Component-based Modeling of Heterogeneous Real-time Systems in BIP

Ananda Basu

Université Joseph Fourier – Grenoble 1

T H È S E

pour obtenir le grade de

Docteur de l'Université Joseph Fourier

Spécialité : Informatique

préparée au laboratoire Verimag

dans le cadre de l'École Doctorale (Mathématiques et Informatique)

présentée et soutenue publiquement par

Ananda Shankar Basu

le 15 December 2008

Modélisation à base de Composants de Systèmes Temps réel Hétérogènes en BIP (Component-based Modeling of Heterogeneous Real-time Systems in BIP)

JURY

Président	Jean Bernard Stefani	INRIA
Rapporteurs	Janos Sztipanovits	University of Vanderbuilt, USA
	Mathai Joseph	Tata Consultancy Services, India
Examinateurs	Lothar Thiele,	ETH, Zurich
	Marc Pouzet	LRI, Paris Sud
	Marius Bozga,	Verimag
Directeur de thése	Joseph Sifakis,	Verimag

Contents

Preface

“Saraswati Mahabhaage Vidye Kamalalochane Vishwaroope Vishaalaakshi Vidyam dehi namosthuthe” (O, the great Goddess Saraswati, thou shower us with all the powers and glories of all knowledge that exist).

Part I

Concepts and Methodology

Chapter 1

Introduction

A central idea in systems engineering is that complex systems are built by assembling components. Components are building blocks, and the concept of component-based modeling is common to all engineering disciplines. Components are usually characterized by abstractions that ignore implementation details and describe properties relevant to their composition, e,g., transfer functions, interfaces. It is possible to get larger components by ”gluing” together simpler ones. Gluing or composition can be formalized as an operation that takes in components and their integration constraints. From these, it provides a description of a new, more complex composite component.

System designers deal with a large variety of components, each having different characteristics, from a large variety of viewpoints, each highlighting different dimensions of a system. A central problem is the meaningful composition of heterogeneous components to ensure their correct interoperation [HS06].

One fundamental source of heterogeneity is the composition of subsystems with different execution and interaction semantics. At one extreme of the semantic spectrum are fully synchronized components, which proceed in lockstep with a global clock and interact in atomic transactions. Such a tight coupling of components is the standard model for most synthesizable hardware and for synchronous real-time software. At the other extreme are completely asynchronous components, which proceed at independent speeds and interact non-atomically. Such a loose coupling of components is the standard model for most multi-threaded software. Between the two extremes, a variety of intermediate and hybrid models can be defined (e.g., globally-asynchronous locally-synchronous models).

Another fundamental source of heterogeneity is the use of models that represent a system at varying degrees of detail and are related to each other in an abstraction (or equivalently, refinement) hierarchy. A key abstraction in system design is the one relating application software to its implementation on a given platform. Application software is largely untimed, in the sense that it abstracts away from physical time. The application code running on a given platform, however, is a dynamic system that can be modeled as a timed or hybrid automaton [ACH$^+$95]. The runtime state includes not only the variables of the application software, but also all variables that are needed to characterize its dynamic behavior, such as time variables and other quantities used to model resources. We need tractable theories to relate component-based models at application and implementation levels. In particular, such theories must provide means for preserving, in the implementation, all essential properties of the application software.

Unified frameworks encompassing heterogeneity in systems design have been developed through tools like Ptolemy [EJL$^+$03] and Metropolis [BWH$^+$03]. Other modeling paradigms for unifying interaction in heterogeneous systems have been proposed in [BGK$^+$06, Arb05]. Nevertheless, in these works unification is achieved by reduction to a common low-level semantic model. Interaction mechanisms and their properties are not studied independently of behavior. We need a framework which is not just a disjoint union of sub-models, but one which preserves properties during model composition and supports meaningful analysis and transformations across heterogeneous model boundaries.

1.1 State of the Art

In this section, we provide a brief description of the current state of the art of component-based technology. We analyze the existing technology for different domains encompassing hardware, software and middleware. We see that component-based engineering is widely used in VLSI circuit design methodologies, and is supported by a large number of tools. Software component-based techniques have seen significant development, especially through the use of object technologies supported by languages such as C++, Java, and standards such as UML and CORBA. However, these techniques have not yet achieved the same level of maturity as has been the case for hardware. There exists a large body of literature dealing with components and their use for different purposes and in different context.

The following deal, one way or another, with issues related to component-based engineering:

- Software Design Description Languages [GS04, BFLL04], and Architecture Description Languages focusing on non-functional aspects [VPL99, AVCL02].

- System modeling languages such as UML [Sel04], as well as languages and notations specific to tools such as Simulink/Stateflow, SystemC

[RHG+01], Statecharts, Metropolis [BWH+03], Ptolemy [Lee03, BHLM02, EJL+03], IF-toolset [BGO+04] and PROMETHEUS tool [Gößl01].

The component framework proposed in this thesis extends and improves the ideas of the IF-toolset and PROMETHEUS tool.

The IF toolset [BGO+04], developed at Verimag, is one amongst the well known platforms for component modeling and validation. 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 analyzed using the tool suite CADP [JHA+96], including the minimization and comparison tool ALDEBARAN based on bisimulation, and the alternating-free μ-calculus model-checker EVALUATOR.

The PROMETHEUS tool [Gößl01], a prototype of a compositional modeling tool for real-time systems, was also developed at Verimag. It is based on priority functions as a model for coordination between processes. It also provides a high-level modeling language for specifying real-time processes as well as a scheduler. The language is sufficiently general to specify most frequently used scheduling policies. The possibility of defining and instantiating scheduler templates allow to establish a library of schedulers. Process templates simplify the specification of multiple occurrences of a process type with the same untimed transition structure, but different timing constraints. It uses IF as an intermediate format for timed asynchronous systems and integrates tools operating on different levels of abstraction.

Other developments in component-based modeling and programming technology are:

- Component models based on classical concepts of Component-Based Software Engineering (CBSE) like FRACTAL [Fra] and its implementations, e.g., THINK [FSLM02].
- Coordination language extension of programming languages such as Linda, Javaspaces, TSpaces, Concurrent Fortran, nesC [GLvB+03] and Polyphonic C$^\sharp$ [BCF02].
- Middleware standards such as IDL, Corba, Javabeans, .NET, RMI.
- Software development environments such as PCTE. SWbus, Softbench, Eclipse.
- Theoretical frameworks based on process algebras e.g., the Pi-Calculus [Mil98] or based on automata e.g., [RC03].

There is a difference between the notion of component in software engineering and the notion of component in hardware engineering. In the former, communication between the components are point to point, by function calls. Conventional function calls are blocking, in the sense that the caller makes no progress until the callee completes. In addition, languages like NesC allows joint function calls, and Polyphonic C$^\sharp$ offers declaration of asynchronous methods and synchronization patterns, allowing two or more methods to synchronize. However, in software models, the interconnect between the components is not always easy to determine due to polymorphism. The execution of the behavior of the components is made in the context of the threads, and components do not have any proper activity. The inability to statically determine the component interconnections and the thread module of execution leads to reduced analyzability of software models.

In hardware models, components are concurrent, have their own activity, and communication is through dedicated channels. The execution is inherently synchronous.

One of the requirements is to have a platform that encompasses the heterogeneous features of both paradigms, with clear separation between component behavior and interconnect.

1.2 Shortcomings

There are different requirements for a component-based construction methodology. Firstly, it has to be founded on rigorous semantics and provide concepts supporting separation of concerns, e.g., decoupling behavior from interaction. 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.

Secondly, it needs to encompass heterogeneous descriptions, as most of the platforms and languages previously mentioned, support specific interaction mechanisms and computation models. For instance, software design frameworks are based on interaction by method call and do not allow direct modeling of atomic interaction mechanisms. On the contrary, other frameworks such as SystemC and Matlab/Simulink have built in mechanisms for synchronous execution, and are not adequate for describing asynchronous systems.

Thirdly, it also needs to encompass the description of timing and resource management, which are essential for non functional properties. Standards such as UML and AADL offer only syntactic sugar for time and scheduling policies. The lack of adequate semantic frameworks does not allow checking for inconsistency in timing requirements, or in the meaningful composition of scheduling policies.

And finally, a component-based construction framework should consider architectures as first class entities. The existing theoretical frameworks are too low level, since they only emphasize on behavioral aspects.

To meet the above requirements, we need component frameworks encompassing *heterogeneity* and allowing *constructivity* along the design process. We explain these concepts below.

1.2.1 Encompassing Heterogeneity

Heterogeneity is the property of systems built from components with different characteristics. Heterogeneity has several sources and manifestations, and the existing body of knowledge is largely fragmented into unrelated models and corresponding results.

System designers deal with a large variety of components, each having different characteristics. Two central problems are the meaningful composition of heterogeneous components to ensure their correct inter-operation, and the meaningful refinement and integration of heterogeneous viewpoints during the design process. For this, we need semantic frameworks encompassing heterogeneous composition. Superficial classifications may distinguish between hardware and software components, or between continuous-time (analog) and discrete-time (digital) components, but heterogeneity has two more fundamental sources: the composition of subsystems with different execution and interaction semantics; and the abstract view of a system from different perspectives.

Heterogeneity of Interaction Interactions are combinations of actions performed by system components in order to achieve a desired global behavior. Interactions can be *atomic* or *non-atomic*. For atomic interactions, the state change induced in the participating components cannot be altered through interference with other interactions. As a rule, synchronous programming languages and hardware description languages use atomic interactions. By contrast, languages with buffered communication (e.g., SDL) and multi-threaded languages (e.g., Java) generally use non-atomic interactions.

Both atomic and non atomic interaction may involve strong or weak synchronization. Strongly synchronizing interactions can occur only if all participating components agree (e.g., CSP rendezvous). Weakly synchronizing interactions are asymmetric; they require only the participation of an initiating action, which may or may not synchronize with other actions (e.g., outputs in Esterel [BC85]).

Heterogeneity in interactions may also arise due to the different number of participants. Interactions can be binary (point to point) or n-ary for $n \geq 3$. Interactions in CCS and SDL, function calls in most programing languages and message passing through channels are typical examples of binary interactions, while some high level modeling languages/platforms allow for n-ary synchronizations,e.g., Polyphonic C$^\sharp$ [BCF02]. The implementation of n-ary interactions by using binary interaction primitives is a non-trivial problem.

Clearly, there exists no formalism supporting directly all these types of interaction.

Heterogeneity of Execution Presently, there is a lack of formalisms encompassing both synchronous and asynchronous execution. Synchronous execution is typically used in hardware, in synchronous programming languages, and in time-triggered systems. It considers that a system's execution is a sequence of global steps. It assumes synchrony, meaning that the environment does not change during a step, or equivalently, that the system is infinitely faster than its environment. In each execution step, all the system components contribute by executing some quantum of computation. The synchronous execution paradigm, therefore, has a built in strong assumption of fairness: in each step all components can move forward.

Asynchronous execution, by contrast, does not use any notion of global computation step. It is adopted in most distributed systems description languages such as SDL [IT00] and UML [Sel04], and in multi threaded programming languages such as ADA [ADA] and Java [Jav]. The lack of built in mechanisms for sharing computation between components can be compensated through scheduling mechanisms, e.g., priorities.

Heterogeneity of Abstraction System development involves the use of languages, models and physical implementations, representing a system and its components at different abstraction levels. For heterogeneity, a key abstraction is the one relating an application software to its implementation on a given platform.

Application software is *untimed* in the sense that it abstracts out physical time. The only references to physical time are time parameters of real time statements, such as timeouts and watchdogs. The expiration of watchdogs or timeouts is treated at the semantic level as an external event. The application code running on a given platform, however, is a dynamic system that can be modeled as a timed or hybrid automaton [Hen96]. The runtime state includes not only the variables of the application software, but also all variables that are needed to characterize its dynamic behavior such as time, quantity of resources e.g., memory and power. We need abstractions and theory relating application software to its implementations. In particular, such abstractions should guarantee the preservation of all essential properties of the application software.

1.2.2 Achieving Constructivity

Constructivity is the possibility to build complex systems that meet given requirements, from building blocks and glue components with known properties. Constructivity can be achieved by algorithms (compilation and synthesis), and also by architectures and design disciplines. In principle, component-based frameworks should allow inferring system properties from properties of their structure. Currently, most of the existing validation techniques e.g, model-checking, need the construction of global models. We need theory, methods and tools for establishing, by construction, overall system correctness from component properties.

For dealing with heterogeneous systems, we need results in two complementary directions. First, we need construction methods for specific, restricted application contexts characterized by particular types of requirements and constraints, and/or by particular types of components and composition mechanisms. Clearly, hardware synthesis techniques, software compilation techniques, algorithms (e.g., for scheduling, mutual exclusion, clock synchronization), architectures (such as time-triggered; publish-subscribe), as well as protocols (e.g.,for multimedia synchronization) contribute solutions for specific contexts.

Second, we need theories that allow the incremental combination of the above results in a systematic process for system construction. Such theories would be particularly useful for the integration of heterogeneous models, because the objectives for individual subsystems are most efficiently accomplished within those models which most naturally capture each of these subsystems. As explained in this section, we need theory allowing constructivity and meeting the following requirements:

Incrementality This means that composite systems can be considered as the composition of smaller parts. Incrementality is necessary for progressive analysis and the application of compositionality rules.

Compositionality Compositionality rules allow inferring global system properties from the local properties of the components. An example is inferring global deadlock-freedom from the deadlock freedom of the individual components.

Composability Composability rules guarantee that a component's essential properties are not affected during the system construction process, i.e., even after gluing together the components, their essential individual properties are preserved. Composability means stability of component properties across integration, e.g., establishing noninterference for two scheduling algorithms used to manage two system resources.

1.3 Key Issues of Component-Based Construction

On exploring the current state of the art, and analyzing their shortcomings, it becomes clear that a component-based framework needs the following:

- Rigorous and general basis for real-time system design and implementation.
- Concept of component and associated composition operators for incremental description and correctness by construction.
- Upliftment from frameworks based on single composition operator to those with families of composition operators.
- Enhanced expressiveness for modeling co-ordination mechanism such as protocols, schedulers, buses.
- Concept for real-time architecture encompassing heterogeneous paradigms and styles of computation e.g., synchronous vs. asynchronous execution; event driven vs. data driven computation; distributed vs. centralized execution.
- Automated support for component integration and generation of glue code meeting given requirements.

1.4 Our Contribution

We present in this thesis, the BIP component framework. The name BIP is derived from **B**ehavior, **I**nteraction and **P**riority, the three main foundations of this framework. BIP serves for modeling heterogeneous real-time components, and integrates results developed at Verimag over the past five years.

Its main characteristics are the following:

- It supports a component construction methodology based on the thesis that components are obtained as the superposition of three layers. The lower layer contains atomic components described by their *behavior*. The intermediate layer includes a set of *connectors* describing the *interactions* between transitions of the behavior. The upper layer is a set of *priority* rules describing scheduling policies for interactions. Layering implies a clear separation between *behavior* and *structure* (connectors and priority rules).
- It uses a parameterized *composition* operator on components. The product of two components consists in composing their corresponding layers separately. Parameters are used to define new interactions as well as new priority rules between the composed components [GS05, Sif05]. The use of such a composition operator allows *incremental* construction. That is, any compound component can be obtained by successive composition of its constituents. This is a generalization of the associativity/commutativity property for composition operators whose parameters depend on the order of composition.

- It encompasses heterogeneity. It provides a powerful mechanism for structuring interactions involving strong synchronization (rendezvous) or weak synchronization (broadcast). Synchronous execution is characterized as a combination of properties of the three layers. Finally, timed components can be obtained from untimed components by applying a structure preserving transformation of the three layers.

- It allows considering the *system construction* process as a sequence of transformations in a three-dimensional space: $Behavior \times Interaction \times Priority$. A transformation is the result of the superposition of elementary transformations for each dimension. This provides a basis for the study of property preserving transformations or transformations between subclasses of systems such as untimed/timed, asynchronous/synchronous and event-triggered/data-triggered.

1.5 Organization of the Report

The document is split into four parts, the first presenting the Concepts and Methodologies (Chapter 2), the second describing the Implementations (Chapter 3 and 4), the third presenting Case studies applying the methodology (Chapter 5), and the last part (Chapter 6) drawing the conclusions.

Chapter 2 presents the basic notions about components, their composition using glues, and the necessary properties for component-based construction of systems. It introduces the BIP component framework, describing its architecture and abstract semantics. A concrete model for BIP is then presented. We then provide a classification of systems, showing heterogeneous domains that can be modeled in the BIP framework. Later a distributed semantics for BIP is presented, relevant for its implementation on real distributed platforms.

The second part of this thesis, which describes the implementation, is presented in two chapters. Chapter 3 introduces the language developed for modeling systems in the BIP methodology. It presents in detail, the language constructs, illustrated with abstract grammar and lucid examples.

Chapter 4 presents the tool-chain that we have developed in Verimag, supporting the presented methodology. It also describes the other tools that can be associated with our tool-chain for specific purposes. It provides vivid description for different types of implementations, e.g., centralized and distributed, with benchmarks for comparing performance of different implementations.

In the third part (Chapter 5), we present two real applications that were modeled and executed in BIP. We describe modeling of two categories of systems. The first is a mixed HW/SW system, where we model and analyze wireless sensor network systems, with motes running NesC applications on TinyOS. The second is an example of componentization techniques for big software systems. We model an MPEG4 encoder and analyze the pros and cons of componentization.

We conclude the document in Chapter 6, with an overview of the work and its future perspectives.

Chapter 2

Component Composition in BIP

We describe in this chapter, the basic notions about components and their composition. We introduce the BIP component framework and its formal semantics. We present an abstract model, followed by a concrete model of the methodology, with examples of modeling simple systems. We then present a distributed semantics of BIP. We give a classification of systems that can be modeled in BIP with examples of timed and synchronous systems. We conclude the chapter with a discussion on the system modeling space.

2.1 Basic Ideas

A *component* is a behavioral entity, having a well defined interface. It denotes an executable description whose runs can be modeled as sequences of actions. Tasks, processes, threads, functions, blocks of code can be considered as components.

A basic component, representing only behavior is called an *atomic* component. We denote atomic components graphically as boxes containing behavior inside. Behavior is represented by a labeled transition system (LTS).

Definition 2.1.1 (Labeled Transition System) *A labeled transition system is a triple $B = (Q, P, \rightarrow)$, where Q is a set of states, P is a set of actions, and $\rightarrow \subseteq Q \times P \times Q$ is a set of transitions, each labeled by an action.*

For any pair of states $q, q' \in Q$ and an action $p \in P$, we write $q \xrightarrow{p} q'$, iff $(q, p, q') \in \rightarrow$. If such q' does not exist, we write $q \overset{p}{\not\rightarrow}$.

A set of atomic components can be glued together producing a larger component, called a *compound* component. A *glue* combines a set of components. It is a set of operators mapping tuples of behavior into behavior.

It restricts the combined behavior of the components by memoryless coordination in order to meet some global properties. Given $B_1, B_2, \ldots B_n$ a set of atomic components, their composition with the glue $\mathcal{GL}$ produces a transformed behavior B, as shown in figure 2.1, where

$$B = \mathcal{GL}(B_1, B_2, \ldots B_n)$$

The glue is a separate layer, composing the underlying layer of behaviors. The behavior B is the product of the behaviors of $B_1, B_2, \ldots B_n$, with some restrictions implied by the meaning of the glue. The new component B can be further used for composition with other components.

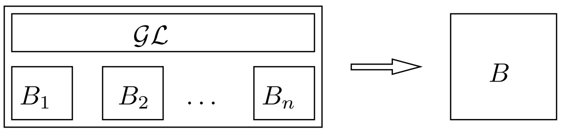


Figure 2.1: Component Composition.

Component-based design consists in building a component satisfying a given property from:

- a set of components $B_1, B_2, \ldots B_n$, described by their behavior, and
- a set of glue operators $\mathcal{GL} = \{gl_1, gl_2, \ldots gl_n\}$

The meaning of a glue is defined by operational semantics. We provide below a definition for glue based on Structural Operational Semantic (SOS) [Plo81, AFV01] rules. The definition is adopted from [BS08b].

Definition 2.1.2 (Glue Operator) *A glue operator is any behavior transformer defined by derivation rules of the form*

$$\frac{\{q_i \xrightarrow{a_i}_i q_i'\}_{i \in I} \qquad \{q_k \not\xrightarrow{a_k}_k\}_{k \in K}}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

where

- $I, K \subseteq \{1, \ldots n\}$
- $I \neq \emptyset$, $I \cap K = \emptyset$
- $a = \bigcup_{i \in I} a_i$ *is a transition of the composed component, and*
- $q_i' = q_i$ *for* $i \notin I$.

Premises of the form $q \xrightarrow{a} q'$ *are called positive, and those of the form* $q \not\xrightarrow{a}$ *are called negative. That is:*

1. *a derivation rule has at most one positive premise for each component.*
2. *there is at least one positive premise.*
3. *a label can appear either in positive or in negative premises, but not in both, i.e., no contradictory premises.*

A glue is a set of glue operators. The definition of a glue operator considers the transition of a compound component ($\rightarrow$) as a result of the transitions of its constituents ($\rightarrow_i$).

Notice that glue is defined by stratified rules. The transition relations involved in the premises ($\rightarrow_i$) and the transition relation of the conclusion ($\rightarrow$) are different. For example, non-deterministic choice and sequential composition are not glue operators.

The composition of a set of behaviors by a glue gives a restricted behavior, contained in the product of their behaviors. This is illustrated with an example (figure 2.2). We have two behaviors, B_1 and B_2, specified as labeled transition systems.

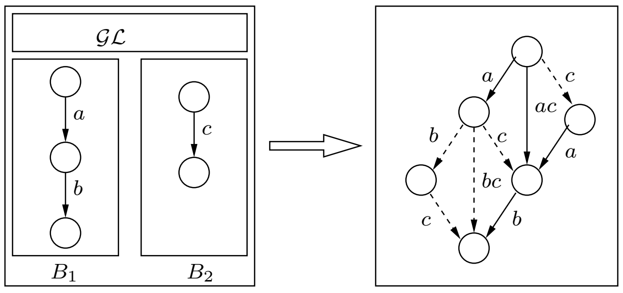


Figure 2.2: Composition with Glue.

The glue $\mathcal{GL}$ is the set of the following operators

$$\frac{q_1 \xrightarrow{a}_1 q_1'}{q_1q_2 \xrightarrow{a} q_1'q_2} \qquad \frac{q_1 \xrightarrow{a}_1 q_1' \qquad q_2 \xrightarrow{c}_2 q_2'}{q_1q_2 \xrightarrow{ac} q_1'q_2'} \qquad \frac{q_1 \xrightarrow{b}_1 q_1' \qquad q_2 \not\xrightarrow{c}_2}{q_1q_2 \xrightarrow{b} q_1'q_2}$$

The composed behavior obtained after the application of the glue $\mathcal{GL}$ 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 glues.

Our 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. For example, as shown in figure 2.3, components B_1 and B_1' are composed with glue $\mathcal{GL}_1$, and the resulting component is integrated with B_2 by $\mathcal{GL}_{12}$ to produce the global system.

Constructivity is expressed by the following requirements:

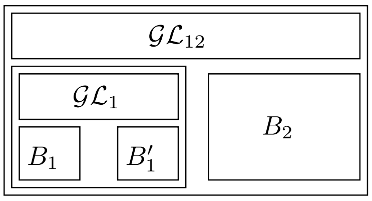


Figure 2.3: Components and Glues.

2.1.1 Incrementality

Incrementality of construction means the following requirements:

2.1.1.1 Decomposition

An n-ary glue operator could be obtained by successive application of a binary glue operator, as shown on figure 2.4. In general, we should be able to write

$$\mathcal{GL}\left(B_1, \ldots, B_n\right) = \mathcal{GL}_1\left(B_1, \mathcal{GL}_2\left(B_2, \ldots, B_n\right)\right)$$

That is, any compound component can be obtained by successive composition of its atomic components.

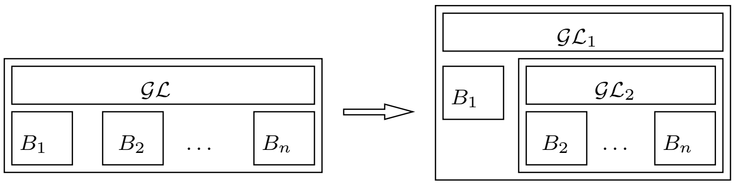


Figure 2.4: Decomposition.

2.1.1.2 Flattening

It is the dual of the decomposition operation. Any given structure can be flattened to a component which is the composition of its atomic components by using a single glue operator.

$$\mathcal{GL}_1\left(B_1, \mathcal{GL}_2\left(B_2, \ldots\right)\right) = \mathcal{GL}\left(B_1, \ldots, B_n\right)$$

An example is shown in figure 2.5.

The combination of decomposition and flattening leads to incrementality. By the above two mechanisms, a given system of behavior can be partitioned into any required structure.

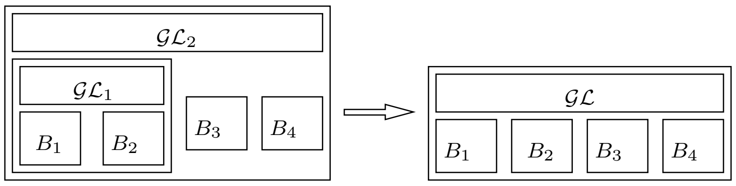


Figure 2.5: Flattening.

2.1.2 Compositionality

Compositionality means inferring global system properties from the properties of the individual system. It can be formally defined by rules of the form

$$\frac{\{B_i \models f_i\}_{i=1}^{n}}{\mathcal{GL}(B_1, \ldots B_n) \models \widetilde{\mathcal{GL}}(f_1, \ldots f_n)}$$

where f_i is a property of the component B_i; $\mathcal{GL}$ is a glue composing the components; and $\widetilde{\mathcal{GL}}$ is an operator on properties depending on the glue $\mathcal{GL}$.

2.1.3 Composability

Composability guarantees that during the component construction process, all essential properties of subcomponents are preserved. It is defined by the following rule.

$$\frac{\{\mathcal{GL}_i\{B_j\}_{j\subseteq[1,n]} \models f_i\}_{i=1}^{m}}{\mathcal{GL}(B_1, \ldots B_n) \models \bigwedge_{i=1}^{m} f_i}$$

where $\mathcal{GL}_i$ is a glue satisfying a property f_i on the set of components $\{B_j\}_{j\in[1,n]}$; and $\mathcal{GL} = \bigodot_{i=1}^{m} \mathcal{GL}_i$ is a composition of the glues.

2.2 Abstract Model of BIP

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 presented in definition 2.1.2. The framework is based on a 3-tier architecture, the layers being **B**ehavior, **I**nteraction and **P**riority.It defines a mechanism for composition of *behavior* using the *interaction* and *priority* glues.

2.2.1 The 3-Tier Architecture

The BIP component framework uses an abstract layered model of components. A component consist of the superposition of three layers: *behavior*, *interaction* and *priority*, as shown in figure 2.6, where:

1. Behavior describes the dynamic behavior of a component. It consists of a set of labeled transition system. Each transition has a port, a guard and a function. Guards are conditions depending on local state. Ports characterize the component's ability to interact with a given environment.

2. Interactions describe architectural constraints on behavior. They are joint state changes of composed components used to coordinate their execution.

3. Priorities provide a mechanism for restricting the global behavior of the layers underneath by filtering amongst possible interactions. They help reducing non-determinism in the execution of the interactions between the components. They are useful for enforcing state invariant properties such as mutual exclusion and scheduling policies.

The interaction and priority layers are the glue operators for BIP. In the fol-

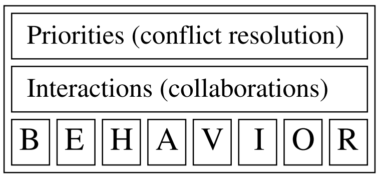

Priorities (conflict resolution)
Interactions (collaborations)
B E H A V I O R

Figure 2.6: BIP Architecture.

lowing section, we give a formal description of each of the layers, introduced here.

2.2.2 Abstract Semantics

We provide a formalization of the BIP component model focusing on the individual layers of behavior, interaction and priority glue, and provide the operational semantics for composition of behavior with respect to interaction and priority layers.

Definition 2.2.1 (Behavior) *A behavior B is a labeled transition system represented by a triple $(Q, P, \rightarrow)$, where Q is a set of control states, P is a set of communication ports, $\rightarrow \subseteq Q \times P \times Q$ is a set of transitions, each labeled by a port.*

For any pair of control states $q, q' \in Q$ and a port $p \in P$, we write $q \xrightarrow{p} q'$, iff $(q, p, q') \in \rightarrow$. When the port is irrelevant, we simply write $q \rightarrow q'$. Similarly, $q \xrightarrow{p}$ means that there exists $q' \in Q$ such that $q \xrightarrow{p} q'$.

A port p of B is *enabled* iff B is at a state q and $q \xrightarrow{p}$. Otherwise, p is *disabled.*

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

Definition 2.2.2 (Interaction) *For a set of ports P, an interaction is a non-empty subset $a \subseteq P$ of ports.*

When we write $a = \{p_i\}_{i \in I}$, $I \in 1 \ldots n$, we suppose that for each $i \in I$, $p_i \in P_i$. The interaction model is specified by a set of interactions $\gamma \subseteq 2^P$.

Interactions of γ can be *enabled* and *disabled.* An interaction a is *enabled* iff, for all $i \in [1, n]$, the port $a \cap P_i$ is enabled in B_i. It is *disabled* if, there is an $i \in [1, n]$, the port $a \cap P_i$ is disabled in B_i.

The enabledness of an interaction does not necessarily entails its execution, as it might be inhibited by another enabled interaction, in the priority model. We now provide the operational semantics for the composition of a system of behavior with respect to an interaction model. The composition is derived from the universal glue (2.1.2), consisting of only positive premises.

Definition 2.2.3 (Composition for Interactions) *The composition of a set of components $\{B_i\}_{i=1}^n$, parameterized by a set of interactions $\gamma \subseteq 2^P$, is a transition system $B = (Q, \gamma, \rightarrow_\gamma)$, where $Q = \bigotimes_{i=1}^n Q_i$ and $\rightarrow_\gamma$ is the least set of transitions defined by the rule*

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

We write $B = \gamma(B_1, \ldots, B_n)$. The inference rule says that the obtained behavior $\gamma(B_1, \ldots, B_n)$ can execute a transition $a \in \gamma$, iff for each $i \in I$, the action $a \cap P_i$ is enabled in B_i; the states of the transition system, that do not participate in the interaction a remain unchanged.

Observe that, it is possible for a composed behavior to further communicate on the ports initially provided by its constituent behaviors.

Notice that interactions are glue operators with positive premises only. Hence, in a behavior, more than one interactions can be enabled at the same time, introducing a degree of non-determinism. This can be restricted by priorities, which constitute the third layer in the proposed 3-tier BIP architecture.

Priorities are a powerful tool for restricting nondeterminism, and allows straightforward modeling of urgency and scheduling policies for real time systems. For example, execution constraints like run to completion and synchronous execution can be modeled by priority models on threads. Moreover, as the priority model is dynamic, it can advantageously overcome the static restrictions of other execution models, like process algebra.

A priority model is a memoryless controller. It filters the possible interactions from the interaction model, based on the current global state of the system. We provide below a definition of priority, followed by the composition of behavior using the priority glue.

Definition 2.2.4 (Priority) *A priority is a relation $\prec \subseteq \gamma \times Q \times \gamma$, where γ is the set of interactions, and Q is the global set of states. We write $a \prec_q a'$ for $(a, q, a') \in \prec$. Furthermore, we require that for all $q \in Q$, $\prec_q$ is a strict partial order on γ. $a \prec_q a'$ means that interaction a has less priority than a' at state q.*

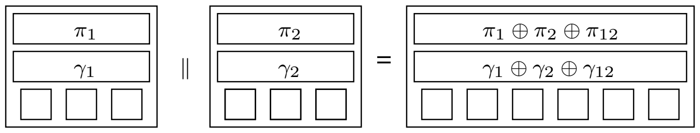
A priority model π is a set of priorities. The composition using the priority glue is derived from the universal glue definition(2.1.2).

Definition 2.2.5 (Restriction w.r.t Priority Model) *Given a behavior $B = (Q, P, \rightarrow_\gamma)$, its restriction by the priority model π is the behavior $B' = (Q, P, \rightarrow_\pi)$ defined by the rule*

$$\frac{q \xrightarrow{a}_\gamma q' \qquad \{q \overset{a'}{\not\rightarrow}_\gamma\}_{a \prec_q a'}}{q \xrightarrow{a}_\pi q'}$$

Notice that priorities are glue operators with only one positive premise and an arbitrary number of negative premises. The interaction and the priority glue encompass the universal glue.

Given a set of components $B_1, \ldots, B_n$, an interaction model γ, and a priority model π, the compound component is obtained by application of γ and π, i.e., $\pi\gamma(B)$. An interaction is enabled in $\pi\gamma(B)$ only if it is enabled in γ and maximal according to π among the enabled interactions in B.



π_1
γ_1
$\|$
π_2
γ_2
=
$\pi_1 \oplus \pi_2 \oplus \pi_{12}$
$\gamma_1 \oplus \gamma_2 \oplus \gamma_{12}$

Figure 2.7: Composition.

Given two components $\pi_1\gamma_1(B_1)$ and $\pi_2\gamma_2(B_2)$, we can define a binary composition operator $\|$ parameterized by the interaction model γ_{12}, and priority model π_{12} (figure 2.7). The layers of the composed component are obtained by composing separately the individual layers of the composed components, and with the corresponding parameters of the composition operator. That is,

$$\pi_1\gamma_1(B_1)\|\pi_2\gamma_2(B_2) = \pi_1 \oplus \pi_2 \oplus \pi_{12}(\gamma_1 \cup \gamma_2 \cup \gamma_{12}(B_1, B_2))$$

The meaning of the priority composition operator $\oplus$ will be defined later in section 2.3.3.

We now provide simple examples, illustrating the layered architecture of BIP and demonstrating component composition through the use of interaction and priority layers.

