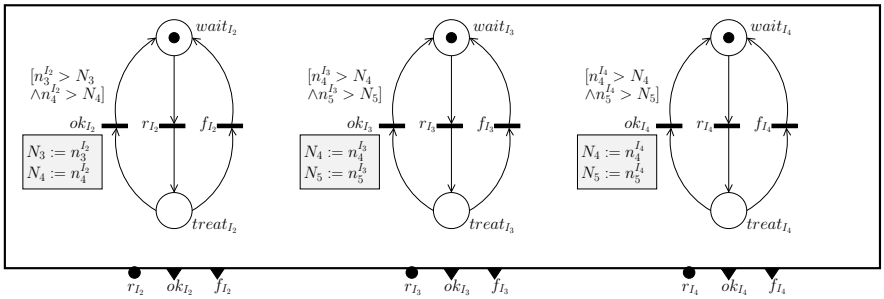


Figure 4.11: A centralized Conflict Resolution Protocol for Figure 4.8.

number of participations of B_i. Whenever a reserve message r_a for interaction $a = \{p_i\}_{i \in I}$ is received by the Conflict Resolution Protocol, the message provides a set of participation numbers ($\{n_i^a\}_{i \in I}$) for all components involved in a. If for each component B_i, the participation number n_i^a is greater than N_i, then the Conflict Resolution Protocol acknowledges successful reservation through port ok_a and the participation numbers in the Conflict Resolution Protocol are set to values sent by the Interaction Protocol. On the contrary, if there exists a component whose participation number is less than or equal to what Conflict Resolution Protocol has recorded, then the corresponding component has already participated for this number and the Conflict Resolution Protocol replies failure via port f_a.

Now, since the structure and behavior of the Conflict Resolution Protocol components depend on the employed algorithm, we only specify an abstract set of minimal restrictions of this layer as follows:

- For each component B_i, the Conflict Resolution Protocol maintains a variable N_i indicating the last participation number reserved for B_i.
- For each interaction $a = \{p_i\}_{i \in I}$ handled by the Conflict Resolution Protocol, we include three ports: r_a, ok_a and f_a. The receive-port r_a accepts reservation requests containing fresh values of variables n_i^a. The send-ports ok_a and f_a accept or reject the latest reservation request, and N_i variables are updated in case of positive response.
- Each r_a message should be acknowledged by exactly one ok_a or f_a message.
- Each component of the Conflict Resolution Protocol should respect the message-count properties described above.

4.6.3.1 Centralized Implementation

Figure 4.11 shows a centralized Conflict Resolution Protocol for the model in Figure 4.8. In fact, the component in Figure 4.11 is the component $CENT_1$ in Figure 4.8. A reservation request, for instance, r_{a_2}, contains fresh variables $n_3^{a_2}$ and $n_4^{a_2}$ (corresponding to components B_3 and B_4). The token representing interaction a_2 is then moved from place

$wait_{a_2}$ to place $treat_{a_2}$. From this state, the Conflict Resolution Protocol can still receive a request for reserving a_3 and a_4 since $wait_{a_3}$ and $wait_{a_4}$ still contain a token. This is where message-counts play their role. The guard of transition ok_{a_2} is $(n_3^{a_2} > N_3) \wedge (n_4^{a_2} > N_4)$ where N_i is the last known used participation number for B_i. Note that since execution of transitions are atomic in BIP, if transition ok_{a_2} is fired, it modifies variables N_i atomically (i.e., before any other transition can take place). We denote this implementation by $CENT$.

4.6.3.2 Token Ring Implementation

Another example of a Conflict Resolution Protocol is inspired by the token-based algorithm due to Bagrodia [Bag87], where we add one reservation component per externally conflicting interaction. Figure 4.12 shows the respective components for the model presented in Figure 4.8. Exclusion is ensured using a circulating token carrying N_i variables; i.e., the component that owns the token compares the value of the received n_i variables with the N_i variables from the token. If they are greater, an ok message is sent to the component that handles that interaction and the N_i values on the token are updated. Otherwise, a fail message is sent. Subsequently, the reservation component releases the token via port ST, which is received by the next component via port RT. Obviously, this algorithm allows a better level of distribution at the Conflict Resolution Protocol layer. We denote this implementation by TR.

4.6.3.3 Implementation Based on Dining Philosophers

A third choice of Conflict Resolution Protocol algorithm is an adaption of the hygienic solution to the dining philosophers problem presented in [CM88, Bag89]. Its Send/Receive BIP implementation is presented in Figure 4.13. Similar to token ring, each externally conflicting interaction is handled by a separate component. If two interactions are conflicting, the two corresponding components share a fork carrying N_i variables corresponding to the atomic components causing the conflict. In order to positively respond to a reserve, a component has to fetch all forks shared with its neighbors. Then, it compares participation numbers received from the reservation request and from the forks and responds accordingly. After such a response, the forks become dirty. Finally, the component sends the forks if it is asked to do so. We denote this implementation by DP.

4.6.4 Cross-Layer Interactions

In this subsection, we define the interactions of our 3-layer model. Following Definition 4.4.1, we construct Send/Receive interactions by specifying which one is the sender. Given a BIP model $\gamma(B_1 \cdots B_n)$, a partition $\gamma_1 \cdots \gamma_m$, and the obtained Send/Receive components $B_1^{SR} \cdots B_n^{SR}$, Interaction Protocol components $IP_1 \cdots IP_m$, and Conflict Resolution

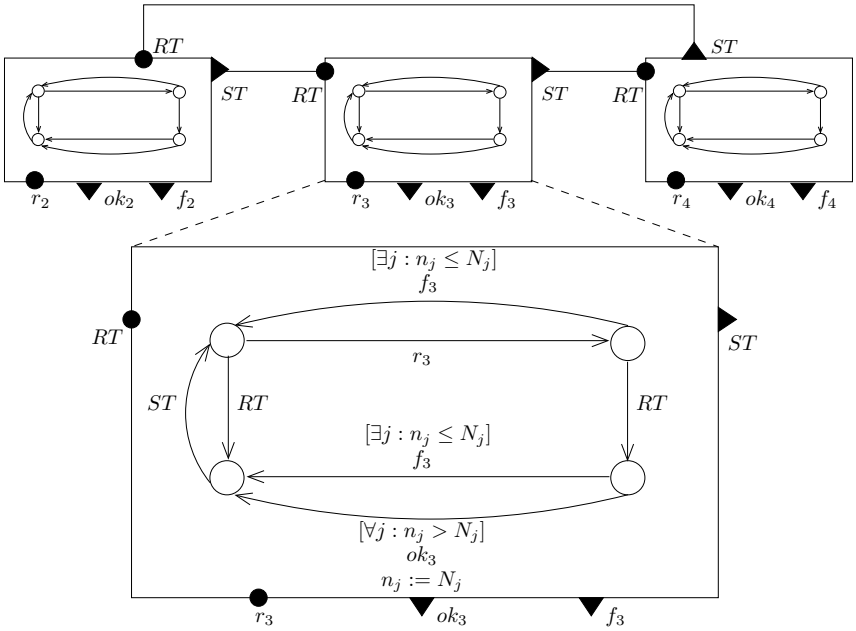


Figure 4.12: Token-based Conflict Resolution Protocol for the BIP models in Figures 4.1 and 4.8.

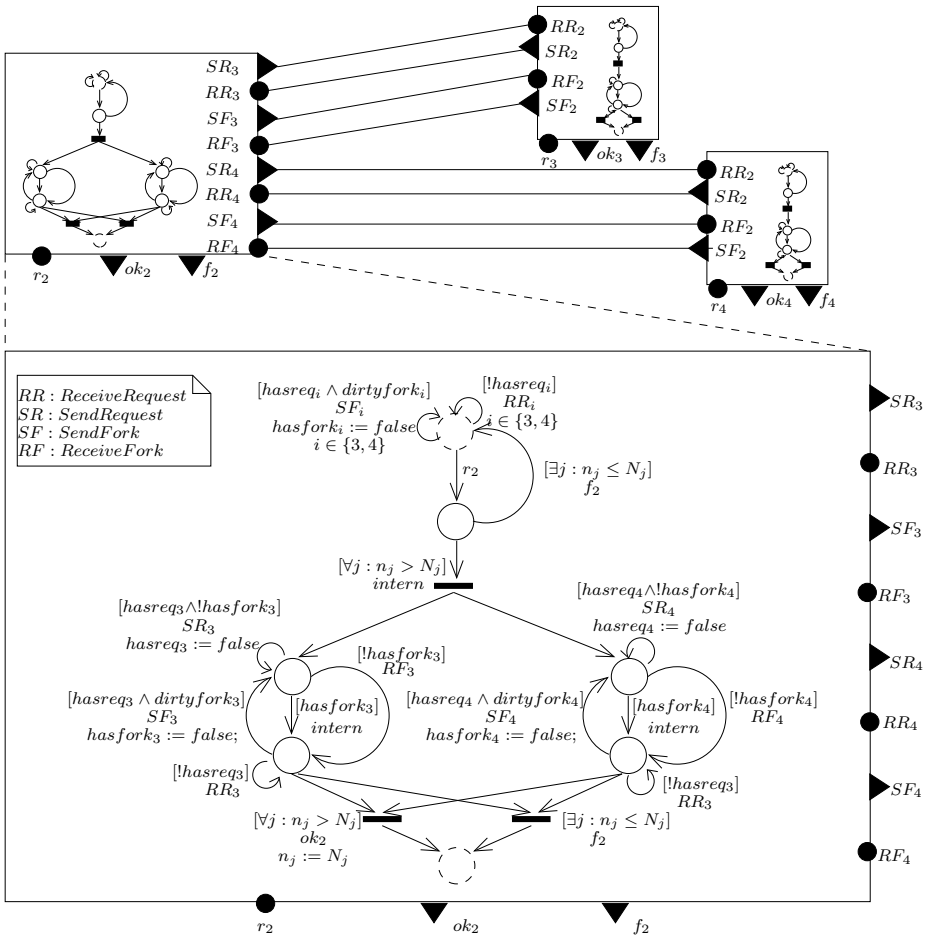


Figure 4.13: Dining philosophers-based Conflict Resolution Protocol for the BIP models in Figures 4.1 and 4.8.

Protocol components $CENT_1 \cdots CENT_k$, we construct the Send/Receive interactions γ^{SR} according to Definition 4.4.1 as follows:

- For each component B_i, γ^{SR} contains a multicast connector formed by all ports o_i, where B_i is the sender.
- For each Interaction Protocol component IP_j and port p in IP_j, we include a binary interaction, such that port p of IP_j is the sender, and, port p of the corresponding component in the components layer is the receiver.
- For each interaction a that is in external conflict, γ^{SR} contains an interaction between r_a ports, such that the Interaction Protocol is the sender and Conflict Resolution Protocol is the receiver. Likewise, γ^{SR} contains interactions between ok_a and f_a ports.

