Component-Based Design of Real-Time Systems

Marius Bozga

Document d'Habilitation à Diriger des Recherches
Université Joseph Fourier, Grenoble I

Component-Based Design of Real-Time Systems

Marius Bozga

February 4, 2010

Composition du Jury :

Valérie Issarny	DR Inria Rocquencourt, rapporteur
Jean-Marc Jézéquel	Professeur à l'Université de Rennes, rapporteur
Bengt Jonsson	Professeur à l'Université de Uppsala, rapporteur
Jean-Bernard Stefani	DR Inria Grenoble Rhone Alpes, examinateur
Joseph Sifakis	DR CNRS, Verimag, exminateur

Contents

Chapter 1

Introduction

1.1 System Design Challenge

The design of large and reliable IT systems is a challenging engineering problem. Traditional engineering disciplines such as civil engineering or mechanical engineering are based on solid theory for building artifacts with predictable behaviour over their life-time. In contrast, we lack constructivity results for computing systems: computer science provides only partial answers to particular system design problems. With few exceptions, in our domain predictability is impossible to guarantee at design time and therefore, a posteriori validation remains the only means for ensuring correctness.

For example, today we master in a satisfactory way the construction of two categories of systems. On one hand, there are critical systems of low to medium complexity such as flight controllers or industrial plant controllers. For such systems, the tight relation with control theory and control engineering influence the development of synchronous languages and associated technology, with their wellknown success today. On the other hand, there are complex best-effort systems such as telecommunication systems. Here, the layered model of network protocols provides a solid technology that sustain the construction of extremely complex and functional computer networks among which Internet is probably the most remarkable example.

Nevertheless, in addition to the two above categories of systems there exist many others, less well deserved by the current design practice, but with a broad utility foreseen in the near future. In particular, the development of affordable critical systems including automotive, medical or robotic devices still rely on a large variety of ad-hoc design techniques. The same situation is observed for the integration of systems of systems e.g., the Internet of things, smart grids, ambient intelligence, etc.

In general, the design of large IT systems is facing several difficulties, all of them being derived from our limited ability to handle, represent and communicate information:

- *integration complexity*: Complex software systems are built by reusing and assembling components, that are, simpler sub-systems. Clearly, this is the only way to master the inherent complexity and to ensure the correctness of the design, and also to increase the productivity. However, system level integration becomes extremely difficult because

components are highly heterogeneous: they have different characteristics, are often developed using different technologies, and highlight different features from different viewpoints.

- *incomplete requirements*: The requirements are often incomplete and ambiguous because expressed in natural languages.

- *design approaches*: Design approaches are often empirical and based on expertise and experience of design teams. On one hand, this situation confers several advantages because people tend to solve new problems by reusing, extending and improving past solutions proven to be efficient and robust. Consequently, this favors components reuse and avoids re-inventing and re-discovering design solutions every time. On the other hand, this situation also acts like a barrier: teams are not always able to adapt in a satisfactory manner to new requirements and moreover, they tend to reject better solutions simply because they do not fit their design know-how.

For these reasons, it is not surprising that large IT projects are most likely to exceed their budget and timing while delivering poor quality results.

There are several requirements that a design methodology for complex software systems must ensure:

- *Correctness:* Avoiding design errors or at least, detect and eliminate them as early as possible is a major issue. The following requirements are a prerequisite for correct and scalable design methodologies. First, the design methodology should rely on the use of models with well-defined semantics. Second, it should rely on constructivity results allowing to infer global properties of a system from properties of its components. Third, it should lead to correct implementations by application of transformations preserving functional properties.

- *Productivity:* The design flow must allow enhanced productivity, especially for programming complex distributed applications. This can be achieved by offering programmers domain-specific languages allowing in particular natural expression of parallelism, both data or functional parallelism. The semantics of these languages must be defined through translation to a reference semantic model in order to ensure coherent and seamless integration of components developed using different heterogeneous programming models. The current design practice is fairly much more complex e.g., software design frameworks are based on interaction by method call and do not allow direct modeling of atomic interaction mechanisms. On the contrary, modeling frameworks for mixed hardware/software or control systems such as SystemC and Simulink/Stateflow have built-in mechanisms for synchronous execution, and are not adequate for describing asynchronous systems.

- *Performance:* The performance of a system is as important as its functional correctness. Resources such as memory, time and energy must be first class concepts. Moreover, it should be possible to analyze and evaluate efficiency in using resources as early as possible along the design process. Unfortunately, widely used modeling formalisms such as the Unified Modeling Language UML [RJB04], its MARTE profile [OMG08b] or the Architecture Analysis and Design Language AADL [FLV03] offer only syntactic sugar

for expressing timing constraints 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.