Example 2.2.6 (Rendezvous) *Figure 2.8(a) shows three components, a sender (S) and two receivers (R1, R2). The sender has a port s for sending messages, and each receiver has a port r_i ($i = 1, 2$) for receiving messages. Additionally, the sender has a port i and the receivers have ports i_k ($k = 1, 2$) respectively, representing independent (internal) events. Rendezvous is modeled by specifying the interaction sr_1r_2 in the interaction model of the composed system. This interaction synchronizes the s event of the sender with the r_i ($i = 1, 2$) events of the receivers, and hence a rendezvous. Note that the interaction model also contains the independent events of the sender and receivers. No priorities are necessary, hence the priority model is empty.*

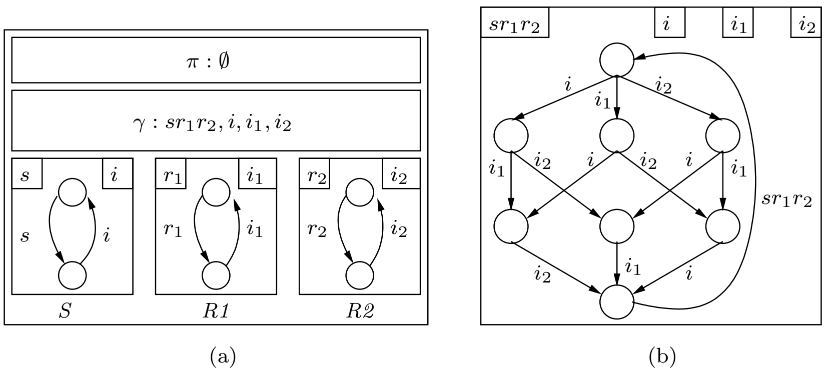
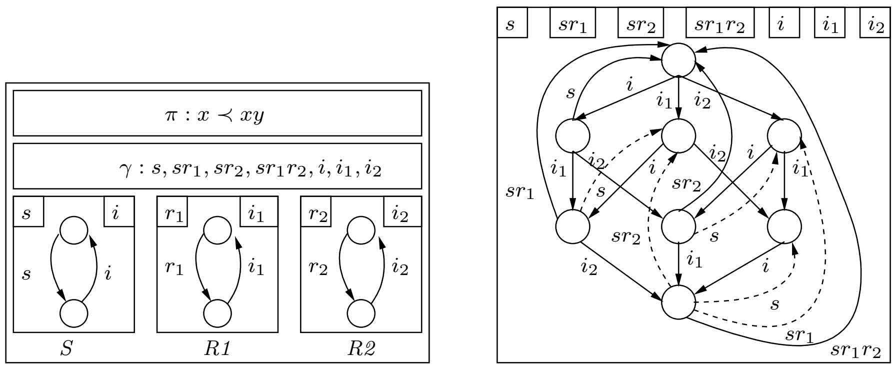


Figure 2.8: Rendezvous.

The equivalent behavior of the composition is shown in figure 2.8(b).

Example 2.2.7 (Broadcast) *Figure 2.9(a) models a broadcast from the sender to the receivers. The interaction model contains the set of all interactions which synchronizes the send event of the sender with the receive event of the receivers, i.e., s, sr_1, sr_2, and sr_1r_2. However, amongst the possible interactions, the one with the maximum number of receivers should be selected in a broadcast. To ensure this selection policy, a priority model* *is specified, describing the (static) rule that an interaction (xy) containing another one (x) has higher priority over it.*

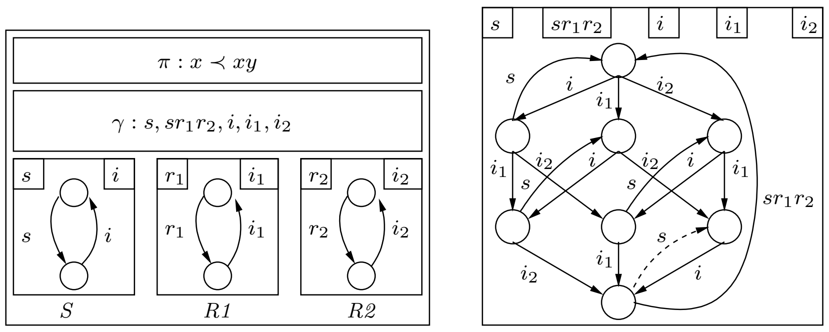

$\pi : x \prec xy$
$\gamma : s, sr_1, sr_2, sr_1r_2, i, i_1, i_2$
s i S
r_1 i_1 $R1$
r_2 i_2 $R2$
s sr_1 sr_2 sr_1r_2 i i_1 i_2

(a) (b)

Figure 2.9: Broadcast.

The behavior after the composition with interaction and priority model is shown in figure 2.9(b). The transitions inhibited by the priority model are shown by dotted arrows.

Example 2.2.8 (Atomic Broadcast) *Figure 2.10(a) models an atomic broadcast. It is a specific type of a broadcast, where either all or none of the receivers synchronize with the sender. The synchronizing interactions are s and sr_1r_2. The priority model is required to enable the maximal interaction. The composed behavior is presented in figure 2.10(b).*


$\pi : x \prec xy$
$\gamma : s, sr_1r_2, i, i_1, i_2$
s i S
r_1 i_1 $R1$
r_2 i_2 $R2$
s sr_1r_2 i i_1 i_2

(a) (b)

Figure 2.10: Atomic Broadcast.

2.3 Concrete Model of BIP

2.3.1 Modeling Behavior

In the concrete model, an atomic component represents behavior B as a transition system, extended with variables and functions, represented by $(X, P, S, \rightarrow)$, where:

- P is a set of *ports*, $P = \{p_1 \dots p_n\}$.
- S is a set of *control states* $S = \{s_1 \dots s_k\}$. Control states denote places at which the components await for synchronization.
- X is a set of *variables* used to store (local) data. 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.
- $\rightarrow$ is a set of transitions modeling computation steps of components. Each transition is a tuple of the form (s_1, p, g_p, f_p, s_2), representing a step from control state s_1 to s_2, denoted as $s_1 \xrightarrow{p,g_p,f_p} s_2$.

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

The semantics of execution of a transition is an atomic sequence of two micro-steps: 1) an interaction including p which involves synchronization between components with possible exchange of data, followed by 2) an internal computation specified by the function f_p on X. That is, if x is a valuation of X after the interaction, then $f_p(x)$ is the new valuation when the transition is completed.

Formally, the behavior is a labeled transition system with moves of the form $(s_1, x) \xrightarrow{\alpha} (s_2, x')$, where s_1 is a control state of the automaton and x is a valuation of its variables. The move $(s_1, x) \xrightarrow{\alpha} (s_2, x')$ is possible if there exists a transition (s_1, p, g_p, f_p, s_2), such that $g_p(x) = \mathit{true}$. As a result of the move, the set of variables are modified to $x' = f_p(x)$.

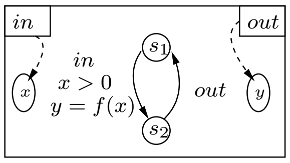


Figure 2.11: Component behavior

Figure 2.11 shows an atomic reactive component with two ports *in*, *out*, control states s_1, s_2, and variables x, y. x and y are associated with the ports *in* and *out* respectively. At s_1, the transition labeled *in* is possible if $x > 0$. When an interaction through *in* takes place, the variable x is eventually modified and a new value for y is computed by the action $f(x)$. From control state s_2, the transition labeled *out* can occur. The omission of guard and function for this transition means that the associated guard is *true* and the internal computation micro-step is empty.

2.3.2 Modeling Interactions

Composition operators allow to build a system as a set of components that interact by respecting constraints of an interaction model. We propose a means for structuring interactions by using connectors.

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. Graphically, connectors are represented as trees with ports at their leaves.

Based on the structure of interactions represented by connectors, they are classified as Simple Connector and Structured Connector.

2.3.2.1 Simple Connector

A simple connector is defined by a set of ports. An interaction of a simple connector is any non empty subset of its set of ports. For example, a simple connector consisting of the ports $p1$, $p2$ and $p3$ has seven possible interactions: $p1$, $p2$, $p3$, $p1p2$, $p2p3$, $p1p3$ and $p1p2p3$. Each non trivial interaction, i.e., interaction with more than one port, represents a synchronization between transitions labeled with its ports. We use the term connector to mean simple connectors.

Following results in [GS05], we introduce a typing mechanism for the ports of a connector, in order to specify the feasible interactions of a connector, and in particular to express the following two basic modes of synchronization:

- Strong synchronization or rendezvous, when the only feasible interaction of a connector is the maximal one, i.e., containing all the ports.
- Weak synchronization or broadcast, when feasible interactions are all those containing a particular port which initiates the broadcast.

It is possible to represent any arbitrary interaction through a connector by structured combination of the above two basic synchronization protocols.

To characterize these protocols, we associate types with ports: *trigger* and *synchron*. 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 connectors and

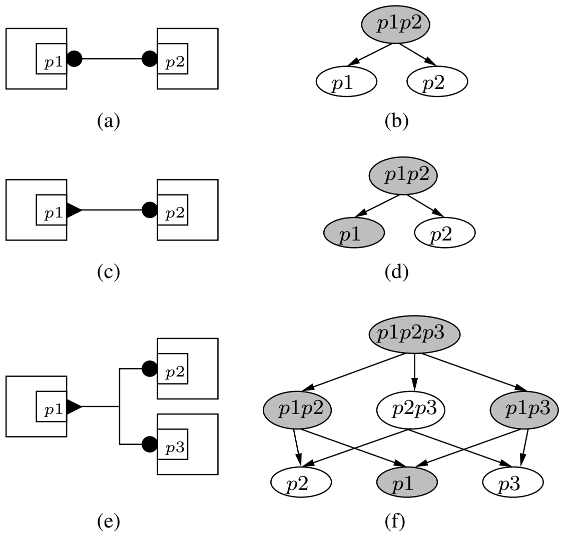


Figure 2.12: Connectors and their interactions.

their feasible interactions are shown in figure 2.12. 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 $p1p2$, as shown in (b). It represents a rendezvous, meaning that both actions are necessary for the synchronization. In (c), 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 $p2$ requires the occurrence of $p1$. The feasible interactions are $p1$ and $p1p2$, shown in (d). In (e), the interactions between $p1$, $p2$ and $p3$ are also asymmetric. The interactions $p1$ can occur alone or synchronize with either or both $p2$ and $p3$, as shown in (f).

Example 2.3.1 (Connectors) *Figure 2.13(a) shows an example of graphical representation of connectors in BIP. The corresponding layered notation is shown in figure 2.13(b). Connector $C1$ contains the ports $tick1$, $tick2$ and $tick3$. It assigns to each port the attribute synchron. Thus $C1$ represents the interaction describing a rendezvous between the ports $tick1$, $tick2$ and $tick3$. The only possible interaction in this case is $tick1tick2tick3$.*

The other connector $C2$ consists of the ports out, $in1$ and $in2$, where the port out is assigned the attribute type trigger. The other ports are synchrons. $C2$ describes a broadcast from the port out to the ports $in1$ and $in2$. The set of possible interactions are out, $outin1$, $outin2$, and $outin1in2$ (i.e., any *interaction containing the trigger port out). The complete set of interactions are included in the interaction model of figure 2.13(b).*

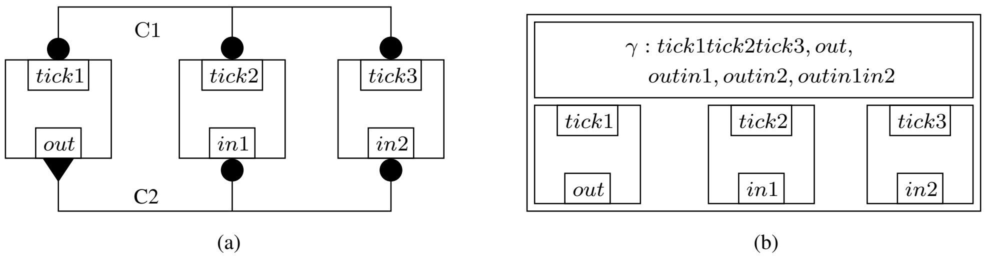


Figure 2.13: Connector.

2.3.2.2 Structured Connector

So far we have seen a notation for connectors, which are essentially flat, i.e., having types (triggers and synchrons) associated to the individual ports only. However, connectors sometimes need to be structured, i.e., having types associated to groups of ports. This is necessary to represent some interactions, which otherwise cannot be represented by a flat connector. Consider for example the atomic broadcast (Example 2.2.8), where we need to restrict the feasible interactions to s and $sr_1r_2r_3$. It is impossible to represent this set of interactions through a simple connector as presented in the previous section.

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 [BS07b], as described below.

Algebra of Connectors

The Algebra of Connectors is a compact notation for algebraic representation and manipulation of connectors. The Algebra of Connectors $\mathcal{AC}(P)$, introduced in [BS07b], 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. Given two connectors involving sets of ports s_1 and s_2, it is possible to obtain by fusion a new connector involving the set of ports $s_1 \bigcup s_2$, as shown in figure 2.14(a). It is also possible to structure connectors hierarchically, as shown in figure 2.14(b) where terms p_1p_2 and p_3p_4 are typed and then fused to obtains a new connector. The semantics of the algebra of connectors associates with a connector (a term)

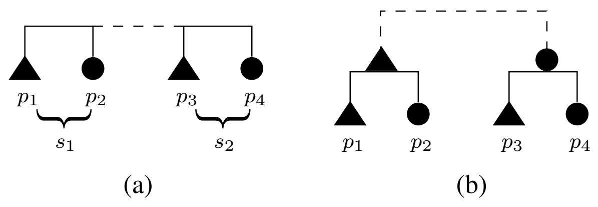


Figure 2.14: Fusion (a) and structuring (b) of connectors.

the set of its feasible interactions. The algebra and its laws can be used to represent and handle symbolically, complex interaction patterns.

The Algebra of Connectors, as presented in [BS07b], has two operations: union and fusion. Union operation allows to combine several connectors into a single expression. Fusion represents synchronization of connectors by merging their interactions. We briefly introduce the syntax and semantics of the algebra of connectors as follows.

Syntax

Let P be a set of ports, such that $0, 1 \notin P$. The syntax of $\mathcal{AC}(P)$ is defined by

$$\begin{array}{rcll} s & ::= & [0] \mid [1] \mid [p] \mid [x] & (\textit{synchrons}) \\ t & ::= & [0]' \mid [1]' \mid [p]' \mid [x]' & (\textit{triggers}) \\ x & ::= & s \mid t \mid x \cdot x \mid (x)\,, & \end{array} \tag{2.1}$$

for $p \in P$, and where $\cdot$ is a binary operator called *fusion*, and $[\cdot]$ and $[\cdot]'$ are unary *typing* operators.

Fusion is used to merge two connectors, whereas typing is used to form hierarchically structured connectors: $[\cdot]'$ defines *triggers* (which can initiate an interaction), and $[\cdot]$ defines *synchrons* (which need synchronization with other ports). In order to simplify notation, we often omit brackets on 0, 1, and ports $p \in P$, as well as '$\cdot$' for the fusion operator.

Semantics

The semantics of $\mathcal{AC}(P)$ is given in terms of sets of interactions. Intuitively it consists in recursively applying the following rule: *For a connector of the form* $[x_1]' \dots [x_n]'[y_1] \dots [y_m]$*, a possible interaction is a union of interactions from the sub-connectors* $x_1, \dots, x_n, y_1, \dots, y_m$*, comprising an interaction from* at least one *of the triggers* $x_1, \dots, x_n$*. When there are no triggers (*$n = 0$*), an interaction from every synchron sub-connector* $y_1, \dots, y_m$ *is required to form an interaction of the complete connector.* A formal definition is given by the function $\|\cdot\| : \mathcal{AC}(P) \to 2^{2^P}$, defined by

$$\|0\| = \emptyset, \quad \|1\| = \{\emptyset\}, \quad \|p\| = \Big\{\{p\}\Big\}, \quad \left\|\prod_{i=1}^{n}[x_i]\right\| = \left\{\bigcup_{i=1}^{n} a_i \;\middle|\; a_i \in \|x_i\|\right\}, \tag{2.2}$$

$$\left\|\prod_{i=1}^{n}[x_i]' \prod_{j=1}^{m}[y_j]\right\| = \bigcup_{i\in[1,n];J,K} \left\|[x_i] \prod_{j\in J}[x_j] \prod_{k\in K}[y_k]\right\|, \tag{2.3}$$

where for $p \in P$, $x_1, \ldots, x_n, y_1, \ldots, y_m \in \mathcal{AC}(P)$, and the union in 2.3 is taken over all (potentially empty) subsets $J \subseteq [1, n]$ and $K \subseteq [1, m]$.

$\mathcal{AC}(P)$ allows compact and intuitive representation of interactions, and captures explicitly the difference between broadcast and rendezvous. A full presentation of the properties of $\mathcal{AC}(P)$ is given in [BS07b]. Figure 2.15 shows the $\mathcal{AC}(P)$ notation for the connectors of examples 2.2.6, 2.2.7 and 2.2.8. Note the application of the typing operator to groups of ports, repre-

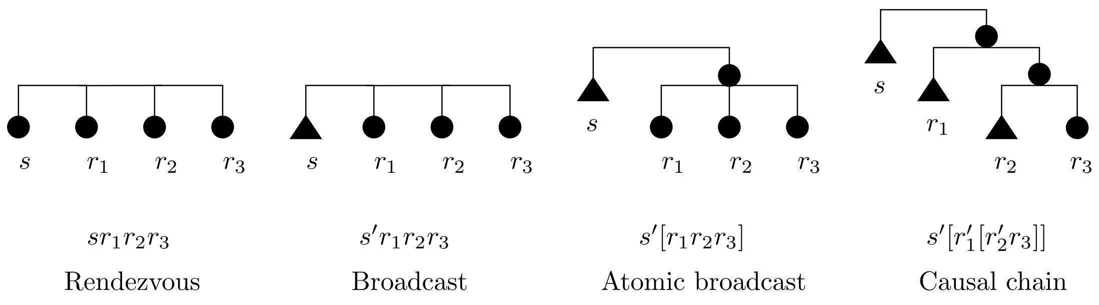


Figure 2.15: $\mathcal{AC}(P)$ notation of connectors.

senting structured connectors. For atomic broadcast, the type synchron is attributed to the group of ports $r_1r_2r_3$, which can be seen as a sub-connector. The causal chain is another example of a structured connector.

Incremental Construction of Connectors

The fusion and typing operators on connector allow for their incremental construction. Consider the broadcast from a sender port s to two receiver ports r_1 and r_2, as shown in figure 2.16(a). The same set of interactions can be obtained incrementally by an equivalent structured connector, as shown in figure 2.16(b). It is constructed by first creating a broadcast from s to r_1, specified by $s'r_1$, and then by the adding the second receiver r_2 through the structured connector $[s'r_1]'r_2$. Notice that both represents the same set of interactions,$\gamma = \{s,\, sr_1,\, sr_2,\, sr_1r_2\}$.

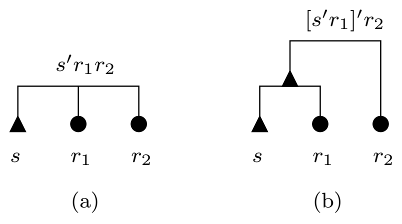


Figure 2.16: Incremental construction of connectors.

2.3.2.3 Semantics of Composition by Connectors

Connector Guards and Transfer Functions

In BIP the execution of interactions may involve transfer of data between the synchronizing components. If α is the interaction of a connector, we use a guard G_α (boolean condition) and data transfer function F_α to specify data transfer. Guard is a global condition, spanning over the variables of the interacting components. An enabled interaction is executable if its guard is true. The execution of the interaction leads to execution of the data transfer F_α associated with it. As a result, the variables of the synchronizing components are updated. The mechanism for execution of guard and data transfer of a connector is formally presented below.

Consider the composition of n concrete components $\{B_i\}_{i=1}^n$ parameterized by a set of connectors γ. The composed system is a concrete component $B = (X, P, S, \rightarrow_\gamma)$ with:

- A set of variables X, which is the union of the sets of variables of the composed components $X = \bigcup_i^n X_i$,
- A set of interactions P defined by γ, $P = \{\alpha— \alpha$ is an interaction in $\gamma\}$,
- A set of control states S, which is the Cartesian product of the sets of control states of the composed components $S = \bigotimes_i^n S_i$,
- A set of transitions $\rightarrow_\gamma$ of the form (s, α, g, f, s'), where:
  - $s = (s_1, \ldots, s_n)$, s_i being a control state of the i^{th} component.
  - α is a feasible interaction in γ associated with a guarded command (G_α, F_α), such that there exists a subset $J \subseteq \{1, \ldots, n\}$ of components with transitions $\{(s_j, p_j, g_j, f_j, s'_j)\}_{j \in J}$ and $\alpha = \{p_j\}_{j \in J}$.
  - $g = (\bigwedge_{j \in J} g_j) \wedge G_\alpha$.
  - $f = F_\alpha; [f_j]_{j \in J}$. That is, the computation starts with the execution of F_α followed by the execution of all the functions f_j in some arbitrary order. The result is independent of this order as components have disjoint sets of variables.

– $s'(j) = s'_j$ if $j \in J$; otherwise $s'(j) = s_j$. That is, the states from which there are no transitions labeled with ports in α, remain unchanged.

The composition of the concrete system is shown in figure 2.17.

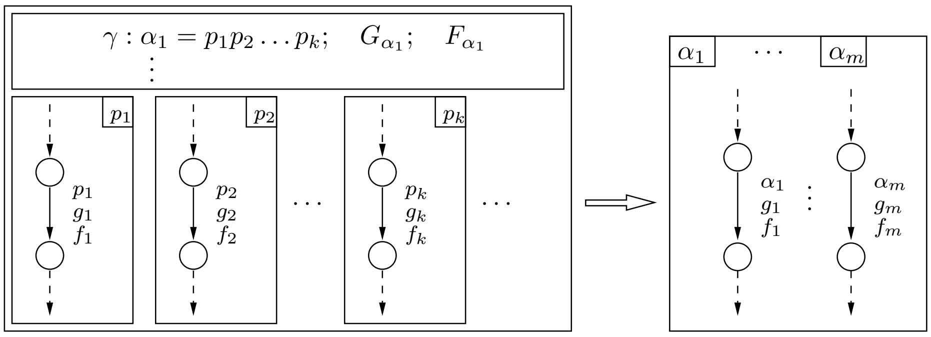


Figure 2.17: Composition of concrete components.

2.3.3 Modeling Priority

For the concrete mode, a priority order $\prec$ is represented by a priority rule. It is a tuple $(C, \prec)$, where C is a state predicate (boolean condition) characterizing the states where the priority applies. C also plays the role of a dynamic guard for the priority $\prec$.

A priority rule is textually expressed as $C \rightarrow \prec$. It means that when the state predicate is true and both the interactions specified in the priority are enabled, the higher order interaction is selected for execution. For static priorities, C is true and is omitted from the priority rule.

A priority model π consists of a set of priority rules, $\{C_i \rightarrow \prec_i)\}_{i=1}^{m}$, with a disjoint set of state predicates, i.e., for any two $i \neq j$, we have $C_i \cap C_j = \emptyset$. Consider the behavior given by an automaton, as shown in the left of figure

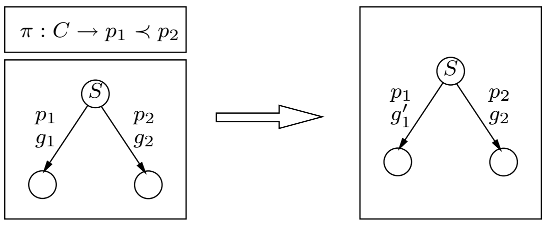


Figure 2.18: Priority restriction.

2.18, with a non deterministic choice between two interactions p_1 and p_2, from the control-state S. The corresponding guards are respectively g_1 and g_2. Non-determinism is resolved by a priority rule $C \rightarrow p_1 \prec p_2$, which selects the interaction p_2 to be taken, when C holds, by disabling p_1. This is implemented by modifying the guard g_1, of the interaction with the lower priority, to g_1', given by $g_1' = g_1 \wedge (C \Rightarrow \neg g_2)$. The restricted behavior is shown in the right of the same figure. If the predicate C is true, the interaction p_1 can occur only if p_2 is disabled. However if C is false, the priority holds no more, and we have a non-deterministic choice between p_1 and p_2.

The transformation of the underlying behavior through restriction is imposed by modifying the guards of all the interactions which are dominated by priority rules. For an interaction p_k dominated by priority rules from π, its modified guard is given as

$$g_k' = g_k \wedge \bigwedge_{C \rightarrow \prec \in \pi} (C \Rightarrow \bigwedge_{p_k \prec p_i} \neg g_i)$$

2.3.3.1 Composition of Priorities

The restriction of a behavior B by the successive application of priorities π_1 and π_2 may not be the same. It can depend on the order of application of π_1 and π_2. That is, in general

$$\pi_2(\pi_1(B)) \neq \pi_1(\pi_2(B))$$

Consider for example, the behavior in the left of figure 2.19, showing a transition system with interactions p_1, p_2 and p_3, possible from a control state. Consider the two priorities, $\pi_1 : true \rightarrow p_1 \prec p_2$ and $\pi_2 : true \rightarrow p_2 \prec p_3$. The order of their application leads to different behavior, as evident from the figure. It is shown in [GS03] that non commutativity of priorities is due to the fact that the union of the priority orders is not a priority order.

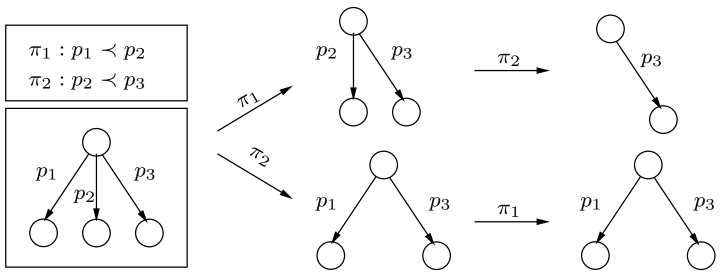


Figure 2.19: Priority Composition is non-commutative.

To avoid this anomaly, we define for two priorities π_1, π_2, their composition $\pi_1 \oplus \pi_2$ as the least priority contained in $\pi_1 \cup \pi_2$ (if it exists). Computing $\pi_1 \oplus \pi_2$ amounts to computing the transitive closure of $\pi_1 \cup \pi_2$. The operation is partial. It is defined only if this is a strict partial order, i.e., no cycles.

Priorities restrict the component behavior and do not add behavior. Components that are deadlock free, on restriction by the priority model continues to preserve the deadlock freedom. However, composition of priority models that leads to cyclic priority dependencies may result in deadlock situations. This is explained by the example in figure 2.20. The problem models two identical tasks T_1 and T_2, using shared resources W_1 and W_2. On arrival (a_1) the task T_1 first acquires the resource W_1 by the action b_1, and then W_2 by the action b'_1. It releases both the resources upon finishing (f_1). T_2 has a similar behavior, except that it first acquires W_2 (by b'_2) and then W_1 (by b_2). We use priorities for mutual exclusion in the use of W_1 and

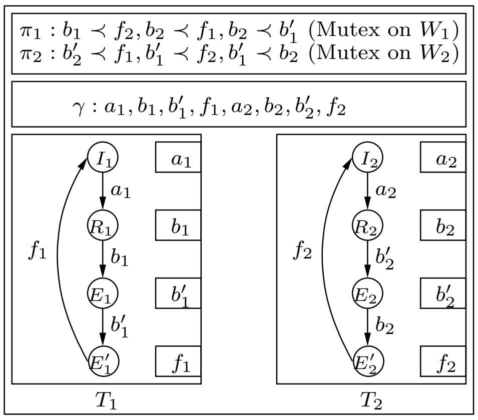

$\pi_1 : b_1 \prec f_2, b_2 \prec f_1, b_2 \prec b'_1$ (Mutex on W_1)
$\pi_2 : b'_2 \prec f_1, b'_1 \prec f_2, b'_1 \prec b_2$ (Mutex on W_2)
$\gamma : a_1, b_1, b'_1, f_1, a_2, b_2, b'_2, f_2$
I_1
a_1
R_1
b_1
E_1
b'_1
E'_1
f_1
T_1
I_2
a_2
R_2
b_2
b'_2
E_2
E'_2
f_2
T_2

Figure 2.20: Priority Composition leading to deadlock.

W_2 between T_1 and T_2. The rules of the priority model π_1 ensures exclusive use of W_1, and that of W_2 by the priority model π_2.

We would now like to ensure the mutual exclusion of both resources by the combination of the two given priority models. However, the composition of the rule $b_2 \prec b'_1$ of π_1 and $b'_1 \prec b_2$ of π_2 is undefined (as it leads to a cycle). This reflects the existence of a possible deadlock when b'_1 and b'_2 are enabled.

We now demonstrate through an example how an execution policy, e.g., synchronous execution, can be expressed with the help of priorities. As described in 1.2.1, synchronous execution adopts a strong fairness assumption, as in a computation step, all the components are offered the possibility to execute some quantum of computation. We show that synchronous execution can be obtained by appropriately restricting the behavior and interaction layers.

Example 2.3.2 (Synchronous Execution)

Consider the example of figure 2.21, a system which is a serial composition of three strongly synchronized components R_k, with port i_k (denoting an input action) and port o_k (denoting an output action), $k = 1, 2, 3$. Assume

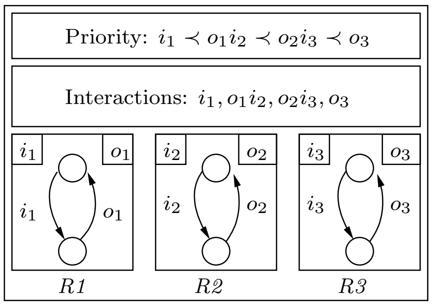

Priority: $i_1 \prec o_1i_2 \prec o_2i_3 \prec o_3$
Interactions: i_1, o_1i_2, o_2i_3, o_3
i_1
o_1
i_2
o_2
i_3
o_3
R1
R2
R3

Figure 2.21: Enforcing synchronous execution.

the components are reactive in the sense that they are triggered from some idle state when an input arrives, and eventually produces an output before reaching some idle state, from where a new input can be accepted. For the sake of simplicity, the components have simple cyclic behaviors with alternating inputs and outputs.

The interaction model consists of the interactions i_1, o_1i_2, o_2i_3 and o_3. We assume that the system is closed, so the interactions i_1 and o_3 are autonomous. In the product of the behaviors restricted by the interaction model, each component can perform computation independently of the others, provided the constraints resulting from the interaction model are met. This corresponds to asynchronous execution.

The behavior can be constrained by the priority model in order to enforce synchronous execution, in the sense that a run of the system is a sequence of steps, each step corresponding to the treatment of an input i_1 until an output o_3 is produced. This can be enforced by the priority order $i_1 \prec o_1i_2 \prec o_2i_3 \prec o_3$. This order reflects the causality between the interactions of the system. In fact, if all the components are at some idle state, then all are awaiting for an input. Clearly, only i_1 can occur to make R_1 evolve to a state from which o_1i_2 can occur. This will trigger successively o_2i_3 and finally o_3. Notice that i_1 cannot be executed as long as a computation takes place in some component.

We now provide a set of examples showing the application of priorities to enforce execution constraints on a system. We also provide generic models of task (both preemptable and non-preemptable), which are frequently used in a wide variety of applications.

Example 2.3.3 (Mutual Exclusion)

In this example, we show the enforcement of *mutual exclusion*, a very common execution constraint, needed when we have multiple components (tasks) sharing a single resource. In the example shown in figure 2.22, we have two identical tasks, T_1 and T_2, modeled as BIP components. The control states of the i-th task are I_i (Idle), R_i (Ready) and E_i (Executing). The actions (ports) are a_i (activate), b_i (begin) and f_i (finish). Each task is initialized to state I_i, from where it can voluntarily activate through the action a_i and become ready (R_i) for execution. The start of execution is marked by the action b_i, by which the task acquires the resource and moves to the execution state (E_i). The interaction model does not enforce any restriction and con-

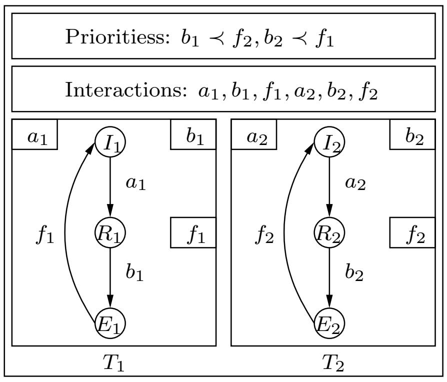

Prioritiess: $b_1 \prec f_2, b_2 \prec f_1$
Interactions: $a_1, b_1, f_1, a_2, b_2, f_2$
a_1
b_1
f_1
I_1
R_1
E_1
T_1
a_2
b_2
f_2
I_2
R_2
E_2
T_2

Figure 2.22: Enforcing Mutual Exclusion.

sists of all the independent actions of the tasks. Under mutual exclusion, a task cannot acquire the resource if the other task is already in the execution state. This is enforced by the priority model, which assigns the actions to obtain the resource, lower priority than the actions to release the resource, i.e., $b_1 \prec f_2$ and $b_2 \prec f_1$. For T_1, the priority $b_1 \prec f_2$ prevents it from obtaining the resource, unless T_2 releases it (by the action f_2). Similarly, T_2 is restricted by $b_2 \prec f_1$.

When both tasks are at state R_i, either of them can acquire the resource in a non-deterministic fashion. However, static priority rules can be used to prioritize the tasks in any desired order. For example, $b_2 \prec b_1$ sets a higher priority for T_1 to acquire the resource, compared to T_2.

Note that mutual exclusion can also be expressed by a restricted interaction model. The interactions: a_1, $b'_1 f_2$, f_1, a_2, $b'_2 f_1$, f_2, would by maximal progress, enforce mutual exclusion. The interaction $b'_i f_j$ lets task i to obtain the resource, either by executing b_i alone, when task j is not in E_j, or by executing the synchronization $b_i f_j$, where task j releases the resource to task i.

In this example we have shown the use of static priorities. The following examples illustrate the use of dynamic priorities.

Example 2.3.4 (A FIFO Scheduler)

We provide another example showing the enforcement of a scheduling policy using the priority model. In particular, it shows the application of dynamic priorities. Figure 2.23 shows the model of scheduling two tasks under FIFO.