Note that the interaction do not depend on the Conflict Resolution Protocol. The entire model obtained is denoted B^{SR}_{CENT}, B^{SR}_{TR} or B^{SR}_{DP} following the embedded Conflict Resolution Protocol. The interactions between the three layers of our running example are presented in Figure 4.8. The send-ports are graphically denoted by triangles and receive-ports by bullets.

4.7 Correctness

In Subsection 4.7.1, we show that our 3-layer model meets the constraints of the Send/Receive model specified in Section 4.5. In Subsection 4.7.2, we prove that a BIP model is observationally equivalent with the BIP model obtained by the transformation of Section 4.6. Finally, we prove the correctness of models embedding different implementations of Conflict Resolution Protocol in Subsection 4.7.3.

4.7.1 Compliance with Send/Receive Models

Proposition 3 *Given a BIP model B, the model B^{SR} obtained by transformation of Section 4.6 meets the constraints of Definition 4.4.1.*

Proof The send-ports and receive-ports are clearly determined in subsection 4.6.4 and respect the syntax presented in the two first points of definition 4.4.1. We now prove the third point, that is whenever a send-port is enabled, all its associated receive-ports are enabled.

Between the Interaction Protocol and Conflict Resolution Protocol layers, for reserve, ok and fail interactions related to $a \in \gamma$ it is sufficient to consider places fr_a and e_a in the Interaction Protocol layer, $wait_a$ and $treat_a$ in the Conflict Resolution Protocol layer. Initially the configuration is $(fr_a, wait_a)$ from which only the send-port r_a in Interaction Protocol might be enabled, and the receive-port r_a is enabled. If the r_a interaction takes place, we reach the configuration $(e_a, treat_a)$, in which only send-ports ok_a and f_a in Conflict Resolution Protocol might be enabled, and the associated receive-ports in Interaction

Protocol are enabled. Then if either ok or fail interaction takes place we switch back to the initial configuration.

Between components and Interaction Protocol layers, for all interactions involving component B_i, it is sufficient to consider only the places w_i, r_i and s_p for each port p exported by B_i in the Interaction Protocol. Whenever one of the places w_i or r_i is enabled in each Interaction Protocol component, the property holds for the o_i interaction. In this configuration, no place s_p might be active since it would require one of the token from a w_i or a r_i, thus no send port p is enabled.

If there is an Interaction Protocol component such that the token associated to B_i is an a place s_p, it comes either from an a or an ok_a labeled-transition. In the first case, no other interaction involving B_i can take place, otherwise it would be externally conflicting with a. In the second case according to the Conflict Resolution Protocol, the ok_a was given for the current participation number in the component B_i and no other interaction using this number will be granted. Thus in all cases, there is only one active place s_p with p exported by B_i. The response can then take place and let the components continue their execution. ■

4.7.2 Observational Equivalence between Original and Transformed BIP Models

In this subsection, our goal is to show that B and B^{SR} are observationally equivalent. Let us first recall the definition of *observational equivalence* of two transition systems $A = (Q_A, P \cup \{\beta\}, \rightarrow_A)$ and $B = (Q_B, P \cup \{\beta\}, \rightarrow_B)$. It is based on the usual definition of weak bisimilarity [Mil95], where β-transitions are considered unobservable.

Notice that, a state of an atomic components is defined as a pair $q = (s, v)$ where $s \in L$ is the control state, $v : X \mapsto Data$ is a valuation of the variables X of the atomic component. For simplicity of reasoning and clarity about correctness we omit the variables defined in the original atomic components, hence, functions and guards defined in the transitions and interactions is not necessary. Moreover, a state of an atomic component q become the actual control state l. In this case, an interaction can be seen as a set of ports, and a composite component $B = \gamma(B_1, \ldots, B_n)$ is a transition system $(Q, \gamma, \rightarrow)$, where $Q = \bigotimes_{i=1}^{n} Q_i$ $(Q_i = B_i.L)$ and $\rightarrow$ is the least set of transitions satisfying the rule:

$$\frac{a = \{p_i\}_{i \in I} \in \gamma \qquad \forall i \in I.\ q_i \xrightarrow{p_i}_i q_i' \qquad \forall i \notin I.\ q_i = q_i'}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

Definition 4.7.1 (Weak simulation.) *A* weak simulation *over A and B, denoted $A \subset B$, is a relation $R \subseteq Q_A \times Q_B$, such that we have $\forall (q, r) \in R,\ a \in P :\ q \xrightarrow{a}_A q' \implies \exists r' :$ $(q', r') \in R \wedge r \overset{\beta^* a \beta^*}{\rightarrow}_B r'$ and $\forall (q, r) \in R :\ q \xrightarrow{\beta}_A q' \implies \exists r' :\ (q', r') \in R \wedge r \xrightarrow{\beta^*}_B r'$*

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

We consider the correspondence between actions of B and B^{SR} as follows. For each interaction $a \in \gamma$, where γ is the set of interactions of B, we associate either the binary interaction ok_a or the unary interaction a, depending upon existence of an external conflict. All other interactions (offer, response, reserve, fail) are unobservable and denoted β.

We proceed as follows to complete the proof of observational equivalence. Amongst unobservable actions β, we distinguish between β_1 actions, that are communication interactions between the components layer and the Interaction Protocol (namely offer and response), and β_2 actions that are communications between the Interaction Protocol and and Conflict Resolution Protocol (namely reserve and fail). We denote q^{SR} a state of B^{SR} and q a state of B. A state of B^{SR} from where no β_1 action is possible is called a *stable state*, in the sense that any β action from this state does not change the state of the component layer.

Lemma 1 *From any state q^{SR}, there exists a unique stable state $[q]^{SR}$ such that $q^{SR} \xrightarrow{\beta_1^*} [q]^{SR}$.*

Proof The state $[q]^{SR}$ exists since each Send/Receive component B_i^{SR} can do at most two β_1 transitions: receive a response and send an offer. Since two β_1 transitions involving two different components are independent (i.e do not change the same variable or the same place), the ordering of β_1 action does not change the final state. Thus $[q]^{SR}$ is unique. ■

We now show a property of the participation numbers. Let $B.n$ mean 'the variable n that belongs to component B'.

Lemma 2 *When B^{SR} is in a stable state, for each couple (i, j), such that B_i is involved in interactions handled by IP_j, we have $B_i.n_i = IP_j.n_i > CENT.N_i$.*

Proof When in stable state, all offers have been sent, thus the participation numbers in Interaction Protocol correspond to those in components $B_i.n_i = IP_j.n_i$.

Initially, for each component B_i, $CENT.N_i = 0$ and $B_i^{SR}.n_i = 1$ thus the property holds. The N_i variables in Conflict Resolution Protocol are updated on a ok transition, using values provided by the Interaction Protocol, that is by the components. We show that after each ok_a transition, the property still holds. For each component B_i^{SR} participant in a, it holds that $B_i^{SR}.n_i = CENT.N_i$ after the offer. Then, the response transitions increments participation numbers in components such that in the next stable state $B_i^{SR}.n_i > CENT.N_i$. For components $B_{i'}$ not participating in a, by induction we have $B_i^{SR}.n_{i'} > CENT.N_{i'}$ and only participation numbers in components can be incremented. ■

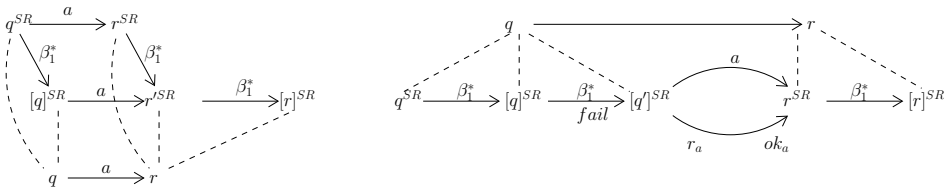
Since we need to take into account participation numbers n_i, we introduce an intermediate centralized model B^n. This new model is a copy of B that includes in each atomic component an additional variable n_i which is incremented whenever a transition is executed. As B and B^n have identical set of states and transitions labeled by the same ports, they are observationally equivalent. (They are even strongly bisimilar.)

Lemma 3 $B \sim B^n$.

Proof We say that two states (q, q^n) of B and B^n are equivalent if they have the same control states. This defines a bisimulation. ■

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

Proposition 4 $B^{SR} \sim B^n$.



(a) Point(ii) (b) Point(ii)

Figure 4.14: Proof of observational equivalence.

Proof We define a relation R between the states Q^{SR} of B^{SR} and the states Q of B^n as follows: $R = \{(q^{SR}, q) \mid \forall i \in I : [q]_i^{SR} = q_i\}$ where q_i denotes the state of B_i^n at state q and $[q]_i^{SR}$ denotes the state of B_i^{SR} at state $[q]^{SR}$. The three next assertions prove that R is a weak bisimulation:

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

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

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

(i) If $q^{SR} \xrightarrow{\beta} r^{SR}$, either β is a β_1 action and $[q]^{SR} = [r]^{SR}$, either β is a β_2 action which does not change the state of component layer and does not enable any send-port.

(ii) The action a in B^{SR} is either a unary interaction a or a binary interaction ok_a. In both cases, $a = \{p_i\}_{i \in I}$ has been detected to be enabled in IP_j by the tokens in received places and the guard of the a or r_a transition in Interaction Protocol, with the participation numbers n_i. We show that a is also enabled at state $[q]^{SR}$:

- If a has only local conflicts, no move involving B_i can take place in another Interaction Protocol, and no β_1 move involving B_i can take place in IP_j since a is enabled.
- If a is externally conflicting, no move involving B_i has taken place in another Interaction Protocol (otherwise ok_a would not have been enabled), nor in IP_j since the fr_a place is empty.

At stable state $[q]^{SR}$, the lemma 2 ensures that $IP_j.n_i = B_i^{SR}.n_i$. Following the definition of R, we have $B_i.n_i = B_i^{SR}.n_i$ when B^n is at state q. Thus a is enabled with the same participation numbers at state q and in IP_j at state q^{SR} and $[q]^{SR}$, which implies $q \xrightarrow{a}$.

Since the β_1 actions needed to reach the state $[q]^{SR}$ did not interfere with action a, we can replay them from r^{SR} to reach a state r'^{SR}, as shown on figure 4.14. The state r'^{SR} is not stable because of response and offers that can take place in each component participant in B_i. Executing these actions brings the system in state $[r']^{SR}$ which is clearly equivalent to r, and by point (i) we have $(r^{SR}, r) \in R$.