- *Parsimony:* The design flow should not enforce any particular programming or execution model. Designers can use degrees of freedom in the design process, e.g. parallelism or non-determinism, for choosing amongst possible implementations guided only by system requirements. Nonetheless, the most successful design methodologies nowadays privilege a unique programming model together with an associated compilation chain oriented towards a given execution model. For example, synchronous design methodologies strongly rely on synchronous programming models and targets sequential implementations on single processors. Alternatively, real-time programming based on scheduling theory for periodic tasks targets dedicated real-time multitasking platforms, etc.

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

1.1.1 Component-based Frameworks

In this section, we provide a brief description of the current state of the art of component-based 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 modern programming languages, modeling standards and middleware. However, these techniques have not yet achieved the same level of maturity as has been the case for hardware. There exists a huge body of literature dealing with components and their use for different purposes and in different contexts. The following deal, one way or another, with issues related to component-based engineering for modeling and/or programming complex software systems:

- Software Design Description Languages [GS04, BFLL04], and Architecture Description Languages focusing on non-functional aspects [VPL99, AVCL02].
- Normalized system modeling languages such as UML [OMG09], SysML [OMG08a], SDL [ITU99] and associated tools.
- Languages and notations specific to system design tools such as SystemC [Pan01, RHG+01], Metropolis [BWH+03], Ptolemy [EJL+03], GME [BGK+06], Simulink/Stateflow [Mat], Autofocus [HS01], 42 [MB07]
- Component models based on classical concepts of Component-Based Software Engineering (CBSE) like Fractal [BCS04] and its implementations, e.g., Julia, Think, etc
- Middleware standards such as Corba, Javabeans, .NET
- Software development environments such as PCTE, SWbus, Softbench, Eclipse.

- Coordination language extension of programming languages such as Javaspaces [FHA99], TSpaces [FLN+03], Polyphonic C♯ [BCF02], nesC [GLvB+03] and
- Theoretical frameworks based on process algebras e.g., the Pi-Calculus [Mil98] or based on automata e.g., [RC03].

There are significant differences 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. Exceptions are languages such as Polyphonic C♯ which offers declaration of asynchronous methods and synchronization patterns, allowing two or more methods to synchronize. Moreover, in software models, the interconnect between the components is not always easy to determine due to polymorphism and dynamic linking, in general. Furthermore, the execution of the behavior of the components is often made in the context of asynchronous threads, and components do not have any proper activity. The inability to statically determine the component interconnections and the thread of execution leads to reduced analyzability of software models.

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

1.1.2 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 [Hoa78]). 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$^\sharp$. The implementation of n-ary interactions by using binary interaction primitives is a non-trivial problem.

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 and UML, and in multi threaded programming languages such as ADA and Java. The lack of built in mechanisms for sharing computation between components can be compensated through scheduling and coordination mechanisms, e.g., priorities, locks, semaphores, etc.

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 [AD94] 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.

1.1.3 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. More precisely, we need theory for composition 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.

1.2 Our contribution

1.2.1 The BIP Component Framework

We present in this work, the BIP component framework [BBS06, Bas08]. The name BIP is derived from Behavior, Interaction and Priority, 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). E.g., synchronous execution is characterized as a combination of properties of the three layers. 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.

1.2.2 BIP-centric System Design

We propose a system design methodology based on BIP. The design flow is illustrated in figure 1.1 and its main characteristics are the following:

- The design flow is *model-based*. It relies on the BIP component framework to represent both application models, that is, pure functional models of the software as well as system models, that is implementations of the final system, where constraints and specific characteristics of the execution platform are taken into account.

- In order to increase *productivity*, we use translations from common programming models into BIP. We already define and experiment with translations from general languages (e.g, synchronous languages) as well as for domain specific languages.

- A key idea in our methodology is to generate the system models (i.e., implementations) from the application model of the software and a model of the target platform by using a set of *correct-by-construction model transformations*. These transformations preserve functional properties. Furthermore, they take into account extra-functional constraints. We propose several types of model transformations:

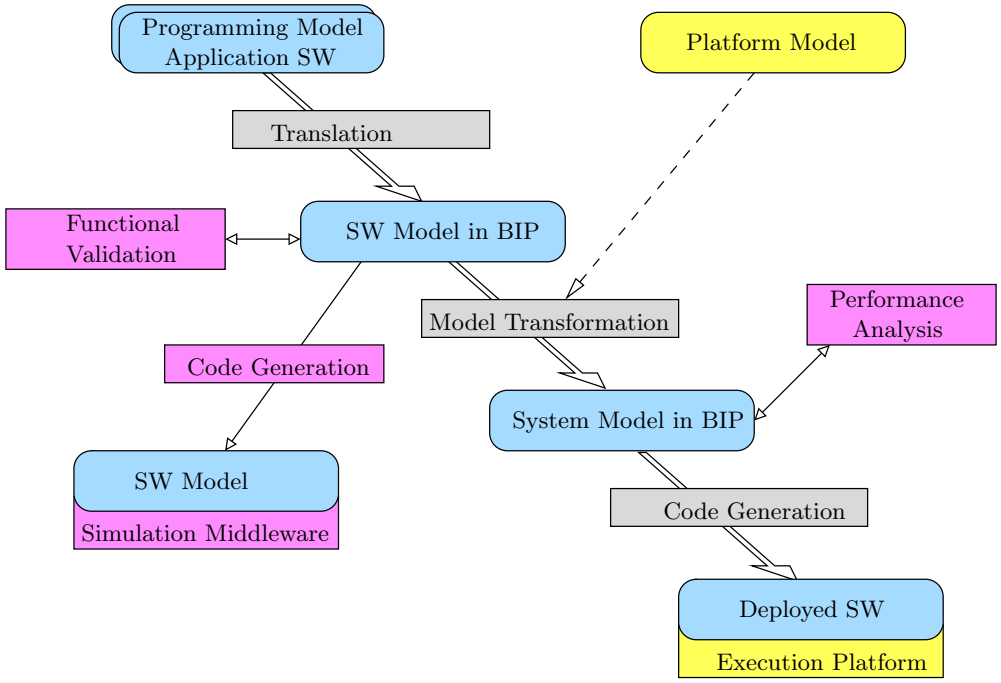


Figure 1.1: BIP-centric design flow

1. *Architecture Optimization:* These transformations take BIP models and transform them into functionally equivalent BIP models with different architectures. Such transformations have already been implemented in the BIP2BIP tool[BJS09]. They allow in particular to generate from a hierarchical model an equivalent flat model or a single atomic component by composing the behavior of the constituent components. Flat models have been proven optimal for implementation on single-processor platforms. We will study transformations for deriving optimal implementations with respect to given execution platform.
2. *Distributed Implementation:* Coordination in BIP is achieved through multiparty interactions and scheduling by using dynamic priorities. The associated semantics is defined on a global state model. This makes reasoning about systems easy, however, it is harder to obtain distributed implementations where the primitives available for communication and coordination are less powerful[BBBS08]. For example, the implementation of a multiparty interaction as an asynchronous send/receive protocol can be done: either in a decentralized manner by adding to each one of the components involved in the interaction a controller; or in a centralized manner by using a single controller coordinating the behavior of the components. In general, we plan to reuse and adapt existing distributed algorithms for realizing multiparty interactions or solving related problems. We will prove their correctness in this particular context by developing composability arguments e.g, non-interference of the algorithms with the functional behaviour of the system. Finally, we will assess their performance with respect to different criteria such as the degree of parallelism or the overhead for coordination.

3. *Memory management:* BIP adopts a private memory model which is safe for programming but may lead to inefficient implementations. The aim is to realize memory transformation from private to shared memory and conversely. We are also interested in transformations leading to mixed solutions combining private and shared memory and determining tradeoffs.

- We provide theory, methods and tools for establishing the *correctness* of application models. We favor a correct-by-construction development, for example, we provide generic constructivity results allowing to establish by construction deadlock-freedom and confluence of synchronous systems represented in BIP. Moreover, we also provide tools based on compositional and incremental methods for discovering key functional properties such as invariants. Furthermore, we provide tools based on simulation and state space exploration for debugging and tuning the application models, in specific situations.

- In order to estimate *performance* of the final system implementation, we envisage specific methods inspired by functional verification, such as compositional analysis and abstraction of timed automata models, as well as connections to well-established performance evaluation methods e.g., modular performance analysis based on analytic models or discrete event simulation. Moreover, in addition to time, the scope of performance analysis will be extended to other important resources such as memory and energy, and this will require further enrichment of the BIP component model.

1.3 Organization of the Report

In chapter 2, **System Construction**, we introduce the key concepts of component-based construction using the BIP component framework. We begin with an abstract model for components and their composition using abstract glue. This abstract model help us to formalize and reason about constructivity properties such as incrementality, compositionality and composability. Moreover, it is used to define an abstract measure of the expressive power of component based frameworks. Then, we present the concrete BIP model. We introduce the three layers – atomic behaviour, interaction models and priority models – and provide the operational semantics for composition. We close the chapter with a discussion on the BIP system construction space.

In chapter 3, **Language Factory** we provide an overview of the expressive power of BIP for modeling applications developed using different programming models. First, we illustrate how timed systems can be effectively represented using a discrete time version of BIP. Second, we consider synchronous systems. In general, for such systems there exist general constructivity results allowing to establish properties such as deadlock-freedom and confluence (determinism) of computations. We show how synchronous systems can be represented using a specialized version of BIP and how the constructivity results can be reformulated and proven in this particular setting. Finally, we close the chapter by briefly presenting how applications developed using domain specific languages can be represented in BIP. We consider the case of mixed hardware/software applications for sensor networks and the case of software controllers for autonomous robots

In chapter 4, **System Implementation**, we present the main transformations leading from

BIP application models to efficient implementations with specific architectures. As a first step, we define semantics preserving transformations allowing to flatten (partially or totally) hierarchical components and/or hierarchical interaction models. Then, we define two different methods of implementation, sequential and distributed, to be used according to the execution platform underneath. In particular, for distributed implementation, we investigate different architecture solutions, ranging from fully centralized, where a global controller is used to coordinate the execution of all interactions between atomic components, to fully decentralized, where each interaction can be executed on its own, and coordination between them is realized using a decentralized protocol. Finally, we present specific model optimization allowing to merge several atomic components into one atomic component. Such an optimization allow to reduce the coordination overhead between components at runtime and to achieve similar performance as for hand-written implementations, for particular applications.