The tasks share a common resource, so we enforce mutual exclusion between the tasks, similar to the previous model (example 2.3.3). The behavior model

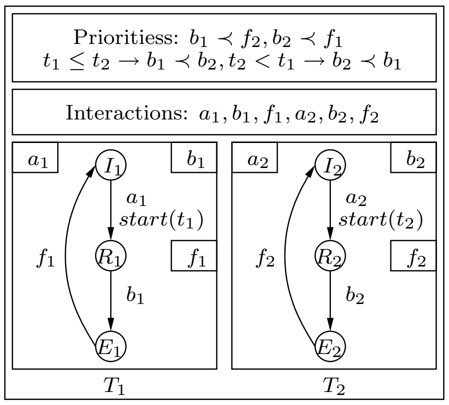

Prioritiess: $b_1 \prec f_2, b_2 \prec f_1$
$t_1 \leq t_2 \rightarrow b_1 \prec b_2, t_2 < t_1 \rightarrow b_2 \prec b_1$
Interactions: $a_1, b_1, f_1, a_2, b_2, f_2$
a_1
b_1
I_1
a_1
$start(t_1)$
f_1
R_1
f_1
b_1
E_1
T_1
a_2
b_2
I_2
a_2
$start(t_2)$
f_2
R_2
f_2
b_2
E_2
T_2

Figure 2.23: FIFO.

of a tasks is the same as in example 2.3.3. Except that each task has an additional clock variable t_i, used to measure the duration of its waiting time in the Ready state (R_i). Moreover, on the action a_i, it executes the statement $start(t_i)$, which starts a timer on its clock variable t_i, to measures the time it has been waiting in R_i.

The FIFO constraints resolve the conflict between the processes competing for the acquisition of the common resource. Unlike a non-deterministic choice of the previous example, when both the tasks are ready, the task which has waited longer is allowed to get the resource. This is represented in the priority model by the following dynamic priority rules: $b_1 \prec b_2$ if $(t_1 \leq t_2)$, and $b_2 \prec b_1$ if $(t_2 \leq t_1)$.

The models of tasks we have seen so far are non-preemptable. We now propose models of preemptable tasks, needed for preemptive scheduling policies.

Example 2.3.5 (Fixed priority preemptive scheduling.)

We model fixed priority scheduling with preemption for n tasks sharing a common resource (figure 2.24). The scheduler gives preference to low index tasks.

The behavior model of a preemptable task (T_i) is a simple extension of a non-preemptable tasks, with an additional control state S_i (suspended), and actions p_i (preempt) and r_i (resume).

Mutual exclusion between execution states E_i is guaranteed by the connectors in the model, which are $b_i' p_j$ and $f_i' r_j$. As b_i is a trigger and p_j is a synchron, the possible interactions in the connector $b_i' p_j$ are either b_i or $b_i p_j$. Now the interaction b_i alone cannot take place if $b_i p_j$ is possible (by maximal progress rule within a connector). Hence mutual exclusion is guaranteed by preemption. Similarly, the connector $f'_i r_j$ guarantees that a preempted task can resume (by executing the action r_j) only if a task terminates by executing f_i. However, as f_i is a trigger, the executing task can terminate also when no tasks are preempted.

Scheduling constraints resolve conflicts between processes (b_i and r_i actions) competing for the acquisition of the common resource. They are implemented by adding a third layer with the following priority rules. The priorities $b_i \prec b_j$, $b_i p_k \prec b_j p_k$ and $f_k r_i \prec f_k r_j$ for all k and $i > j$ enforces the static priority for accessing the resources, with the lower index process (j) getting higher preference. The rules $b_i p_j \prec f_j$ for $i > j$ ensures that a high priority task (index j) is not preempted by a low priority task (index i), i.e., the interaction $b_i p_j$ is suppressed by f_j. This ensures termination of the high priority task than preemption. The same scheduling policy can

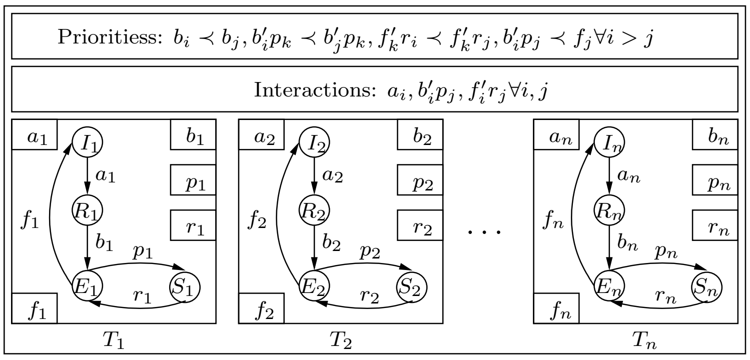

Prioritiess: $b_i \prec b_j, b'_i p_k \prec b'_j p_k, f'_k r_i \prec f'_k r_j, b'_i p_j \prec f_j \forall i > j$
Interactions: $a_i, b'_i p_j, f'_i r_j \forall i, j$
a_1 b_1 p_1 r_1 f_1 I_1 R_1 E_1 S_1 T_1
a_2 b_2 p_2 r_2 f_2 I_2 R_2 E_2 S_2 T_2
$\dots$
a_n b_n p_n r_n f_n I_n R_n E_n S_n T_n

Figure 2.24: Fixed-priority preemptive scheduling.

also be implemented by a different set of glues, as shown in figure 2.25.

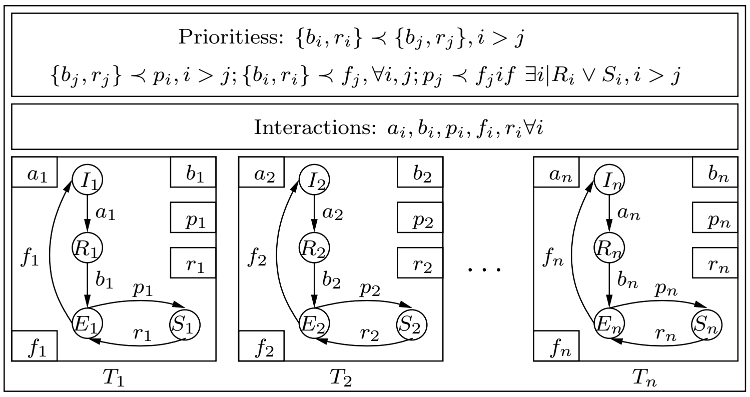

Prioritiess: $\{b_i, r_i\} \prec \{b_j, r_j\}, i > j$
$\{b_j, r_j\} \prec p_i, i > j; \{b_i, r_i\} \prec f_j, \forall i, j; p_j \prec f_j if \exists i | R_i \vee S_i, i > j$
Interactions: $a_i, b_i, p_i, f_i, r_i \forall i$
a_1 b_1 p_1 r_1 f_1 I_1 R_1 E_1 S_1 T_1
a_2 b_2 p_2 r_2 f_2 I_2 R_2 E_2 S_2 T_2
$\dots$
a_n b_n p_n r_n f_n I_n R_n E_n S_n T_n

Figure 2.25: Fixed-priority preemptive scheduling (with a different glue).

Maximal Progress Priority in Connectors

A particular priority rule, that favors, among the enabled interactions of a connector, the maximal one, i.e., the one with maximum number of ports, is known as maximal progress priority. This can be explicitly represented through priority rules amongst the interactions, of the form $x \prec xy$, where x and xy are interactions of the connector. As an example, maximal progress is necessary to model a broadcast. Maximal progress is implicitly assumed in connectors for their compact and natural representation.

2.4 Classification of Systems

In this section, we show that timed and synchronous systems can be represented as BIP components.

2.4.1 Timed Systems

Timed systems are modeled as timed automata with urgency [BS00, BST98], where the execution of a transition is an action defining a change in control state, whereas time progresses at control states. Urgency is expressed by means of an urgency attribute on transitions. This attribute can take the values *eager*, *lazy* or *delayable*. The latter is a composite type very useful in practice. It means that the transition is considered to be lazy at some state if it remains enabled at the next time unit; otherwise, it is eager. *Eager* transitions are executed at a point of time at which they become enabled, if they are not disabled by another transition. *Delayable* transitions cannot be disabled by time progress. *Lazy* transitions may be disabled by time progress.

In the graphical notation, urgency types are associated with ports (transition labels). We use the notation p^τ, where p is a port and τ can be either ϵ (eager), λ (lazy), or δ (delayable).

Like in timed automata, time distances between actions are measured by clock variables, declared in atoms. They can be set to some value or reset in transitions. They can be used in timed guards to restrict the time points at which transitions can be taken.

Local clock variables allow the specification of timing constraints, such as duration of tasks (modeled by time passing in a control state associated with the task), and deadlines for actions in a process. This is illustrated later in the chapter through examples.

We define *t*imed components and provide a structure preserving mapping from timed components to BIP components defined in section 2.3. Timed components are built from atomic timed components. The definition of atomic timed components for discrete time is inspired from [AGS02]. They have:

- A set of ports $P = \{p_1 \dots p_n\}$.
- A set of control states $S = \{s_1 \dots s_k\}$.
- A set of *v*ariables V, partitioned into two sets U and X, respectively the set of *u*ntimed and *t*imed variables.
- A set of transitions of the form $(s_1, p^\tau, g_p^u \wedge g_p^t, f_p, s_2)$, representing a step from control state s_1 to s_2. The *u*rgency type τ which can be either *e*ager or *l*azy is used to characterize the urgency of the transition. As for ordinary transitions, $g_p = g_p^u \wedge g_p^t$ is a guard, conjunction of conditions on untimed and timed variables respectively and f_p is a function on V.
- A set of *evolution functions* $\{\phi_i\}_{1 \leq i \leq k}$ in bijection with control states. The function ϕ_i gives for a given valuation x of X at control state s_i and a given value of a discrete time parameter t, the valuation $\phi_i(x,t)$ reached when time progresses by t. Further, it satisfies the following properties:
  - $\phi_i(x, 0) = x$, and
  - $\phi_i(x, t_1 + t_2) = \phi_i(\phi_i(x, t_1), t_2)$.

An atomic timed component C represents a transition system [AGS02] in the following manner.

Let s_i be a control state of C and (u, x) be a valuation of (U, X). From the state (s_i, u, x),

- Either an enabled transition can be executed independently of its urgency type - the semantics is the same as for transitions of BIP components.
- Or time can progress by one time unit Δ, leading to state $(s_i, u, \phi_i(x, \Delta))$, if all eager transitions are disabled.

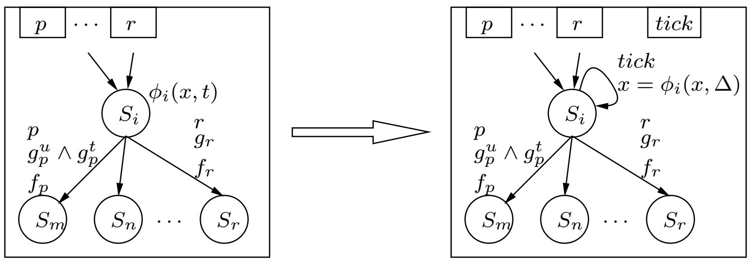


Figure 2.26: From Timed Component to BIP Component.

We provide a translation from timed to BIP atomic components. The principle of the translation is explained in figure 2.26. It consists in implementing for each atomic component, time progress from s_i and subsequent state changes, by a loop transition with guard *true* and function $\phi_i(x, \Delta)$. This transition is labeled by a special port called *tick*, for synchronization with other timed components. The translation is parameterized by Δ, which must be the same for all the atoms in the system.

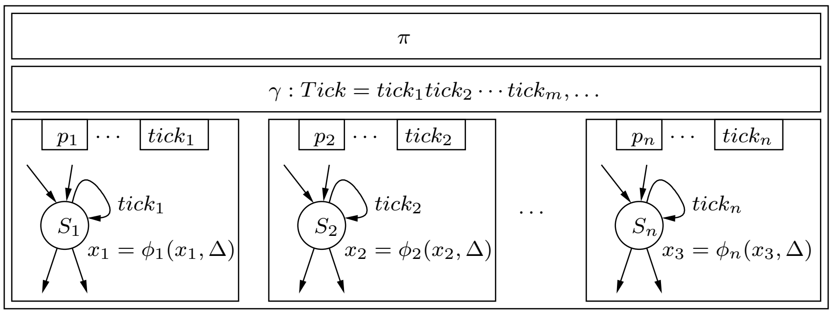

π
$\gamma : Tick = tick_1 tick_2 \cdots tick_m, \ldots$
p_1 $\cdots$ $tick_1$
$tick_1$
S_1
$x_1 = \phi_1(x_1, \Delta)$
p_2 $\cdots$ $tick_2$
$tick_2$
S_2
$x_2 = \phi_2(x_2, \Delta)$
$\ldots$
p_n $\cdots$ $tick_n$
$tick_n$
S_n
$x_3 = \phi_n(x_3, \Delta)$

Figure 2.27: Composition of Timed Components.

In the composition of the resulting BIP components from the timed components, strong synchronization is necessary between all the *tick* ports as shown in the architecture of figure 2.27. That is, a connector called *Tick*, relating all the *tick* ports is used. It is defined as a rendezvous between the *tick* ports.

Furthermore, to take into account urgency of the transitions, we use priorities.

$$\{(at\ S_i \wedge g_p) \rightarrow Tick \prec \alpha \mid \exists p\,.\ S_i \xrightarrow{p^{\epsilon} g_p f_p} S_i' \wedge p \in \alpha\}$$

The meaning of the priority rule is that, if there is an eager transition enabled from in a component, time has lower priority than this transition. Hence time can progress only if there are no enabled eager transitions.

If a port p has a unique occurrence in the behavior, then the predicate $(at\ S_i \wedge g_p)$ in the priority rules can be removed, as its will be already included in the guard by the enabledness of α.

Example 2.4.1 (Billiard Balls) *We provide an example of a timed system by modeling the movement of two billiard balls, colliding on a billiards table and with each other. Each ball moves independently, starting from an initial position with given velocities and direction. We assume zero friction between the balls and the board. The balls are reflected at the edges of the billiards table, and may collide with each other. On collision, they get their velocities and directions exchanged, assuming perfectly elastic collision.*

A ball moves in the plane of the board, its motion being the product of its movement in two dimensions, the X and the Y co-ordinate respectively. We identified a co-ordinate to be the basic atomic component of the model.

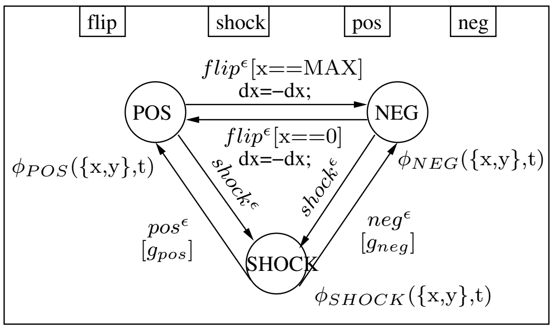


Figure 2.28: Architecture of a co-ordinate.

A timed model for the co-ordinate atom is shown in figure 2.28. It has three control states, POS, NEG and $SHOCK$, and is parameterized by an integer MAX denoting the maximum distance it can travel before being re-bounced. It has a timed variable x representing its current position, and dx representing its velocity. There is also a clock variable y. The port $flip$ denotes a re-bounce event, while *shock* represents a collision. It can remain at the state POS until its has reached the MAX displacement, whence it is reflected, resulting in the reversal of its displacement (modeled by changing the sign of dx). Similarly, it can stay at NEG until it has reached the origin. From either of these states, it can suffer a collision (*shock*), whence it moves to the control state $SHOCK$. From here, the two actions *pos* and *neg* with exclusive guards determine whether to move it to POS or NEG. The guard for the action *pos* (g_{pos}) is $((dx > 0) \wedge (x < MAX)) \vee ((dx < 0) \wedge (x == 0))$, and that for *neg* ($g_{neg}$) is $((dx > 0) \wedge (x == MAX)) \vee ((dx < 0) \wedge (x > 0))$. All the actions, $flip$, *shock*, *pos* and *neg* are eager. The evolution functions specifying the updation of x and y with time, are the same for all the control states, and defined as:

$$\phi_s(\{x, y\}, t) = \begin{cases} x + t.dx \\ y + t \end{cases}$$

where $s = POS, NEG, SHOCK$. The *shock* transition has a guard consisting the clock variable y, in order to prevent zero-delay collisions. This may happen because after the *shock* interaction, the co-ordinates of the balls remain the same. Hence, in order to force the time to progress, we provide the guard, $y > 0$ to the *shock* transition, and reset y ($y = 0$;) as the action.

A ball is a compound component, consisting of two co-ordinates, X and Y. The interaction model specifies the synchronizing between the *shock* ports of X and Y. The architecture of the *Ball* is shown in figure 2.29. It shows the model after the timed transformation of the co-ordinate components. Note that, after the timed transformations, the co-ordinates have

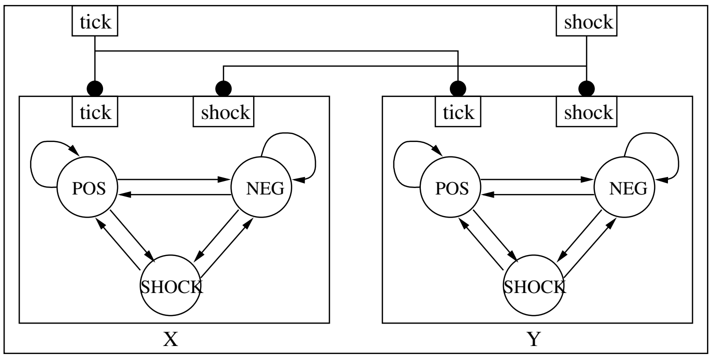


Figure 2.29: Architecture of a Ball.

additionally the *tick* ports, and all of them are synchronized through a global *tick* connector.

The system *Billiard* contains two *Ball*'s, $ball_1$ and $ball_2$, as shown in 2.30. The interaction model specifies a synchronization between all the

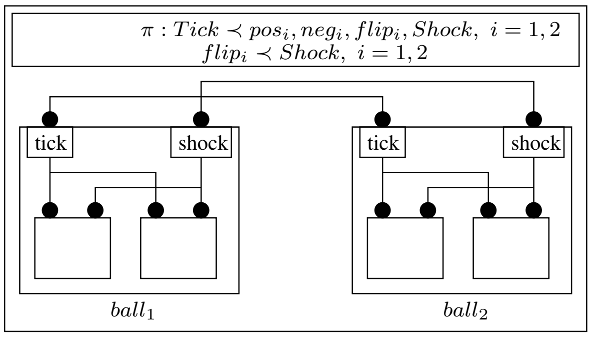

$\pi : Tick \prec pos_i, neg_i, flip_i, Shock, \; i = 1, 2$
$flip_i \prec Shock, \; i = 1, 2$
tick
shock
tick
shock
$ball_1$
$ball_2$

Figure 2.30: Architecture of the system with two balls.

shock connectors. It is a rendezvous of all the *shock*s, with a guard which checks the condition that the co-ordinate position (X and Y) of the balls are the same, i.e., the condition necessary for a collision are met. It is associated with a transfer function, which interchanges the velocities (dx) of the balls in both the dimensions, X and Y. Structured connectors are also created to synchronize the $Tick$ connectors of the balls. The system also contains the connectors modeling the individual interactions for $flip$, *pos* and *neg* for each co-ordinate, not shown in the figure.

The urgency criteria for the timed transitions leads to assigning the $Tick$ interaction the lowest priority. Also, we need the *shock* interaction to have higher priority than the $flip$ interactions for correct behavior.

Example 2.4.2 (Scheduling of Timed Tasks)

This example taken from [WDE05], is a performance evaluation problem with timed tasks processing events from a bursty event generator. There are three tasks T_1, T_2 and T_3, connected serially to the event generator. A task is activated by an event from its predecessor, and executes on dedicated CPU's. On completion, the task passes the event to its successor. The block diagram of the *System* is shown in figure 2.31(a). We design a model to measure the total delay of an event, starting from its creation from the event generator till it is processed by T_3. The basic components for the model are

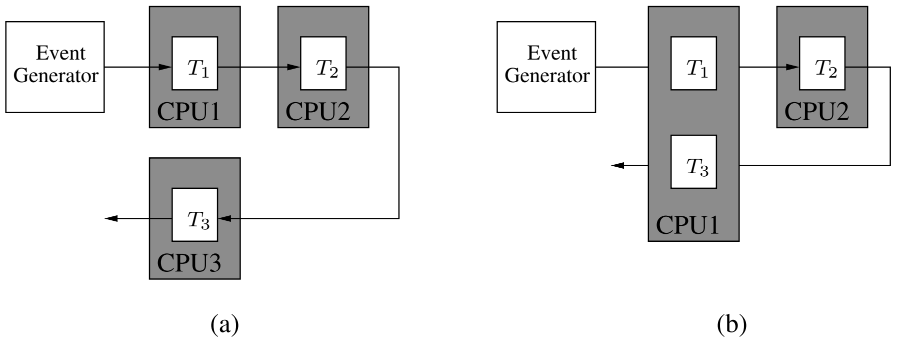


Figure 2.31: Scheduling of timed tasks.

Task and *EventGenerator*. A generic timed model of *Task* is shown in figure 2.32, which can be used either as a simple task, or as a preemptable task. It has a timed variable d to keep track of the execution time. The evolution functions specifying the updation of d with time is given as:

$$\begin{aligned}\phi_{Rdy}(d,t) &= d\\ \phi_{Susp}(d,t) &= d\\ \phi_{Exe}(d,t) &= d+t\end{aligned}$$

The urgency on the transition labeled by $finish^\delta$ in figure 2.32 means that the transition is lazy when $d < WCET$ and eager when $d = WCET$.

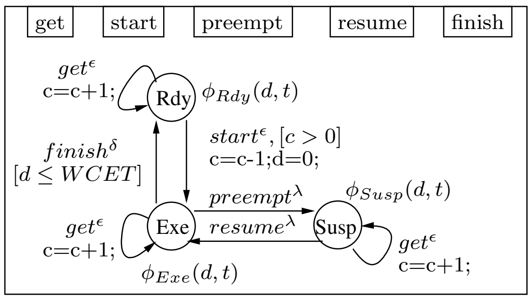


Figure 2.32: Task Component.

The ports and behavior of *EventGenerator* are shown in figure 2.33. It has a period T, a jitter J with $J > T$, and a minimum inter-arrival time

D between successive events being generated. The compound component

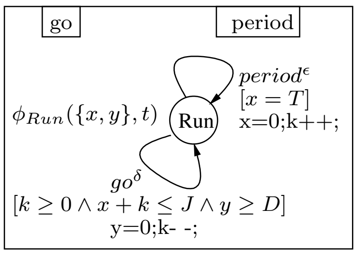

go
period
$period^{\epsilon}$
$[x = T]$
x=0;k++;
$\phi_{Run}(\{x, y\}, t)$
Run
go^{δ}
$[k \geq 0 \wedge x + k \leq J \wedge y \geq D]$
y=0;k- -;

Figure 2.33: EventGenerator Component.

System is a serial connection of the *EventGenerator* instance *EvntGen*, and three instances of $Task$, T_1, T_2 and T_3, as shown in figure 2.34. Component instances are parameterized: *EventGenerator* instance by the time-period T, jitter J and minimum inter-arrival time D; $Task$ instances by their worst case execution time $WCET$. The transmission of the events, i.e., synchronization between the *EvntGen* and $Task$s are modeled by connectors. We use the notation T_1*.tick* to mean the *tick* port of T_1. With this notation for ports, the connectors are as follows:

$$\gamma = \begin{cases} EvntGen.tick\ T_1.tick\ T_2.tick\ T_3.tick & (C_0) \\ EvntGen.period' & (C_1) \\ EvntGen.go\ T_1.get & (C_2) \\ T_1.finish\ T_2.get & (C_3) \\ T_2.finish\ T_3.get & (C_4) \\ T_2.start' & (C_5) \\ T_1.start' & (C_6) \\ T_3.start' & (C_7) \\ T_3.finish' & (C_8) \end{cases}$$

C_0 is a rendezvous between the *tick* ports of all the components. C_2 is a rendezvous between port *go* of *EvntGen* and port *get* of T_1. Synchronization on this connector represents an event generated by *EvntGen* and its transmission to T_1. Similarly C_3 represents an event flow from T_1 to T_2. The connectors C_1, C_5, C_6, C_7 and C_8 represents trivial interactions, each with only one trigger port. In this model, the tasks are non-preemptable, so the ports *preempt* and *resume* are not associated to connectors. They are not involved in any interaction. Priorities are used to enforce the urgency criteria of the transitions (not shown in the figure). For example, to enforce the urgency for the *period* interaction in *EvntGen*, we have the rule

$$(x == T) \rightarrow EvntGen.tick\ T_1.tick\ T_2.tick\ T_3.tick \prec EvntGen.period$$

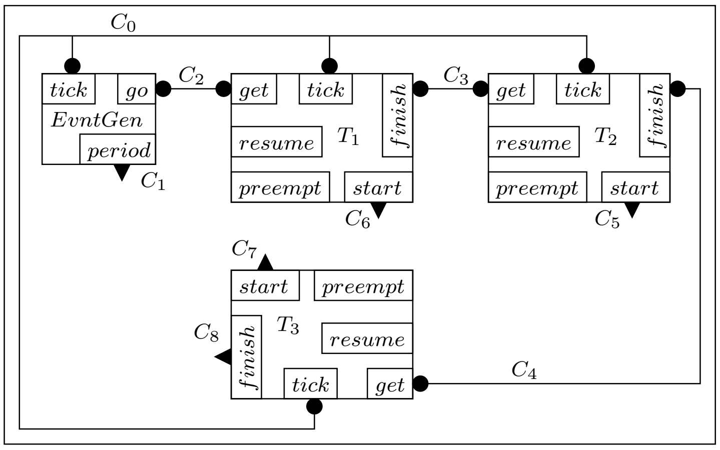


Figure 2.34: Timed task architecture.

The second part of the the problem is to model the *System* where tasks T_1 and T_3 share CPU1 and T_1 can preempt T_3, whereas T_2 runs on CPU2, as shown in figure 2.31(b). The BIP architecture of the model (without the priorities enforcing urgency) is shown in figure 2.35.

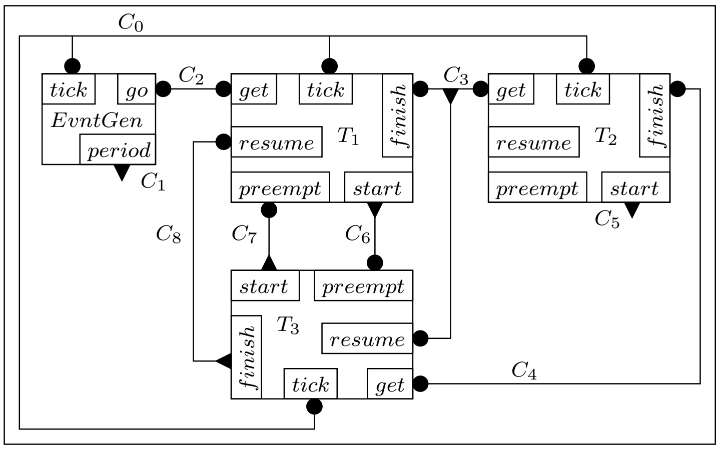


Figure 2.35: Timed task architecture (with mutual exclusion).

The connectors in this architecture are

$$\gamma = \begin{cases} EvntGen.tick\ T_1.tick\ T_2.tick\ T_3.tick & (C_0) \\ EvntGen.period' & (C_1) \\ EvntGen.go\ T_1.get & (C_2) \\ [T_1.finish\ T_2.get]'\ T_3.resume & (C_3) \\ T_2.finish\ T_3.get & (C_4) \\ T_2.start' & (C_5) \\ T_1.start'\ T_3.preempt & (C_6) \\ T_3.start'\ T_1.preempt & (C_7) \\ T_3.finish'\ T_1.resume & (C_8) \end{cases}$$

Mutual exclusion between T_1 and T_3 is enforced by the connectors C_3, C_6, C_7 and C_8. They guarantee that a task will preempt if the other has to start, and similarly a task can resume only when the executing task finishes. Note that C_3 is a structured connector, representing the interactions $T_1.finish\ T_2.get$ and $T_1.finish\ T_2.get\ T_3.resume.$ In the first case T_1 finishes and transmits an event to T_2, in the second case it additionally releases the resource to T_3 by resuming its execution.

Static priority between T_1 and T_3 is enforced by the rule

$$true \rightarrow T_3.start \prec T_1.start$$

and non-preemption of T_1 by T_3 is realized by the rule

$$true \rightarrow T_3.start\ T_1.preempt \prec EvntGen.tick\ T_1.tick\ T_2.tick\ T_3.tick$$

Note that for the urgency of the $finish$ transition, we have the rule

$$(d == WCET) \rightarrow EvntGen.tick\ T_1.tick\ T_2.tick\ T_3.tick \prec T_1.finish$$

and the composition of the above two rules produce

$$(d == WCET) \rightarrow T_3.start\ T_1.preempt \prec T_1.finish$$

This ensures that T_1 will $finish$ instead of releasing the CPU1 to T_3.

2.4.2 Synchronous Systems

Synchronous components are a subclass of components built from synchronous atomic components which have:

- A set of control states S, partitioned into three sets $S^{st} \neq \emptyset$, S^{un}, $S^{syn} \neq \emptyset$, respectively the sets of *stable*, *unstable* and *synchronization* states.
- A particular class of transitions, *synchronization* transitions. These have a special port *syn*, their guards are true and functions are empty. From synchronization states only synchronization transitions are possible and lead to stable states.

- Transitions from unstable states leading either to unstable states or to synchronization states.

- Stable states with synchronization transition loops. All other transitions from these states lead to unstable or synchronization states. There is a finite number of transitions in a path between two successive distinct stable states in the transition graph. This guarantees that for deadlock free execution, stable states are visited infinitely often.

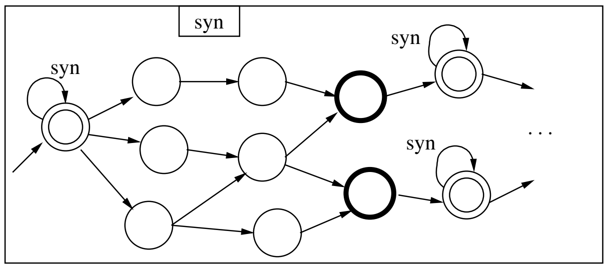


Figure 2.36: Synchronous Atomic Component.

The general form of a synchronous atomic component is shown in figure 2.36. Composition of synchronous components is characterized by the following requirements:

- Ports of transitions from unstable states must belong to triggers, that is, their execution cannot be blocked due to synchronization constraints.

- All the *syn* ports are strongly synchronized via a connector *Syn* with empty set of complete interactions.

- Any interaction has higher priority than the *syn* interaction. This enforces progress from stable states and prevents live-lock by looping on *syn* transitions.

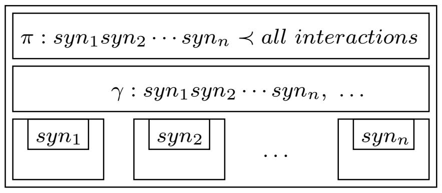

$\pi : syn_1 syn_2 \cdots syn_n \prec all\ interactions$
$\gamma : syn_1 syn_2 \cdots syn_n,\ \ldots$
syn_1
syn_2
$\ldots$
syn_n

Figure 2.37: Composition of Synchronous Components.

The general form of a synchronous architecture is shown in figure 2.37. The above requirements ensure lockstep execution. For such an execution, a run is a sequence of steps leading from one global stable state to another. If all the components are at stable states, then each component can execute a sequence of transitions leading to a synchronization state. From synchronization states, strong synchronization through *syn* is enforced. This interaction cannot occur as long as other interactions are possible, as it has the lowest priority. It may happen that during a synchronous step some components stay at the same stable state, if the transitions leaving this state are not enabled. The synchronization transition loops from stable states allow transitions from synchronization states of the other components to be executed.

Example 2.4.3 (Synchronous Counter)

We model synchronous modulo-n counter as a composition of n single bit counters. The i^{th} bit is represented by an atomic component modeling a modulo-2 counter 2.38. We show the construction of a modulo-8 counter by composing three modulo-2 counters, shown in figure 2.39. The behavior of

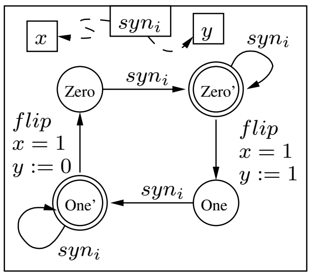


Figure 2.38: A Modulo-2 Counter.

the atomic component *mod*2 is depicted in figure 2.38. *Zero*′ and *One*′ are *stable* states whereas *Zero* and *One* are *synchronization* states respectively. The port *flip* denotes an internal action. The variable x is the input and y is the output. The compound component *mod*8 consists of 3 instances of

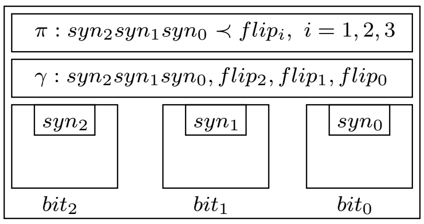

$\pi : syn_2 syn_1 syn_0 \prec flip_i,\ i = 1, 2, 3$
$\gamma : syn_2 syn_1 syn_0, flip_2, flip_1, flip_0$
syn_2
syn_1
syn_0
bit_2
bit_1
bit_0

Figure 2.39: A Modulo-8 Counter.

*mod*2 namely bit_0, bit_1, and bit_2, where bit_0 is the least significant bit. The interaction model consists of the interaction synchronizing the syn_i ports of all the bits. The architecture of the composition is shown in figure 2.39.

The interaction is associated with a data transfer relation, which modifies the variables of the respective components on every synchronization step as follows: $bit_0.x := 1$; $bit_1.x := bit_0.y$; $bit_2.x := bit_1.y \wedge bit_1.y$;

Priority rules enforce that the synchronization interaction has lower priority than the internal actions of the components.The y values constitute the output of the system, and varies from 000 to 111.

2.5 The System Construction Space

The BIP framework shares features with existing ones for heterogeneous components, such as [BWH+03, EJL+03, BGK+06, Arb05]. A common key idea is to encompass high-level structuring concepts and mechanisms. Ptolemy was the first tool to support this by distinguishing between behavior, channels, and directors. Similar distinctions are also adopted in Metropolis and BIP, which offer interaction-based and control-based mechanisms for component integration. The two types of mechanisms correspond to *cooperation* and *competition*, two complementary fundamental concepts for system organization.