(iii) In figure 4.14, we show the different actions and states involved in this part. From q^{SR}, we reach $[q]^{SR}$ by doing β_1 actions. Then we execute all possible fail interactions (that are β_2 actions), so that all fr_a places are empty, to reach a state $[q']^{SR}$. At this state, if a has only local conflicts, the interaction a is enabled, else the sequence $r_a\ ok_a$ can be executed since lemma 2 ensures that guard of ok_a is true. In both cases, the interaction corresponding to a brings the system in state r^{SR}. From this state, the responses corresponding to each port of a are enabled, and the next stable state $[r]^{SR}$ is equivalent to r, thus $(r^{SR}, r) \in R$. ■

4.7.3 Interoperability of Conflict Resolution Protocol

As mentioned in Subsection 4.6.3, the centralized implementation *CENT* of the Conflict Resolution Protocol can be seen as a specification. We also proposed two other implementations, respectively, token-ring *TR* and dining philosophers *DP*. However, these implementations are not observationally equivalent to the centralized implementation. More precisely, the centralized version defines the most liberal implementation: if two reservation requests a_1 and a_2 are received, the protocol may or may not acknowledge them, in a specific order. This general behavior is not implemented neither by the token ring nor by the dining philosophers implementations. In the case of token ring, the response may depend on the order the token travels through the components. In the case of dining philosophers, the order may depend on places and the current status of forks.

Nevertheless, we can prove an observational equivalence if we consider *weaker* versions of the above implementations. More precisely, for the token ring protocol, consider the weaker version $TR^{(w)}$ which allows to release the token or provide a fail answer regardless of the values of counters. Likewise, for the dining philosophers protocol, consider the weaker version $DP^{(w)}$, where forks can always be sent to neighbors, regardless of their status and the values of counters. Clearly, a weakened Conflict Resolution Protocol is not desirable for

a concrete implementation since they do not enforce progress. But, they play a technical role in proving the correctness of our approach. The following proposition establishes the relation between the different implementations of the Conflict Resolution Protocol.

Proposition 5 *(i)* $CENT \sim TR^{(w)} \sim DP^{(w)}$
(ii) $TR \subset TR^{(w)}$, $DP \subset DP^{(w)}$.

Let us denote by B^{SR}_X the 3-layer model obtained from the initial system B and embedding algorithm X in the Conflict Resolution Protocol. Also, let us denote $Tr(B)$ the set of all possible traces of observable actions allowed by an execution of B. The following proposition states the correctness of our implementation.

Proposition 6 *(i)* $B \sim B^{SR} \sim B^{SR}_{TR^{(w)}} \sim B^{SR}_{DP^{(w)}}$
(ii) $Tr(B) \supseteq Tr(B^{SR}_{TR})$ *and* $Tr(B) \supseteq Tr(B^{SR}_{DP})$.

Proof (i) The leftmost equivalence is is a consequence of lemma 3 and proposition 4. The other equivalences come from proposition 5 and the fact that observational equivalence is a congruence with respect to parallel composition. (ii) The trace inclusions follows from the simulations $TR \subset TR^{(w)}$ respectively $DP \subset DP^{(w)}$ ■

4.8 Transformation from Send/Receive BIP into C++

In this section, we describe how we generate for a Send/Receive BIP component pseudo C++ code. Notice that since the behavior of these components are formalized as Petri nets, we only present generation of C++ code for a Petri net whose transitions are labeled by send-ports, receive-ports, or unary ports (see C++ Pseudo Code 1).

Initially, each component creates a TCP socket and establishes reliable connections with all components that it needs to interact (Lines 1-2). These interactions and their corresponding physical connections are determined according to the complete Send/Receive BIP model and a *configuration file*. This file specifies the IP address and port number of all components for final deployment. We assign one Boolean variable to each place of the given Petri net, which shows whether or not the place contains the token. Thus, the initial state of the Petri net is determined by an initial assignment of these variables (Line 3).

After initializations, the code enters an infinite loop that executes the transitions of the Petri net as follows. For each step, the code scans the list of all possible transitions and gives priority to transitions that are labeled by a send-port (Lines 6-10) or unary ports of the given Petri net (Lines 11-15). Actual emission of data is performed by an invocation of the TCP sockets system call `send()` in Line 7. Once data transmission or an internal computation is completed, tokens are removed from input places and put to output places of the corresponding transitions (Lines 8 and 13).

C++ Pseudo Code 1 Petri net

Input: A Petri net of a Send/Receive BIP component and a configuration file.
Output: C++ code that implements the given Send/Receive Petri net

```
    // Initializations
 1: CreateTCPSocket();
 2: EstablishConnections();
 3: PrepareInitialState();

 4: while true do
 5:     // Handling send-ports and internal computations
 6:     if there exists an enabled transition labeled by a send-port then
 7:         send(...);
 8:         PrepareNextState();
 9:         continue;
10:     end if
11:     if there exists an enabled transition labeled by a unary port then
12:         DoInternalComputation();
13:         PrepareNextState();
14:         continue;
15:     end if

16:     // Handling receiving messages
17:     select(...);
18:     recv(...);
19:     PrepareNextState();
20: end while
```

Finally, if no send-port is enabled and all internal computations are completed, execution stops and waits for messages from other components (Line 17). Once one of the sockets contains a new message, the component resumes its execution and receives the message (Line 18).

It is straightforward to observe that our code avoids creating deadlocks by giving priority to send-ports and unary-ports. Moreover, sending messages before doing internal computation triggers receivers components waiting for a response and increases parallelism.

Note that we also provide the generation of C++ code by using MPI for communications. Generally speaking, the same principle is applied as above, however, we use the communication primitives offered by MPI (e.g., *MPI_Send*(), *MPI_ISend*(), *MPI_Recv*(), etc.) instead of TCP sockets.

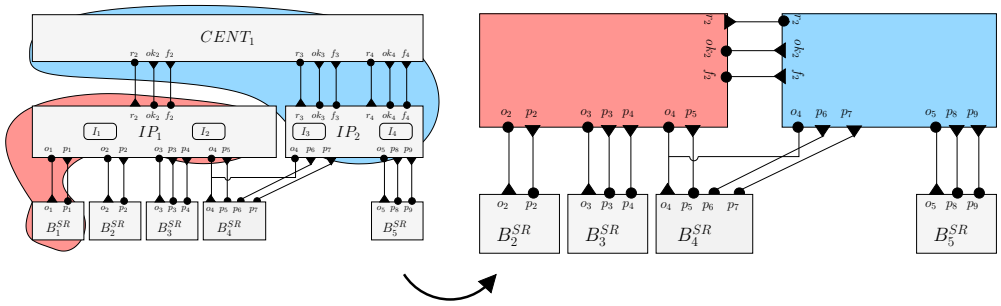


Figure 4.15: The impact of merging components.

4.9 Component Composition

This technique is applied to the intermediate 3-layer Send/Receive model developed in Section 4.6. It consists in composing some components from the 3-layer Send/Receive model. For instance, if we consider an Interaction Protocol handling one interaction, in this case, it is possible to merge it into one of its corresponding components without losing any parallelism. This method will be also useful in the case when generating MPI code, since an important overhead will appear due to context switching between processes. For example, if we consider a platform consisting of two cores, it is preferable to generate exactly two processes. Thus, we need to merge components from the 3-layer Send/Receive model to obtain the less number of components and without killing parallelism. Figure 4.15 illustrates an example of merging components. As input we take a set of partitioning of components and we obtain as output a new equivalent BIP model by merging with respect to the partition given as input. This can be done by applying the third transformation (Component Composition) presented in Chapter 3.

4.10 Experimental Validation

In this section, we present the results of our experiments. Recall that, our implementation automatically generates C++ code from the 3-layer BIP model developed in Sections 4.5 and 4.6, where Send/Receive interactions are implemented by TCP sockets or MPI primitives. We have implemented and integrated the transformations in the BIP toolset. The tool takes a composite BIP model in the global state semantics and a network configuration file as input and generates the corresponding C++ executable for each Send/Receive component for all layers of the intermediate BIP model (e.g, Atomic Component, Interaction Protocol, Conflict Resolution Protocol). Each executable can be run independently on a different machine or a processor core.

We denote each experiment scenario by (i, X), where i is the number of interaction

partitions and X is the choice among the three Conflict Resolution Protocols described in Subsection 4.6.3 (i.e., *CENT*, *TR*, or *DP*). For the case where partitioning of interactions results in having no external conflicts, hence, requiring no reservation component, we use the symbol '$-$' to denote an empty Conflict Resolution Protocol. All experiments in this section are conducted on quad-Xeon 2.6 GHz machines with 6GB RAM running under Debian Linux. The machines are connected via a 100Mbps Ethernet network.

For three non trivial examples, Diffusing Computation, Utopar Transportation System and Bitonic Sorting, described bellow we show that different conflict resolution algorithms and partitioning may result in significantly different performance depending on the initial BIP model and the deploying of the distributed implementation over target platforms.

4.10.1 Diffusing Computation

We model a simplified version of Dijkstra-Scholten termination detection algorithm for diffusing computations [DS80] in BIP. *Diffusing computation* is the task of propagating a message across a distributed system; i.e., a wave that starts from an initial node and diffuses to all processes in a distributed system. Diffusing computation has numerous applications such as traditional distributed deadlock detection and reprogramming of modern sensor networks. One challenge in diffusing computation is to detect its termination. In our version, we consider a torus (wrapped around grid) topology for a set of distributed processes, where a spanning tree throughout the distributed system already exists; each process has a unique parent and the root process is its own parent. Termination detection is achieved in two phases: (1) the root of the spanning tree possesses a message and initiates a *propagation wave*, so that each process sends the message to its children, and (2) once the first wave of messages reaches the leaves of the tree, a *completion wave* starts, where a parent is complete once all its children are complete. In this setting, when the root is complete, termination is detected.

Our BIP model has $n \times m$ atomic components (see Figure 4.16 for a partial model). Each component participates in two types of interactions: (1) four binary rendezvous interactions (e.g., $a_0 \cdots a_3$) to propagate the message to its children (as in a torus topology, each node has four neighbors, hence, potentially four children), and (2) one 5-ary rendezvous interaction (e.g., a) for the completion wave, as each parent has to wait for all its children to complete.