In chapter 5, **System Validation**, we present the methods currently available for validation of (significant subsets of) BIP models: compositional generation of invariants, explicit state space exploration and model-checking, and automatic generation of timing abstractions. The first method, actually implemented in the D-Finder tool [BBNS09], allows to compute two categories of invariants, namely component invariants and interaction invariants, in a compositional and incremental manner. This method scales well in practice and the generated invariants have been proven useful to establish deadlock-freedom of large applications. For debugging purposes and/or finer grain analysis, we provide state space exploration and model-checking techniques. In fact, a significant subset of BIP is covered by the IF model-checker [VER], a state of the art toolbox for validation of distributed real-time systems. Finally, we briefly introduce the automatic generation of timing abstractions. This recent method has been experimented in the context of asynchronous circuits. It allows to generate tractable validation models following a compositional and incremental generation approach.

Chapter 2

System Construction

We describe in this chapter the main notions about components and their composition. We begin with an abstract model, where basic definitions of components and component composition are introduced. This abstract model is then used to formalize generic properties of component-based construction such as incrementality and constructivity. Then, we introduce the concrete model used within the BIP component framework. We provide an abstract syntax and operational semantics for all the key concepts including atomic and composite components, interaction and priority models. We conclude the chapter with a discussion on the BIP system construction space.

2.1 Basic Ideas

2.1.1 Components and Glue

A *component* is a behavioral entity, having a well defined interface. It denotes an executable specification whose runs can be modeled as sequences of discrete actions.

We distinguish two kinds of components: atomic and composite. Atomic components are the basic elements in the components hierarchy. Their behavior is explicitly represented as labeled transition systems.

Definition 2.1 (labeled transition system)**.**

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

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

Composite components are obtained by composing together other components (atomic or composite) using a *glue*. Their behavior is the product of behaviors of the inner components, with restrictions implied by the glue. Formally, a glue *gl* consists of a set of memory-less glue operators with the precise meaning defined by operational semantic rules. The following definition is adopted from [BS08].

Definition 2.2 (glue operator)**.**

A glue operator for behaviors $(B_i)_{i=1,n}$ is any behavior transformer defined by derivation rules of the form

$$[\textit{glue operator}]\quad \frac{\{q_i \xrightarrow{a_i}_i q_i'\}_{i\in I} \qquad \{q_i = q_i'\}_{i\notin I} \qquad \{q_k \not\xrightarrow{a_k}_k\}_{k\in K}}{(q_1,\ldots,q_n) \xrightarrow{\cup_{i\in I} a_i} (q_1',\ldots,q_n')}$$

where $I, K \subseteq \{1,\ldots n\}$, $I \neq \emptyset$ and $I \cap K = \emptyset$. That is, there is at least one positive premise and negative and positive premises are not contradictory.

Let us remark that glue operators are defined by stratified rules. They define transitions of composite components ($\rightarrow$) as a result of composition of transitions of their constituents ($\rightarrow_i$).

Our ultimate goal is to provide a methodology for component description and integration in a meaningful manner. The methodology must be incremental, i.e., components can be composed through a meaningful hierarchy of glues. Moreover, it must provide support for compositionality and composability, as follows.

2.1.2 Incrementality: Flatenning and Decomposition

We consider a component-based development framework as being incremental if it allows to (re)partition an existing component-based system into any required structure.

Incrementality can be achieved as a combination of decomposition and flattening operations. Decomposition consists in transforming a n-ary glue operator into a successive application of binary glue operators, as shown on figure 2.1. In general, we should be able to write:

$$gl(B_1,\ldots,B_n) = gl_1\left(B_1, gl_2\left(B_2,\ldots,B_n\right)\right)$$

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

$$gl_{1,n}\left(gl_{1,2}\left(B_1,B_2\right)\ldots B_n\right) = gl(B_1,\ldots,B_n)$$

2.1.3 Compositionality and Composability

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

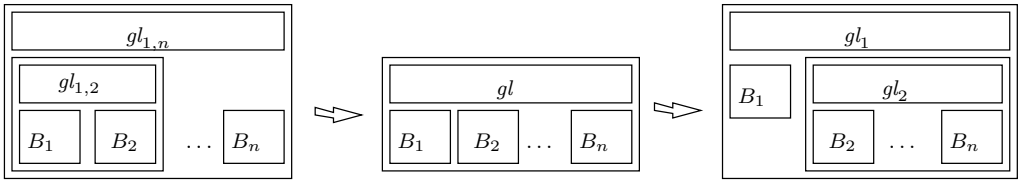


Figure 2.1: Incrementality: flatenning and decomposition

$$[\textit{Compositionality}] \quad \frac{\{B_i \models \phi_i\}_{i=1,n}}{gl(B_1, \ldots B_n) \models \widetilde{gl}(\phi_1, \ldots \phi_n)}$$

where ϕ_i is a property of the component B_i, gl is a glue composing the components, and $\widetilde{gl}$ is an operator on properties depending on the glue gl.