This is a significant progress with respect to languages directly supporting only interaction-based mechanisms of such as CSP, Lotos, Java. There is evidence through numerous examples treated in BIP, that the combination of interactions and priorities allow enhanced modularity and direct modeling of schedulers, quality controllers and quantity managers. Of course, one could advocate that ease of description for rich languages without an adequate methodology may be at the detriment of simplicity and insight gained through the use of a smaller number constructs and concepts. The comparison of languages based on a set of rigorous and pertinent criteria is an issue that deserves further investigation.

BIP characterizes systems as points in a three-dimensional space: $Behavior \times Interaction \times Priority$, as represented in figure 2.40. Elements of the $Interaction \times Priority$ space characterize the overall *architecture*. Each dimension, can be equipped with an adequate partial order, e.g., refinement for behavior, inclusion of interactions, inclusion of priorities. Some interesting features of this representation are the following:

Separation of concerns: Any combination of behavior, interaction and priority models meaningfully defines a component. This is not the case for other formalisms e.g., in Ptolemy [EJL+03], for a given model of computation, only particular types of channels can be used. Separation of concerns is essential for defining a component's construction process as the superposition of elementary transformations along each dimension.

Unification: Different subclasses of components e.g.,untimed/timed, asyn-

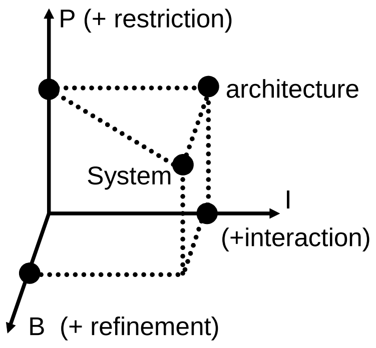


Figure 2.40: The Construction Space.

chronous/synchronous, event-triggered/data-triggered, can be unified through transformations in the construction space. These transformations often involve displacement along the three coordinates. They allow a deeper understanding of the relations between existing semantic frameworks in terms of elementary behavioral and architectural transformations. For instance, as explained in section 2.4.1, timed system can be obtained from an untimed system by 1)refinement of its untimed behavior (adding timers and tick transitions); 2) by adding a synchronous interaction between ticks; 3) by adding priorities to express urgency of timed transitions with respect to ticks.

Correctness by construction: The component construction space provides a basis for the study of architecture transformations allowing preservation of properties of the underlying behavior. The characterization of such transformations can provide (sufficient) conditions for correctness by constructions such as *compositionality* and *composability* results for deadlock-freedom [GS05]. In an ongoing work, we try to determine regions of the system construction space where properties are preserved, in particular deadlock-freeedom and state invariance.

2.6 Distributed Model of BIP

The semantic model of BIP presented in section 2.2.2 is based on the notion of global states of the system, i.e., complete state information of all the components are available, and the system moves from one global state to another. This is known as the global state semantics. In this section, we provide a distributed implementation method for systems in BIP, based on partial information about the individual component states [BBBS08]. The implementation method is a translation from BIP models into distributed models involving two steps. The first translates BIP models into partial state models where are known only the states of the components which are ready to communicate. The second implements interactions in the partial state model by using message passing primitives.

The main objectives of this work has been to identify the conditions for which the three models are observationally equivalent. We show that in general, the translation from global state to partial state models does not preserve observational equivalence. Preservation can be achieved by strengthening the premises of the operational semantics rules by an oracle. This is a predicate depending on the priorities of the BIP model. We show that there are many possible choices for oracles. Maximal parallelism is achieved for dynamic oracles allowing interaction as soon as possible.

2.6.1 Basic Concepts

A distributed system is a collection of loosely coupled independent components, communicating by explicit message passing. The components are intrinsically concurrent and their states may be known only through communication. We cannot determine the exact global state of a distributed system, we can only approximate it [CL85].

We study a distributed implementation method for the BIP (Behavior, Interaction, Priority) component framework for modeling heterogeneous systems [BBS06]. The method allows to get a distributed implementation for a given BIP model, in there steps:

• It starts from a *global state* model of the system to be implemented described in BIP. The model represents the system behavior as a transition system where transitions are atomic. The BIP execution platform uses an Engine which coordinates the execution of the components. Atomicity of transitions implies a strict alternation between the execution of components and the Engine: no interaction is possible when some component is performing a computation.

• From the global state model, a *partial state* model is derived where we distinguish between states from which components are *ready* for interaction and states where components are *busy* by executing some internal computation. For this model partial state knowledge may suffice for executing interactions. We study conditions for the partial state model to be equivalent to the global state model. The conditions are in the form of an *oracle* used by the BIP Engine to safely execute interactions in the presence of uncertainty about the global state.

• From the partial state model, a *distributed model* is obtained where atomic multiparty interactions of the partial state models are replaced by communication protocols. In this model, components exchange messages to communicate with the Engine represented by an additional component.

The main results are conditions for which the three models are observationally equivalent by considering as silent the actions corresponding to internal computations of the initial global state model. They are described in more details below.

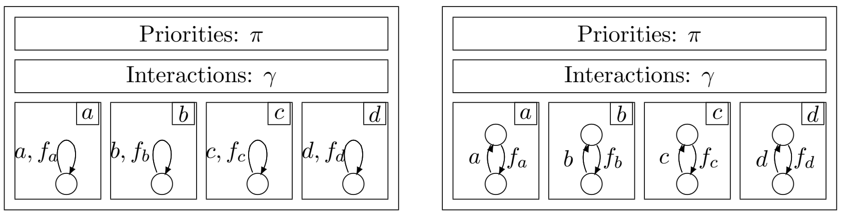
BIP combines two powerful mechanisms for describing multiparty interactions between components: *interactions* and *priorities*. A system model is layered. The lowest layer contains atomic components whose behavior is described by state machines with data and functions described in C. As in process algebras, atomic components can communicate by using ports. The second layer contains interactions which are relations between communication ports of individual components. Priorities are used to express scheduling policies by selecting amongst the enabled interactions of the layer underneath.

The current implementation of BIP is based on global state semantics. From a BIP model, a compiler is used to generate C++ code for a dedicated platform. The platform uses an Engine that directly interprets the operational semantics rules. For a given global state, the Engine computes from the set of the communication ports offered by individual components and the set of interactions, the set of the enabled interactions. Amongst these, the Engine chooses a maximal one, according to the priorities of the third layer, and notifies the involved components which can continue their computation.

We define partial state semantics for BIP where the assumption of atomic execution of transitions does not hold. This is a straightforward generalization of global state semantics where interactions are separated from internal computation in the components. A component may be either in a busy state or in a ready state. A busy state corresponds to the execution of some internal computation. When the computation terminates, some ready state is reached. From this state the component can participate in interactions and move again to some busy state.

The implementation problem for a partial state model is to find an Engine that may execute interactions even for partially known states, while preserving (observational) equivalence with the corresponding global state model. The following example shows that in general, the two models are not equivalent.

Example 2.6.1 (Broadcast) *Consider a BIP model consisting of four components A, B, C, D each one offering cyclically an interaction through ports a, b, c, d followed respectively by the execution of functions f_a, f_b, f_c, f_d (Figure 2.41(a)). We assume that A is a sender and B, C, D are receivers. A can broadcast a message through a and the set of the possible interactions is $\gamma = \{a, ab, ac, ad,\ abc, abd, acd, abcd\}$. Priority rules are used to ensure that amongst all the possible interactions from a state only a maximal one is possible. This is expressed by using a priority order on interactions π with rules of the form $x \prec xy$ where x and xy are interactions. These rules say that whenever both interactions x and xy are enabled, only interaction xy*



(a) Global State Model (b) Partial State Model

Figure 2.41: A System with Four Components

can be executed. That is, maximal progress is enforced. For this example, the only possible interaction is abcd and thus the functions f_a, f_b, f_c, f_d are executed synchronously.

The partial state model for this system is shown in Figure 2.41(b). It is possible, due to the separation between interaction and internal computation, to reach a configuration where the receivers are in a busy state. In that case, only the ready components will be synchronized. Thus an arbitrary desynchronization of the receivers with respect to the sender is possible.

Example 2.6.2 (Rendezvous) *Consider again the previous example where broadcast is replaced by three rendezvous: $\gamma = \{ab, bc, cd\}$ and π is such that $ab \prec bc$, $cd \prec bc$ in the global state system. This system executes forever the interaction bc. Consider the corresponding partial state system where interactions are separated from functions. For this system, it is possible to execute the sequence $ab.(f_a.cd.f_c.f_b.ab.f_d)^\omega$ which goes through states never enabling the interaction bc.*

The above examples motivate the definition of partial state semantics where the premises of the operational semantics rules include an oracle, a predicate parameterized by a dependency relation between interactions. The dependency relation is an abstraction of the priorities of the initial BIP model. The oracle characterizes the partial states from which an interaction can be safely executed: if an interaction a_1 depends on an interaction a_2, then a_1 cannot be executed if the system has some internal evolution leading to a state enabling a_2. We show that there are many possible choices for oracles. If the time for computing them is negligible, best performance is achieved for oracles allowing interaction as soon as possible in order to reduce waiting times of ready components. The worst performing oracle is the one allowing interaction only when all the components are at ready states. For this oracle partial and global state semantics coincide.

We study a transformation from the partial state model to a distributed one. This consists in replacing atomic interactions by protocols using message passing. For distributed semantics, the Engine becomes an additional component. The results are applied to obtain a multi-threaded implementation for BIP. We analyze performance of this implementation for different types of oracles as well as with respect to the global state semantics model.

The method presented is not specific to BIP and can be applied for the implementation of systems in particular in two cases. First, for concurrent systems with fairness constraints which at implementation level, become scheduling policies expressed by dynamic priorities. Second, for systems involving communication by broadcast. This requires mechanisms for identifying the maximal set of interacting components that can be specified by using priorities. Consequently, the proposed method can be used for correct implementation.

The chapter is organized as follows. In section 2.6.2, we present the partial state semantics for BIP. In section 2.6.4, we study oracles and their properties. We show correctness of partial state semantics enforced by an oracle with respect to global state semantics.

2.6.2 Partial State Semantics

The model with global state semantics presented in section 2.2.2 is based on the fact that transitions are atomic and a global state is always defined. To obtain the partial state model corresponding to a global state model, we 1) replace atomic components by their partial state models; 2) extend the operational semantics rules for interactions and priorities.

Atomic Components To model concurrent behavior, we associate with each atomic component, its corresponding *partial state model.* Atomic components with partial states behave as atomic components with the difference that each transition is decomposed into a sequence of two transitions: an interaction (visible transition) followed by an internal computation or *busy transition.* Between these two transitions, a new *busy* state is added. Busy states are transient states considered by the Engine as undefined states of the component.

Definition 2.6.3 (Atomic Component with Partial States) *Given an atomic component $B = (Q, P, \rightarrow)$, we define the associated partial state model as the transition system $B^{\perp} = (Q \cup Q^{\perp}, P \cup \{\beta\}, \rightsquigarrow)$ where*

- $Q^{\perp} = \{q_t \mid t \in\rightarrow\}$ *such that* $Q^{\perp} \cap Q = \emptyset$. $Q^{\perp}$ *is a set of busy states in bijection with the set of transitions* $\rightarrow$.
- β *is a port name not in* P
- $\rightsquigarrow\subseteq (Q \cup Q^{\perp}) \times P \cup \{\beta\} \times (Q \cup Q^{\perp})$ *where if* $t = (q_1, p, q_2) \in\rightarrow$, *then* $q_1 \overset{p}{\rightsquigarrow} q_t$ *and* $q_t \overset{\beta}{\rightsquigarrow} q_2$.

Interaction We define below interactions for partial state models.

Definition 2.6.4 *Given a BIP model built from a set of atomic components $\{B_i = (Q_i, P_i, \rightarrow_i)\}_{i=1}^n$, of the form $\gamma(B_1, \ldots, B_n)$, we define the corresponding partial state model $\gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$ such that*

- $B_i^\perp$ *is the partial state model* $B_i^\perp = (Q_i \cup Q_i^\perp, P_i \cup \{\beta_i\}, \rightsquigarrow_i)$
- $\gamma^\perp = \gamma \cup \{\beta_i\}_{i=1}^n$

Notice that $\gamma^\perp(B_1^\perp, \ldots, B_n^\perp) = (\bigotimes_{i=1}^n (Q_i \cup Q_i^\perp), \gamma^\perp, \rightsquigarrow)$. The transition relation $\rightsquigarrow$ can be equivalently defined by the rules:

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

$$\frac{q_i \overset{\beta_i}{\rightsquigarrow} q_i'}{(q_1, \ldots, q_i, \ldots, q_n) \overset{\beta_i}{\rightsquigarrow} (q_1, \ldots, q_i', \ldots, q_n)}$$

The first rule is the same as the composition rule for the global state semantics. The second rule defines the busy transitions of the composite system.

The state space can be split into two disjoint sets $\bigotimes_{i=1}^n (Q_i \cup Q_i^\perp) = Q^g \cup Q^p$. The set of *global states* $Q^g = \bigotimes_{i=1}^n Q_i$ which is the set of states of $\gamma(B_1, \ldots, B_n)$. The set of *partial states* Q^p where at least one component is busy.

Definition 2.6.5 *For $q, q' \in Q^p \cup Q^g$, we write $q \overset{\beta}{\rightsquigarrow} q'$ if $q \overset{\beta_i}{\rightsquigarrow} q'$ for some i.*

Property The relation $\overset{\beta}{\rightsquigarrow}$ is terminating and confluent. Thus, from any partial state, a unique global state is eventually reached by executing β-transitions.
Priority The above property is used to define priorities for partial state models. The priority relation at some partial state should agree with the priority relation at the global state reached by executing β-transitions.

Definition 2.6.6 *Given a BIP model $\pi\gamma(B_1, \ldots, B_n)$, the corresponding partial state model is $\pi^\perp\gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$ where $\pi^\perp \subseteq \gamma \times (Q^g \cup Q^p) \times \gamma$ such that $a_1 \pi_q^\perp a_2$ if $\exists q' \in Q^g.\ q \overset{\beta^*}{\rightsquigarrow} q' \wedge a_1 \pi_{q'} a_2$.*

Note that $\pi^\perp$ is a priority and it coincides with π on Q^g.

Example 2.6.7 (Broadcast/Rendezvous with partial states) *The partial state model for the system in example 2.6.1 has the atomic components $A^\perp$, $B^\perp$, $C^\perp$ and $D^\perp$ with two states and two transitions defined by*

$$X^\perp = (\{q_X, q_X^\perp\}, \{x, \beta_X\}, \{(q_X, x, q_X^\perp), (q_X^\perp, \beta_X, q_X)\})$$

where $(X, x) \in \{(A, a), (B, b), (C, c), (D, d)\}$. For the broadcast, $\gamma^\perp = \{a, ab, ac, ad, abc, abd, abcd\} \cup \{\beta_A, \beta_B, \beta_C, \beta_D\}$ and $\pi^\perp$ is such that for all states q in $\gamma^\perp(A^\perp, B^\perp, C^\perp, D^\perp)$, we have $\pi^\perp_q = \{(x, xy) \mid (x, xy) \in \gamma^2\}$. For the rendezvous (example 2.6.2), we have $\gamma^\perp = \{ab, bc, cd\} \cup \{\beta_A, \beta_B, \beta_C, \beta_D\}$ and $\pi^\perp$ is such that for all states q in $\gamma^\perp(A^\perp, B^\perp, C^\perp, D^\perp)$, we have $\pi^\perp_q = \{(ab, bc), (cd, bc)\}$.

2.6.3 Comparing Global and Partial State Semantics

We study sufficient conditions for partial state models to be behaviorally equivalent to global state models. We use observational equivalence [Mil95] for this comparison by considering that β-transitions are not observable. As noticed in the introduction (Example 2.6.1), observational equivalence is not preserved. The systems $\pi\gamma(A, B, C, D)$ and $\pi^\perp\gamma^\perp(A^\perp, B^\perp, C^\perp, D^\perp)$ are not observationally equivalent. The global state model can perform only the maximal interaction *abcd* while in the partial state model, non maximal synchronization is possible. For instance, we have the transitions:

$$(q_A, q_B, q_C, q_D) \overset{abcd}{\leadsto} (q_A^\perp, q_B^\perp, q_C^\perp, q_D^\perp) \overset{\beta}{\leadsto} (q_A, q_B^\perp, q_C^\perp, q_D^\perp) \overset{a}{\leadsto} (q_A^\perp, q_B^\perp, q_C^\perp, q_D^\perp)$$

Thus, in general, a BIP model is not observationally equivalent to its partial state model. Nonetheless, the following theorem shows that if $\pi = \emptyset$, $\gamma(B_1, \ldots, B_n)$ and $\gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$ are observationally equivalent.

We define observational equivalence of two transition systems $A = (Q_A, L \cup \{\beta\}, \rightarrow_A)$ and $B = (Q_B, L \cup \{\beta\}, \rightarrow_B)$. It is based on the usual definition of weak bisimilarity where β-transitions are considered unobservable.

Definition 2.6.8 (Weak Simulation) *A weak simulation over A and B is a relation $R \subseteq Q_A \times Q_B$ such that we have $\forall (q, r) \in R,\ a \in L.\ q \xrightarrow{a}_A q' \implies \exists r'.\ (q', r') \in R \wedge r \xrightarrow{\beta^* a \beta^*}_B r'$ 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 simulations. We say that A and B are observationally equivalent and we write $A \sim B$ if for each state of A there is a weakly bisimilar state of B and conversely.

We use this definition to compare partial state and complete state semantics.

Theorem 2.6.9 $\gamma(B_1, \ldots, B_n) \sim \gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$

2.6.4 Partial State Semantics with Oracles

Let $\gamma(B_1, \ldots, B_n)$ be a system obtained as the composition of atomic components $B_i = (Q_i, P_i, \rightarrow_i)$ by using a set of interactions $\gamma \subseteq 2^P$ where

$P = \bigcup_{i=1}^{n} P_i$. The corresponding partial state system $\gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$ consists of the components $B_i^\perp = (Q_i \cup Q_i^\perp, P_i \cup \{\beta_i\}, \rightsquigarrow_i)$ composed by using interactions in $\gamma^\perp$. As above, we take $\bigotimes_{i=1}^{n}(Q_i \cup Q_i^\perp) = Q^g \cup Q^p$. We also suppose that π is a priority for $\gamma(B_1, \ldots, B_n)$, and $\pi^\perp$ is its extension to partial states.

2.6.4.1 Basic Definitions and Properties

For a system $\gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$, a state $q \in Q^g \cup Q^p$ and an interaction $a \in \gamma$, we say that a is *enabled* at state q and we write $Act(q, a)$, if the transition a is possible from state q. That is, $q \overset{a}{\rightsquigarrow} q'$ for some state q'. We say that a is *disabled* at state q and we write $\texttt{disabled}(q, a)$, if there is an atomic component in a *ready state* that prevents synchronization on a. That is, if $a = \{p_i\}_{i \in I}$ there is $i \in I, q_i \in Q_i$ such that $q_i \overset{p_i}{\not\rightsquigarrow}$.

For global states, $\texttt{disabled}(q, a)$ is equivalent to $q \overset{a}{\not\rightsquigarrow}$ and in particular we always have either $\texttt{disabled}(q, a)$ or $Act(q, a)$. However, for partial states the status (disabled or enabled) of an interaction a at a given state may be unknown if some components involved in a are in busy states.

To compare partialness of states, we define a partial order relation over the states of composite components.

Definition 2.6.10 (State Ordering) *For $q, r \in Q^g \cup Q^p$, $q \leq r \iff \forall i \in \{1..n\}. (r_i = q_i \vee q_i \in Q_i^\perp)$.*

For a given relation $\pi^\perp$, an oracle is a predicate $\mathcal{O}$ on $(Q^p \cup Q^g) \times \gamma$ used to strengthen the premises of the semantic rule for $\gamma^\perp(B_1^\perp, \ldots, B_n^\perp)$. Oracles are defined so that $\pi^\perp\gamma_{\mathcal{O}}^\perp(B_1^\perp, \ldots, B_n^\perp)$ is observationally equivalent to $\pi\gamma(B_1, \ldots, B_n)$ where $\gamma_{\mathcal{O}}^\perp(B_1^\perp, \ldots, B_n^\perp)$ is the behavior restricted by the oracle. We introduce first a notion of composition with an oracle and in Subsection 2.6.4.2, we introduce oracles.

Definition 2.6.11 (Composite Components with Oracle) *Given an oracle $\mathcal{O}$ on $(Q^p \cup Q^g) \times \gamma$, we define $B \overset{def}{=} \gamma_{\mathcal{O}}^\perp(B_1^\perp, \ldots, B_n^\perp)$ as the transition system $(Q^p \cup Q^g, \gamma^\perp, \rightsquigarrow)$ where $\rightsquigarrow$ is the least set of transitions satisfying the rules*

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

$$\frac{q_i \overset{\beta_i}{\rightsquigarrow}_i q_i'}{(q_1, \ldots, q_i, \ldots, q_n) \overset{\beta_i}{\rightsquigarrow} (q_1, \ldots, q_i', \ldots, q_n)}$$

The following proposition says that a system with an oracle $\mathcal{O}$ strongly simulates ([Mil95]) a system with oracle $\mathcal{O}'$ such that $\mathcal{O} \implies \mathcal{O}'$.

Proposition Let $\mathcal{O}$ and $\mathcal{O}'$ be two oracles for the system $\gamma^{\perp}(B_1^{\perp}, \ldots, B_n^{\perp})$, such that $\mathcal{O} \implies \mathcal{O}'$. They define two systems $B = \gamma^{\perp}_{\mathcal{O}}(B_1^{\perp}, \ldots, B_n^{\perp}) = (Q^g \cup Q^p, \gamma^{\perp}, \rightarrow_{\mathcal{O}})$ and $B' = \gamma^{\perp}_{\mathcal{O}'}(B_1^{\perp}, \ldots, B_n^{\perp}) = (Q^g \cup Q^p, \gamma^{\perp}, \rightarrow_{\mathcal{O}'})$. Every state of B is strongly similar to some state of B'.

2.6.4.2 Oracles

We defines oracles parameterized by a dependency relation $\sqsubseteq$ on interactions. This relation contains $\pi^{\perp}$ but it need not be an order as shown below.

Definition 2.6.12 (Oracle) *A $\sqsubseteq$-oracle for a system $\gamma^{\perp}(B_1^{\perp}, \ldots, B_n^{\perp})$ and a* dependency relation $\sqsubseteq \subseteq \gamma \times (Q^g \cup Q^p) \times \gamma$, *is a predicate $\mathcal{O}$ on $(Q^g \cup Q^p) \times \gamma$ such that:*
•*(Dependency Enforcement)*

$$\mathcal{O}(q,a) \implies \big(\forall a'.\ a \sqsubseteq_q a' \implies \textit{\texttt{disabled}}(q,a') \vee \mathit{Act}(q,a')\big)$$

•*(Soundness) $q \in Q^g \implies \forall a \in \gamma.\ \mathcal{O}(q,a)$*

The dependency enforcement condition means that the oracle allows execution of a from state q if the status (enabled or disabled) of the interactions a' that dominate a (i.e., $a \sqsubseteq_q a'$) is known.

Property If $\sqsubseteq_1 \subseteq \sqsubseteq_2$ and if $\mathcal{O}$ is a $\sqsubseteq_2$-oracle, then it is a $\sqsubseteq_1$-oracle.

We will now define several $\pi^{\perp}$-oracles for the system $\gamma^{\perp}(B_1^{\perp}, \ldots, B_n^{\perp})$ providing various degrees of parallelism and cost of implementation. There is a compromise to make between the degree of parallelism allowed by an oracle, and the cost for its implementation.
Ideal Oracle The best possible oracle is defined by

$$\mathcal{O}_{ideal}(q,a) \iff \big(\forall a'.\ a\pi_q^{\perp}a' \implies \texttt{disabled}(q,a') \vee \mathit{Act}(q,a')\big)$$

However, such an oracle is difficult to implement. It requires that at a given partial state q, the Engine is able to compute the relation $\pi_q^{\perp}$ which according to the definition of $\pi^{\perp}$ (Definition 2.6.6) boils down to computing the global state q' reachable from q. For this, in the general case, the Engine has to know the transition relation of the global state system.
Dynamic Oracle We use now a dynamic approximation $\sqsubseteq^{dyn}$ of $\pi^{\perp}$. The reachability condition $q \overset{\beta^*}{\rightsquigarrow} q'$ in the definition of $\pi^{\perp}$ is replaced by a comparison $q \leq q'$, i.e., $a \sqsubseteq_q^{dyn} a' \iff \exists q' \in Q^g.\ q \leq q' \wedge a\pi_{q'}a'$. The dynamic oracle is defined by:

$$\mathcal{O}_{dyn}(q,a) \iff (\forall a'.\ a \sqsubseteq_q^{dyn} a' \implies \mathit{Act}(q,a') \vee \texttt{disabled}(q,a'))$$

For the dynamic oracle, the Engine does not need a complete knowledge of the state of the system in order to compute $\sqsubseteq_q^{dyn}$ for a given partial state q.

Static Oracle The static oracle $\mathcal{O}_{static}$ is defined via a static approximation $\sqsubseteq^{st}$ of $\pi^{\perp}$: $a \sqsubseteq^{st}_{q} a' \iff \exists q' \in Q^g.\ a \pi_{q'} a'$. We write $\sqsubseteq^{st}$ instead of $\sqsubseteq^{st}_{q}$ as the relation does not depend on q. The static oracle is defined by:

$$\mathcal{O}_{static}(q,a) \iff (\forall a'.\ a \sqsubseteq^{st} a' \implies Act(q,a') \vee \texttt{disabled}(q,a'))$$

Lazy Oracle The lazy oracle forbids all interactions from partial states. It waits for all the atomic components to finish their computation in order to know all the possible interactions. It is defined by $\mathcal{O}_{lazy}(q,a) \iff q \in Q^g$.
Proposition $\mathcal{O}_{ideal}$, $\mathcal{O}_{dyn}$, $\mathcal{O}_{static}$ and $\mathcal{O}_{lazy}$ are $\pi^{\perp}$-oracles and we have, $\mathcal{O}_{lazy} \implies \mathcal{O}_{static} \implies \mathcal{O}_{dyn} \implies \mathcal{O}_{ideal}$.

The above result with Proposition 2.6.4.1 shows that these oracles provide an increasing degree of parallelism.

2.6.4.3 Correctness with Respect to Global State Semantics

The systems $\pi\gamma(B_1,\ldots,B_n)$ and $\pi^{\perp}\gamma^{\perp}_{\mathcal{O}}(B^{\perp}_1,\ldots,B^{\perp}_n)$ are observationally equivalent when $\mathcal{O}$ is a $\pi^{\perp}$-oracle.

Theorem 2.6.13 *Let π be a priority relation for the system $\gamma(B_1,\ldots,B_n)$ and $\mathcal{O}$ a $\pi^{\perp}$-oracle for the system $\gamma^{\perp}(B^{\perp}_1,\ldots,B^{\perp}_n)$. The systems $\pi\gamma(B_1,\ldots,B_n)$ and $\pi^{\perp}\gamma^{\perp}_{\mathcal{O}}(B^{\perp}_1,\ldots,B^{\perp}_n)$ are observationally equivalent.*

Part II

Implementation: Language and Tool-chain

Chapter 3

The BIP Language

For representing system models in the BIP framework, we propose, in this chapter, the BIP language. The BIP language provides syntactic constructs for describing systems conforming to the formal framework presented in Section 2.3. It is a co-ordination language, and is an extension of the C programming language. It leverages on C style variables and data type declarations, expressions and statements, and provides additional structural syntactic constructs for defining component behavior, specifying the co-ordination through connectors, and describing the priorities. The language enables us to express the concurrent and sequential behavior of systems, as an interconnection of components. A system can be described hierarchically, and timing information can also be expressed in the same description.

The principal constructs are:

1. *atom*: to specify behavior, with an interface consisting of ports. Behavior is described as a set of transitions. Transitions are labeled by ports.

2. *connector*: to specify the co-ordinations between the ports of components, and the associated guarded actions.

3. *priority*: to restrict the possible interactions, based on conditions depending on the state of the integrated components.

4. *compound*: to specify systems hierarchically, from other atoms or compounds, with connectors and priorities.

5. *model*: to specify the entire system, encapsulating the definition of the components, and specify the top level instance of the system.

The language allows defining types for port, atom, connector, and compound, defining priorities, and instantiating objects of the defined types. Every instantiated object has a scope. The scope in a BIP description are:

1. atom
2. compound
3. model
4. package

3.1 Basic Language Elements

Identifiers

An identifier is composed of a sequence of one or more characters. A legal character is an upper case letter (A . . . Z), a lower-case letter (a . . . z), a digit (0 . . . 9) or the underscore (_) character. The first character of an identifier may be a letter or underscore. Identifiers are case sensitive.

Reserved Words

The following identifiers are reserved words in the language (also called keywords), and therefore cannot be used as basic identifiers in a BIP description.

atomic	**component**	**compound**	**connector**	**data**
define	**delayable**	**do**	**down**	**eager**
else	**end**	**export**	**extern**	**from**
header	**if**	**in**	**is**	**initial**
lazy	**model**	**multishot**	**package**	**place**
on	**port**	**priority**	**provided**	**singleshot**
timed	**to**	**type**	**use**	**up**

Comments

The language supports C style comments. Single line comments begin with two consecutive front slashes (//) and extends until the end of the line. Multi-line comments can be enclosed between a pair of /* and */. Examples are:

```
// This is a single line comment
/* This is a
multi line
comment */
```

Special Pragma

A description enclosed within the character pairs **{#** and **#}** is treated specially for code generation purpose. It acts as a special directive for the BIP compiler to pass the description as it is to the code generator. This is useful for enclosing standard C header files, arbitrary C type and function definition, or to specify any arbitrary C code. For example:

```
{# #include<stdarg.h>
   typedef char* String; #}
```

Data Objects and Types

A data object holds a value of a specified type. Objects are created by object declaration. Basic C data types and C style declaration is adopted in the BIP language. The keyword **data** is used to specify a data object declaration. An example is:

data `String name`

which defines `name` as an instance of the type `String`.

3.2 Modeling Atoms

3.2.1 Port Type

Ports in BIP are typed. The language allows the definition of typed ports, i.e., ports associated with typed variables. A port type definition is characterized by the number and type of its associated variables. The syntax for a port type definition is:

port type *port-type-name* [**(** *typed-variable-list* **)**]

An example of a port type definition, named `intPort` is the following:

port type `intPort(` **int** `i)`

It defines a port which can be associated with an integer variable. The variable can be referred by the name `i`.

3.2.2 Atomic Type

An atomic type defines an atom. Atoms are the leafs in the component hierarchy. An atomic type is characterized by its ports, and an optional list of parameters. The syntax is:

atomic type *atom-type-name* [**(** *parameter-list* **)**]
    variable-definition
    port-definition
    place-definition
    [*initial-block*]
    transition-definition
    [*priority-definition*]
    [*export-definition*]
end

Variable

Variables used in an atom are declared as data objects. They are typed, as in the C language. They may have initial values. A variable may be qualified as **extern** if its definition is in an opaque C code. A variable may be qualified as **timed**, when it is supposed to evolve with time. The syntax is:

> [**extern**] [**timed**] **data** *data-type-name data-instance-name*
> [= *initial-value*]

Port

Ports are instances of port-types, and may be associated with atom variables. A port may refer to multiple variables, and a variable may be referenced by multiple ports. The syntax is:

> **port** *port-type-name port-name* [**(** *atom-variable-list* **)**]

The variables in the port instantiation are positionally mapped to the variables in the corresponding port type definition. For example, to instantiate a port named `p` of type `intPort`, associated with an integer variable `x`, one would write,

> **port** `intPort p(x)`

Implicitly, a port is internal, i.e., representing an internal action which does not need explicit synchronization for its triggering. However, a port may be exported by the keyword **export** to make it visible in the component interface. Exporting a port has an option to specify the exported name of the port, i.e., a name by which a port is known in the component interface. The syntax is the following:

> **export port** *port-type-name port-name*
> [**(** *atom-variable-list* **)**] [= *port-export-name*]

For example, the port `p` can be exported with a name `out` as:

> **export port** `intPort p(x) = out`

Port can be used in the scope of atoms, in defining connectors and in compounds.

Place

The behavior of an atomic component may be represented either as an automata or as a petri-net. The places represents the control-states for the automata, or places for the petri-net. They are identifiers, preceded by the keyword **place**. As an example, to describe three places `idle`, `empty` and `full`, we write the following,

> **place** `idle, empty, full`

Initial Block

The **initial** statement is used for initializing the control-states and component variables. The syntax is:

> **initial to** *initial-place-list* [**do** *statement-list*]

As an example, we can specify `idle` as the initial place, and initialize an integer variable as follows:

> **initial to** `idle` **do** `x=0;`

Transition

The behavior is given by a set of transitions modeling atomic computation steps. The syntax is given below:

> **on** *port-name* **from** *place-list* **to** *place-list*
> [**provided** *guard-expression*]
> [*timed-guard-expression*]
> [**do** *statement-list*]

A transition is specified by a label (a port name) following the keyword **on**. The source(s) of a transition follow the keyword **from**, while the destination(s) follows **to**. The guard of the transition is specified after the **provided** keyword. The guard has two parts, an untimed guard and a timed guard. The untimed guard is a boolean expression of the atomic variables. Timed guards are useful for modeling timed systems. A timed guard is an expression of the timed variables of the atom. Timed guards are elaborated in section 3.6. The guard of a transition is optional, and if not provided, it is assumed to be true. The action statements of the transition are specified following the **do**. The action statements are C statements. An example of a transition from the place `empty` to `full` is

> **on** `in`
> **from** `empty` **to** `full` **provided** `0<x` **do** `y=f(x);`

We assume that the C expressions and statements used in the **provided** and **do** sections are adequately restricted, e.g., they do not have side effects, and respect the atomicity assumption for transitions i.e., guaranteed termination.

Priority in Atom

The language offers the option for specifying priority between the ports of an atom. The syntax is:

> **priority** *priority-name port-name* `<` *port-name*
> [**provided** *expression*]

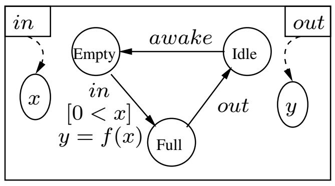


Figure 3.1: An atomic type.

Figure 3.1 shows an atomic reactive component with three ports `in`, `out`, `awake`, variables `x`, `y`, and control states `Idle`, `Empty` and `Full`. The automaton is initialized to control state `Idle`, from which it can move to `Empty` by the internal transition `awake`. At `Empty`, the transition labeled `in` is possible if `0<x`. When an interaction through `in` takes place, the variable `x` is eventually modified and a new value for `y` is computed. From `Full`, the transition `out` can occur. The omission of guard and function for this transition means that the associated guard is `true` and the internal computation step is empty.