Our first set of experiments is on a 4×6 torus. We apply different partitioning scenarios as illustrated in Figure 4.17. Figure 4.18 shows the time needed for 100 rounds of detecting termination of diffusing communication for each scenario. In the first two scenarios, the interactions are partitioned, so that all conflicts are internal and, hence, resolved locally by the Interaction Protocol. In case of $(2, -)$, all interactions of the propagation wave are grouped into one component of the Interaction Protocol and all interactions related to the completion wave are grouped into the second component. Such grouping does not allow parallel execution of interactions. This is the main reason that the performance of $(1, -)$

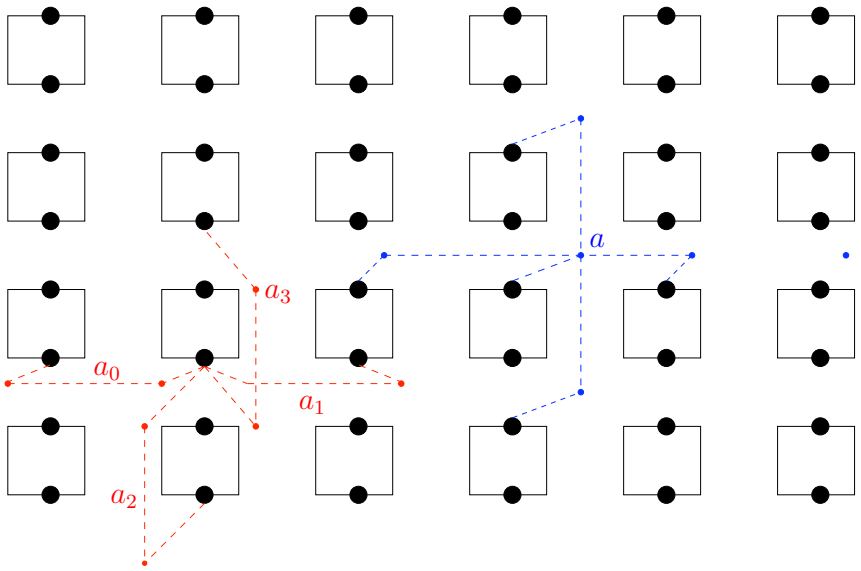


Figure 4.16: Partial BIP model for diffusing computations.



Figure 4.17: Different scenarios for diffusing computations.

and $(2, -)$ are the worst in Figure 4.18.

Next, we group all interactions involved in components $1 \cdots 12$ into one component and the rest in a second component of the Interaction Protocol. This constitutes experiments $(2, CENT)$, $(2, TR)$, and $(2, DP)$. Such partitioning allows more parallelism during propagation and completion waves, as an interaction in the first partition can be executed in parallel with an interaction in the second partition [1]. This is why the performance of $(2, CENT/TR/DP)$ is better than $(1, -)$ and $(2, -)$. Now, since almost all propagation interactions conflict with each other and so do all completion interactions, in case of the dining philosophers algorithm, the conflict graph is not dense. Hence, a small number of decisions can be made in a local neighborhood of philosophers. It follows that the performance of $(2, TR)$ is quite competitive with $(2, DP)$. It can also be seen that $(2, CENT)$ performs as good as $(2, TR)$ and $(2, DP)$. This is due to the fact that there exist only two partitions, which results in a low number of reservation requests.

Figure 4.18 also shows the same type of experiments with 4 and 24 partitions. Similar to the case of two partitions, the performance of TR and $CENT$ for 4 and 24 partitions are almost the same. However, $CENT$ and TR outperform DP. This is due to the fact that in case of DP, each philosopher needs to acquire 4 forks, which requires considerable communication. On the other hand, TR does not require as much communication, as the only task it has to do is releasing and acquiring the token. Moreover, the level of parallelism in DP in case of a 6×4 torus is not high enough to overcome the communication volume.

In the next experiment, following the lesson learned from the tradeoff between communication volume and parallelism, we design a scenario where we exploit the fact that each reservation component in DP resolves conflicts through communicating with its neighboring components. This is not the case in TR. Thus, we consider a 20×20 torus. As can be seen in Figure 4.19, the performance of DP is significantly better than TR. This is solely because when we have a large number of components, in TR. The token has to travel a long way in order to allow parallel execution of interactions. To the contrary, in DP, the Conflict Resolution Protocol components act in their local neighborhood and although more communication is needed, it allows better concurrency, hence, higher simultaneous execution of interactions. We expect that by increasing the size of the torus, DP outperforms $CENT$ as well.

We conclude from this example by stating the main lesson learned from our experiments:

> *Different partitioning schemes and choice of committee coordination algorithm for distributed conflict resolution suit different topologies and settings although they serve a common purpose. Designers of distributed applications should have access to a library of algorithms and choose the best according to parameters of the application.*

1. Execution of each interaction involves 10ms suspension of the corresponding component in the Interaction Protocol to perform and I/O command.

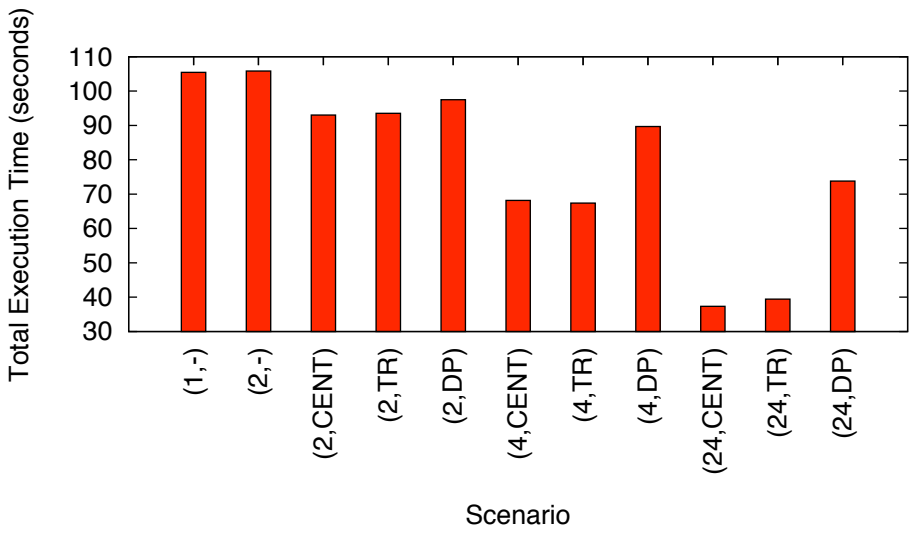


Figure 4.18: Performance of termination detection in diffusing computation in different scenarios (Torus 4×6).

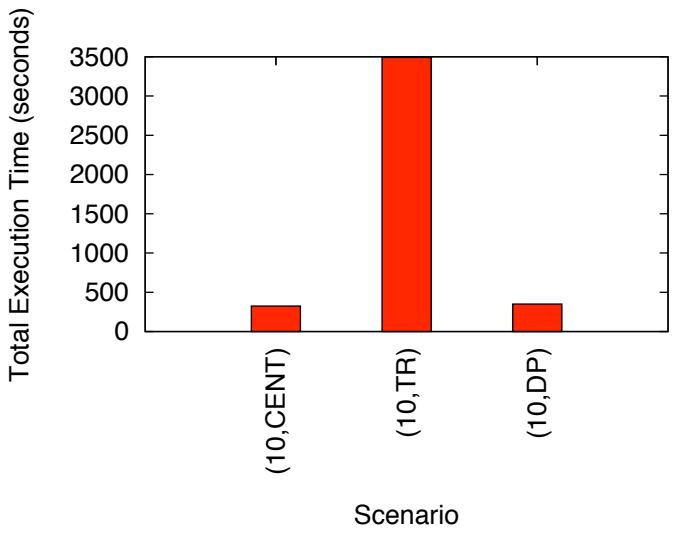


Figure 4.19: Performance of termination detection in diffusing computation in different scenarios (Torus 20×20).

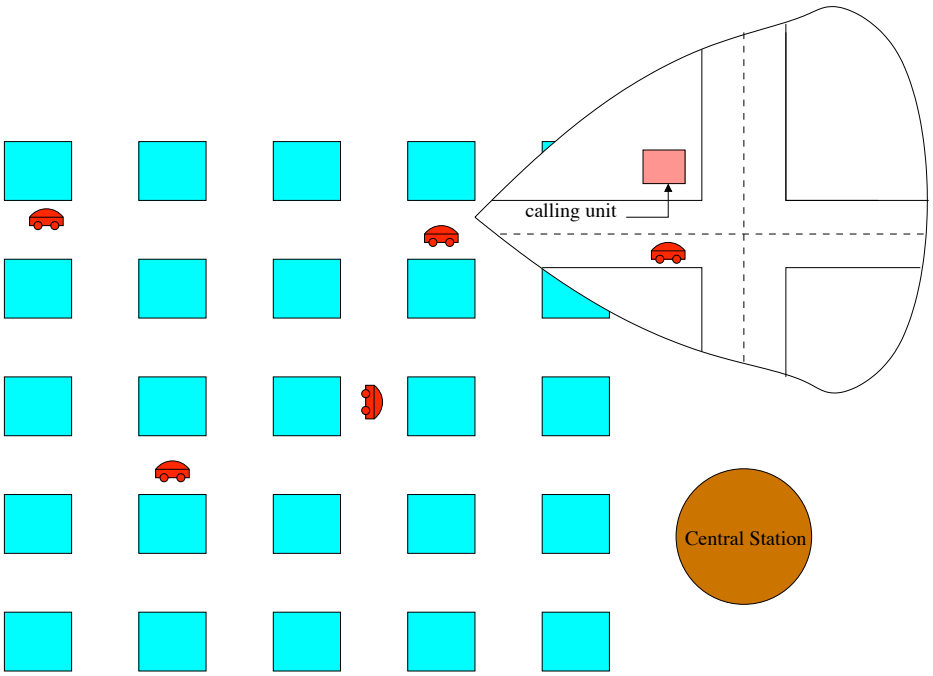


Figure 4.20: Utopar transportation system.

4.10.2 Utopar Transportation System

The second example is Utopar, an industrial case study of the European Integrated project SPEEDS [2]. Utopar is an automated transportation system managing various requests for transportation. The system consists of a set of autonomous vehicles, called U-cars, a centralized automatic control (Central-Station) and calling units (see Figure 4.20).

We modeled a simplified version of the Utopar Transportation System in BIP. The overall system architecture is depicted in Figure 4.21. It is a composition of an arbitrary (but fixed) number of components of three different types: U-Cars, Calling-Units and Central-Station. The Utopar system interacts with external users, i.e., the passengers. For sake of completeness, users are also represented in the Figure 1 as components, however, their behavior is not explicitly modeled.