Composability guarantees that during the system construction process, all essential properties of subcomponents are preserved. It is defined by rules of the following form:

$$[\textit{Composability}] \quad \frac{\{gl_i(B_{i_1}, B_{i_2}, ...) \models \phi_i\}_{i=1,m}}{gl(B_1, \ldots B_n) \models \bigwedge_{i=1}^{m} \phi_i}$$

where gl_i is a glue satisfying a property ϕ_i on a subset of components $\{B_{i_1}, B_{i_2}, ...\}$; and $gl = \odot_{i=1}^{m} gl_i$ is a composition of the glues.

2.1.4 Expressivity

The paper [BS08] introduces a meaningful way to compare the expressivity of glues, that is, composition operators used in component-based frameworks.

To determine whether one glue is more expressive than another, we compare their respective sets of behaviors composable from the same atomic ones. Several approaches to comparing the expressiveness of glues can be considered according to the type of modifications of the system that one allows in order to perform the comparison. In any case, this consists in exhibiting for each operator of one glue an equivalent operator in the other one. Below, we define two criteria for the comparison of glue expressiveness:

1. *Strong expressiveness*, where the exhibited glue operator must be applied to the same set of behaviors as the original one,

2. *Weak expressiveness*, where the exhibited glue operator must be applied to the same set of behaviors as the original one, with potentially an addition of some fixed set of coordination behaviors.

2.2 The BIP Framework

The BIP component framework is a concretization of the abstract framework introduced previously. BIP considers behavior defined using extended transition systems and composition of behaviors using two kinds of glue, interactions and priorities. It is shown in [BS08] that these encompass the universal glue presented in definition 2.2, that means, BIP has the (maximal) expressive power of the universal glue. The framework is based on a 3-tier architecture as illustrated in figure 2.2, the layers being behavior, interaction and priority, where:

1. *Behavior* describes the dynamic behavior of atomic components. It consists of a set of extended transition systems. 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 define 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 and/or scheduling policies.

BIP defines mechanisms for composition of *behavior* using the *interaction* and *priority* glues. In the following sections, we give a formal description of each of the layers, introduced here.

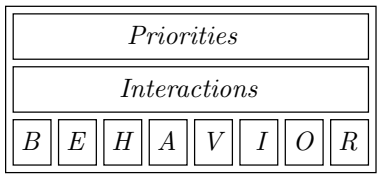


Figure 2.2: BIP 3-tier Architecture

2.2.1 Ports and Interfaces

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

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

A set of ports is called an *interface*.

2.2.2 Atomic Behavior

Definition 2.3 (atomic behavior)**.**

An atomic behavior B is a tuple (P, X, N) where:

- *P is a finite set of ports, the interface of the behavior,*
- *X is a finite set of variables, including the ones associated to ports in P,*
- *$N = (L, T, F)$ is an extended 1-safe Petri net:*
  - *L is a finite set of places,*
  - *T is a finite set of transitions τ labelled by (p_τ, g_τ, f_τ) where*
    - *$p_\tau \in P \cup \{\bot\}$ is the port triggered by the transition τ, if any*
    - *g_τ is the guard of τ, that is a predicate on X and*
    - *f_τ is the update function associated with τ, that is a state transformer defined on X,*
  - *$F \subseteq L \times T \cup T \times L$ is the token flow relation.*

Example 2.1.

Figure 2.3 presents examples of atomic behaviour using an intuitive and self-explanatory graphical notation.