The BIP description of the reactive component of figure 3.1 is:

```
atomic type Reactive
   data int x, y
   export port intPort in(x) = in
   export port intPort out(y) = out
   port ePort awake
   place Idle, Full, Empty
   initial to Idle do x=0;
   on awake
      from Idle to Empty
   on in
      from Empty to Full provided 0<x do y=f(x);
   on out
      from Full to Idle
end
```

The variables associated with a port may be modified on executing the the transfer function of an interaction, in which the port participates.

A pure event port does not have any associated variables, and provides the mechanism for event synchronization only (i.e., without any data transfer). The port type `ePort` used in the code-sample above is an example of a pure event port.

In the above example, the ports `in` and `out` are instances of a predefined port type `intPort`, which associates an integer variable with a port. The variable `x` is associated with the port `in`, and `y` with `out` respectively. On the other hand, the internal port `awake` is of type `ePort`, which is an event port, and is not associated with any variable.

Note that a port of an atom is not visible to its environment unless it is exported explicitly. In the above example, the ports `in` and `out` are exported, whereas the port `awake` acts as an internal port. It is necessary to export a port if it has to be used in some connector for synchronization purpose, as will be evident in the following section.

3.3 Modeling Connectors

A connector defines the set of possible interaction between ports of components and the corresponding data-transfer between the variables associated with the ports. The BIP language allows the definition of connector types.

3.3.1 Connector Type

A connector type defines a connector template, parameterized by a list of ports. The syntax is as follows:

```
connector type connector-type-name ( port-list ) [ ( parameter-list ) ]
    define port-expression
    variable-definition
    on interaction_1 provided guard_1 up { up-statement-list_1 }
                                      down { dn-statement-list_1 }
    on interaction_2 provided guard_2 up { up-statement-list_2 }
                                      down { dn-statement-list_2 }
    ...
end
```

The keyword **define** is followed by an expression of its ports in $\mathcal{AC}(P)$, (2.3.2, Algebra of Connectors) specifying the possible set of interactions.

A connector may contain local variables. It also specifies a list of guarded interactions, with the associated data transfers. The guard follows the keyword **provided**, and is restricted to boolean expressions on the variables associated with the ports.

For every interaction, the data transfer is specified by an `up` and a `down` action. The action `up` is supposed to update the local variables of the connector, from the values of variables associated with the ports. Conversely, the action `down` is supposed to update the variables associated with the ports, from the values of the connector variables. The guards are C expression and the `up` and `down` actions consist of C statements.

Additionally, a connector type definition may contain C type parameters.

Simple Connector

Figure 3.2 shows graphically, examples of simple connector types. The connector type named Rendezvous (figure 3.2(a)) is parameterized by three

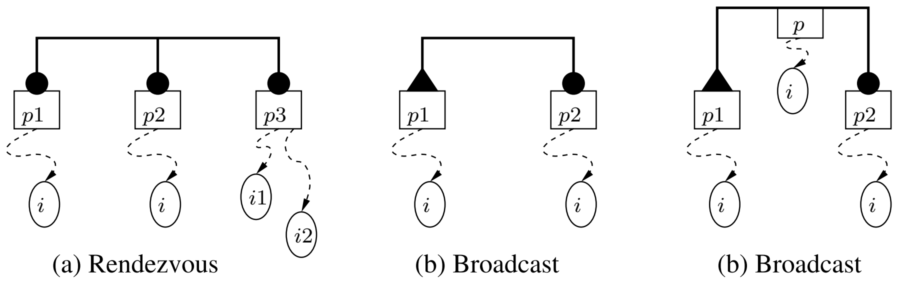


Figure 3.2: Connector Types.

ports, $p1$, $p2$, and $p3$. $p1$ and $p2$ are of type *intPort*, whereas $p3$ is of type *int2Port*, which associates a port with two integer variables. The connector type defines a rendezvous between the ports, specified by the $\mathcal{AC}(P)$ expression $p1p2p3$, with associated guard and transfer functions as described below:

```
connector type Rendezvous (intPort p1, intPort p2, int2Port p3)
  define [p1 p2 p3]
  data int x
  on p1 p2 p3 provided (p1.i + p2.i != p3.i1 + p3.i2)
    up {x = MAX(p1.i, p2.i, p3.i1, p3.i2);}
    down {p1.i = p2.i = p3.i1 = p3.i2 = x;}
end
```

The connector declares a local integer variable x. The interaction involving the three ports can occur only when the guard expression (`p1.i + p2.i != p3.i1 + p3.i2`) is true. The data transfer consists of the combined effect of the `up` and `down` actions. In the `up` action, the MAX of the variables associated with the ports is calculated and stored in the connector variable `x`. In the `down` action, the value of `x` is assigned to each port variable. As a result of the data transfer, the variables associated with the ports are set to the maximum of their values.

The second example (figure 3.2(b)) defines a connector type named `Broadcast`, between two `intPorts`, $p1$ and $p2$. $p1$ is the trigger of the broadcast, defined by the $\mathcal{AC}(P)$ expression $p1'p2$. The BIP description is:

```
connector type Broadcast (intPort p1, intPort p2)
  define [p1′ p2]
  data int x
  on p1
    up {x = p1.i;}
    down {}
  on p1 p2
```

```
        up {x = p1.i;}
        down {p2.i = x;}
    end
```

The data transfer methods describe transfer of value from the variable associated with the trigger $p1$ to the variable associated with the synchron $p2$. Also in this example, as no guards are provided with the interactions, they are, by default, taken as true.

Notice that contrary to other formalisms, BIP does not allow explicit distinction between input and output ports. For simple data flow relation, variables can be interpreted as inputs or outputs. For instance, in the connector type `Broadcast`, $p1$ can be thought as an output port and $p2$ as an input port.

3.3.1.1 Exporting port from Connector

A connector has an option to define a port and export it. This allows a connector to be used as a port in other connectors, and create structured connectors. The port export statement can be provided in the connector type definition, the syntax is:

> **export** *port-type-name port-name port-instance-name*
> [(*port-data-list*)]

Local variables of the connector can be associated with its exported port. This allows to manage hierarchical data transfer mechanism. As an example, consider the connector type definition of `Broadcast`, which consists of two `intPort` and exports an `intPort`.

```
connector type Broadcast (intPort p1, intPort p2)
    define [p1' p2]
    data int x
    on p1
        up {x = p1.i;}
        down {}
    on p1 p2
        up {x = p1.i;}
        down {p2.i = x;}
    export intPort p(x)
end
```

The export statement creates a port `p` of type `intPort`, and associates the connector variable `x` with it. This leads to two possibilities: 1) a connector of type `Broadcast` can be used as an `intPort` in another connector, leading to a structured connector, 2) it can be exported as a port of type `intPort` in a compound type (ports of compounds are described later).

For a connector type definition, the type of its exported port forms an integral part of its signature, in addition to the list of its ports along with their types, and the C parameters.

3.4 Modeling Compounds

3.4.1 Compound Type

A compound type defines a new component type from existing components (atom or compound) by creating their instances, instantiating connectors between them and specifying the priorities. Hence it defines a structural model from existing components. A compound offers the same interface as an atom, hence externally, there is no difference between a compound and an atom.

The syntax for compound type is:

> **compound type** *compound-type-name* [(*parameter-list*)]
> *component-instantiation*
> *connector-instantiation*
> *priority-definition*
> **end**

Instantiating Component

A component to be instantiated must be first defined, either as an atomic type or a compound type. The syntax for instantiating a component is:

> **component** *component-type-name component-instance-name*
> [(*component-parameter-list*)]

To instantiate a component of type `Reactive`, we would write:

> **component** `Reactive R1`

which creates an instance of `Reactive` named `R1`.

Instantiating Connector

A connector instance associates the ports of instantiated components through the interactions defined by the connector type. The ports specified in the connector instantiation are mapped positionally to the port parameters in the connector type definition. The syntax is:

> **connector** *connector-type-name connector-name*
> (*port-list*) [(*connector-parameter-list*)]

As an example, an instance `C0` of connector type `Broadcast`, parameterized by the ports `out` and `in` of the components `R1` and `R2` respectively, is written as:

connector `Broadcast C0(R1.out, R2.in)`

Structured Connector

Structured connectors are created by the combined mechanism of exporting port from a connector and instantiating connectors, where a port of the connector is an exported port of another instantiated connector. Figure 3.3 shows an example of a structured connector. The connector `C0` relates the

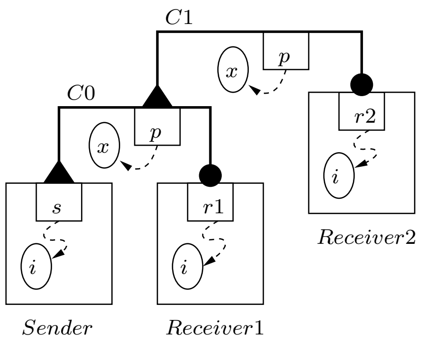


Figure 3.3: Structured Connector

port `s` (trigger) of the `Sender` component with the port `r1` (synchron) of the `Receiver1` component, and exports the port `p`. It represents the $\mathcal{AC}(p)$ interaction $sr1'$. `C1` is a structured connector joining the port `p` (trigger) of connector `C0` with the port `r2` (synchron) of the`Receiver2` component. It represents the $\mathcal{AC}(p)$ interaction $[sr1']'r2$. The BIP code is:

connector `Broadcast C0 (Sender.s, Receiver1.r1)`
connector `Broadcast C1 (C0.p, Receiver2.r2)`

Exporting Connectors: Ports of a Compound

The language allows a compound to export a connector as a port, similar to exporting ports from atoms. Exporting makes the port (and hence the connector) visible in the compound interface, and hence allows for its further synchronization with other components. This also provides a homogeneous interface for both atoms and compounds, facilitating their seamless usage.

In order to export a connector as a port, the connector itself must have an exported port. The type of the exported port of the compound is then the same as the type of the exported port of the connector. The connector export statement is provided in the body of the compound type definition. The syntax is:

export *port-type-name export-port-name* **is** *connector-name*

We provide an example (figure 3.4) of exporting a connector as a port from a compound component. The connector `C0` in the compound `SendRecv` is exported as a port named `sr1`, of type `intPort`. The BIP description is:

```
compound type SendRecv
  component Reactive Sender
  component Reactive Receiver1
  connector Broadcast C0 (Sender.s, Receiver1.r1)
  export intPort sr1 is C0
end
```

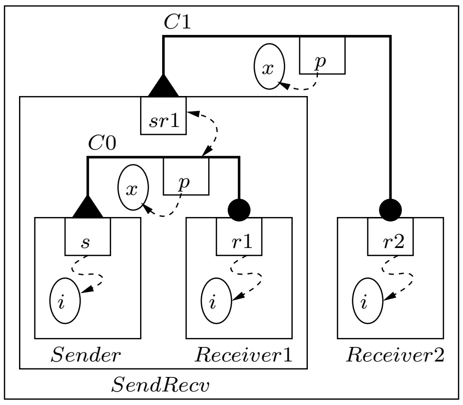


Figure 3.4: Exporting Connector as Compound Port

3.5 Modeling Priority

Priorities are expressed by a set of rules, each specifying an order between a pair of interactions or connectors. The syntax, similar to that of priorities in atoms, is as follows:

priority *priority-name interaction-or-connector-name* <
  interaction-or-connector-name
  [**provided** *expression*]

The keyword **provided** is followed by an expression (boolean expression in C) representing the global guard for dynamic priorities. The expression can only refer to variables associated to the ports of the interactions specified in the priority, or exported variables. The **provided** statement is optional, and if omitted, the guard is considered to be true. This is the case for static priorities.

When a priority is expressed between connectors, the rule apply between all pairs of possible interactions between the connectors.

The maximal progress priority is enforced implicitly by the BIP engine: if an interaction is contained in another one, the latter has higher priority to the former.

Shown below is an example of priority specification in BIP.

```
priority P1 C1:R1.in < C2:R1.out, R2.in
priority P2 C2 < C4 provided (R2.x > 0)
```

The priority labeled `P1` is a static priority between the interaction `R1.in` and `R1.out, R2.in`. Note that the syntax requires also to specify the label of the connector with the interaction. The priority `P2` is dynamic, and is specified between the connectors `C2` and `C4`.

Note that an interaction corresponding to an *internal* port has a higher priority over other interactions.

Compound types can also contain C type parameters, similar to atomic types. The language provides a homogeneous interface for both atomic and compound types. Similar to atomic types, a compound type is characterized by its type name, list of parameter, and the ports of its interface.

An example of a compound type, named *System*, is shown in figure 3.5. It is the serial connection of three `Reactive` components `R1`, `R2` and `R3`, with connectors and priorities. The BIP description is:

```
compound type System
   component Reactive R1, R2, R3
   connector Singleton C1(R1.in)
   connector Broadcast C2(R1.out, R2.in)
   connector Broadcast C3(R2.out, R3.in)
   connector Singleton C4(R3.out)
   priority P1 C1 < C3
   priority P2 C1 < C4
   priority P3 C2 < C4
end
```

3.6 Modeling Timed Systems: Timed Guards

The BIP language allows modeling of timed systems by using timed variables and associating deadlines with the atomic transitions. They are conditions on the timed variables that determine when a transition must be triggered by having priority over time progress. This is achieved by associating urgency type with the transitions, namely eager, delayable and lazy, as described in 2.4.1.

The syntax for representing timed guards in BIP is the following:

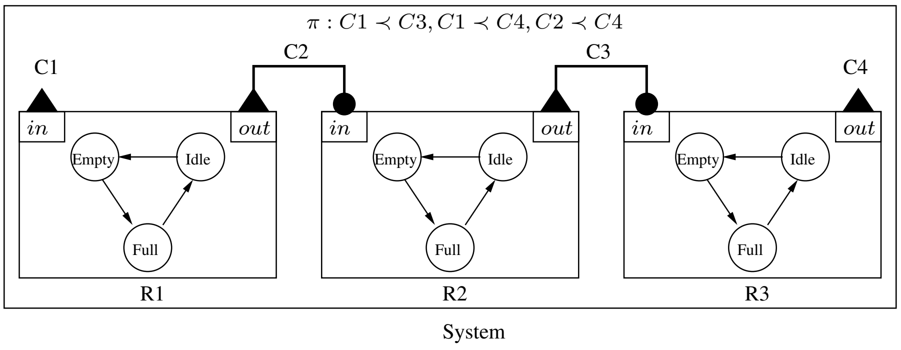


Figure 3.5: A compound type.

> **eager** | **delayable** | **lazy**
> *timed-var* **in** (*lower-bound*, *upper-bound*)

where *lower-bound* and *upper-bound* are the bounds of the deadline. They are expressions in C, which evaluates to real values. An example of a timed component is:

```
atom type Worker
   timed data clock
   export port intPort work
   ...
   place working, idle
   initial to working
   on work from working to idle delayable clock in (2,5)
   ...
end
```

3.7 Arrays

The language allows to create static arrays of components and connectors. This is similar to defining arrays, as in the C language, by specifying a static dimension. Individual elements of the array can be accessed by indexing.

Array of Components

The syntax for creating an array of component instance is:

> **component** *component-type-name component-instance-name* [*dimension*]
> [(*component-parameter-list*)]

Below we define a compound type named `System` that instantiates an array of `Reactive` components.

```
compound type System (int length)
   component Reactive R[length]
   ...
   export port intPort input is R[0].in
   export port intPort output is R[length-1].out
end
```

It also exports port `in` and `out` of the 0^{th} and $length - 1^{th}$ instance respectively as ports of the compound, named `input` and `output`.

Array of Connectors

> **connector** *connector-type-name connector-instance-name* [*dimension*]
> **(** *component-port-list* **)** [**(** *connector-parameter-list* **)**]

We also introduce a macro notation, denoted as `$` which refers to all the instances of the array obtained by substituting `$` by indices of the array dimension. This is useful to specify the component port instances for the connectors. An example of an array of `Broadcast` connectors is:

```
compound type System (int length)
   component Reactive R[length]
   connector Broadcast Cnx[length-1] (R[$].out, R[$+1].in)
   ...
end
```

Here `Cnx` is an array of `Broadcast` connector, where the i^{th} connector associates the ports `R[i].out` and `R[i+1].in`.

3.8 Package and System

The package declaration allows to create a library of BIP component and connector types to be used in different designs. In addition, common C datastuctures and functions can also be defined inside a package. A package is identified by a name, and may use other defined packages. The syntax is:

> **package** *package-name*
> [**use** *other-package-name*]
> *C-data-structure-function-list*
> *component-definition-list*
> *connector-definition-list*
> **end**

The **use** statement imports the definition of a package. A package is compiled to a library by the BIP compiler, which can be imported in the design of a system. Hence packages support for separate compilation. An example package definition is shown below:

```
package my_pack
   {# typedef char* String; #}
   port type IntPort(int x)
   port type ePort
   connector type SendRec(IntPort s, IntPort r)
      define [ s r ]
      data int val
      on s r up {val = s.x;} down {r.x = val;}
      export IntPort p(val)
   end
   atomic type Producer
      data int i
      export port IntPort comm(i)
      port ePort work(i)
      place idle
      place working
      initial to idle do i=0;
      on work
         from working to idle
         do i=i+1;
      on comm
         from idle to working
   end
end
```

It defines a package named my_pack. The package defines a C typedef, a port type and a connector type.

The design of an entire system is enclosed in a **model**. The model may include packages, and define components, connectors, and C data-strucures and functions, as in a package. It also specifies the top level instance of the design, known as the root component. The syntax of model is:

> **model** *model-name*
> [**use** *package-name*]
> *C-data-structure-function-list*
> *component-definition-list*
> *connector-definition-list*
> *root-component-instantiation*
> **end**

We provide the description of an entire system below. It models a Producer and Consumer component. They work independently, and synchronize by a rendezvous which transfers data (integer) from the Producer to the Consumer.

```
model ProdCons
```

```
      use my_pack
      atomic type Consumer
         data int j
         export port IntPort comm(j)
         port ePort work
         place idle
         place working
         initial to idle
         on work
            from working to idle
            do printf("j=%d\n", j);
         on comm
            from idle to working
      end
      compound type System
         component my_pack.Producer P
         component Consumer C
         connector my_pack.SendRec cnx (P.comm, C.comm)
      end
      component System S
   end
```

The model `ProdCons` uses the package `my_pack`. It defines an atom `Consumer`, and a compound `System`. In `System`, `P` is an instance of the `Producer` component which is defined in the package `my_pack`, while `C` is an instance of `Consumer`, defined in the model itself. The connector `cnx` is of type `SendRec`, which is also defined in `my_pack`. The root component `S` is an instance of the top level component `System`.

3.9 Expression and Statement

The expressions in BIP are essentially C expressions, with the same set of operators, arithmetic and logical. The language has also some limited support for array indexing and pointer dereferencing. However, arbitrary expressions may also be specified, enclosed within {# ··· #}. The only restriction in this case is that the expressions are not parsed by the BIP compiler, and directly embedded in the generated application code.

The statements are a subset of C statements. The compiler currently supports assignment and conditional statements. However, arbitrary statements can be specified within {# ··· #}. C style function call statements are supported, but the validation of the function parameters is delayed until the compilation of the application code.

The complete grammar of the BIP language is provided in the appendix 6.3.2.3.

Chapter 4

The BIP Tool-Chain

This chapter presents the implementation of the BIP framework, formally introduced in chapter 2, in the form a tool-chain called the BIP tool-chain [BIP]. The framework has been partially implemented in the IF toolset [BGO$^+$04] and the PROMETHEUS tool [Gößl01]. The BIP tool-chain provides a complete implementation, with a rich set of tools for modeling, execution, analysis (both static and on-the-fly) and static transformations. The tools can be broadly classified into:

- a frontend for editing and parsing BIP programs, and generating an intermediate model. The model can be used to generate code for execution and analysis on a backend platform, as well as for other source level transformations and static analysis.
- a backend platform consisting of an engine and the infrastructure for executing the generated code.
- connections to external analysis tools.

4.1 Overview of the Tool-Chain

The BIP tool-chain developed at Verimag is shown in figure 4.1. It includes:

- An *editor*, for textual description of a system in the BIP language.
- A *parser*, to analyze a BIP program, and to generate the model conforming to the BIP meta-model.
- A *deparser*, to produce BIP description source, back from the model.
- A *code generator*, for generating C++ code executable on the BIP engine.
- Other *source level transformation* tools, at the model level, for optimization and static analysis.

The code generator provides, in addition, option to generate THINK specification [FSLM02], from which the Think tool-chain can generate code to be executed over a choice of target platforms. The source level tools being

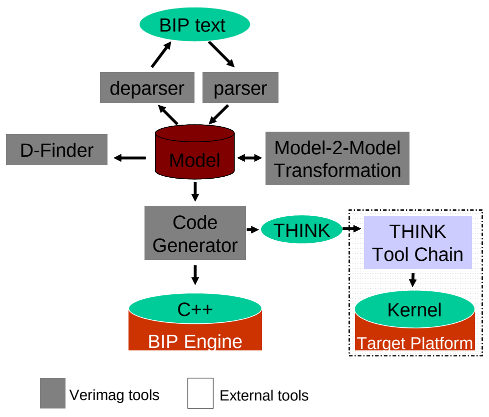


Figure 4.1: The BIP tool-chain.

developed at Verimag to enrich the BIP tool-chain includes the following:

- *D-Finder* tool [SBS08, BBSN08], to detect deadlocks in BIP models by static analysis of the model.
- *Model-2-model transformation* tools, in order to perform useful transformations, like timed BIP model to a basic BIP model, and transformations for run-time optimizations, such as flattening the hierarchy and transforming structured connectors to flat connectors.

The editor, parser, deparser and code generator forms the frontend of the tool-chain. The backend consists of the software infrastructure for executing the BIP models. They are explained in the following sections.

4.2 The Frontend

The frontend compilation chain transforms BIP programs into an intermediate model. It consists of a parser and code generator. The parser performs syntactic analysis of the input program and reports the programming errors. It creates an intermediate representation of the program in the form a model that conforms to a full-fledged meta-model for BIP, developed using EMF[1] [EMF]. The code generator takes the model generated by the parser, and produces an application code, which can be executed on the backend. We provide below a succinct description of the BIP meta-model.

[1]Eclipse Meta-modeling Framework

4.2.1 BIP Meta Model

The meta-model represents a template of the structure of the intermediate model to be generated from a BIP program. All the modeling elements, presented in the BIP language, have a representation in the BIP model in the form of a data structure. Class diagrams are used to define the relations between the different modeling elements, through inheritance and containment. The meta-model has been designed in a manner so that it is intuitively close to the BIP grammar, and efficient in terms of compact representation and re-use.

An example of the meta-mode class diagram for a BIP **connector type** is shown below in figure 4.2.

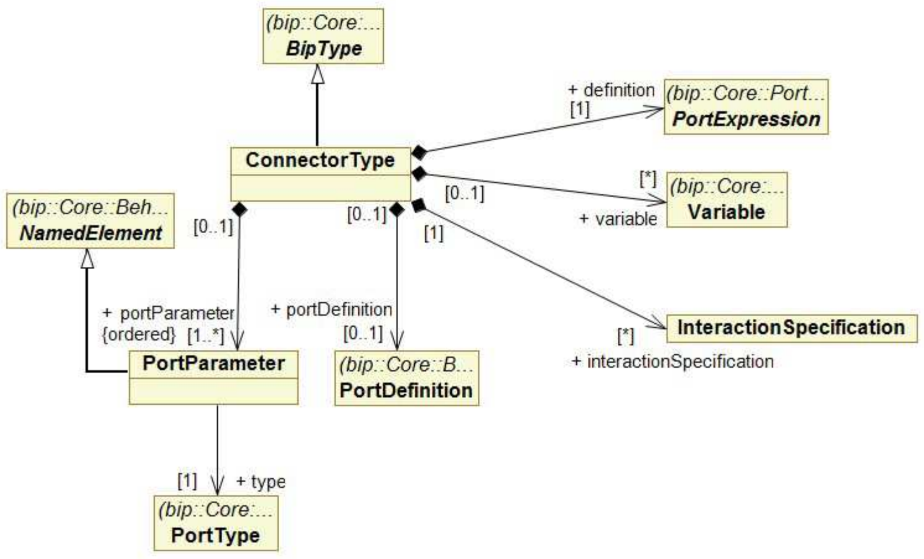


Figure 4.2: Meta model representing a Connector Type.

It is defined by a class `ConnectorType` derived from a base class `BipType`. It consists of a list of `PortParameter` objects, which represents the list of ports for the connector type. Note that the list has a multiplicity of at least one, meaning that there must be at least one port parameter for the connector. Moreover, the list is attributed as *ordered* to enforce the positional association of the list elements with the actual arguments.

The class `PortExpression` defines the $\mathcal{AC}(P)$ expression of the connector, and `Variable` is a list of the declared connector variables. The interactions specified in the connector are stored as a list of `InteractionSpecification` objects, and the exported port of the connector is stored as a `PortDefinition`.

A detailed description of the individual meta-model classes mentioned here and the complete meta model of the BIP intermediate representation is provided in the appendix 6.3.2.3.

4.2.2 Code Generator

The code generator takes as input a model, generated by the parser, and transforms it to a C++ application code. The application is an executable model of the original BIP program. Code is generated for each atomic component, connectors and priorities, i.e., the code is modular and preserves the structure of the initial model.

The code generator has options for generating application code for two types of platforms:

- single-threaded
- multi-threaded

For single-threaded platform, the code generator has further options for generating code for two types of execution:

- running an execution
- performing exhaustive exploration

In the second case, the generated code is instrumented with additional data-structures and routines which enables to store the states of the components, and monitor the global state space, needed for exhaustive simulation.

The frontend heavily relies on tools and model-based technologies available for Java [Jav] under the Eclipse platform. The parser has been developed using the parser generator ANTLR [ANT], and the meta model has been developed using the UML tool PAPYRUS [PAP], both of which are integrated in Eclipse. The code generator bas been implemented in Java. The parser consists of 4000 lines of code, and the code generator has 5,000 lines of Java code, excluding the auto generated files.

4.3 The Backend

The backend of the BIP tool chain provides a platform for executing and analyzing the C++ application code, generated by the frontend. It includes an engine and the associated software infrastructure.

The engine directly implements the BIP operational semantics. It plays the role of the co-coordinator in selecting and executing interactions between the components, taking into account the glue specified in the input component model. It monitors the state of the components and considering the interaction model, finds all the enabled interactions. It then applies the priority rules to eliminate the interactions with low priority, and selects one amongst the maximal enabled, for execution.

We have three different implementations for the BIP engine:

1. A *centralized enumerative* engine, which decides the selection and execution of the system interactions based on the complete knowledge of the state of the system. It represents interactions by enumeration.

2. A *centralized symbolic* engine, where the previous enumerative representation of interaction is replaced by a symbolic representation using BDDs. This implementation allows for efficient execution for some specific classes of systems.

3. A *distributed* engine, which computes the possible interactions based on partial information about the system state. This version is relevant for the deployment of the system in real distributed platforms.

The engine is entirely implemented in C++ on Linux. This choice has been made mainly to allow a smooth integration of components with behavior expressed using plain C/C++ code. For the distributed implementation, POSIX threads are used.

4.3.1 Centralized Enumerative Engine

In the centralized implementation, the engine works based upon the complete state information of the components. The execution follows a two phase protocol, marked by the execution of the engine, and the execution of the atomic components.

In the execution phase of the engine, it computes the interactions possible from the current state of the atomic components, and guards of the connectors. Then, between the enabled interactions, priority rules are applied to eliminate the ones with low priority. During this phase, the components are blocked, and await to be triggered by the engine. The engine selects a maximal enabled interaction, executes its data transfer, and triggers the execution of the atomic components associated with this interaction.

The second phase is the execution of the local transitions of the notified atomic components. They continue their local computation independently and eventually reach new control states. Here, the atomic components their enabled transitions to the engine and get blocked. The two phases are repeated, unless a deadlock is reached or the user wants to terminate the simulation. The scheme of the protocol is shown in figure 4.3. The execution of the engine has the following algorithm:

```
1. initialize the atomic components [init()
2. wait for all atoms to be blocked [sync()]
3. compute the enabled interactions from $\gamma$ [evaluate()]
4. filter interactions by priority rules from $\pi$ [filter()]
5. if no enabled interaction, then deadlock, exit.
6. select one out of the maximal interactions
7. perform it's data transfer [execute()]
```

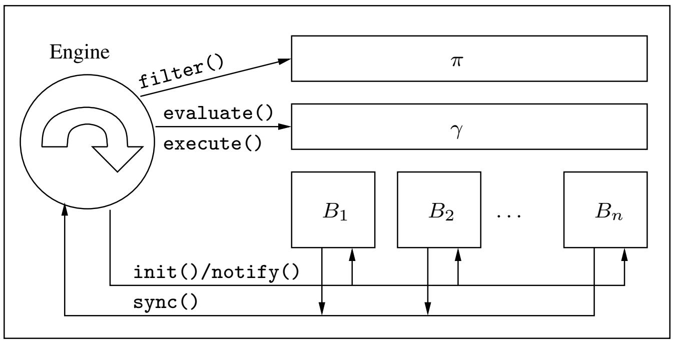


Figure 4.3: Centralized engine architecture.

8. trigger the involved atoms [`notify()`]
9. goto step 2.

where the routine names provided in square brackets are the ones called at the corresponding step of the algorithm.

Figure 4.3 also shows the communication scheme between the engine and the components, taking into account the interaction and priority models. The communication is initiated by the engine by calling `init()`, which initializes the components and executes them for the first time. Further communication from the engine to the components are by calling `notify()`, which triggers the atoms concerned in the execution of the selected interaction. The atoms communicate to the engine, specifying their eagerness to communicate through their enabled ports by calling `sync()`, and eventually getting blocked.

The BIP Engine has run-time options for:

- execution
- enumerative state-space exploration

For the execution, the engine offers the possibilities of running either a random trace (by randomly selecting an enabled interaction for execution), or an interactive trace, where the user is offered to choose an interaction out of the enabled ones. When a trace is executed, the engine displays the sequence of interactions.

The state space exploration mode generates state graphs in the form of labeled transition systems, as shown in figure 4.4. The state graphs can be analyzed by model checking [QS82, CE81] and by Observers [Tre94, Pha94]. The state graph can also be minimized and compared with tools like Aldebaran [BFKL97].

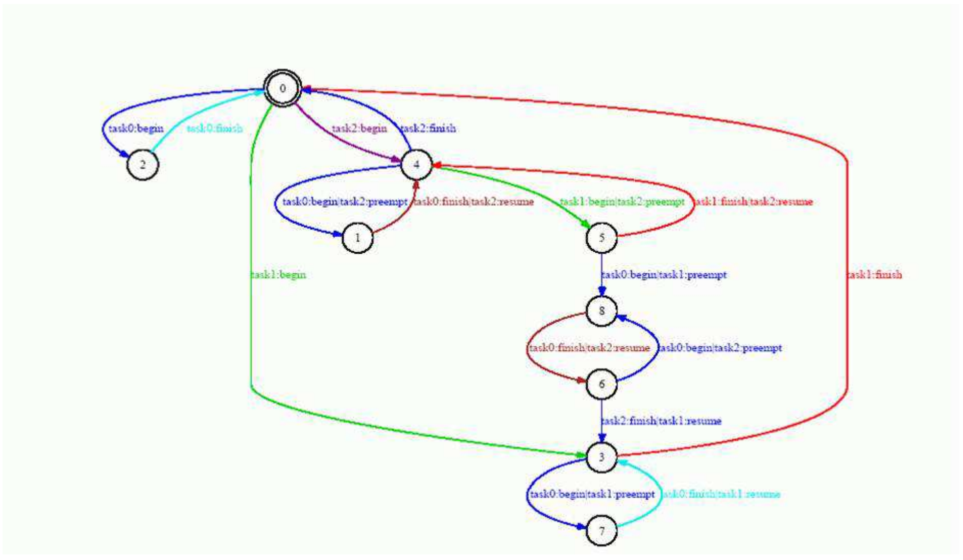

0
task0:begin
task0:finish
2
task2:begin
task2:finish
4
task0:begin|task2:preempt
task0:finish|task2:resume
1
task1:begin|task2:preempt
task1:finish|task2:resume
5
task1:begin
task0:begin|task1:preempt
task1:finish
8
task0:finish|task2:resume
task0:begin|task2:preempt
6
task2:finish|task1:resume
3
task0:begin|task1:preempt
task0:finish|task1:resume
7

Figure 4.4: LTS generated by exploration.

4.3.1.1 Model checking

We can used the model-checker tool Evaluator [MS00] to perform on-the-fly verification of temporal properties on the state-space generated by the BIP engine on exploration of a system. For example, properties like specific order of execution of interactions in a system can be efficiently verified, as has been applied in the verification of a robotic controller modeled in BIP [BBG+08].

4.3.1.2 Verification using Observers

For a given system S and a safety property P, we construct first an observer for P, i.e. an automaton which monitors the behavior of S and reports an error on violation of P. The verification consists of exploring the state-space of the product system. Observers can be modeled in BIP itself. They consists of control states with special labels, like `ERROR` and `PRUNE`. During exploration, if a global system state containing the `ERROR` state of the observer is reached, an error is reported. This technique has been used in the verification of timing properties of modules in the BIP model of a robot controller [BBG+08].

The `PRUNE` state is used to skip exploration of some specific branch of the state graph, and is useful in reducing the exploration time for big systems.

4.3.2 Centralized Symbolic Engine

In the enumerative BIP engine, for each connector, the engine needs to compute all the possible interactions, check which are enabled by evaluating the guards, and select the maximal enabled one to be executed. As the number of interactions is exponential in the number of ports in the connector in the worst case, the performance of this engine degrades for examples where the topology of the connectors and the typing of their ports produces exponential number of interactions.

In the boolean BIP engine, component behavior and connectors are represented as boolean functions. The implementation of the boolean functions is made using the BDD package CUDD [Som].

For an atomic component, all ports, control states, and guards 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 of its ports and guards. The global behavior is obtained as a boolean operation on the expressions representing atoms and connectors.

The choice of an interaction to be executed boils down to evaluating the guards and 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 iteration step by efficient BDD manipulations. In comparison to the potentially exponential cost of the enumerative engine, this renders a more efficient engine with evaluation that, in general, remains linear. The following sections describe in details the boolean representation of the individual BIP elements (atoms, connectors, priorities) and its evaluation by the engine.