The overall behavior of the system is obtained by composing the behavior of the inner components according to the following set of interactions:

– request: handling car requests (made by passengers) at Calling-Units;

2. http://www.speeds.eu.com/

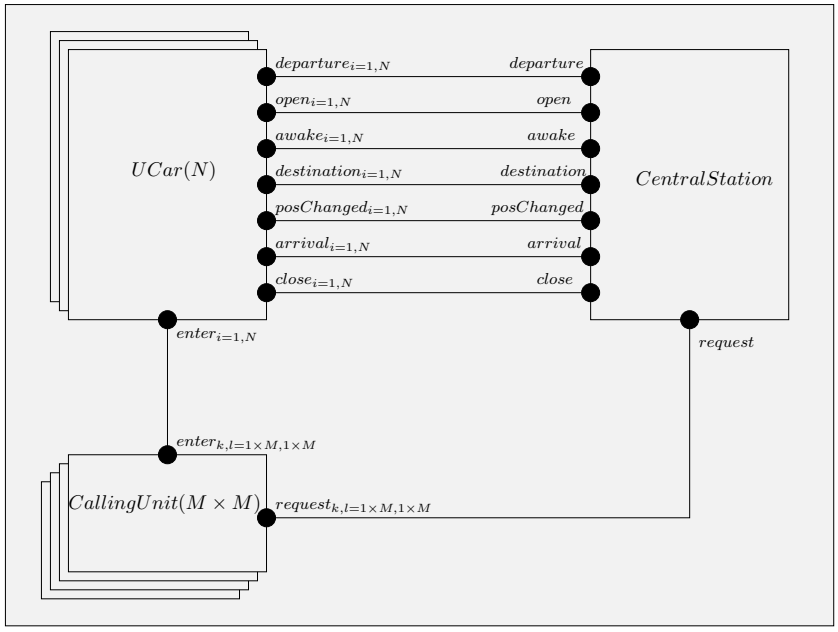


Figure 4.21: High-level BIP model for Utopar system.

– destination: handling destination requests (made by passengers) seating within U-Cars;
– enter: handling the step on (resp. off) for passengers into U-Cars;
– departure: handling departure commands issued by Central-Station towards the U-Cars;
– posChanged, arrival: information provided by moving U-Cars towards the Central-Station;
– open, close: handling the opening/closing of the U-Cars doors, while parked at Calling-Units.

Our first set of experiments consists of $25 = 5 \times 5$ calling units and 4 cars. For each calling unit we group all the interactions connected on it in the same Interaction Protocol. Moreover, for each car we group all the interactions connecting the car with the central station in the same Interaction Protocol. Thus, we obtain 29 Interaction Protocol components. Using this partitioning we generate the corresponding 3-layer Send/Receive model for the three Conflict Resolution Protocols.

We simulate the target platform as follows. We consider that there exists a machine on each calling unit. Moreover, each machine is connected to their four neighbours and the communication between two neighbours machines takes $1ms$.

We generate the corresponding C++ executable for each Send/Receive component for

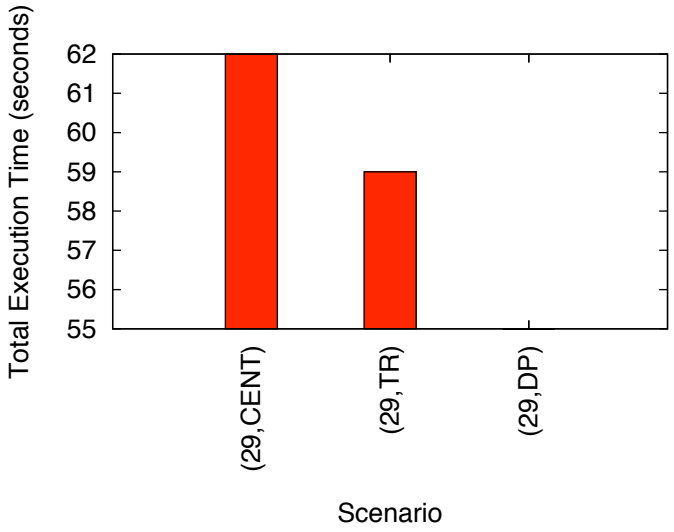


Figure 4.22: Performance of responding 10 requests per calling unit ($25 = 5 \times 5$ calling units, and 4 cars).

all layers, and we embedded each C++ executable as follows. We embedded the code corresponding to the calling unit with its Interaction Protocol into the machine located on the calling unit. Furthermore, we embedded the code corresponding to the central station into the central machine that is located in the center of the calling units. Regarding the Conflict Resolution Protocol, in the case of *CENT* the best choice is to embed its corresponding code into the central machine. Concerning *TR* and *DP* algorithm we embedded the code of each component of this layer in its corresponding Interaction Protocol.

Figure 4.22 shows the time needed for responding 10 requests by each calling unit. It is clear that the performance of $(29, DP)$ is better than $(29, TR)$ and $(29, CENT)$. This is due to the overhead of communications for the case of *TR* and *CENT*. More precisely, regarding *CENT* the overhead is due to the communication between the components of Interaction Protocol layer and Conflict Resolution Protocol layer, since the centralized Conflict Resolution Protocol is placed in the central machine. Regarding *TR* the overhead is due to the communications between Conflict Resolution Protocol which depends on the

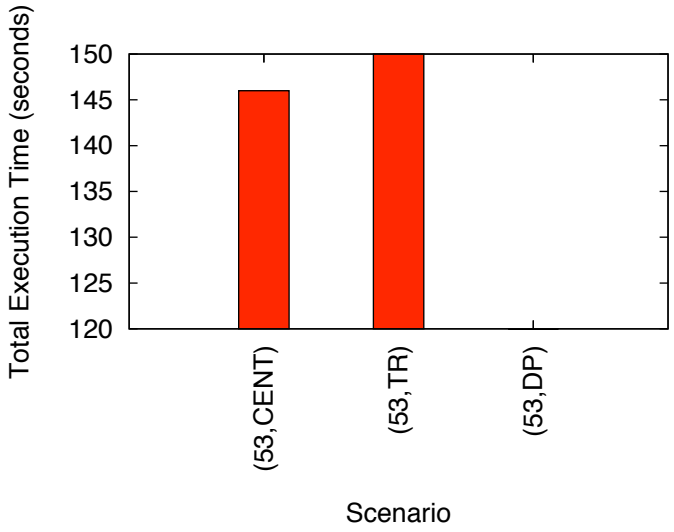


Figure 4.23: Performance of responding 10 request per calling unit ($49 = 7 \times 7$ calling units, and 4 cars).

number of components in this layer. To the contrary, in *DP*, the Conflict Resolution Protocol components act in their local neighborhood although more communication is needed.

Figure 4.23 also shows the same type of experiments by taking 4 cars and $49 = 7 \times 7$ calling units. The performance becomes worse for *TR* since the token has to travel a long way through the components of the Conflict Resolution Protocol layer.

4.10.3 Bitonic Sorting

In the two previous examples, the system itself is geographically distributed, hence the generation of a distributed implementation is necessary. However, in other case the generation of a distributed implementation is necessary for deriving more computational power by using multiple processors (e.g., sorting algorithm, genetic algorithm).

The aim of this example is to show that our methodology assists developers of parallel and multi-core applications to start developing from high-level BIP models and not get

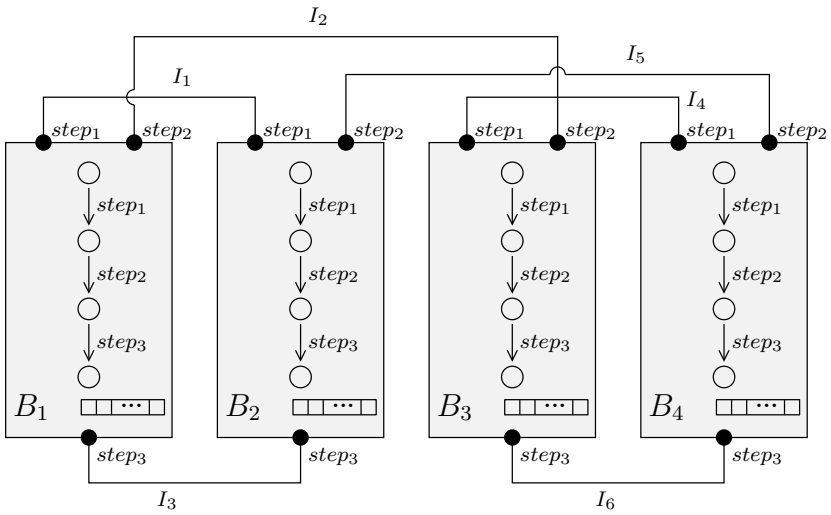


Figure 4.24: High-level BIP model for Bitonic Sorting Algorithm.

involved in low-level synchronization details.

Bitonic sorting [Bat68] is one of the fastest sorting algorithms suitable for distributed implementation in hardware or in parallel processor arrays. A sequence is called *bitonic* if it is initially nondecreasing then it is nonincreasing. The first step of the algorithm consists in constructing a bitonic sequence. Then, by applying a logarithmic number of bitonic merges, the bitonic sequence is transformed into totally ordered sequence. We provide an implementation of the bitonic sorting algorithm in BIP using four atomic components, each one handling one part of the array. These components are connected as shown in Figure 4.25. The six interactions are non conflicting. Moreover, interactions a_1, a_2 and a_3 cannot run in parallel. The same holds for interactions a_4, a_5, a_6. Thus, to obtain maximal parallelism between interactions it sufficient to create only two components for the Interaction Protocol layer. Where, the first one handles the interactions a_1, a_2 and a_3 and the second one handles the interactions a_4, a_5 and a_6. Furthermore, since all interactions are non conflicting, there is no need for the Conflict Resolution Protocol layer (detected automatically by the tool). In this example each component sends only three messages, each one containing its own array.

We run experiments for three configurations: $1c$, $4c$, $4c'$. For $1c$, we use one single-core machine, where the four atomic components along with the two *IP* components. For $4c$, we use two dual-core machines and place each atomic component on a different core. We also distribute the *IP* components over two cores, such as to reduce the network communication overhead. For $4c'$, we use the same distribution for components and *IP*. The results are reported in Table 4.1 for arrays of size $k \times 10^4$ elements, and $k = 20, 40, 80, 160$. As can be

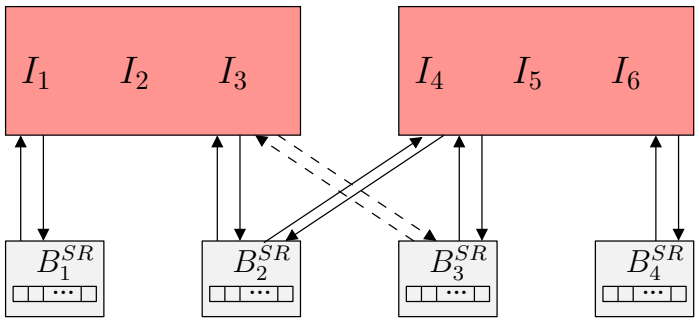


Figure 4.25: 3-layer Send/Receive BIP model for Bitonic Sorting Algorithm.

k	C++/Socket (generated)		
	$1c$	$4c$	$4c'$
20	96	23	24
40	375	96	100
80	1504	390	397
160	6024	1539	1583

Table 4.1: Performance of Bitonic Sorting Algorithm.

seen in Table 4.1

Table 4.1 shows the performance of the automatically C++ code generated using TCP sockets. It is clear that if we consider larger arrays, then increasing the number of cores leads to a proportional performance gain. For example, the execution time for sorting an array of size 160×10^4, for the configuration $4c$ is 1539 seconds, and for the configuration $1c$ is $6024 \approx 4 \times 1539$ seconds.