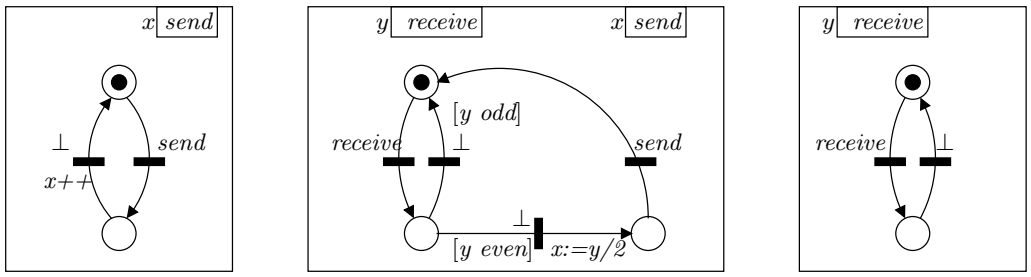


Figure 2.3: Examples of atomic behaviour

In order to define the operational semantics for atomic behavior, let us first introduce some notations. Given a Petri net $N = (L, T, F)$ we define the set of 1-safe markings $\mathcal{M}$ as the set of functions $m : L \to \{0, 1\}$. Given two markings m_1, m_2, we define inclusion $m_1 \leq m_2$ iff for all $l \in L$, $m_1(l) \leq m_2(l)$. Also, we define addition $m_1 + m_2$ as the marking m_{12} such that, for all $l \in L$, $m_{12}(l) = m_1(l) + m_2(l)$. Given a set of places $K \subseteq L$, we define its characteristic marking m_K by $m_K(l) = 1$ for all $l \in K$ and $m_K(l) = 0$ for all $l \in L \setminus K$. Moreover, when no confusion is possible from the context, we will simply use K to denote its characteristic marking m_K. Finally, for a given transition τ, we define its pre-set ${}^\bullet\tau$ (resp. post-set $\tau^\bullet$) as the set of places flowing to (resp. from) that transition ${}^\bullet\tau = \{l \mid (l, \tau) \in F\}$ (resp. $\tau^\bullet = \{l \mid (\tau, l) \in F\}$).

Definition 2.4 (atomic behavior semantics)**.**

The semantics of a atomic behavior $B = (P, X, N)$ *with* $N = (L, T, F)$ *is defined as the labelled transition system* $\mathcal{S}_B = (Q_B, \Sigma_B, \underset{B}{\rightarrow})$ *where*

- $Q_B = \mathcal{M} \times \mathcal{V}$ *is the set of states defined by:*
  - $\mathcal{M} = \{m : L \to \{0, 1\}\}$ *the set of 1-safe markings,*
  - $\mathcal{V} = \{\mathbf{v} : X \to \mathcal{D}\}$ *the set of valuations of variables,*
- $\Sigma_B = P \times \mathcal{D}^2 \cup \{\beta\}$ *is the set of labels,*
- $\underset{B}{\rightarrow} \subseteq Q_B \times \Sigma_B \times Q_B$ *is the set of transitions defined by the following rules:*

$$[B_1] \quad \frac{\tau \in T \quad p_\tau = \bot \quad {}^\bullet\tau \leq m \quad g_\tau(\mathbf{v}) = \mathit{true} \quad m' = m - {}^\bullet\tau + \tau^\bullet \quad \mathbf{v}' = f_\tau(\mathbf{v})}{(m, \mathbf{v}) \xrightarrow[B]{\beta} (m', \mathbf{v}')}$$

$$[B_2] \quad \frac{\tau \in T \quad p_\tau \in P \quad \mathit{stable}_B(m, \mathbf{v}) \quad {}^\bullet\tau \leq m \quad g_\tau(\mathbf{v}) = \mathit{true} \quad v^{up} = \mathbf{v}(x_{p_\tau}) \quad m' = m - {}^\bullet\tau + \tau^\bullet \quad \mathbf{v}' = f_\tau(\mathbf{v}[x_{p_\tau} \mapsto v^{dn}])}{(m, \mathbf{v}) \xrightarrow[B]{p(v^{up}/v^{dn})} (m', \mathbf{v}')}$$

where $\mathit{stable}_B(q) = \neg\exists q'.q \xrightarrow[B]{\beta} q'$.

At semantic level, we distinguish two kinds of transitions. *Internal transitions*, defined by the rule B_1, correspond to the firing of behavior transition labeled with $\bot$. These transitions can be taken as soon as they are enabled by the marking and the guard, and update the data valuation and the marking, according to the net flow and annotations of the transition. *Visible transitions*, defined by the rule B_2, correspond to transitions labeled by ports p. These transitions have most likely the same definition as the internal ones, except that they also perform an instantaneous data exchange through the port: the current value v^{up} is sent and a new value v^{dn} is received for x_p, before the update.

Moreover, visible transitions have implicitly lower priority than internal transitions: their execution is postponed until the configuration is stable, that is, no internal transitions are enabled anymore. This choice is motivated by the intuition that, usually, ones expect that components engage in interactions (that may need to consult and update their local data) only when their state is stable.

2.2.3 Connectors and Interaction Models

Composition operators allow to build composite components from a set of sub-components that interact simultaneously by respecting constraints of an interaction model. We propose

a means for structuring interactions by using connectors.

Connectors are used to define sets of related interactions, that means, involving ports from the same support set of ports, in a compact manner. For each interaction, the connector defines its guard, an enabling condition, as well as two transfer predicates defining how the data available on ports is transferred within the connector. As we will see later in more details, the data transfer within connectors is decomposed in two phases. In the first phase, data is moving up from support ports to the connector. In the second phase, data is moving down, from the connector back to the ports. The transfer predicates define the local data transfer rules, for every interaction of the connector.