4.3.2.1 Boolean representation of Atomic Components

For each atomic component $B_i = (Q_i, P_i, \rightarrow)$ and each state $q \in Q_i$, we define boolean functions $f_q, f_{B_i} \in \mathbb{B}[Q_i, P_i]$ as follows

$$f_q \;=\; q \wedge \bigwedge_{q' \neq q} \overline{q'} \wedge \bigvee_{q \xrightarrow{a}} \left(a \wedge \bigwedge_{p \in P_i \setminus a} i\,\overline{p} \right), \qquad f_{B_i} \;=\; \bigvee_{q \in Q_i} f_q \vee \bigwedge_{p \in P_i} \overline{p}\,.$$

Assuming that the value of a state variable $q \in Q_i$ is set to *true*, valuations of port variables in P_i satisfying f_{B_i} correspond to possible transitions of B_i from the state q. Notice that the constant *false* valuation means that the component does not change its state. The boolean function, representing all the possible transitions of the product automaton, is then the conjunction $f_B = \bigwedge_{i=1}^{n} f_{B_i}$.

Example 4.3.1 *Consider the example of a causality chain, shown in figure 4.5. It has a sender (S) and two receivers (R1, R2). For a message to be received by R2, it has to be received at the same time by R1 also. This co-ordination scheme is common in reactive systems. To avoid confusion*

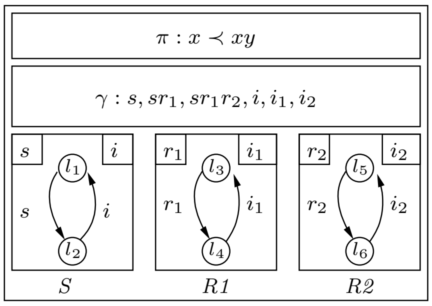


Figure 4.5: Causality Chain.

in notations, we denote the control states of S by l_1 and l_2. The boolean function representing S is then $f_S = l_1 s \bar{i} \vee l_2 \bar{s} i \vee \bar{s}\,\bar{i}$, and the functions representing the two receivers are computed similarly. Taking the conjunction, we obtain the boolean function representing the product of the three atomic components:

$$\begin{aligned} f_B &= (l_1 \overline{l_2}\, s\, \bar{i} \vee \overline{l_1}\, l_2\ \bar{s}\, i \vee \bar{s}\, \bar{i}) \wedge (l_3 \overline{l_4}\, r_1 \overline{i_1} \vee \overline{l_3}\, l_4\ \overline{r_1}\, i_1 \vee \overline{r_1}\, \overline{i_1}) \wedge \\ &\quad (l_5 \overline{l_6}\, r_2 \overline{i_2} \vee \overline{l_5}\, l_6\ \overline{r_2}\, i_2 \vee \overline{r_2}\, \overline{i_2})\,. \end{aligned}$$

4.3.2.2 Boolean representation of Connectors

Let P be the set of all ports in the system. In order to obtain a boolean representation for connectors, we compute, for each connector x, the causal tree $t = \tau(x)$ ([BS08a]). The boolean function $f_C \in \mathbb{B}[P]$ representing a connector is obtained from the causal tree essentially by inverting the arrows in order to obtain, for each $p \in P$, the *causal rule* of the form $p \Rightarrow \bigvee_{i=1}^{n} a_i$ (cf. [BS08a]).

The meaning of such a causal rule is that, for the port p to participate in an interaction, at least one of a_i must be part of it. Finally, observe that at least one of the nodes forming the roots of causal trees must participate in the interaction. Therefore, to the conjunction of the above causal rules, we add the disjunction of roots of t.

Example 4.3.2 *In the model of example , the causal order is realized by the connector $s'[r_1'r_2]$. The boolean function representing this connector (as described in [BS07b]) is*

$$f_x = s \wedge (r_1 \Rightarrow s) \wedge (r_2 \Rightarrow r_1)\,.$$

When several connectors $C_1, \ldots, C_m$ are used to describe the interactions in a system, boolean functions are individually computed as above for each of the C_i and combined by taking $f_C = \bigvee_{i=1}^{m} \left(f_{C_i} \wedge \bigwedge_{p \notin C_i} \overline{p} \right)$, where $p \notin C_i$ means that the port p is not used in C_i.

4.3.2.3 The Engine Protocol

The following protocol is used at each step of the execution to choose an interaction to be fired. It starts with an initialization phase, where the following boolean functions are computed: $f_B \in \mathbb{B}[\bigcup_{i=1}^n Q_i, P]$, representing the atomic behaviors; and $f_C \in \mathbb{B}[P]$, representing the connectors. The conjunction $f_S = f_B \wedge f_C$ is also computed at this stage. The main loop of the engine consists of the following steps:

1. Each atomic component B_i sends to the engine current its state $q_i \in Q_i$.

2. The engine picks any valuation a on P, such that $(a, q) \models f_S \wedge \bigwedge_{i=1}^n q_i$, where q is the valuation on $\bigcup_{i=1}^n Q_i$ representing the global state of the system.

3. The engine notifies components of their respective transitions to take, by communicating to each component B_i the label $a \cap P_i$ of the transaction to take.

$(a, q) \models f_S$ implies $a \models f_C$, which means that $a \in \gamma$, i.e., the interaction a is authorized by the interaction model. Similarly, $(a, q) \models f_B$ means that a is active in the current global state q of the system. Thus, any such interaction a represents an enabled interaction.

The computation in the engine is limited to taking the conjunction of f_S with state variables representing current states of atomic components. The BDD for f_S is only computed once and remains constant throughout the execution of the BIP model. Thus, this computation is proportional to the number of atomic components in the system.

4.3.2.4 Benchmarks

We compare the engine execution times of the enumerative and boolean engines for two benchmark examples. The BIP models for both examples are limited to synchronization, i.e., do not have any data transfer. Below we present the two examples and the simulation results.

4.3.2.4.1 Bus Consider a system of N independent clusters of components communicating through a "bus", i.e., a single common connector (see fig 4.6). Each cluster consists of four components that alternate computation (transitions labeled c_i, for $i \in [1, 4]$) and communication (s_i, for $i \in [1, 4]$). Computations of the four components in a cluster are completely independent and cannot be synchronized. Thus, for each $i \in [1, 4]$, there is a singleton connector c_i. On the other hand, communications s_i are weakly synchronized by the connector $s_1' s_2' s_3' s_4$. In this connector, the ports s_1, s_2,

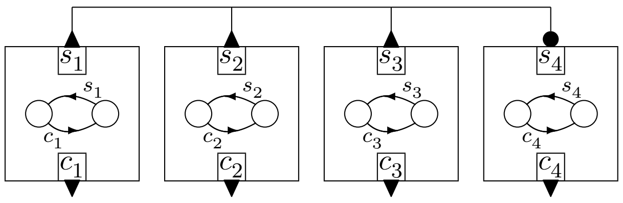


Figure 4.6: A unit cluster for the Bus example

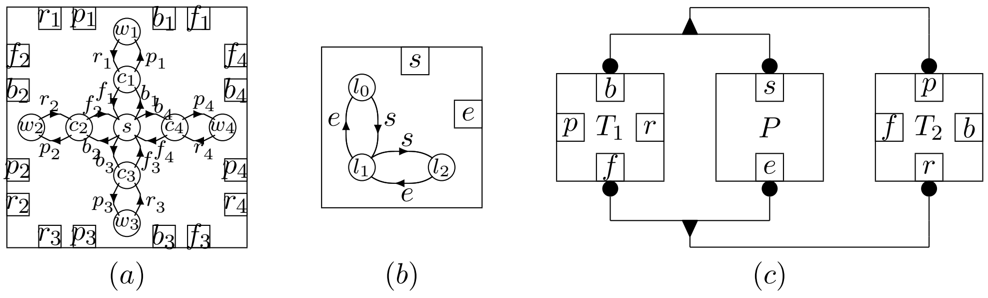


Figure 4.7: BIP models of a task (a) and a processor (b); connectors (c)

and s_3 are triggers, whereas s_4 is a synchron. This means that communication is only possible through s_4 when at least one other component is ready to communicate—the fourth component is an *observer*.

In a system of N clusters (i.e., $4N$ components), there are $5N$ connectors. We say that connectors are *sparse* in this system, which favors the enumerative engine: execution times of both enumerative and boolean engines are linear in the number of components.

4.3.2.4.2 Preemptable Tasks This example originates from [WDE05], where a performance evaluation problem is considered with timed tasks running concurrently on shared processors. Here, we disregard the timed aspects of this example and only consider the aspect of the task behavior concerned with processor sharing.

Consider M processors and N tasks that can be executed on any processor. A processor can have at most two tasks assigned to it at a time: one running and one preempted. On arrival of a new task, the running one is preempted. A task is resumed, when the one that has preempted it, terminates.

The BIP model of the task component type is shown in figure 4.7(a). It has an "idle" state s, and, for each processor $i \in [1, M]$, a "compute" state c_i and a "wait" state w_i. An idle task (in state s) can begin execution on the processor i by taking the transition labeled b_i from the state s to the state c_i. It can finish execution by taking the transition labeled f_i from the state c_i back to the state s.

A task running on the processor i can be preempted (transition labeled p_i from the state c_i to the state w_i) and, subsequently, resumed (transition labeled r_i from the state w_i to the state c_i).

The BIP model of the processor component type is shown in figure 4.7(b). A processor k is free in the control state l_0, and can start executing a new task by taking a transition labeled s to the state l_1. To do so, it must synchronize with the "begin" port b_k of the task to be allocated.

From the state l_1, the processor can move back to state l_0, if the running task finishes (transition labeled by e). Otherwise, it can preempt the running task and start a newly arriving task by taking a transition to l_2, labeled by the port s. To do so, it must synchronize with the "begin" port b_k of the newly arrived task and "preempt" port p_k of the currently running task. Similarly, for a processor with two tasks (state l_2) an interaction ef_kr_k ends the running task and resumes the preempted one.

Each task is connected with every processor and every other task. Figure 4.7(c) shows the corresponding connectors $[bs]'p$ and $[fe]'r$ between a task T_1, a processor P, and another task T_2. For the sake clarity, we show only the relevant ports.

Thus, in a system of N tasks and M processors, there are $2N(N-1)M$ connectors. We say that connectors are *dense* in this system, which favors the boolean engine: execution time of the boolean engine is linear in the number of components, whereas that of the enumerative engine is linear in the number of connectors and quadratic in the number N of tasks.

Observe that e.g., connector $[bs]'p$ has two interactions *bs* and *bsp*. Whenever a task is already running on a processor, it has to be preempted before a new one can be started. This is realized by the maximal progress rule, i.e., giving priority to *bsp* over *bs*. Both enumerative and boolean engines automatically pick the maximal interaction, which does not increase computational complexity of the underlying algorithms (contrary to arbitrary priorities).

4.3.2.4.3 Simulation results We measured the engine execution times for both examples for 10^6 iterations of the engine loop. Figure 4.8 shows the engine execution times, obtained with both the enumerative and boolean engines, related to the number of components in the system.

As expected, for the Bus example, the execution times of both engines are close and linear in the number of components (dashed lines in figure 4.8). The enumerative engine outperforms the boolean one. This is due to the fact that the basic operation of the boolean engine (BDD conjunction with the state variable) is more expensive than that of the enumerative engine (connector evaluation). However, one can reasonably expect that, even if the number of connectors is linear in the number of components, when this proportion increases, execution time of the enumerative engine will increase

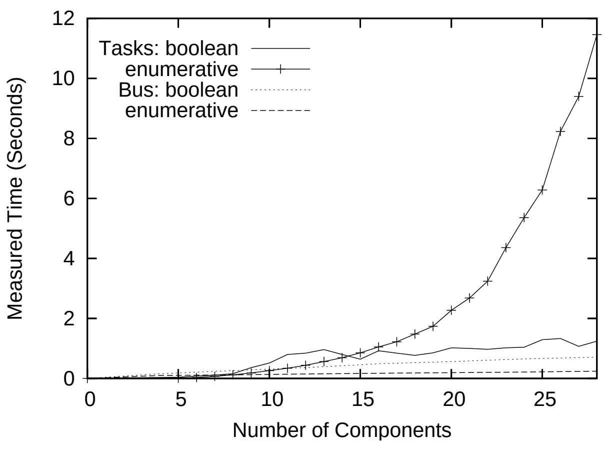


Figure 4.8: Engine execution times.

accordingly, whereas that of the boolean one should stay the same.

In the Preemptable Tasks example, we fixed the number of processors to $M = 4$. The execution time of the enumerative engine is linear in the number of connectors, i.e., quadratic in the number of components (solid lines in figure 4.8). The execution time of the boolean engine is linear in the number of components. Thus boolean engine considerably outperforms the enumerative one.

4.3.2.4.4 Remarks We presented the symbolic implementation of the BIP execution framework. This implementation is based on computing boolean representation for components and connectors by using an existing BDD package. The boolean representation is used by the engine at run time to compute the interaction to be executed at each iteration of the engine loop. The aim of the symbolic implementation is to reduce the overhead observed in the original enumerative engine due to this computation.

The main goal of this work is to demonstrate the feasibility of this approach. Therefore the focus is on the boolean representation of connectors, i.e., disregarding priorities and guards.

We have compared the execution times of the two engines. For the symbolic implementation, the engine execution time is proportional to the number of components, whereas, for the enumerative engine, it is proportional to the number of connectors.

The engine execution times were evaluated for two examples favoring respectively the two engines. For systems with dense connectors (as in the Preemptable Tasks example), the execution time of the enumerative engine explodes, whereas that of the boolean engine remains small. For systems where connectors are sparse (as in the Bus example), the execution times of both engines are close, with the enumerative one potentially outperforming the symbolic one.

As we have mentioned above, the execution time of the boolean engine depends on the number of components and not on the number of connectors. Therefore, even when the number of connectors is linear in the number of components, the boolean engine can outperform the enumerative one, provided that the average number of connectors per component is sufficiently high.

The computation of the boolean representation must only be performed once for each model (e.g., at code generation), and consequently is irrelevant for the performance comparison. This, together with the above observations, allows us to conclude that, in general, the symbolic implementation of the BIP execution model considerably reduces the engine overhead. Furthermore, performance degradation for systems with sparse connectors is relatively small. It should also be observed that, for systems with dedicated engine, the choice of the enumerative or symbolic implementation can be made at code generation phase, in order to optimize the engine performance.

Future work includes extending symbolic implementation to all BIP elements, i.e., priorities and guards. Boolean representation of the latter is straightforward, but remains to be fully formalized.

4.3.3 Distributed engine

In this section, we present an implementation for the distributed semantics of BIP, introduced in section 2.6. The model of BIP components with partial states is a first step towards a distributed implementation of BIP by separating internal computations from interactions. However, this model uses strong synchronization and therefore is still not directly implementable on arbitrary platforms where rendezvous is usually not available as a communication primitive.

We present here, a distributed implementation of BIP components with partial states, where multiparty interactions are replaced by asynchronous communication protocols (see figure 4.9). The target model is *input-output*

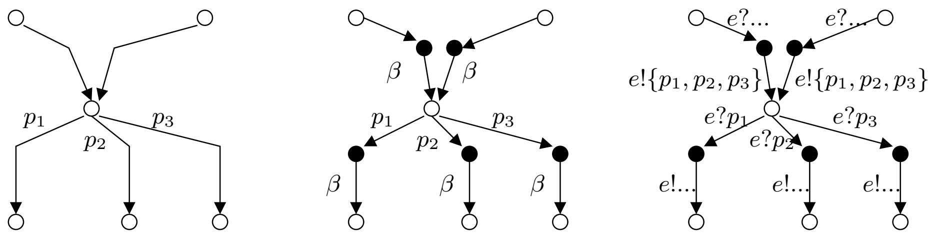


Figure 4.9: Transformation from atomic BIP components (left) towards atomic components with partial states (middle) and io-machines (right)

systems (io-systems) that are collections of parallel *input-output machines* (io-machines) communicating asynchronously by message passing through FIFO channels. This model is conceptually simple and directly encompasses primitives offered by languages used for modeling of distributed systems (such as SDL[IT99] or IO-automata[GL98]) or primitives usually available on distributed execution platforms (e.g. asynchronous execution of threads or processes, inter-process and inter-thread communication through FIFO queues, network protocols).

The principle of implementation is sketched in figure 4.10. Given $\pi^\perp\gamma^\perp(B_1^\perp, B_2^\perp, ..., B_n^\perp)$ and a $\pi^\perp$-oracle $\mathcal{O}$, the implementation is an io-system consisting of io-machines B_i^{io} emulating the behavior of $B_i^\perp$ and an additional io-machine, the *Engine* $E(\gamma^\perp, \pi^\perp, \mathcal{O})$ realizing the coordination between them. Communication takes place only between the atomic components and the Engine, and never directly between different atomic components – this leads to an io-system with a centralized architecture.

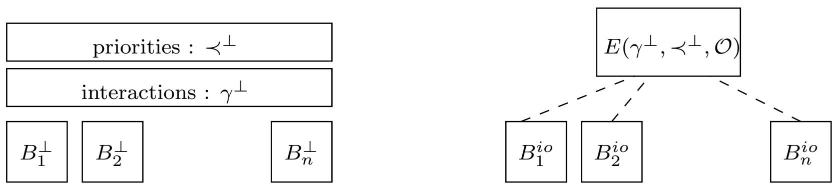

priorities : $\prec^\perp$
interactions : $\gamma^\perp$
$B_1^\perp$
$B_2^\perp$
$B_n^\perp$
$E(\gamma^\perp, \prec^\perp, \mathcal{O})$
B_1^{io}
B_2^{io}
B_n^{io}

Figure 4.10: Implementation: The Overall Structure

Formally, an io-system is a tuple $\mathcal{S} = (\mathcal{M}, Act, \{A_i = (Q_i, \hookrightarrow_i)\}_{i\in I})$ where

- $\mathcal{M}$ is a set of *messages*,
- *Act* is a set of actions α including *outputs* $j!m$ – output of the message $m \in \mathcal{M}$ to machine $j \in I$, *inputs* $j?m$ – input of message $m \in \mathcal{M}$ sent by machine $j \in I$ or *uninterpreted actions* a,
- $\{A_i = (Q_i, \hookrightarrow_i)\}_{i\in I}$ is a finite set of io-machines, where
  - Q_i is a finite set of states,
  - $\hookrightarrow_i \subseteq Q_i \times Act \times Q_i$ is a finite set of transitions labeled with actions.

States of io-systems are represented by configurations $\{(q_i, w_i)\}_{i\in I}$ where $q_i \in Q_i$ is a local state and $w_i \in (I \times \mathcal{M})^*$ is the FIFO-queue content of io-machine i. The semantics of io-systems is given as a labeled transition system on configurations. For each transition $q_i \overset{\alpha}{\hookrightarrow}_i q_i'$ of the io-machine i, we consider the following transitions on configurations corresponding respectively to input, output and uninterpreted actions:

- $\{..., (q_i, (j, m) \bullet w_i'), ...\} \overset{\tau}{\hookrightarrow} \{..., (q_i', w_i'), ...\}$ when $\alpha = j?m$,

- $\{..., (q_i, w_i), (q_j, w_j), ...\} \overset{\tau}{\hookrightarrow} \{..., (q'_i, w_i), (q_j, w_j \bullet (i, m)), ...\}$ when $\alpha = j!m$
- $\{..., (q_i, w_i), ...\} \overset{a}{\hookrightarrow} \{..., (q'_i, w_i), ...\}$ when $\alpha = a$,

The implementations of atomic components are io-machines obtained as follows. Whenever a ready state is reached, they output a message to the Engine containing (1) the sets of ports on which they are willing to interact and (2) their local ready state. Then, they wait for a notification from the Engine indicating the port selected for interaction. Depending on this port, they continue their execution. Formally, given $B_i^\perp = (Q_i \cup Q_i^\perp, P_i \cup \{\beta_i\}, \rightsquigarrow_i)$, its corresponding io-machine $B_i^{io} = (Q_i \cup Q_i^\perp, \hookrightarrow_i)$ has the same set of states as $B_i^\perp$ and transitions defined by the following rules (see Figure 4.9):

- $q_i \overset{e!(X,q'_i)}{\hookrightarrow_i} q'_i$ *interaction request* whenever $q_i \overset{\beta_i}{\rightsquigarrow_i} q'_i$ and $X = \{p \mid q'_i \overset{p}{\rightsquigarrow_i}\}$
- $q_i \overset{e?p}{\hookrightarrow_i} q'_i$ *interaction notification* whenever $q_i \overset{p}{\rightsquigarrow_i} q'_i$

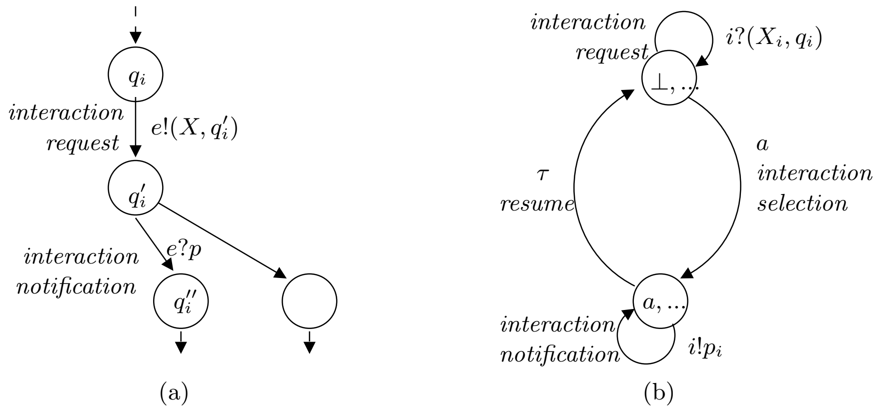


Figure 4.11: Principle of Implementation: (a) io-machines for atomic components and (b) io-machine for the Engine

The Engine $E(\gamma^\perp, \pi^\perp, \mathcal{O})$ is an io-machine (see Figure 4.11) realizing the coordination between atomic io-machines for a given set of interactions $\gamma^\perp$, priorities $\pi^\perp$ and a $\pi^\perp$-oracle $\mathcal{O}$. Iteratively, the Engine receives and stores the sets of ports and the local states of components ready to interact. Depending on this information, it seeks a feasible interaction, which is maximal with respect to priorities and allowed by the oracle $\mathcal{O}$. If such an interaction exists, the Engine *executes* it by notifying sequentially, in some arbitrary order, all the involved components. Formally, given $\pi^\perp\gamma^\perp(B_1^\perp, B_2^\perp, ..., B_n^\perp)$ and an oracle $\mathcal{O}$, the Engine is the io-machine $(Q_e, \hookrightarrow_e)$ where

• $Q_e = (\gamma \cup \{\perp\}) \times \bigotimes_{i=1}^n 2^{P_i} \times \bigotimes_{i=1}^n (Q_i \cup \{\perp\})$ is the set of states of the form $(a^\perp, \mathbf{X}, \mathbf{q}^\perp)$ with $\mathbf{X} = (X_1, ..., X_n)$ and $\mathbf{q}^\perp = (q_1^\perp, ..., q_n^\perp)$ where

- $a^{\perp} \in \gamma \cup \{\perp\}$ is the interaction being currently executed, $\perp$ if none;
- $X_i \in 2^{P_i}$, is the set of ports on which component i is able to interact, empty if still busy;
- $q_i^{\perp} \in Q_i \cup \{\perp\}$ is the state q_i if component i is ready to interact, $\perp$ if still busy.

- $\hookrightarrow_e$ contains the following transitions
- $(\perp, \mathbf{X}, \mathbf{q}^{\perp}) \overset{i?X_i,q_i}{\hookrightarrow_e} (\perp, \mathbf{X}[X_i/i], \mathbf{q}^{\perp}[q_i/i])$ *interaction request*, stores information received from component i ready to interact.
- $(\perp, \mathbf{X}, \mathbf{q}^{\perp}) \overset{a}{\hookrightarrow_e} (a, \mathbf{X}, \mathbf{q}^{\perp})$ *interaction selection*, whenever an interaction a exists such that $a \subseteq \cup_{i=1}^{n} X_i$, a is maximal with respect to priorities $\pi^{\perp}$ and a is allowed by the oracle $\mathcal{O}$ at state $\mathbf{q}^{\perp}$. It consists in executing the interaction and moving to a state from which all the components involved will be notified.
- $(a, \mathbf{X}, \mathbf{q}^{\perp}) \overset{i!p_i}{\hookrightarrow_e} (a, \mathbf{X}[\emptyset/i], \mathbf{q}^{\perp}[\perp/i])$ *interaction notification* and *cleanup* of the i component involved in the interaction a, that is when $a \cap X_i = \{p_i\} \neq \emptyset$,
- $(a, \mathbf{X}, \mathbf{q}^{\perp}) \overset{\tau}{\hookrightarrow_e} (\perp, \mathbf{X}, \mathbf{q}^{\perp})$ *resume*, when all atomic components have been notified, that is $a \cap \cup_{i=1}^{n} X_i = \emptyset$. It consists in moving back to a state where requests are handled.

The correctness of the implementation is formally established by the following theorem.
Distributed Execution Platform The distributed semantics is driven by the engine, for multi-threaded execution of the generated code of the BIP model. Each atomic component is assigned to a thread, the engine being a thread itself. The engine is parameterized by a dynamic or lazy oracle. Iteratively, the engine computes feasible interactions available on ready components. Then, if such interactions exist and the oracle allows them, the engine selects one for execution and notifies the involved components.

4.3.3.1 Benchmarks

We present two examples illustrating the application of the results on a prototype implementation. We evaluate for two different types of oracles, the degree of parallelism over time, measured as the number of simultaneously executing atomic components. Before providing experimental results, we analyze the relationship between degree of parallelism and parameters of the system.

To simplify the analysis, consider a system consisting of n atomic components always able to interact through their ports. We distinguish the following cases, illustrated in Figure 4.12:

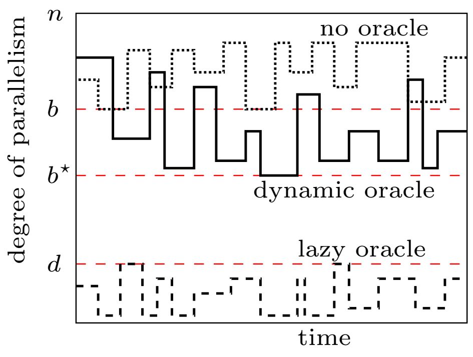


Figure 4.12: Performance analysis

• For an implementation without oracle, the degree of parallelism is related to the minimal cardinality b of blocking subsets of atomic components. A subset of atomic components is *blocking* iff every interaction in the system requires at least one component of the subset to participate. Now, the degree of parallelism l is such that $b \leq l \leq n$. In fact, whenever less than b components are running some interaction is possible and the Engine can eventually launch it;

• For an implementation with the lazy oracle, the maximal degree of parallelism is related to the maximal degree of interaction d, that is the maximal number d of components involved in a single interaction. In this case, the degree of parallelism l is such that $0 \leq l \leq d$. Interactions can be executed only from global states so there is no possibility of concurrency between interactions - the Engine is not able to keep running more than d atomic components at time;

• Finally, for dynamic oracles, the degree of parallelism is related again to the minimal cardinality $b^\star$ of some particular blocking sets of atomic components, the ones which block *all the maximal* interactions. We have $b^\star \leq b$ and the degree of parallelism l achieved in this case is such that $b^\star \leq l \leq n$. Using a similar reasoning as in the case without oracle, whenever less than $b^\star$ components are running, there should exist a maximal interaction ready and the Engine can eventually launch it.

As a first benchmark, we consider a linear chain consisting of a set of identical components connected serially as shown in Figure 4.13. A component C_i has two ports, l_i and r_i. It has a single control state S_i, and two transitions labeled by l_i and r_i. The transition r_i is always enabled, its guard being *true*, whereas the transition l_i has a non-trivial guard g_i. We model broadcast from each component to its right neighbor by considering two types of interactions, 1) a set of singleton interactions consisting of the ports r_i; 2) a set of binary interactions $r_i l_{i+1}$ between the neighboring components, and 3) the priority $r_i \pi r_i l_{i+1}$ for the above interaction pair.

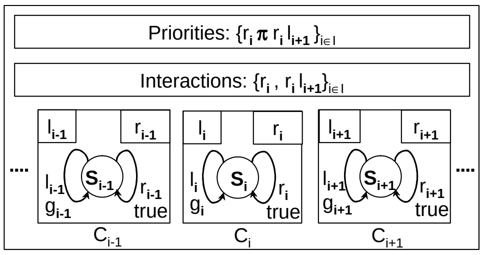


Figure 4.13: The Linear Chain

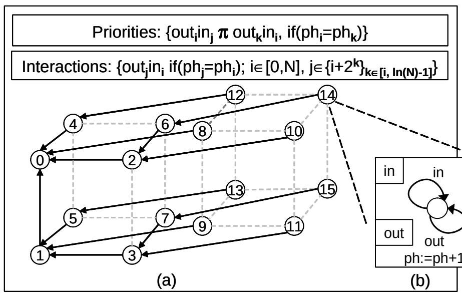

Priorities: {out_i in_j π out_k in_i, if(ph_i=ph_k)}
Interactions: {out_j in_i if(ph_j=ph_i); i∈[0,N], j∈{i+2^k}_{k∈[i, ln(N)-1]}}
in
out
ph:=ph+1
(a)
(b)

Figure 4.14: The Parallel Adder

Our experiment considers a system with 25 such components. Each component executes 100 steps (transitions), getting busy for 50-60 milliseconds on an l transition and 5-6 milliseconds on an r transition respectively. We performed the experiment on a single-processor PC running linux. The busy times of the atomic components were simulated by *sleep* system calls. We measured the degree of parallelism in the system with respect to the execution time. Figure 4.15 shows the results obtained for dynamic and lazy oracles.

Without oracle, the degree of parallelism is 25 continuously. In fact, whenever a component is ready, it can continue alone on the r interaction and the Engine notifies it immediately. For the lazy oracle, the maximal degree of parallelism equals the maximal degree of an interaction, which is 2. Whenever an interaction takes place, the two participating atomic components are active simultaneously for the first 5-6 ms, after which only the atomic component performing the l transition remains busy for 50-60 ms. Therefore, the degree of parallelism stays at an average close to 1. Finally, for dynamic oracle, the minimal blocking set has cardinality 12 (as for a linear chain with n atoms, the minimal cardinality is $n/2$, when every alternate atoms are busy blocking all the maximal interactions). Hence, we have at least 12 atomic components executing at any time. The measured degree of parallelism in this case, remains in average higher than 15.

The second benchmark treats a parallel adder originally presented in [Qui86], which adds 2^m values in a hypercube multi-processor machine. When the algorithm begins, the nodes hold the values to be added. On termination, the node labeled 0 contains their sum. Figure 4.14 presents the BIP model of a pipelined parallel-adder in a 4-dimensional hypercube with 2^4 nodes. Each node is modeled as a BIP component with ports *in* and *out*, labeling two transitions from a single control state, as shown in Figure 4.14(b). It also contains an array of values to be added (not shown on the figure) and the variable *ph* which records the index of current running addition on that

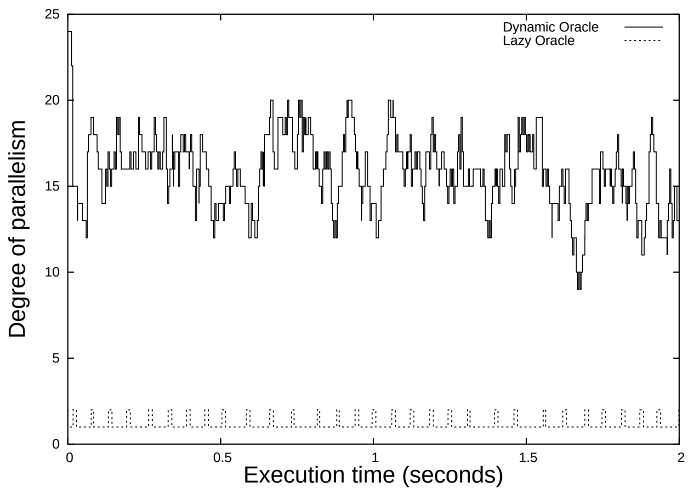


Figure 4.15: Degree of Parallelism for Linear Chain

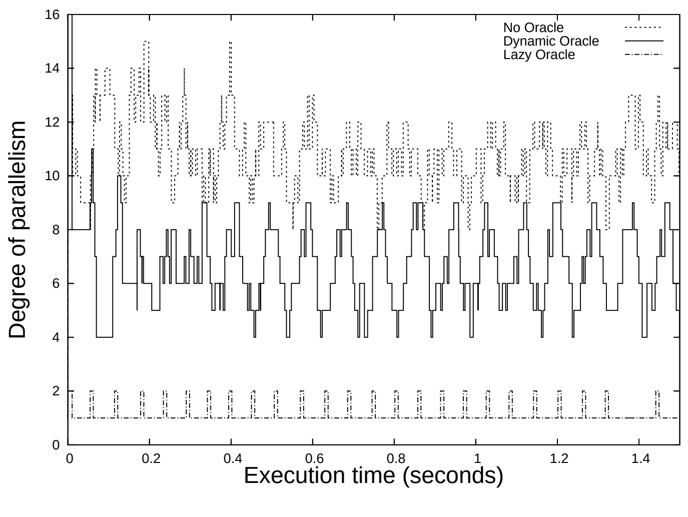


Figure 4.16: Degree of Parallelism for Parallel Adder

node.

For each addition, every node receives partial addition results from its predecessors, adds them to its own value, sends the resulting sum to its unique successor and increments its *ph* variable. Communications between nodes are modeled as interactions between the *out* port of a node and the *in* port of its successor, with a transfer of value from the node to the successor. Priorities are used to enforce correct order of the computation, i.e., a node cannot perform an *out* unless it has synchronized through its *in* port with all its predecessors. The final result of every addition is generated by the root node labeled 0.

The degrees of parallelism achieved, respectively without oracle and with lazy and dynamic oracles, are shown in Figure 4.16. Without oracle, the degree of parallelism is in average equal to 10. Let us notice that, without oracle, the functional behavior is completely wrong as priorities are used to enforce the right order of computation between nodes. With the lazy oracle, the maximal degree of parallelism equals the maximal degree of interaction which is 2. However, due to specific timing constraints on the execution of *in* and *out* transitions, the degree of parallelism stays in average close to 1. Finally, the dynamic oracle achieves a much better performance with an average degree of parallelism equal to 7.