The performance of case $4c$ (2 computers with two cores each) configuration is shown in Table 4.2. Observe that the performance of the C++/Socket code is approximately identical in both cases. This is because socket operations are interrupt-driven. Thus, if a component is waiting for a message, it does not consume CPU time. On the other hand, MPI uses active waiting, which results in CPU time consumption when the *IP* is waiting. Since we have four cores for six processes, the MPI code generated from the original Send/Receive model is much slower than the socket code. Nevertheless, as it appears in the table, reducing the number of components to one per core by merging (see Figure 4.26) allows the MPI code to reach the same speed as in the C++/socket implementation.

k	S/R BIP		Merged S/R BIP	
	Socket	MPI	Socket	MPI
20	23	63	24	24
40	96	271	96	96
80	390	964	391	394
160	1539	4158	1548	1554

Table 4.2: The impact of component composition on Send/Receive models.

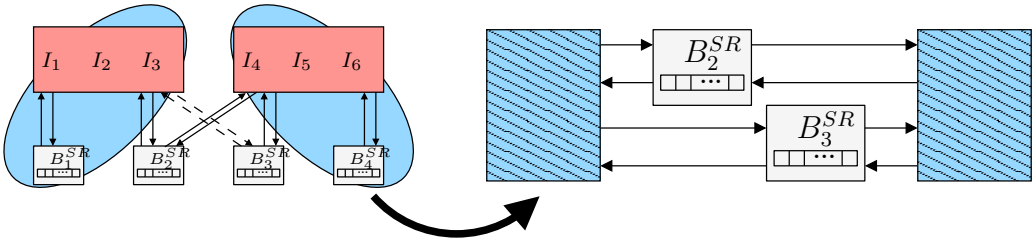


Figure 4.26: Component composition components in Bitonic Sorting 3-layer Send/Receive BIP model.

4.11 Summary

In this chapter we proposed a methodology for producing automatically efficient and correct-by-construction distributed implementations by starting from a high-level model of the application software in BIP. The methodology transforms arbitrary BIP models into Send/Receive BIP models, directly implementable on distributed execution platforms. The transformation consists of:

1. breaking atomicity of actions in atomic components by replacing strong synchronizations with asynchronous Send/Receive interactions;
2. inserting several distributed controllers that coordinate execution of interactions according to a user-defined partition;
3. augmenting the model with a distributed algorithm for handling conflicts between controllers.

We shown that the obtained Send/Receive BIP models are observationally equivalent to the initial models. Hence, all the functional properties are preserved by construction in the implementation. Moreover, Send/Receive BIP models can be used to automatically derive distributed implementations. Currently, it is possible to generate stand-alone C++ implementations using either TCP sockets for conventional communication, or MPI im-

plementation, for deployment on multi-core platforms. This method is fully implemented. We report concrete results obtained under different scenarios (i.e., partitioning of the interactions and choice of algorithm for distributed conflict resolution).
In the next chapter, we present the tool which implements the transformations presented in this thesis. Moreover, we give an overview of the integration of our tool in the design methodology for BIP for automatically deriving efficient centralized and distributed from high-level BIP models.

CHAPTER 5

Tool-Chain

Contents

5.1 BIP into Centralized Implementations Tool-Chain - BIP2BIP 107
5.2 BIP into Distributed Implementations Tool-Chain - BIP2Dist 109
5.3 Summary 109

5.1 BIP into Centralized Implementations Tool-Chain - BIP2BIP

The transformations from BIP into centralized implementations in Chapter 3 have been implemented in the BIP2BIP tool, which is currently integrated in the BIP toolset [BIP] as shown in Figure 5.1.

The BIP2BIP tool is written in *Java*. It allows transformation of parsed models. It contains the following modules implementing the presented transformations.

- Component flattening : this module transforms a composite component into an equivalent one consisting only of atomic components of the initial model and a set of connectors.
- Connector flattening : this module transforms an hierarchically structured connector into an equivalent flat one. By successive applications of this module, we obtain a new model with flat connectors.
- Component composition : this module transforms a set of atomic components and a set of flat connectors into an equivalent atomic component. By successive applications of this module, we obtain a new model consisting of a single atomic component.

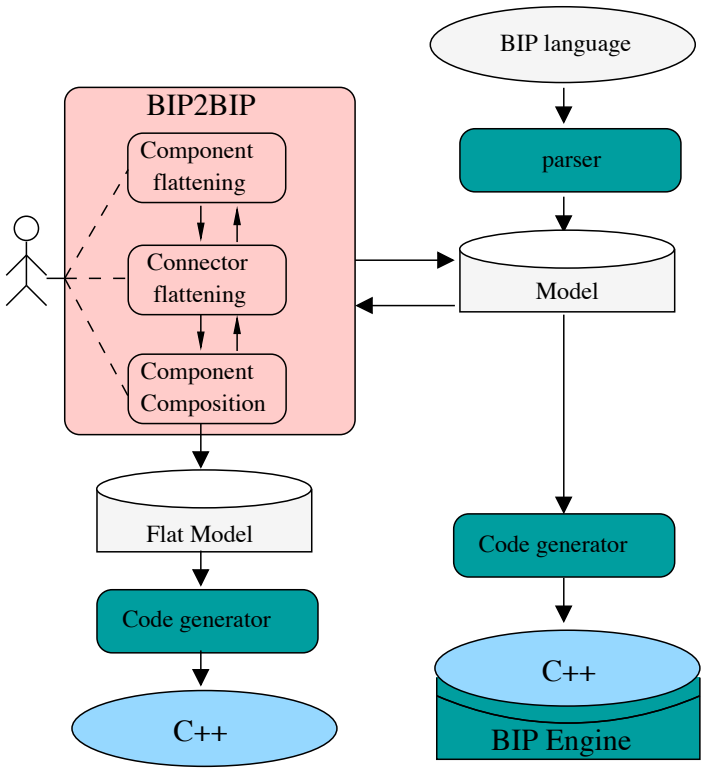


Figure 5.1: BIP2BIP toolset: General architecture

By exhaustive application of these transformations, an atomic component can be obtained. From the latter, the *code-generator* can generate standalone C++ code, which can be run directly without the Engine. In particular, all the remaining non-determinism in the final atomic component is eliminated at code generation by applying an implicit priority between transitions.
It should be noted that the transformations also can be applied independently, to obtain models that respond to a particular user needs. For example, one may decide to eliminate only partially the hierarchy of components, or to compose only some components.
The performance of BIP2BIP is quite satisfactory. For example, when applied to an artificially complex BIP model, consisting of 256 atomic components, composed by using 509 connectors with 7 levels of hierarchy, it takes less than 15 seconds to generate the corresponding C++ program.

5.2 BIP into Distributed Implementations Tool-Chain - BIP2Dist

The transformations from BIP into distributed implementations in Chapter 4 have been implemented in the BIP2Dist tool (see Figure 5.2), which is also integrated in the BIP toolset.

The BIP2Dist tool is written in *Java*. It allows the generation of distributed implementation starting from a high-level BIP model. In the follows, we will describe the design process for that generation using BIP2Dist tool. We illustrate the process on the example shown in Figure 5.3.

1. Starting from a high-level BIP model, we flatten the hierarchy of connectors and components;
2. From the flatten model that only consists of atomic components and flat connectors, we generate 3-layer Send/Receive BIP model by choosing a partition of interactions and a Conflict Resolution Protocol;
3. From the 3-layer Send/Receive BIP model, a designer may merge some components by choosing a partition of component to merge;
4. Finally, from the obtained model we generate C++ code using TCP socket or MPI for communication according to designer demand. Moreover, this step takes as input a mapping of the components over a distributed target platform.

5.3 Summary

We have provided an overview on the implementation of the BIP2BIP and BIP2Dist tool for generating centralized and distributed implementations from BIP models. In the next chapter, we conclude the thesis with an overview of the work and its future perspectives.