Definition 2.5 (connector)**.**

A connector γ is a tuple (P, p_0, A) where

- *P is the support set of ports,*
- *p_0 is the exported port of the connector, such that $p_0 \notin P$,*
- *$A \subseteq 2^P$ is a set of interactions $a = \{p_i\}_{i \in I} \subseteq P$ labelled by $g_a, a\uparrow, a\downarrow$ where:*
  - *$g_a((x_{p_i})_{i \in I})$ is the guard condition, that is, a predicate on $\{x_{p_i}\}_{i \in I}$,*
  - *$a\uparrow\ (x'_{p_0}, (x_{p_i})_{i \in I})$ is the upward transfer, that is, a predicate defining x'_{p_0} depending on $\{x_{p_i}\}_{i \in I}$,*
  - *$a\downarrow\ ((x'_{p_i})_{i \in I}, x_{p_0}, (x_{p_i})_{i \in I})$ is the downward transfer, that is, a predicate defining $\{x'_{p_i}\}_{i \in I}$ depending on x_{p_0} and $\{x_{p_i}\}_{i \in I}$*

Example 2.2.

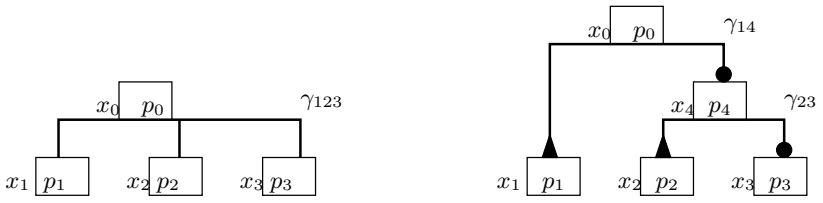
Consider the connector $\gamma_{123} = (\{p_1, p_2, p_3\}, p_0, \{p_1, p_1p_2, p_1p_2p_3\})$ defining a causal chain of interactions on ports p_1, p_2 and p_3. The exported port is p_0. The connector defines three distinct interactions p_1, p_1p_2 and $p_1p_2p_3$. Their associated guards and data transfer are as follows:

a	g_a	$a\uparrow$	$a\downarrow$
p_1	*true*	$x'_0 = x_1$	$x'_1 = x_0$
p_1p_2	*true*	$x'_0 = x_2$	$x'_1 = x_0,\ x'_2 = x_0$
$p_1p_2p_3$	*true*	$x'_0 = x_3$	$x'_1 = x_0,\ x'_2 = x_0,\ x'_3 = x_0$

Clearly, the number of interactions of a connector can grow exponentially to the number of ports in the support set. To manage their complexity, we follow the results in [GS05] and we propose a typing mechanism for the ports of a connector. That is, in order to specify the feasible interactions, we rely on the following two basic modes of synchronization:

- *strong synchronization* or *rendez-vous*, 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 waiting all 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.



(a) (b)

Figure 2.4: Examples of connectors

Example 2.3.

A structural realization of the connector γ_{123} using two broadcast connectors γ_{23} and γ_{14} is illustrated in figure 2.4(b). The associated guards and transfer predicates are as follows:

a	g_a	$a\uparrow$	$a\downarrow$
p_1	*true*	$x'_0 = x_1$	$x'_1 = x_0$
p_1p_4	*true*	$x'_0 = x_4$	$x'_1 = x_0, x'_4 = x_0$

a	g_a	$a\uparrow$	$a\downarrow$
p_2	*true*	$x'_4 = x_2$	$x'_2 = x_4$
p_2p_3	*true*	$x'_4 = x_3$	$x'_2 = x_4,\ x'_3 = x_4$

An interaction model consists of a set of hierarchically structured connectors, built on top of a fixed set of interfaces and satisfying several well-formedness rules such as acyclicity and path determinism.

Definition 2.6 (interaction model)**.**

Let $\{P_j\}_{j\in J}$ be a given set of disjoint interfaces.

Let Γ be a set of connectors, each one exporting a distinct port, and let $P(\Gamma) = \{p_0 \mid \gamma = (P, p_0, A) \in \Gamma\}$ be the set of their (distinct) exported ports.