4.3.3.2 Remarks

We study a distributed implementation method for BIP, a framework for the description of component-based heterogeneous systems. BIP offers two powerful mechanisms for composing components by using interactions and priorities. The combination of interactions and priorities is expressive enough to express usual composition operators of other languages as shown in [BS07a]. In particular to model broadcast, interactions do not suffice and other operators such as restrictions or priorities are needed. Furthermore, priorities are essential for describing scheduling policies, run-to-completion execution, urgency in real-time systems [GS03]. The proposed implementation method is quite general and can be easily adapted to other languages.

A key innovative idea is the translation of languages based on global state semantics to observationally equivalent distributed models from which implementation is straightforward. The decomposition of the translation in two steps allows separation of concerns in solving two main problems: the definition of partial state semantics and the expression of composition in terms of message passing primitives. Operational semantics provide an adequate framework for formalizing the translation. The models are obtained by successive refinements that preserve observational equivalence.

The main results show that whenever priorities are needed to express coordination between components, the operational semantics rules should be strengthened to take into account dependency between interactions. Oracles are very simple controllers enforcing preservation of semantics. Maximal parallelism is achieved for dynamic oracles allowing interaction as soon as possible. Nonetheless, these oracles may entail considerable computational overhead. As illustrated by experimental results the degree of parallelism depends on the type of the oracle and topology of the interactions.

There are many open problems to be investigated in the proposed framework for distributed implementation. These include the preservation of specific classes of properties, and less centralized implementations for the Engine.

Part III

Applications

Chapter 5

Case Studies

This chapter provides some case studies and real applications modeled and analyzed in the BIP framework. The first example shows the modeling of a mixed hardware/software system, strengthening the utility of BIP for heterogeneous systems modeling. We provide two other real applications showing the usefulness of software componentization in BIP for the design and analysis of big control intensive software systems.

5.1 Modeling Mixed HW/SW Systems

5.1.1 Modeling and Verification of Networked Systems – A Case Study on TinyOS-based Networks

Complex heterogeneous systems such as networked systems, composed of hardware and software, are validated by simulation of physical or virtual prototypes. The main obstacle for the application of verification techniques, which are successfully applied to complex software or hardware, is the lack of methods for building global models faithfully representing their behavior. We apply a model construction methodology using the BIP component framework, to TinyOS [Tin] based networks. The methodology consists in building the model of a node as the composition of a model extracted from a nesC [GLvB+03] program describing the application, and models of TinyOS components. Models for networks are obtained by composition of models for nodes by using connectors implementing different types of radio channels. This opens the way for enhanced analysis and early error detection by using verification techniques.

The work has been done in collaboration with Marc Poulhiès, Laurent Mounier, Jacques Pulou and Joseph Sifakis [BMP+07b].

5.1.1.1 Introduction

Modeling and verification techniques have been successfully applied to complex software or hardware. Currently, validation of complex heterogeneous systems such as networked systems, is carried out by simulation or testing of prototype implementations. Existing verification techniques could be applied to heterogeneous systems, provided that we have methods for building executable models faithfully representing their behavior. The construction of such models by composition of models of the application software and of the underlying execution platform is a scientific and technical challenge.

A main difficulty for jointly modeling an application software and its execution infrastructure, is that they adopt very different execution models and views. In component-based software, components are mainly used for structuring functions and associated data. Interactions between components are point-to-point (*e.g.* function calls) through binding interface specifications. This view is far from a system-oriented view needed to model execution mechanisms and their interaction with the external environment. For instance, programs in the nesC language used for programming TinyOS-based applications [GLvB+03], are sets of components and relations between *provided* and *used* interfaces. This programmer's view is not sufficient for determining the interactions between the application software and TinyOS which manages entities such as tasks, commands and events by applying specific scheduling rules.

Wireless sensor networks are complex component-based systems with rich dynamics subject to strong extra-functional requirements. Their design involves the composition of a variety of hardware and software components developed with different methodologies and tools. We have a limited understanding on how specific component features impact the global behavior. To cope with complexity and enhance understanding, it is important to consider wireless sensor networks as the composition of a relatively small set of functions, services and components by using incremental structuring principles. The main obstacle for this is the lack of modeling frameworks encompassing heterogeneity. Most simulation environments use simulation software built in a more or less ad hoc manner, by integrating the application code in specific platforms [LLWC03, GSR+04, TLP05, PBM+04, ECZ06]. They can be useful for debugging purposes but they are not adequate for a more thorough exploration of a network's non-deterministic dynamics.

We apply to TinyOS-based networks, a model construction methodology for building heterogeneous real-time systems. This opens the way for enhanced analysis and early error detection by using verifications techniques. The methodology is not specific to TinyOS, and we believe, can be adapted to networked systems, in general.

For a given sensor node, a global BIP model is built by composing BIP models for its application software and for TinyOS. The latter is obtained by composing controllers for the execution of tasks, events, radio and hardware devices. The models for application software are generated automatically from nesC programs by a translator (shown in figure 5.1) which takes annotated nesC code as input and generates the corresponding BIP components and connectors. BIP models can be analyzed by using state space exploration techniques offered by the toolset.

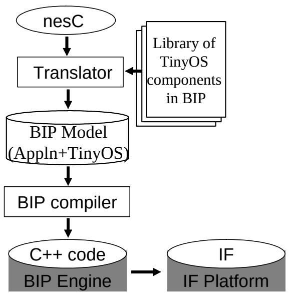


Figure 5.1: The modeling flow.

The methodology presented is characterized as follows:

- A global model for the network is built by composition of BIP components modeling the application software as well as operating system and radio features. This is a main difference with existing simulation approaches, directly using TinyOS and C code generated by the nesC compiler. The BIP model for the TinyOS is an abstract machine driving the execution of the BIP model, obtained by translation of the application software written in nesC.

- A significant difference with existing simulation approaches, is that the obtained BIP models are non-deterministic and fully characterize the behavior of the wireless sensor network. Furthermore, these models have a well-defined notion of state. They can be verified by using state space exploration techniques e.g., model-checking. Even if due to inherent limitations, complete verification of complex networks is intractable, verification is very useful for systematic debugging and early error detection.

- Another important difference is incremental model construction of BIP models [Sif05]. Incrementality means that the global model is obtained by progressively composing its atomic components 2.1.1. This system construction methodology allows defining an architecture hierarchy, with the glue specifying the composition at an architectural level from its subordinate levels. Figure 5.2 shows the architecture of a sensor node (mote) consisting of NesC applications and TinyOS, with their internal contents. This methodology allows preservation of the structure through translation into BIP. That is, it is possible to identify in the global model all its atomic components and their interactions. This allows in particular, to study the impact of changes of a component's behavior or structure on the global behavior and its properties.

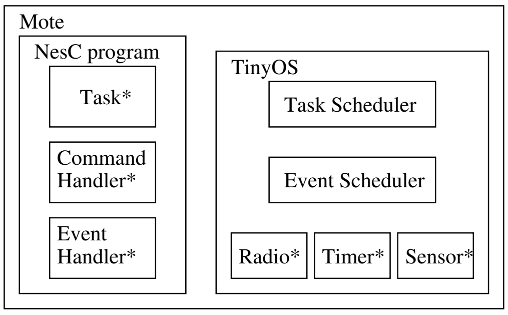


Figure 5.2: Architecture of a Mote.

This work makes the following three main contributions.

- It provides a methodology for building global and faithful models for heterogeneous networked systems.
- It allows a better understanding of the interplay between platform-dependent and platform-independent features. The model of a node is the composition of an abstract machine modeling TinyOS, and a system-oriented model of its application software.
- It provides a single framework supporting both behavioral verification and simulation of networked systems. A comparison on common benchmarks with state-of-the-art simulation environments, shows that this is possible without significant performance degradation.

The presentation of the results is structured as follows. An informal presentation of nesC and its semantics is given in section 5.1.1.2. Section 5.1.1.3 describes the modeling principle for nesC programs. Section 5.1.1.4 describes the modeling principle for TinyOS. The global model construction is explained in section 5.1.1.5, as the composition between application and TinyOS components. We present experimental results for three examples in section 5.1.1.6 and conclude in section 5.1.1.7.

5.1.1.2 The nesC programming model – informal semantics

We briefly present nesC, an extension of C used to develop TinyOS applications [GLvB+03].

nesC applications are built by writing and assembling *components*, representing either software (e.g., a protocol layer) or hardware (e.g., radio devices, timers, sensors). Components *provide* and *use* interfaces, which are groups of services. Interfaces contain *commands* and *events*.

The providers of an interface implement the commands (by means of *command handlers*), while the users implement the events (by means of *event handlers*). This distinction between commands and events within the same interface, allows to properly implement the so-called *split phase* mechanism: the execution of a non atomic operation (e.g., sending a packet) is split into two distinct phases, a command call to request the operation, and an event reception indicating its termination.

It is also possible to use deferred computation mechanisms called *tasks*. A nesC application is therefore written in C code, extended with a few extra primitives, i.e., *call* a command, *signal* an event, and *post* a task.

There are two types of components in nesC: *modules* and *configurations*. Modules provide application code, implementing one or more interfaces. Configurations are used to wire components together. Note that the wiring relation between components is not point to point. In particular, a command call performed by a component can be bound to several Command handlers provided by other components. After a call, the caller waits for completion of *all* the activated callees. Return values are then merged by using a combination function. Event signaling by software components is handled in a similar manner.

Execution of nesC applications is handled by a two-level TinyOS scheduler.

The first level manages task execution, for background computations. The TinyOS scheduler follows a strict FIFO policy for tasks: pending tasks are stored in a FIFO queue, and a task cannot be preempted by another task. Posting a task is a non-blocking operation that returns immediately. A return value indicates either a successful or an unsuccessful post operation (e.g., when the task queue is full).

The second scheduling level is used for event execution. Events represent either hardware interrupts, or indicate the completion of a given requested service. Execution of an event handler is *preemptive*: when an event is received, its corresponding event handler(s) is/are immediately activated, interrupting the current computation (which could be either a task, or another event handler). The suspended execution will resume at the end of event handler execution. Note that this policy may lead to code re-entrance (e.g., when an instance of an event handler preempts another instance of the same event handler).

Sections 5.1.1.3, 5.1.1.4, 5.1.1.5 present three steps for the construction of a global sensor network model in BIP: 1) generation of BIP components from user-defined nesC components, 2) instantiation of predefined BIP components modeling TinyOS, radio and sensors and 3) composition of these components by using connectors modeling communication links.

5.1.1.3 Modeling user-defined nesC components

We use a translator that takes annotated non re-entrant nesC code as input and generates the corresponding BIP components and connectors. Annotations are used to extract the structure characterized by the set of atomic components and the connectors between them. The modeling of the behavior of the atomic components is left to the user.

The method consists in transforming implementations of the Commands, Events and Tasks in a nesC program into atomic BIP components representing Command handlers, Event handlers, and Task handlers, respectively. The non re-entrancy limitation can be overcome by using richer models in BIP. It is possible to detect re-entrance in BIP models by using verification tools.

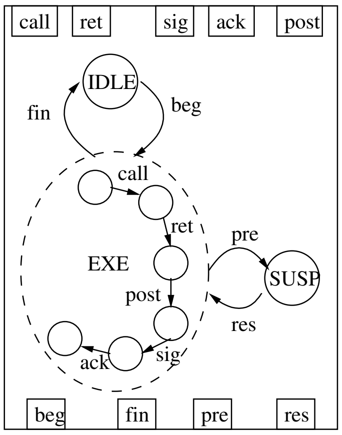


Figure 5.3: A nesC module in BIP.

A generic BIP model for atomic components is shown in figure 5.3. The interface consists of a set of ports with associated types. The behavior is specified by the control states *IDLE*, *SUSP* and *EXE* with transitions between them labeled by ports corresponding to respective actions. *EXE* is a macro state and is further decomposed into states and transitions depending on the specific behavior of the particular component.

The ports are classified in two groups:

- The first consists of the ports *beg, fin, pre* and *res* labeling the transitions for beginning, finishing, preempting and resuming execution of a component. These ports may be used in interactions between the component and TinyOS or in interactions implementing call/return mechanisms for Command handlers. They are *incomplete* as they require triggering from other components.

- The second consists of the ports *call, ret, sig, ack, post* labeling the transitions for call and return of commands, signaling and acknowledgment of events and posting of tasks. The ports *call* and *sig* are of type *complete* as they are triggers of broadcast connectors.

A generated component also contains, in addition to specific local variables, generic variables representing its unique identifier (*ID*), the identifier of a callee (*id*) and the identifier of a posted Task (*t*).

5.1.1.4 Modeling TinyOS in BIP

Our TinyOS model is the composition of two sets of components: 1) schedulers for Events and Tasks, 2) models for hardware components representing Timers, Sensors and Radio.

5.1.1.4.1 Scheduler modeling We use two schedulers to model the two-level scheduling mechanism of TinyOS.

The *Event Scheduler* (figure 5.4(a)) is responsible for the management of events generated by hardware components. When a hardware-generated event *e* is received through the port `sig`, the scheduler first preempts any running component by synchronizing through the port `pre` and stacks the *id*'s of the preempted components received . Then, it triggers the execution of the Event handlers identified by *e* by broadcasting *e* through the port `beg`. From state *BUSY1*, the *Event Scheduler* can either be triggered by a new hardware generated signal (port `sig`), or by a finish notification (port `fin`). In the first case, it preempts the currently running component, in the second case, depending on the state of the stack (empty or not), it goes to *IDLE* or to *BUSY2* from which it resumes the last preempted component.

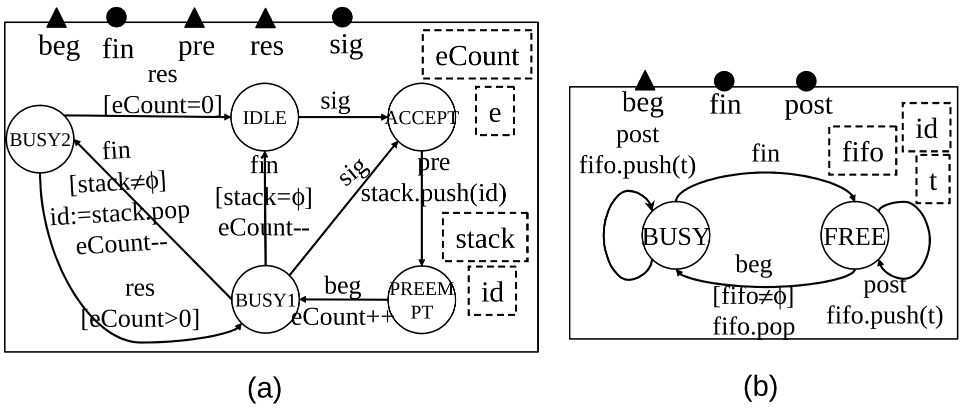


Figure 5.4: Event(a) and Task(b) Schedulers.

The *Task Scheduler* (figure 5.4(b)) is responsible for the scheduling of tasks. It treats the tasks in FIFO order and waits for a task to finish before starting a new one. It has two states: *FREE* and *BUSY*, depending on whether a task is executing or not. In any of these states, it can synchronize through its port *post* to receive new task postings. In the *BUSY* state, it waits for the currently executing task to finish and goes back to the *FREE* state. It can start a new task only if the *Event Scheduler* is *IDLE*.

5.1.1.4.2 Radio Controller Each node has a radio controller composed of a *Radio Sender* (figure 5.5(a)) and a *Radio Receiver* (figure 5.5(b)). We consider a packet level radio model where packet sending is an atomic operation. Sending a packet is a split-phase mechanism modeled by the Command handler *send* and the Event handler *sendDone*. The *send* Command handler is called from the application, and is a request to send a packet through the radio. It synchronizes with the *Radio Sender* through the *sync_send* port which passes the packet to the *Radio Sender*. Then, the *Radio Sender* broadcasts the packet. This is followed by triggering the Event handler *sendDone*.

The *Radio Receiver* receives a packet through the *listen* port, and then, it triggers the Event handler *receive*.

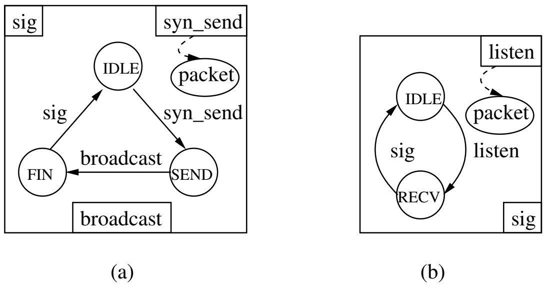


Figure 5.5: Radio controller components.

5.1.1.4.3 Timers and sensors A Timer component is a simple BIP component with a single state and two transitions. One transition is labeled by port *sig* to signal an expiration event. The other is labeled by a special port *tick* and is used to count time steps. To ensure time consistency, the *tick* ports of all the Timers are incomplete and strongly synchronized by using a single connector.

In nesC, Sensors are hardware modules offering interfaces for split-phase operation. The BIP description consists of a model for the Sensor itself, along with the models for the Command handler *getData* and Event handler *dataReady*. The actual value read by the Sensor component can be either a random value or a value provided by a model of the environment. The latter can also be explicitly modeled in BIP.

5.1.1.5 Modeling interaction between the components - the global architecture

In this section we describe the composition of the BIP components using connectors, to build the model of a node as well as the model of the network by specifying interactions between the nodes.

5.1.1.5.1 Interactions in a node We explain the principles of construction of BIP model for nodes by using two sets of connectors.

The first set models interactions for *call* statements and *signal* statements issued by software. A typical *call* statement will generate a *Call* connector and a set of $Return_i$ connectors as shown in figure 5.6.

The *Call* connector is a *broadcast* connecting the *call* port of the caller (c) to the *beg* ports of the possible callees (p, q, r). The component c may call either p and q jointly leading to the interaction (`c.call, p.beg, q.beg`), or call r leading to the interaction (`c.call, r.beg`).

The selection of interactions is by using activation conditions involving comparisons between callee identifiers (ID) and the calling identifier (id).

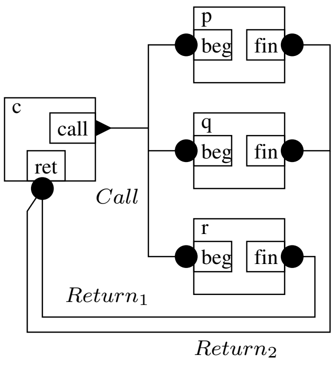


Figure 5.6: BIP connectors for a nesC call command.

The $Return_i$ connectors synchronize the *fin* ports of the callees to the *ret* port of the caller.

The *signal* statements representing software event signalling are handled exactly in the same manner as the *call* statements explained above. However, signals representing hardware events are treated separately and are processed by the event scheduler.

The second set of connectors deal with interactions between BIP components for the application and BIP components for TinyOS (see figure 5.7).

The connectors *TBegin* and *EBegin* deal respectively with interactions between Tasks handlers/Task Scheduler and Event handlers/Event Scheduler. The connectors $TFinish_i$ and $EFinish_i$ are used by Tasks and Event handlers to notify their completion. The *Preempt* connector triggers preemption of the application components. The *Resume* connector is used to

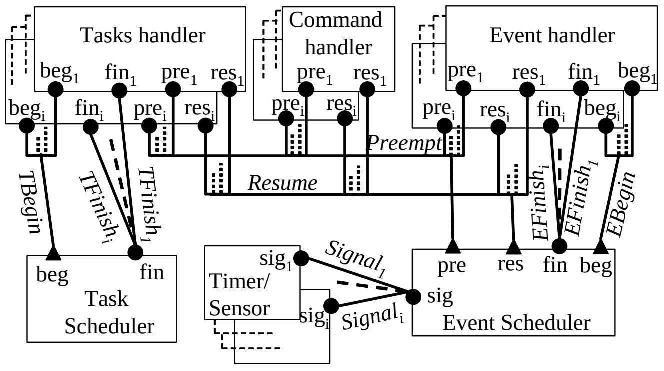


Figure 5.7: The global architecture in BIP.

resume execution of the last suspended component. The connectors *Signal*$_i$ are used to signal any hardware-generated events.

Task posting is through connectors between the port *post* of the Task Scheduler and the ports *post* of software components (not shown in the figure).

5.1.1.5.2 Interactions between nodes - Radio Links Radio links are modelled as BIP connectors linking the ports *broadcast* and *listen* of the radio controller. We consider networks with static topology and use only one connector per *broadcast* port. This connector links the *broadcast* port with all the receivers, through their *listen* port. For each connector, activation conditions depending on the distance between sender and receiver are used to define the feasible interactions. More complex activation conditions allow modelling lossy links.

5.1.1.6 Experimental results

We consider 3 examples: *BlinkTask*, *SenseToLeds* and *SenderReceiver*.

The first example illustrates the utilization of verification techniques. The two others compare our method to specific state-of-the-art simulation methods. One would expect that the use of a general purpose modeling technique instead of a specific one, well-tuned for a particular execution platform, would have a strongly negative impact on performance. Furthermore, the use of rich (non-deterministic) models instead of deterministic ones, could also have a similar effect. Experimental results show no significant performance degradation.

BlinkTask[Tin] describes a node with a variable *state* representing the state of its LED. This variable is shared between the Task *processing*, which reads it, and the Event handler *Timer.fired()*, which modifies it. For *BlinkTask* we generated a timed BIP model with 4 user-defined atomic components, 3 TinyOS components (2 schedulers and 1 Timer) and 11 connectors. Exhaustive state space exploration allows detecting error states where a new timer interrupt arrives while the Task *processing* is still being executed. Traces leading to such error states can be obtained by modeling an *Observer* component in BIP, keeping track of the sequence of interactions of the node. As an example, the analyzed state graph has 28,701 states and 46,197 transitions for the following execution time intervals: *Timer* period $[50, 50]$, *Timer.fired()* $[2, 9]$, *Leds.redOn()* $[2, 7]$, *Leds.redOff()* $[2, 7]$, *processing()* $[20, 32]$. The selected values ensure a correct behavior of the example. However, changing the timer period to values less than $[48, 48]$ leads to error states as detected by the *observer*.

The second example is *SenseToLeds*[Tin] which is a node sampling data from a photo Sensor and displaying them in the LEDs. Its nesC code consists of 4 components. The translation to BIP produces 8 user-defined components, 4 TinyOS components (2 schedulers, 1 Timer and 1 Sensor), and 21 connectors.

We consider a network of *SenseToLeds* nodes without radio links. We show in figure 5.8, simulation times as a function of the number of nodes for a virtual run time of 300 seconds, considering a 4 Hz timer on each node. We performed the tests on an AMD Athlon XP 2800+, 1Gb of RAM running GNU/Linux. The execution time for the network increased linearly with the number of nodes, as expected.

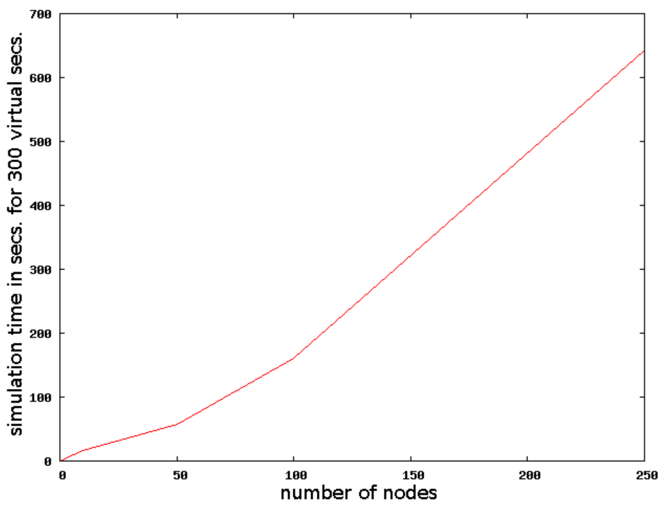


Figure 5.8: *SenseToLeds* example.

The third example *SenderReceiver* is a network of senders and receivers, with lossless channels and static topology. Each sender is connected to a fixed number of receivers y. Each receiver has a unique sender (no collision). The sender nodes execute the *CntToLedsAndRfm*[Tin] nesC program, and the receiver nodes execute the *RfmToLeds*[Tin] program. Figure 5.9 shows real execution times for 300 virtual seconds considering a 4 Hz timer on each node, as a function of the number of senders x and the number of receivers per sender y.

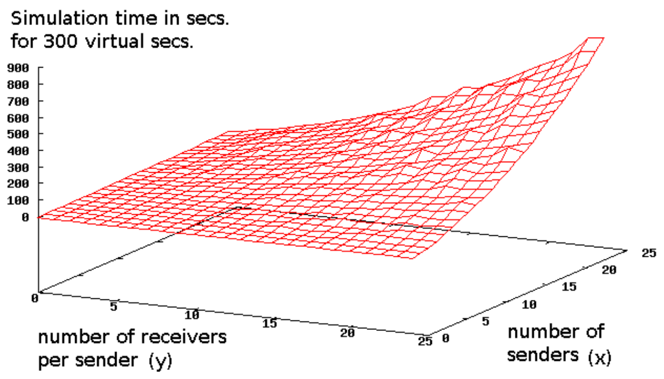


Figure 5.9: *SenderReceiver* example.

5.1.1.7 Remarks

Currently, validation of complex heterogeneous systems: such as networked i systems, is carried out by simulation or testing of prototype implementations. Verification techniques such as model-checking and static analysis are already successfully used for software or hardware. They could be extended to heterogeneous systems, provided that we have methods for building executable models for these systems.

We have applied to TinyOS, a methodology for modeling and verification of networked systems. The methodology is based on the use of the BIP component framework which encompasses description of heterogeneous real-time systems. It allows the construction of global models obtained as the composition of models of nodes. These are obtained by composition of models of the application software and of the execution platform.

The methodology is general and can be applied to building global models of heterogeneous systems. It consists in modeling the execution platform as an abstract machine driving the execution of the application software. For this, a formalization of the language in which application software is written must be provided, in terms of the primitives offered by the platform. This is certainly not an easy task. The formalization should be made at the right abstraction level. Computation granularity should be chosen so as to include in the model all the events which are relevant for the properties to be verified. Furthermore, to keep model complexity low, it should ignore computation sequences not involving such events. For instance, for the verification of synchronization and resource properties, it should assemble atomic sequences of code. The model generation methodology applied to nesC, can be adapted to any language used for programming applications. Its parser can be adequately engineered to identify in the source code, constructs generating relevant events and determine computation granularity. This can be used for (compositionally) generating BIP code.

We spent two man$\times$months for developing the methodology for TinyOS. For other platforms, much more effort would be needed for feature componentization at the right abstraction level. Such an investment seems to be the only way for overcoming current limitations of model-based design and for designing systems of guaranteed quality.

5.2 Software Componentization

5.2.1 MPEG encoder

We present the description of a target application, an MPEG4 video encoder, adopted from [CFSS08]. The video encoder architecture is shown in figure 5.10. The considered application corresponds to a video-phone application. It captures a sequence of frames with a camera, transmits the sequence, and then displays the frames on a screen. From a captured frame, the video encoder produces a corresponding bit-stream. The latter is transmitted to a video decoder which decodes the bit-stream and displays decoded frame on a screen. This architecture uses input and output buffers of the

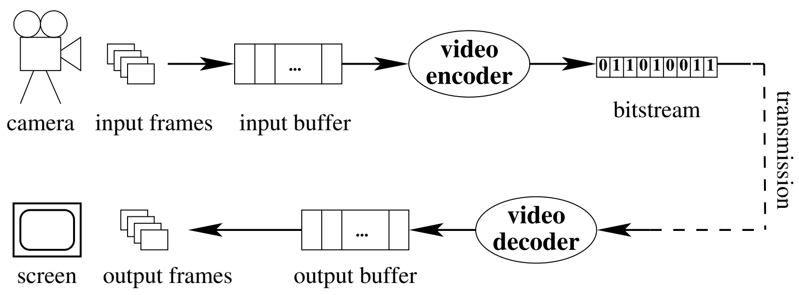


Figure 5.10: Video encoder architecture.

same size K, to cope with changes of load and avoid as much as possible frame skips. These may happen when the input buffer is full.

The initial monolithic MPEG4 encoder is written in C, containing 12000 lines of code. The encoder treats frames cyclically. A frame is captured by the block `GrabPicture`, encoded by the encoder core, and the encoded frame is produced by `OutputPicture`.

In the encoder core, a frame is split into macro-blocks, each of size 256 pixels. Given the height (H pixel) and width (W pixel) of a frame, the generated number of macro-blocks (N) is $(HxW)/256$. The generation of macro-block is done by the module `GrabMacroBlock`. The macro-blocks then pass through a number of modules, and finally get integrated back to a frame by the `Reconstruct` block. The final encoded frame is generated by the `OutputPicture` block. The precedence graph corresponding to the

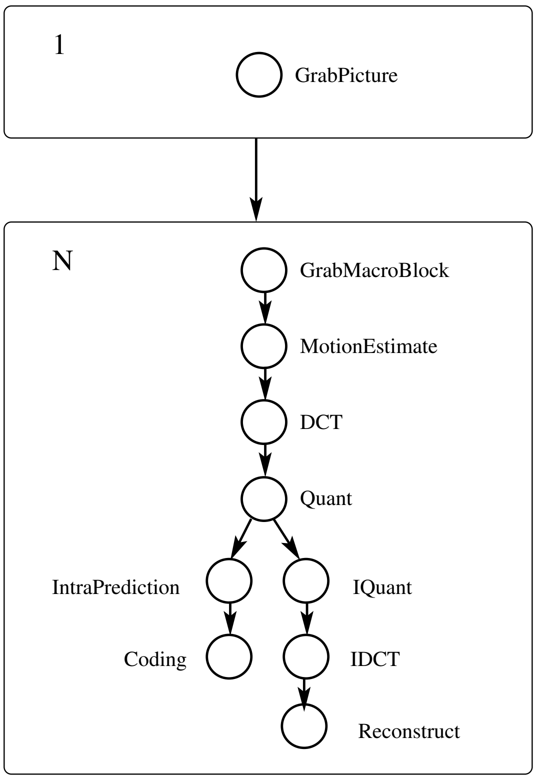


Figure 5.11: Precedence graph of the video encoder.

treatment of a frame is shown in figure 5.11. It is composed of the first action `GrabPicture` – followed by N iterations of the same precedence graph. We have a precedence constraint between two consecutive iterations of the same node of this precedence graph. For instance, the i^{th} iteration of `DCT` must be scheduled before its $i+1^{\text{th}}$ iteration. However, all iterations of `MotionEstimate` can be scheduled before the first iteration of `DCT`.

The goal of this exercise is to faithfully model the encoder in BIP. The fist step is to identifying the functional blocks from the monolithic C code, and determine the BIP components. We do a top-down approach to identify the encoder blocks. As shown in figure 5.12, we identified three main components, `GrabPicture` to grab the frames; `Encode`, the encoder core which operates on frames and encodes them; and the `OutputPicture` component to output the encoded frames. As the encoding is done one frame at a time, the components are connected serially with connectors. The data

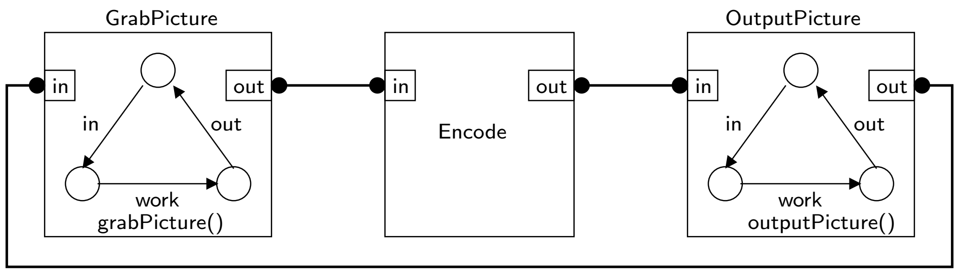


Figure 5.12: Encoder Core.

transfer associated with these connectors perform the transfer of a frame between the respective components. The connector from the output (*out*) of `OutputPicture` to the input (*in*) of `GrabPicture` denotes the event of completion of encoding of a frame, allowing `GrabPicture` to process the next frame.

The block `Encode` performs the core operation of encoding a frame. By analyzing the C code, we obtains the functional blocks and their data dependencies, as already specified in the precedence graph of figure 5.11. The corresponding BIP architecture is shown in figure 5.13. It consists of nine functional blocks, each modeled as an atom. In addition, there are buffer components (not shown in the figure). The `GrabMacroBlock` and `Reconstruct` acts as the source and the sink respectively, in processing the macro-blocks. The behavior of `GrabMacroBlock` is depicted in figure 5.14. It has a counter c, and a parameter MAX set to the number of macro-blocks (N). Upon receiving a frame, it splits it into MAX number of macro-blocks, achieved by calling the method `GrabMacroBlock()` iteratively N times. The other functional blocks of `Encode`, namely `MotionEstimate`, `DCT`, `Quant` etc., have a similar behavior. The BIP model for `MotionEstimate` is shown in figure 5.15. It has three control states, and transitions labeled by ports *in*, *work* and *out*. *in* and *out* are ports of the interface, while *work* is a internal action. It gets a macro-block by an interaction through *in*, processes it internally by the action *work*, when it calls the routine `MotionEstimate()`. The processed macro-block is despatched to the successor functional block on synchronizing through the port *out*. The behavior of the other functional blocks are the same, each calling their respective routines on executing the `work` action.