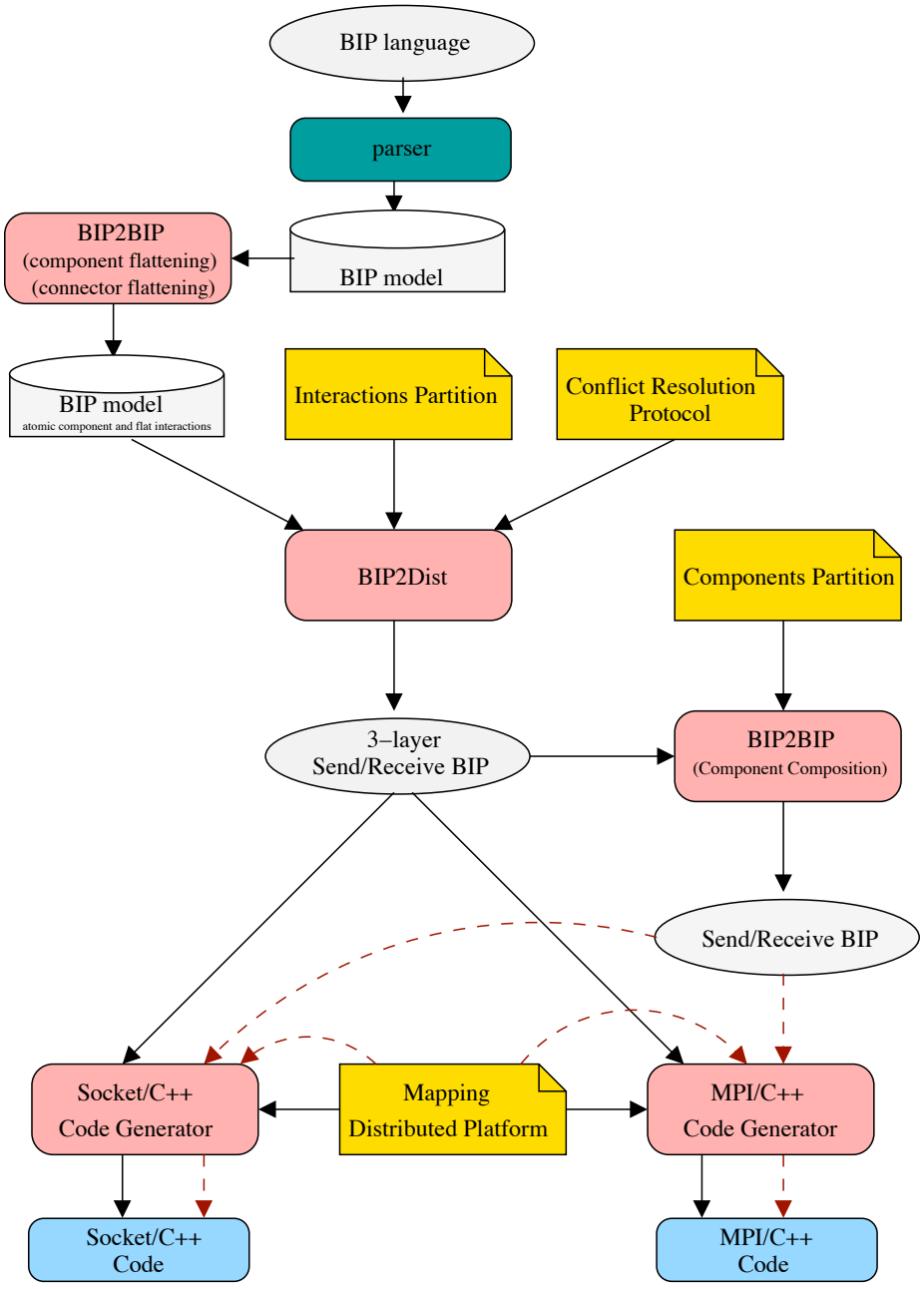


Figure 5.2: BIP2Dist toolset: General architecture

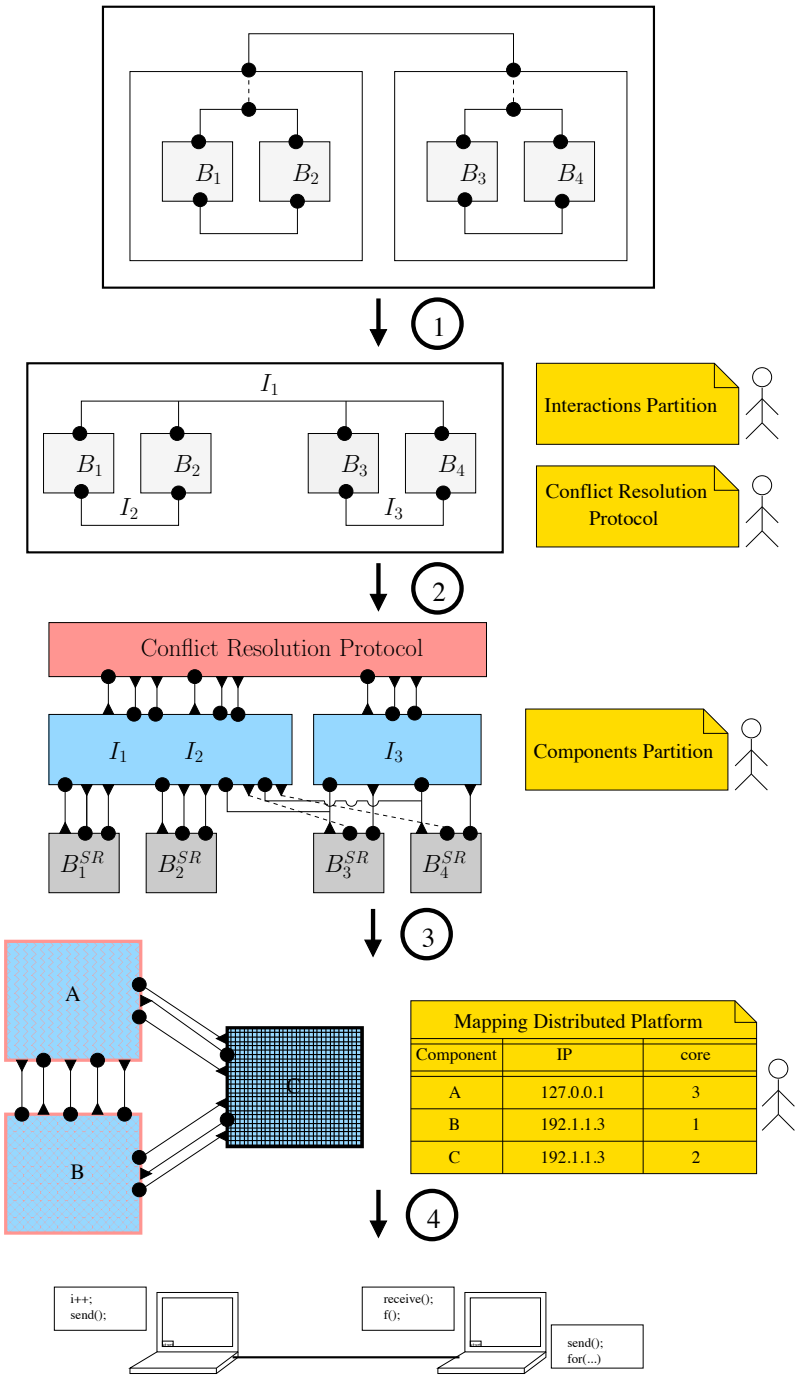


Figure 5.3: Design process to automatically generate distributed implementations

CHAPTER 6

Conclusions and Perspectives

Contents

6.1 Conclusions 113
6.2 Perspectives 115

In this chapter, we conclude the thesis describing the main objectives of the work, the goals we have achieved, the future work directions and its perspectives.

6.1 Conclusions

The thesis shows that it is possible to reconcile component-based incremental design and efficient code generation by applying a paradigm based on the combined use of:

1. A high-level modelling language, BIP, based on well-defined operational semantics and supporting powerful mechanisms for expressing structured coordination between components. The design methodology using BIP language involves the following steps:
   (a) The system (software) to be designed is decomposed into components. The decomposition can be represented as a tree which shows how the system can be obtained as the incremental composition of components. Its root is the system and its leaves correspond to atomic components;
   (b) Description of the behavior of the atomic components;
   (c) Description of composite components as the composition of atomic components by using only connectors and priorities.

This is possible because BIP is expressive enough for describing any kind of coordination by using only architectural constraints [BS08b].

Along steps b) and c) it is possible by using the D-Finder tool, to generate and/or check invariants of the components and validate their properties. The methodology provides sufficient conditions for preserving the already established properties of the sub-systems along the construction.

BIP has already successfully been used for the componentization of non trivial systems such as the controller of the DALA robot [BGL+08]. This allowed building component-based models for which enhanced analysis and verification is possible by using tools such as D-Finder [BBNS09, BBSN08] for compositional verification.

2. Semantics-preserving source-to-source transformations that allows to generate automatically efficient centralized or distributed implementations. We have developed two implementation methods for BIP, sequential and distributed, which target respectively single-processor or multi-processor execution platforms.
   - *centralized implementations:* we defined a set of source-to-source transformations that progressively transform architectural constraints between components into internal computation of product components. These transformations include flattening of hierarchical compositions and hierarchical connectors and also static composition of atomic behavior. The aim of these transformation is to transform a composite component into a single atomic component. From the latter an efficient C++ code can be generated. We show that these transformations are semantic preserving and moreover, when used in the implementation flow, they reduce overheads in execution time by reducing modularity introduced by the designer when it is not necessary at implementation level.
   - *distributed implementations:* we defined a set of source-to-source transformations that generate automatically distributed implementation from the high-level models specified in BIP. Although BIP provides a rich set of interactions, we only considered *rendezvous* interactions, as they play an important role in systems whose constituents need to synchronize on some event in order to start some computation. In a distributed setting, implementation of a multi-party rendezvous results in solving the *committee coordination problem* [CM88], where a set of professors are organized in a set of committees and two committees can meet concurrently only if they have no professor in common; i.e., they are not *conflicting*. Conflict resolution is the main obstacle in distributed implementation of multi-party rendezvous interactions.

     Our transformation consists of two steps. First, it takes as input a BIP model in terms of a set of components glued by rendezvous interactions and generates another BIP model which contains component glued by *Send/Receive* interactions in the following three layers: (1) the *Atomic Components* layer consists of a transformation of behavioral components in the original model, (2) the *Interaction Protocol*

layer detects enabledness of interactions of the original model and executes them after resolving conflicts either locally or by the help of the third layer, and (3) the *Conflict Resolution Protocol* layer resolves conflicts unresolved by the interaction protocol. The Conflict Resolution Protocol implements a committee coordination algorithm and our design allows employing any such algorithm. The second step of our transformation takes the intermediate three-layer BIP model as input and generates C++ executables using either TCP sockets or MPI for communications. We conducted several experiments using different algorithms in the Conflict Resolution Protocol. As predicated, our experiments show that each algorithm is suitable for a different topology, size of the distributed system, communication load, and of course the structure of the initial high-level model. Thus, the important lesson learned from our experiments is that there is no silver bullet to automate code generation for distributed applications and designers must have access to a formal framework and a library of algorithms to be able to develop correct and yet efficient distributed applications.

6.2 Perspectives

For future work, we are considering several research directions.

- According centralized and distributed implementations we plan to take into account priorities. Concerning centralized implementations, priority rules can be compiled in the form of restrictions of the guards of components. On the other hand, according to distributed implementations, we agree that priorities complicate the problem, as unlike conflict-resolution, priorities must be applied globally which requires approaches such computing a global snapshot. The rest of the following future works are according to distributed implementations;
- Another direction is introducing the notion of time in distributed semantics of BIP. Providing timing guarantees in a distributed setting has always been a challenge and BIP is not an exception;
- An important extension to explore is to allow the Conflict Resolution Protocol to incorporate different algorithms for conflict resolution simultaneously. This is because each set of conflicting interactions within the same system may react differently to different algorithms and, hence, it is desirable to handle each set with the algorithm that performs the best. In this context, we are also planning to develop and implement other algorithms, such as solutions to distributed graph matching and distributed independent set. Moreover, we need to have a better understanding of tradoffs between parallelism within the components layer and the Interaction Protocol, load balancing, and network traffic;
- Another important line of research is to measure the overhead of our transformation technique as compared to hand-written code. To this end, we plan to design

customized techniques and conduct experiments in large sensor networks where communication and computation tradoffs play an important role in efficiency and energy consumption and the network cannot afford incorporating solutions that add significant overhead. Another potential avenue for our work is large peer-to-peer networks;
– Finally, given the recent advances in the multi-core technology, we plan to customize our transformation for multi-core platforms as well. In these platforms, network communication can be replaced by simple inter-process communication and one can investigate whether it is possible to devise more effective techniques to achieve correct-by-construction concurrency and process synchronization.

Bibliography

[AHJ+09] Matthieu Anne, Ruan He, Tahar Jarboui, Marc Lacoste, Olivier Lobry, Guirec Lorant, Maxime Louvel, Juan Navas, Vincent Olive, Juraj Polakovic, Marc Poulhies, Jacques Pulou, Stephane Seyvoz, Julien Tous, and Thomas Watteyne, *Think: View-based support of non-functional properties in embedded systems*, Embedded Software and Systems, Second International Conference on **0** (2009), 147–156.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi, *Sorting in c log n parallel steps*, Combinatorica **3** (1983), no. 1, 1–19.

[AVCL02] Robert Allen, Steve Vestal, Dennis Cornhill, and Bruce Lewis, *Using an architecture description language for quantitative analysis of real-time systems*, WOSP '02: Proceedings of the 3rd international workshop on Software and performance (New York, NY, USA), ACM, 2002, pp. 203–210.

[Bag87] Rajive Bagrodia, *A distributed algorithm to implement n-party rendevouz*, Foundations of Software Technology and Theoretical Computer Science, Seventh Conference (FSTTCS), 1987, pp. 138–152.

[Bag89] R. Bagrodia, *Process synchronization: Design and performance evaluation of distributed algorithms*, IEEE Transactions on Software Engineering (TSE) **15** (1989), no. 9, 1053–1065.

[Bas08] Ananda Shankar Basu, *Modélisation à base de composants de systèmes temps rél h´et´erog'enes en bip*, PhD thesis, UJF, 2008.

[Bat68] K. E. Batcher, *Sorting networks and their applications*, AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference, 1968, pp. 307–314.

[BBBS08] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis, *Distributed semantics and implementation for systems with interaction and priority*, FORTE (Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, eds.), Lecture Notes in Computer Science, vol. 5048, Springer, 2008, pp. 116–133.

[BBJ+10a] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeauf, and Joseph Sifakis, *Automated conflict-free distributed implementation of component-based models*, SIES, 2010.

[BBJ+10b] ———, *From high-level component-based models to distributed implementations*, EMSOFT, 2010, (submitted).

[BBNS09] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis, *D-finder: A tool for compositional deadlock detection and verification*, CAV (Ahmed Bouajjani and Oded Maler, eds.), Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 614–619.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis, *Modeling heterogeneous real-time components in BIP*, 4^{th} IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM06), September 2006, Invited talk, pp. 3–12.

[BBSN08] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen, *Compositional verification for component-based systems and application*, ATVA (Sung Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, eds.), Lecture Notes in Computer Science, vol. 5311, Springer, 2008, pp. 64–79.

[BCF02] Nick Benton, Luca Cardelli, and Cédric Fournet, *Modern concurrency abstractions for* $c^{\#}$, ECOOP (Boris Magnusson, ed.), Lecture Notes in Computer Science, vol. 2374, Springer, 2002, pp. 415–440.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani, *The fractal component model and its support in java*, Softw., Pract. Exper. **36** (2006), no. 11-12, 1257–1284.

[BCS02] E. Bruneton, T. Coupaye, and J. B. Stefani, *Recursive and dynamic software composition with sharing*, 2002.

[BFL+04] Roberto Bruni, José Luiz Fiadeiro, Ivan Lanese, Antónia Lopes, and Ugo Montanari, *New insights on architectural connectors*, IFIP TCS (Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, eds.), Kluwer, 2004, pp. 367–380.

[BGK+06] Krishnakumar Balasubramanian, Aniruddha S. Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema, *Developing applications using model-driven design environments*, IEEE Computer **39** (2006), no. 2, 33–40.

[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, ECAI (Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, eds.), Frontiers in Artificial Intelligence and Applications, vol. 178, IOS Press, 2008, pp. 631–635.

[BGO+04] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis, *The if toolset*, SFM (Marco Bernardo and Flavio Corradini, eds.), Lecture Notes in Computer Science, vol. 3185, Springer, 2004, pp. 237–267.

[BIP] *BIP*, http://www-verimag.imag.fr/~async/bip.php.

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

[BMFT07] Richard Vincent Bennett, Alastair Colin Murray, Björn Franke, and Nigel P. Topham, *Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems*, LCTES (Santosh Pande and Zhiyuan Li, eds.), ACM, 2007, pp. 83–92.

[BS07] Simon Bliudze and Joseph Sifakis, *The algebra of connectors: structuring interaction in bip*, EMSOFT (Christoph M. Kirsch and Reinhard Wilhelm, eds.), ACM, 2007, pp. 11–20.

[BS08a] ———, *The algebra of connectors - structuring interaction in bip*, IEEE Trans. Computers **57** (2008), no. 10, 1315–1330.

[BS08b] ———, *A notion of glue expressiveness for component-based systems*, CONCUR (Franck van Breugel and Marsha Chechik, eds.), Lecture Notes in Computer Science, vol. 5201, Springer, 2008, pp. 508–522.

[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** (2003), no. 4, 45–52.

[CFLS05a] J. Combaz, J-C. Fernandez, T. Lepley, and J. Sifakis, *QoS Control for Optimality and Safety*, Proceedings of the 5th Conference on Embedded Software, September 2005.

[CFLS05b] J. Combaz, J.C. Fernandez, T. Lepley, and J. Sifakis, *Fine grain QoS control for multimedia application software*, Design, Automation and Test in Europe (DATE'05) Volume 2, 2005, pp. 1038–1043.

[CFSS07] Jacques Combaz, Jean-Claude Fernandez, Joseph Sifakis, and Loic Strus, *Using speed diagrams for symbolic quality management.*, IPDPS, IEEE, 2007, pp. 1–8.

[CKL+02] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Claudio Passerone, and Yosinori Watanabe, *Quasi-static scheduling of independent tasksfor reactive*

systems, ICATPN (Javier Esparza and Charles Lakos, eds.), Lecture Notes in Computer Science, vol. 2360, Springer, 2002, pp. 80–100.

[CKL+05] ———, *Quasi-static scheduling of independent tasks for reactive systems*, IEEE Trans. on CAD of Integrated Circuits and Systems **24** (2005), no. 10, 1492–1514.

[CM84] K. M. Chandy and J. Misra, *The drinking philosophers problem*, ACM Transactions on Programming Languages and Systems (TOPLAS) **6** (1984), no. 4, 632–646.

[CM88] ———, *Parallel program design: a foundation*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[CRBS08] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis, *Translating aadl into bip - application to the verification of real-time systems*, MoDELS Workshops (Michel R. V. Chaudron, ed.), Lecture Notes in Computer Science, vol. 5421, Springer, 2008, pp. 5–19.

[DDM+07] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu, *A next-generation design framework for platform-based design*, DVCon 2007, February 2007.

[DII+99] John Davis, II, John Davis Ii, Mudit Goel, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, and Yuhong Xiong, *Ptolemy ii: Heterogeneous concurrent modeling and design in java*, 1999.

[DS80] E. W. Dijkstra and C. S. Scholten, *Termination detection for diffusing computations*, Information Processing Letters **11** (1980), no. 1, 1–4.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong, *Taming heterogeneity - the ptolemy approach*, Proceedings of the IEEE **91** (2003), no. 1, 127–144.

[FAH99] Eric Freeman, Ken Arnold, and Susanne Hupfer, *Javaspaces principles, patterns, and practice*, Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu, *Cadp - a protocol validation and verification toolbox*, CAV (Rajeev Alur and Thomas A. Henzinger, eds.), Lecture Notes in Computer Science, vol. 1102, Springer, 1996, pp. 437–440.

[FLN+03] Marcus Fontoura, Tobin J. Lehman, Dwayne Nelson, Thomas Truong, and Yuhong Xiong, *Tspaces services suite: Automating the development and management of web services*, WWW (Alternate Paper Tracks), 2003.

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, and Gilles Muller, *Think: A software framework for component-based operating system kernels*, 2002.

[GLvB+03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, *The NesC language: A holistic approach to networked embedded systems*, SIGPLAN Conference on Programming Language Design and Implementation, 2003.

[GS04] David Garlan and Bradley R. Schmerl, *Using architectural models at runtime: Research challenges*, EWSA (Flávio Oquendo, Brian Warboys, and Ronald Morrison, eds.), Lecture Notes in Computer Science, vol. 3047, Springer, 2004, pp. 200–205.

[GS05] Gregor Gößler and Joseph Sifakis, *Composition for component-based modeling*, Sci. Comput. Program. **55** (2005), no. 1-3, 161–183.

[HG06] Benjamin Hindman and Dan Grossman, *Atomicity via source-to-source translation*, Memory System Performance and Correctness (Antony L. Hosking and Ali-Reza Adl-Tabatabai, eds.), ACM, 2006, pp. 82–91.

[HS02] Franz Huber and Bernhard Schätz, *Integrated development of embedded systems with*, 2002.

[JBB09] Mohamad Jaber, Ananda Basu, and Simon Bliudze, *Symbolic implementation of connectors in bip*, CoRR **abs/0911.5446** (2009).

[JHRC08] Lei Ju, Bach Khoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty, *Performance debugging of esterel specifications*, CODES+ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis (New York, NY, USA), ACM, 2008, pp. 173–178.

[Lov77] David B. Loveman, *Program improvement by source-to-source transformation*, J. ACM **24** (1977), no. 1, 121–145.

[Mat] *Mathworks*, http://www.mathwork.com.

[Mil95] R. Milner, *Communication and concurrency*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

[Mil98] Robin Milner, *The pi calculus and its applications*, JICSLP'98: Proceedings of the 1998 joint international conference and symposium on Logic programming (Cambridge, MA, USA), MIT Press, 1998, pp. 3–4.

[OMG] *Unified Modeling Language UML*, Object Management Group, 2009, http://www.uml.org/.

[Pan01] Preeti Ranjan Panda, *Systemc*, ISSS, 2001, pp. 75–80.

[PCT04] J. A. Pérez, R. Corchuelo, and M. Toro, *An order-based algorithm for implementing multiparty synchronization*, Concurrency and Computation: Practice and Experience **16** (2004), no. 12, 1173–1206.

[Pou10] Marc Poulhiès, *Conception et implantation de système fondé sur les composants. vers une unification des paradigmes génie logiciel et système*, PhD thesis, UJF, 2010.

[PPRS06] Marc Poulhiès, Jacques Pulou, Christophe Rippert, and Joseph Sifakis, *A methodology and supporting tools for the development of component-based embedded systems*, Monterey Workshop (Fabrice Kordon and Oleg Sokolsky, eds.), Lecture Notes in Computer Science, vol. 4888, Springer, 2006, pp. 75–96.

[RC03] Arnab Ray and Rance Cleaveland, *Architectural interaction diagrams: Aids for system modeling*, ICSE '03: Proceedings of the 25th International Conference on Software Engineering (Washington, DC, USA), IEEE Computer Society, 2003, pp. 396–406.

[RHG+01] J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, and W. Mueller, *The simulation semantics of systemc*, DATE '01: Proceedings of the conference on Design, automation and test in Europe (Piscataway, NJ, USA), IEEE Press, 2001, pp. 64–70.

[Sif05] Joseph Sifakis, *A framework for component-based construction extended abstract*, SEFM (Bernhard K. Aichernig and Bernhard Beckert, eds.), IEEE Computer Society, 2005, pp. 293–300.

[STS+10] V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis, *Compositional translation of simulink models into synchronous bip*, SIES, 2010.

[VPL99] James Vera, Louis Perrochon, and David C. Luckham, *Event-based execution architectures for dynamic software systems*, WICSA (Patrick Donohoe, ed.), IFIP Conference Proceedings, vol. 140, Kluwer, 1999, pp. 303–318.