The `Reconstruct` block acts as the sink, gathering the macro-blocks and re-generating the frame. The BIP model is shown in figure 5.16. The method `Reconstruct()` is called iteratively upon receiving each macro-block. Upon integrating N macro-blocks, the reconstructed frame is passed to the output of the `Encode` compound. The data-path is implemented by connectors between the atoms, as shown in the architecture of figure 5.13. We tried two different implementations for modeling the data-path:

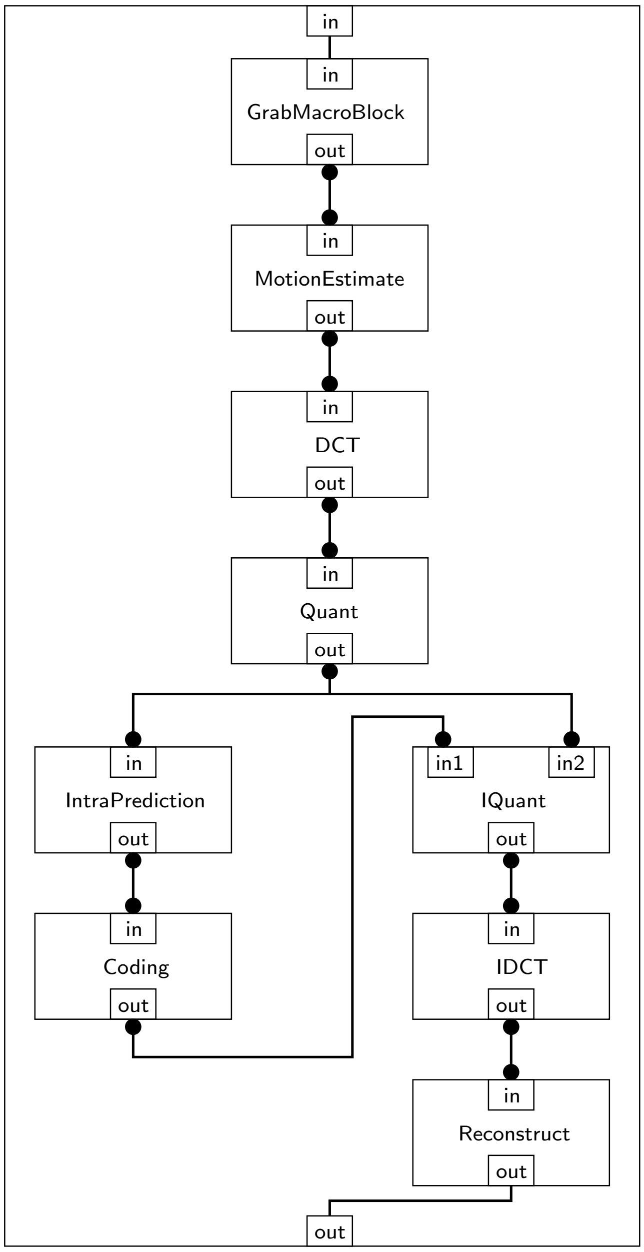


Figure 5.13: Encoder Architecture.

- components connected with buffers in between, with the buffer size large enough to hold N macro-blocks. A component reads a macro-block from its input buffer, performs the computation, and puts the resulting macro-block in the output buffer. The buffers offer interface for writing and reading in an exclusive manner. The complete description contains 20 atomic components (11 functional blocks, 9 buffers)

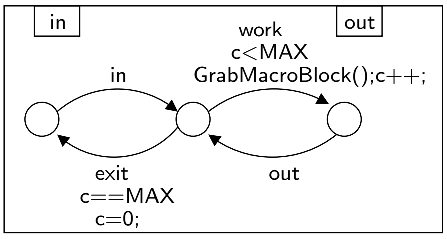


Figure 5.14: Grab-Macroblock Component.

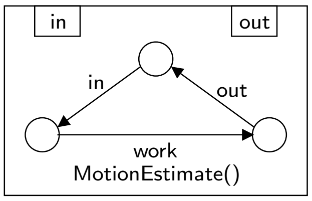


Figure 5.15: Motion-Estimation Component.

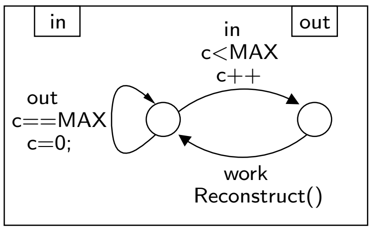


Figure 5.16: Reconstruction Component.

and 11 connectors. This model characterizes the set of all possible scheduling of the computations.

- components connected directly by connectors without using buffer, and using priorities to resolve the non-determinism and correct scheduling. The connectors in `Encode`, which are closer to the final output are assigned higher priority compared to those which lay ahead of them. This enforces a run-to-completion order of execution of the connectors in the `Encode` block. This model contains 11 atomic components, 11 connectors and 27 priority rules.

Performance Results

The BIP description is about 500 lines of code. The computation in the components are done by calling the respective functions, provided by the initial encoder library. The code generated from the BIP description is linked with the library of encoder functions to obtain the executable. The memory foot-print of the BIP executable is 288 Kilo Bytes, as compared to 172 Kilo

Bytes of the monolithic encoder binary. Note that the BIP executable also contains the execution engine integrated with it. The comparison of the run-time of the BIP executable with respect to the original encoder binary showed an overhead of 50% for the BIP executable. The overheads can be broadly classified in the following two category:

- Communication overhead: due to the two-phase execution of the atoms and the engine, with the atoms communicating to the engine by function calls, in order to synchronize with each other. This overhead can me minimized by flattening the componentized model by source level transformations, resulting in a single atomic component.

- Resolving non-determinism: this is due to the fact that the engine needs to evaluate all the feasible interactions, before selecting to execute one. An optimization for this overhead has been tried by selecting feasible connectors based on a given priority order.
  The problem of reducing overheads due to componentization is still an open one, and is currently being studied in the framework of a Ph.D thesis at Verimag.

Part IV

Conclusions and Perspectives

Chapter 6

Conclusion

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

6.1 Objectives of the thesis

We have presented the BIP component framework, that shares features with existing ones for heterogeneous components, such as [BWH$^+$03, EJL$^+$03, BGK$^+$06, Arb05]. A common key idea is to encompass high-level structuring concepts and mechanisms. BIP offers interaction-based and control-based mechanisms for component integration. The two types of mechanisms correspond to *cooperation* and *competition*, two complementary fundamental concepts for system organization.

BIP is based on the notions that a system can be obtained as the composition of three fundamental layers, behavior, interaction and priority. Behavior is represented by atomic components. Interactions are a combination of two protocols, namely rendezvous and broadcast, and we have shown that this is sufficient for expressing any kind of interaction mechanism. Finally, priorities represent elementary controllers, necessary for scheduling.

The presented methodology encompass heterogeneity, a key issue for building systems from components with different characteristics. We have discussed the sources of heterogeneity, arising due to differences in interaction and execution mechanisms. We have provided a classification of heterogeneous systems e.g.,untimed/timed, asynchronous/synchronous, and shown that they can be unified through transformations in the construction space. They allow a deeper understanding of the relations between existing semantic frameworks in terms of elementary behavioral and architectural transformations.

We have discussed that constructivity is necessary for building complex systems from components and glue with known properties, and have provided formalization for its requirements, namely incrementality, compositionality and composability.

We have also shown that the BIP glue, consisting of the interactions and priority rules, is as expressive as the universal glue. BIP is based on a minimal set of primitives for the representation of any kind of system. It provides a mechanism of clear separation of concern regarding behavior and interaction. Global system properties can be achieved by adding separately behavior (as atomic components), creating interaction, specifying restriction between them, and any combination of the above three choices.

BIP characterizes systems as points in a three-dimensional space: $Behavior \times Interaction \times Priority$. Elements of the $Interaction \times Priority$ space characterize the overall *architecture*. Each dimension, can be equipped with an adequate partial order, e.g., refinement for behavior, inclusion of interactions, inclusion of priorities. Separation of concerns is essential for defining a component's construction process as the superposition of elementary transformations along each dimension.

We have provided a system construction methodology, leading to componentization of non trivial systems [CFSS08, BBG$^+$08]. It consists of first identifying the atomic components, determining the basic functionality, defining an architecture hierarchy of composite components and their inclusion relations, and finally defining the glue for building the composite components from the lower level. This has been described in a case study [BMP$^+$07a, BMP$^+$07b]. We have described how a global model of a mote can be obtained after identifying the atomic components of the system, i.e., components for the nesC and those for TinyOS. It considers modeling the execution platform as an abstract machine driving the execution of the application software. The model generation methodology applied to nesC can be adapted for any language used for programming applications.

However, componentization has its costs. As we have shown in some examples, having an explicit engine for driving the semantics has its overheads. We gain analyzability at the cost of increased execution times. However, we now understand that source to source transformations can be performed for generation of efficient implementation for execution purposes.

6.2 Tasks Accomplished

We have came up with a tool-chain and the associated software infrastructure realizing the BIP component framework. A language has been proposed for describing systems in BIP. The tool provides a frontend parser for analysis, generation of an intermediate model, and code generation facilities for execution or enumerative exploration. A full-fledged meta-model for BIP based on EMF [EMF] has been developed. This leads to its seamless integration with tools and model-based technologies available for Java [Jav] under the Eclipse platform. The execution is driven by an engine, which directly implements the operational semantics of BIP. Provisions have been made for connecting the engine with analysis and model-checking tools. We have provided three different implementations for the BIP engine:1) A *centralized enumerative* engine, 2) A *centralized symbolic* engine, and 3) A *distributed* engine.

We have provided evidence through examples treated in BIP, that the combination of interactions and priorities allow enhanced modularity and direct modeling of schedulers. The first set of examples that were modeled and analyzed using the BIP tool-set are presented in section 2.4.1 [BBS06]. We treated some performance evaluation problems consisting of timed tasks, processing events from bursty event generator. The analyzable models generated from the BIP description by the tool-set allows for estimating end to end delays of the events. We modeled different cases with tasks running on dedicated resources as well as on shared resources with preemption. The examples show that different sets of BIP glue can be applied in enforcing protocols and scheduling policies.

In section 5.1.1 [BMP$^+$07a, BMP$^+$07b], the BIP tool-set has been used for the modeling and analysis of TinyOS based wireless sensor network applications. The methodology consists in building the model of a mote as a composition of a model extracted from a nesC program describing the application, and models of TinyOS components. Models of networks are obtained by composition of models for nodes by using connectors implementing different types of radio channels.

Another practical application of the BIP framework is in the construction and verification of a robotic system [BBG$^+$08]. Here we present a methodology for modeling the functional level of an autonomous robot in BIP. The code generated by the tool-set along with the BIP engine provides an automatic synthesis of the execution controller for the robot. The BIP model also offers validation techniques for checking essential "safety" properties.

6.3 Future Work

In this section, we discuss about prospective work directions which are either currently being undertaken, or will be planned for future integration into BIP.

6.3.1 Language Factory

We have described the system construction methodology in BIP, leading to componentization of non trivial systems. This technique has been tested for modeling wireless sensor network applications in nesC [BMP$^+$07a, BMP$^+$07b]. Similar translations have also been made from AADL to BIP [CRBS08], and FXML to BIP [IR07]. We are currently working on translators for BPEL [BPE] and Lustre [Lus].

The translation principle can be generalized for different programming applications meant for different domains. The goal would be to have a library of translators from different domains to BIP. For each domain, we have a language for describing applications, and some underlying platform driving the operational semantics of the language. We need to identify the basic atomic components necessary for the application and provide frontend tools for translating the applications into their corresponding BIP components. We would also need to faithfully integrate the semantics of the underlying platform in the BIP model using specific BIP glues. The obtained global BIP model of the system leads to enhanced analyzability.

6.3.2 Implementation for real platforms

6.3.2.1 Fully Distributed Implementation

We have proposed a distributed implementation for BIP, but it still has a single centralized engine. We plan to have distributed implementation with several engines, where each engine co-ordinates the execution of a group of connectors. For this, we need to understand how to deal with priorities between the interactions, in addition to handling conflicts arising due to local choices offered by the components.

Interesting problems to study would be to identify the number of engines needed for the most efficient implementation, partitioning of the connectors and their deployment into the engines, and optimizations for minimizing the number of exchanged messages. The distributed implementation can suitably take advantage of the current multi-core architectures, for execution.

6.3.2.2 Implementation with Real-time

Another interesting work direction currently being pursued is to allow BIP models to interact with the environment, i.e., to have open systems, with real-time facilities. In the existing system, the environment and the execution platform can be modeled, leading to simulations with logical time. The goal of this work is to allow the engine to directly handle external events and communicate with the platform, without modifying the BIP application of the system.

6.3.2.3 Static Transformations

For efficient execution, static transformation techniques are being integrated into the BIP tool-chain. Currently, we have a source-to-source transformer for converting a hierarchical system to flat system, with static composition of components producing a single atomic component, its behavior being a petri-net. The transformation reduces the overhead due to communication between the components and the engine. Other transformations include flattening of the hierarchical connectors.

Similarly, for efficient execution by the engine, static compilation of the priorities could be made to reduce the non-deterministic choices within an atomic component. Moreover, a similar technique that could be interesting includes providing meta information at model level by specifying sequences of interactions, so that non-determinism can be reduced statically by the engine.

BIP Grammar

BIP program:

bip-definition ::=
  c-definition |
  port-type-definition |
  component-type-definition |
  connector-type-definition

Definition of new port-types allow associating arbitrary data with ports. For pure event ports (i.e., without any data), a predefined port type named **Port** may be used.

port-type-definition ::=
  port type *port-type-name*
    [**(** *c-type-name var-name* { **,** *c-type-name var-name* }* **)**]

Components are classified into Atomic and Compound components. Atomic component defines the basic behavioral unit, with behavior specified either as an automata or a petri-net.

Atom:

atomic-type-definition ::=
  atomic type *atomic-type-name*
    [**(** *c-type-name fpar-name* { **,** *c-type-name fpar-name* }* **)**]
   { *data-definition* }*
   { *port-definition* }*
   { *place-definition* }*
   { *transition-definition* }*
   { **export** *port-type-name port-name* **=** *port-reference* }*
  end

data-definition ::=
  [**extern**] *c-type-name var-name* [**=** *initializer*]

port-definition ::=
  port-type-name port-name **(** [*var-name* { **,** *var-name* }*] **)**

place-definition ::=
  place *place-name* [**=** **initial**]

transition-definition ::=
  on *port-name*[1]
  from *place-name* { **,** *place-name* }* **to** *place-name* { **,** *place-name* }*
  provided *untimed-guard* **when** *timed-guard* **do** *action*

port-reference ::=
  port-name |
  (sub)component-name **.** *port-name* |
  (sub)connector-name

Compound:

compound-type-definition ::=
  compound type *compound-type-name*
      [**(** *c-type-name fpar-name* { **,** *c-type-name fpar-name* }* **)**]
    { *component-definition* }*
    { *connector-definition* }*
    { *priority-definition* }*
    { **export** *port-type-name port-name* **=** *port-reference* }*
  end

component-definition ::=
  component-type component-name **(** *actual-arg* { **,** *actual-arg* }* **)**

connector-definition ::=
  connector-type connector-name **[** *port-reference* { **,** *port-reference* }* **]**
      (*actual-arg* { **,** *actual-arg* }* **)**

Connector:

connector-type-definition ::=
  connector type *connector-type-name*

```
          ( port-type-name port-name  { , port-type-name port-name  }* )
          [ ( c-type-name fpar-name  { , c-type-name fpar-name  }* ) ]
      define  port-expression
      {  var-definition  }*
      {  on port-interaction provided guard up action down action  }*
      [ export  port-definition  ]
  end
```

```
port-expression ::=
    any-valid-AC(P)-expression-on-ports
```

```
port-interaction ::=
    port-name { , port-name }*
```

Priority:

```
priority-definition ::=
    priority priority-name
        interaction < interaction [( provided expression ) ]
```

Expssions and Actions

...

not yet defined, but similar to v1

...

Packages and Models

one package per file, with appropiate naming conventions e.g., Java-like (to be extended to components, connectors and port types...)

```
package-definition ::=
    package package-name
        {  use package-name  }*
        {  bip-definition  }*
```

one model per file

```
model-definition ::=
    model model-name
        {  use package-name  }*
        {  bip-definition  }*
```

root-component-definition

root-component-definition ::=
component-definition

Genericity

Multiple instantiation

A first extension, allows to instantiate arrays of components and of connectors in compounds...

subcomponent-array-definition ::=
component-type (sub)component-name [*dimension* { , *dimension* }*]
(*actual-arg* { , *actual-arg* }*)

subconnector-array-definition ::=
connector-type (sub)connector-name [*dimension* { , *dimension* }*]
[*port-reference* { , *port-reference* }*]
(*actual-arg* { , *actual-arg* }*)

here the actual args and port references may use predefined variables $1, $2, ... and so on up to the number of dimensions to refer to the position in the array. see the Task example below.

Ports with multiplicities

1) Allow to instantiate arrays of ports in components (including the exported ones), using a mechanism as the previous shown... Again, *var-reference***s and** *port-reference***s may use some predefined variables...**

port-array-definition ::=
port-type-name port-name [*dimension* { , *dimension* }*]
[*var-reference* { , *var-reference* }*]

2) Allow for ports with multiplicities in connectors - this will increase a lot the complexity for describing interactions allowed on the connectors, up/down functions, algebra of interactions, etc... Generalization seems quite difficult at this point.

Nevertheless, we can imagine some predefined connectors which strongly synchronize all ports of some type, e.g. ***tick*** **– these connectors should not be given explicitely but through some annotations, e.g,** ***timed***

BIP Meta Model

xxx

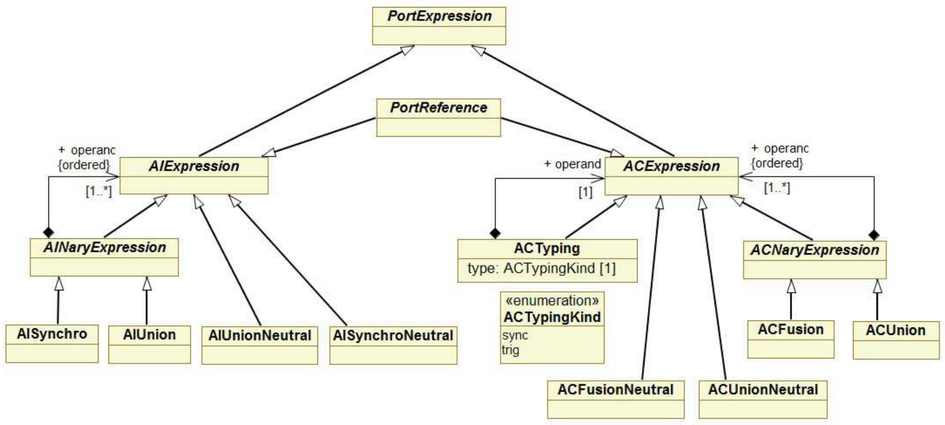


Figure 1: ...

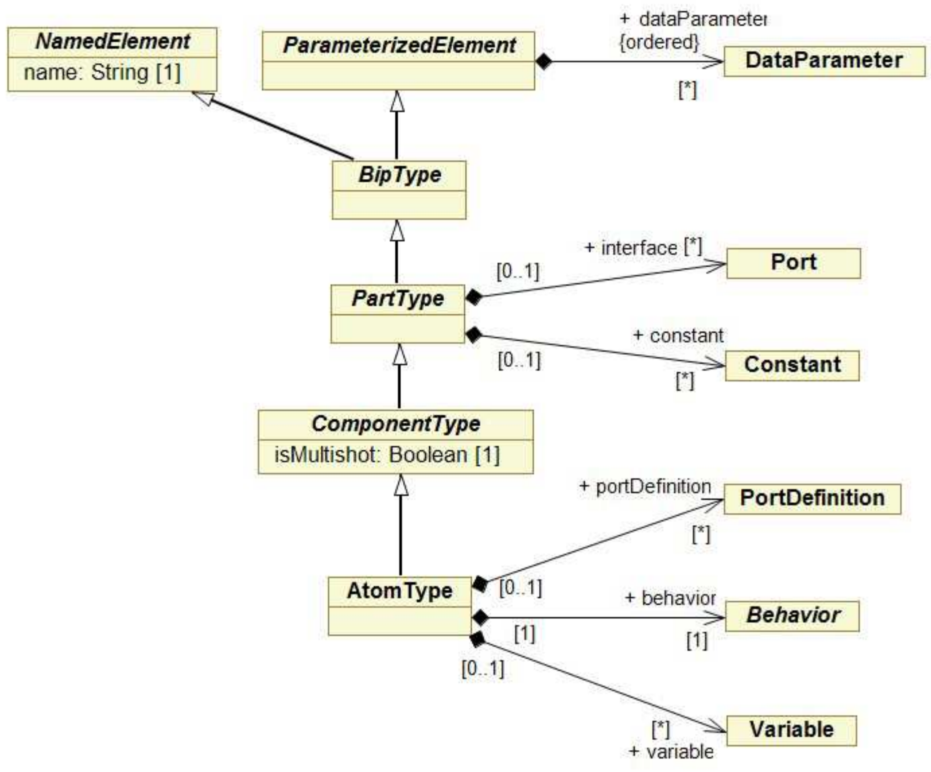


Figure 2: ...

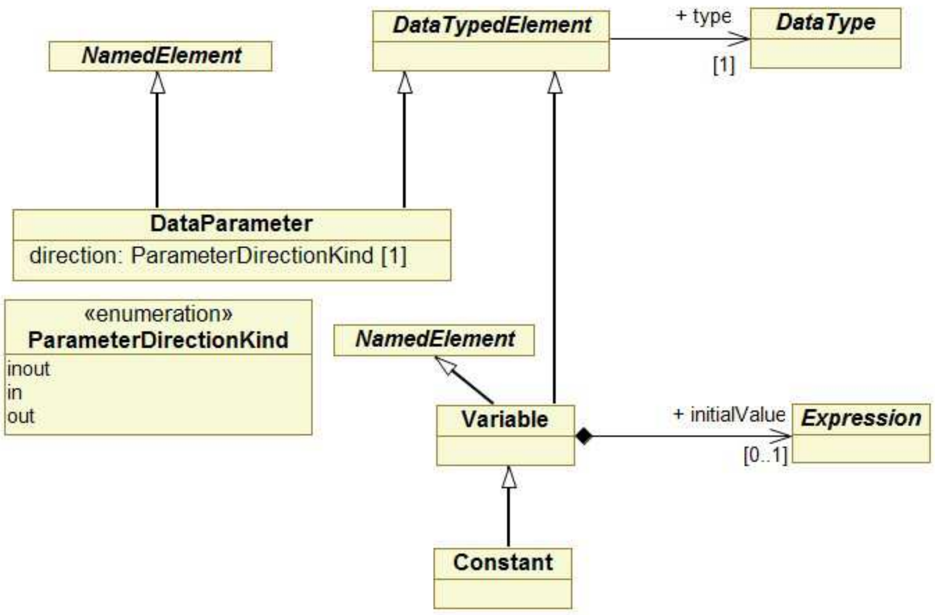


Figure 3: ...

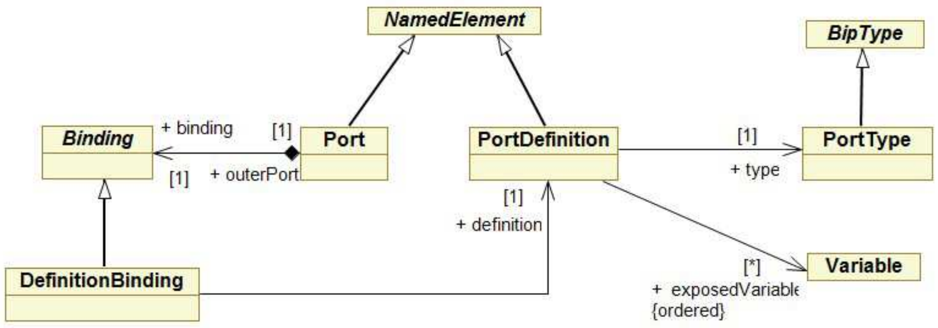

NamedElement
BipType
Binding
+ binding
[1]
Port
[1]
+ outerPort
PortDefinition
[1]
PortType
+ type
[1]
+ definition
DefinitionBinding
[*]
Variable
+ exposedVariable
{ordered}

Figure 4: ...

Bibliography

[ACH$^+$95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, *The algorithmic analysis of hybrid systems*, Theoretical Computer Science **138** (1995), no. 1, 3–34.

[ADA] *ADA*, `http://www.adahome.com/`

[AFV01] Luca Aceto, Wan Fokkink, and Chris Verhoef, *Chapter 3. Structural Operational Semantics*, Handbook of Process Algebra, Elsevier, 2001, pp. 197–292.

[AGS02] Karine Altisen, Gregor Gößler, and Joseph Sifakis, *Scheduler modeling based on the controller synthesis paradigm.*, Real-Time Systems **23** (2002), no. 1-2, 55–84.

[ANT] *ANTLR*, `http://www.antlr.org/`

[Arb05] Farhad Arbab, *Abstract behavior types: a foundation model for components and their composition.*, Sci. Comput. Program. **55** (2005), no. 1-3, 3–52.

[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.

[BBBS08] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis, *Distributed semantics and implementation for systems with interaction and priority*, FORTE, 2008, pp. 116–133.

[BBG$^+$08] Ananda Basu, Saddek Bensalem, Matthieu Gallien, Felix Ingrand, Charles Lesire, Thanh-Hung Nguyen, and Joseph Sifakis, *Incremental component-based construction and verification of a robotic system*, ECAI, Patras, Greece, 2008.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis, *Modeling heterogeneous real-time components in BIP*, 4^{th} IEEE International Conference 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*, 6th International Symposium on Automated Technology for Verification and Analysis (Seoul, South Korea), 2008.

[BC85] Gérard Berry and Laurent Cosserat, *The esterel synchronous programming language and its mathematical semantics*, Seminar on Concurrency, Carnegie-Mellon University (London, UK), Springer-Verlag, 1985, pp. 389–448.

[BCF02] Nick Benton, Luca Cardelli, and Cédric Fournet, *Modern concurrency abstractions for c#*, ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming (London, UK), Springer-Verlag, 2002, pp. 415–440.

[BFKL97] M. Bozga, J-C. Fernandez, A. Kerbrat, and L.Mounier, *Protocol verification with the aldebaran toolset*, Software Tools for Technology Transfer 1, 1997, pp. 166–183.

[BFLL04] Roberto Bruni, Jos Luiz Fiadeiro, Ivan Lanese, and Antnia Lopes, *New insights on architectural connectors*, Proceedings of IFIP TCS 2004, 3rd IFIP International Conference on Theoretical Computer Science, IFIP Conference Proceedings, Kluwer Academics, 2004, pp. 367–379.

[BGK$^+$06] K. Balasubramanian, A.S. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, *Developing applications using model-driven design environments*, IEEE Computer **39** (2006), no. 2, 33–40.

[BGO$^+$04] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis, *The* IF *toolset.*, SFM, 2004, pp. 237–267.

[BHLM02] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt, *Ptolemy: A framework for simulating and prototyping heterogenous systems*, Readings in hardware/software co-design (2002), 527–543.

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

[BMP$^+$07a] Ananda Basu, Laurent Mounier, Marc Poulhiès, Jacques Pulou, and Joseph Sifakis, *Using BIP for modeling and verification of networked systems — A case study on TinyOS-based* *networks*, Tech. Report TR-2007-5, VERIMAG, 2007, `http://www-verimag.imag.fr/index.php?page=techrep-list`.

[BMP$^+$07b] ______, *Using bip for modeling and verification of networked systems – a case study on tinyos-based networks.*, NCA, 2007, pp. 257–260.

[BPE] *Business Process Execution Language*, `http://www.service-architecture.com`

[BS00] Sébastein Bornot and Joseph Sifakis, *An algebraic framework for urgency*, Inf. Comput. **163** (2000), no. 1, 172–202.

[BS07a] Simon Bliudze and J. Sifakis, *The algebra of connectors – structuring interaction in BIP*, EmSoft, 2007, pp. 11–20.

[BS07b] Simon Bliudze and Joseph Sifakis, *The algebra of connectors — Structuring interaction in BIP*, Proceeding of the EMSOFT'07 (Salzburg, Austria), ACM SigBED, October 2007, pp. 11–20.

[BS08a] ______, *Causal semantics for the algebra of connectors (extended abstract)*, FMCO 2007 (Frank de Boer and Marcello Bonsangue, eds.), LNCS, Springer, 2008, (To appear).

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

[BST98] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis, *Modeling urgency in timed systems*, COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference (London, UK), Springer-Verlag, 1998, pp. 103–129.

[BWH$^+$03] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A.L. Sangiovanni-Vincentelli, *Metropolis: An integrated electronic system design environment*, IEEE Computer **36** (2003), no. 4, 45–52.

[CE81] E. M. Clarke and E. A. Emerson, *Synthesis of synchronization skeletons for branching time temporal logic*, Workshop on Logic of Programs (Yorktown Heights, NY, USA), 1981.

[CFSS08] Jacques Combaz, Jean-Claude Fernandez, Joseph Sifakis, and Loïc Strus, *Symbolic quality control for multimedia applications*, Real-Time Syst. **40** (2008), no. 1, 1–43.

[CL85] K. Mani Chandy and Leslie Lamport, *Distributed snapshots: Determining global states of distributed systems*, ACM Trans. Comput. Syst. **3** (1985), no. 1, 63–75.

[CRBS08] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis, *Translating AADL into BIP - Application to the Verification of Real-time Systems*, Model Based Architecting and Construction of Embedded Systems, 2008.

[ECZ06] Edward A. Lee Elaine Cheong and Yang Zhao, *Joint modeling and design of wireless networks and sensor node software*, Tech. Report UCB/EECS-2006-150, EECS Department, University of California, Berkeley, November 2006.

[EJL$^+$03] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, *Taming heterogeneity: The Ptolemy approach*, Proceedings of the IEEE **91** (2003), no. 1, 127–144.

[EMF] *Eclipse Modeling Framework Project*, `http://www.eclipse.org/modeling/emf`

[Fra] *Fractal*, `http://fractal.objectweb.org/`

[FSLM02] J.-Ph. Fassino, J.-B. Stefani, J. Lawall, and G. Muller, *THINK: A Software Framework for Component-based Operating System Kernels*, Proceedings of the Usenix Annual Technical Conference, June 2002.

[GL98] Stephen J. Garland and Nancy A. Lynch, *The ioa language and toolset: Support for designing, analyzing, and building distributed systems*, Tech. Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1998.

[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.

[Gößl01] G. Gößler, PROMETHEUS — *a compositional modeling tool for real-time systems*, Proc. Workshop RT-TOOLS 2001 (P. Pettersson and S. Yovine, eds.), Technical report 2001-014, Uppsala University, Department of Information Technology, 2001.

[GS03] Gregor Gößler and Joseph Sifakis, *Priority systems*, FMCO, 2003, pp. 314–329.

[GS04] David Garlan and Bradley R. Schmerl, *Using architectural models at runtime: Research challenges*, EWSA, 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.

[GSR$^+$04] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer, *A system for simulation, emulation and deployement of heterogeneous sensor networks*, 2nd International Conference on Embedded Networked Sensor Systems, ACM Press, 2004.

[Hen96] T. A. Henzinger, *The theory of hybrid automata*, LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (Washington, DC, USA), IEEE Computer Society, 1996, p. 278.

[HS06] T. Henzinger and J. Sifakis, *The embedded systems design challenge*, Proceedings FM 2006, LNCS (2006).

[IR07] Radu Iosif and Adam Rogalewicz, *An approach to derivation of component-based implementations from data-oriented specifications*, October 2007.

[IT99] ITU-T, *Recommendation Z.100. Specification and Description Language (SDL)*, Tech. Report Z-100, International Telecommunication Union – Standardization Sector, Genève, November 1999.

[IT00] ______, *Recommendation Z-100 Annex F1(11/00): Specification and Description Language (SDL) Formal Definition, International Telecommunication Union, Geneva*, 2000.

[Jav] *Java*, `http://www.java.com/`

[JHA$^+$96] J. -C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, *CADP: a protocol validation and verification toolbox*, Proceedings of the Eighth International Conference on Computer Aided Verification CAV (New Brunswick, NJ, USA) (Rajeev Alur and Thomas A. Henzinger, eds.), vol. 1102, Springer Verlag, 1996, pp. 437–440.

[Lee03] E. A. Lee, *Overview of the ptolemy project*, Tech. report, 2003.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler, *Tossim: accurate and scalable simulation of entire TinyOS applications*, SenSys '03: 1st International Conference on Embedded networked sensor systems (New York, NY, USA), ACM Press, 2003, pp. 126–137.

[Lus] *Lustre*, `http://www-verimag.imag.fr/~synchron/index.php?page=lang-desig`

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

[Mil98] ______, *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.

[MS00] R. Mateescu and M. Sighireanu, *Efficient on-the-fly model-checking for regular alternation-free mu-calculus*, Tech. Report 3899, INRIA Rhône-Alpes, France, 2000.

[PAP] *PAPYRUS*, `http://www.papyrusuml.org/`

[PBM$^+$04] Jonathan Polley, Dionysys Blazakis, Jonathan McGee, Dan Rusk, and John S. Baras, *ATEMU: A Fine-grained Sensor Network Simulator*, Proceedings of SECON, 2004.

[Pha94] M. Phalippou, *Executable testers*, IWPTS (Tokyo, Japan), 1994.

[Plo81] Gordon D. Plotkin, *A structural approach to operational semantics*, Tech. Report DAIMI FN-19, University of Aarhus, 1981.

[QS82] J-P. Queille and J. Sifakis, *Specification and verification of concurrent systems*, Int. Symposium on Programming (Torino, Italy), 1982.

[Qui86] M J Quinn, *Designing efficient algorithms for parallel computers*, McGraw-Hill, Inc., New York, NY, USA, 1986.

[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.

[SBS08] Thanh-Hung Nguyen Saddek Bensalem, Marius Bozga and Joseph Sifakis, *Compositional deadlock-detection and verification for component-based systems*, Tech. report, Verimag, Centre Équation, 38610 Gières, April 2008.

[Sel04] Bran Selic, *Tutorial: An overview of uml 2.0*, ICSE '04: Proceedings of the 26th International Conference on Software Engineering (Washington, DC, USA), IEEE Computer Society, 2004, pp. 741–742.

[Sif05] J. Sifakis, *A framework for component-based construction*, Proceedings of the Third International Conference on Software Engineering and Formal Methods (SEFM), IEEE Computer Society, 2005, pp. 293–300.

[Som] Fabio Somenzi, *CUDD: CU decision diagram package*, http://vlsi.colorado.edu/ fabio/CUDD/, (Release 2.4.1).

[Tin] *http://www.tinyos.net/*.

[TLP05] Ben L Titzer, Daniel K Lee, and Jens Palsberg, *Avrora: Scalable Sensor Network Simulation with Precise Timing*, IPSN 05, 2005.

[Tre94] J. Tretmans, *A formal approach to conformance testing*, IWPTS (Tokyo, Japan), 1994.

[VPL99] James Vera, Louis Perrochon, and David C. Luckham, *Event-based execution architectures for dynamic software systems*, WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1) (Deventer, The Netherlands, The Netherlands), Kluwer, B.V., 1999, pp. 303–318.

[WDE05] *Workshop on distributed embedded systems, lorentz center, leiden*, 2005, http://www.tik.ee.ethz.ch/ leiden05.