We say that Γ is an interaction model for the set of interfaces $\{P_j\}_{j\in J}$ by definition iff:

1. *for each connector $\gamma = (P, p_0, A)$ it holds*
   - *(a) it exports a fresh port: $p_0 \notin \cup_{j\in J} P_j$,*
   - *(b) it has a well defined support: $P \subseteq \cup_{j\in J} P_j \cup P(\Gamma)$,*
   - *(c) it uses at most one port in every interface: $\forall j \in J.\ |P \cap P_j| \leq 1$,*
2. *the dependency relation $\rightarrow_\Gamma$ on the set of ports $\cup_{j\in J} P_j \cup P(\Gamma)$ defined as*

$$p \rightarrow_\Gamma p' \quad \Leftrightarrow \quad \exists \gamma = (P, p_0, A) \in \Gamma.(p = p_0 \text{ and } p' \in P)$$

satisfies the following conditions:

(a) it is acyclic*, that is, there are no circular dependencies,*

(b) it is path deterministic*, that is, there is at most one way to reach a port from another one through dependencies.*

Given an interaction model Γ, we define the subset of its top level connectors $\Gamma^\top$ as follows:

$$\Gamma^\top = \{\gamma = (P, p_0, A) \in \Gamma \mid \forall \gamma' = (P', p'_0, A') \in \Gamma.\ p_0 \notin P'\}$$

The ports exported by top level connectors are called top level ports. An interaction model Γ is called *flat* iff all connectors are top level, that is $\Gamma = \Gamma^\top$. Otherwise it is called *hierarchical*.

Example 2.4.

Figure 2.5 presents a hierarchical interaction model for three interfaces $\{s_i, p_i\}_{i=1,3}$.

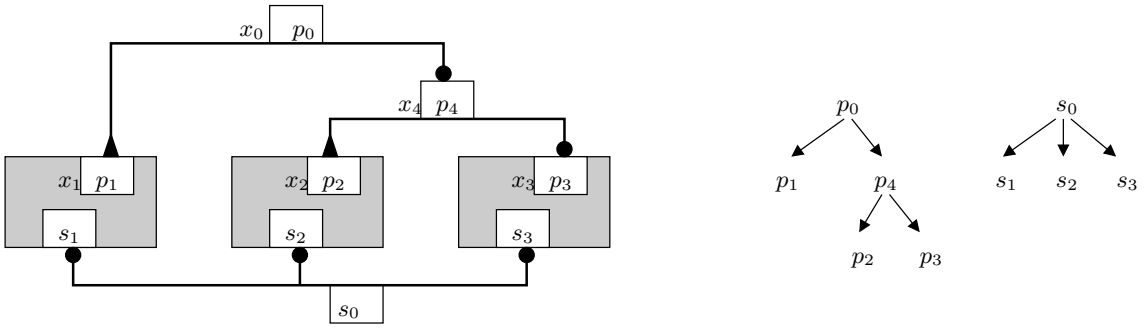


Figure 2.5: Example of interaction model Γ and its associated dependency relation $\rightarrow_\Gamma$

An interaction model defines interaction trees, that are sets of interactions belonging to hierarchical connectors that are executed simultaneously.

Definition 2.7 (interaction tree)**.**

Let Γ *be an interaction model on interfaces* $\{P_j\}_{j\in J}$.

A set of ports $a^t \subseteq \cup_{j\in J} P_j \cup P(\Gamma)$ *is an interaction tree iff*

1. *it contains a uniquely defined port* p_0*, called* $top(a^t)$ *from which all the other ports are recursively dependent*
$$\exists! p_0 \in a^t.\ \ \forall p \in a^t.\ \ p_0 \rightarrow^*_\Gamma p$$
2. *if the interaction tree contains the exported port of some connector, then it includes also exactly one interaction on that connector*
$$\forall \gamma = (P, p_0, A) \in \Gamma.\ (p_0 \in a^t\ \Rightarrow\ \exists! a \in A.\ a \subseteq a^t)$$

The subset of leaf ports in an interaction tree a^t, denoted by $bottom(a^t)$, contains the ports belonging to interfaces, $bottom(a^t) = a^t \cap \cup_{j\in J} P_j$.

An interaction tree is called maximal iff it involves a top level connector, that is $top(a^t) \in P(\Gamma^\top)$. For an interaction model Γ, we denote by $A^t(\Gamma)$ the set of its maximal interactions trees. Moreover, for a given set of ports P, we denote by $A^t(\Gamma, P)$ the set of interaction trees with top ports belonging to P.

Example 2.5.

For the interaction model given in figure 2.5, the set of maximal interaction trees is $\{s_0s_1s_2s_3,\ p_0p_1,\ p_0p_1p_4p_2,\ p_0p_1p_4p_2p_3\}$.

The global data transfer occuring within interaction trees is restricted by the guards and obeys the upward and downward transfer predicates of every triggered connector. More precisely, the operational semantics of connectors is defined in terms of transformations of port valuations, that are, partial functions associating values to (variables associated to) ports.

Let a^t be an interaction tree. Let σ_0 be a partial valuation of interface ports such that $dom(\sigma_0) = bottom(a^t)$. The upward valuation $up_{a^t}(\sigma_0)$ is obtained by propagating values from interface ports upward into the tree, as long as the guard conditions allow them. Formally, it is the smallest port valuation satisfying the following rules:

$$[U_1]\ \frac{(p_0 \mapsto v_0) \in \sigma_0}{(p_0 \mapsto v_0) \in up_{a^t}(\sigma_0)} \qquad [U_2]\ \frac{\{p_i \mapsto v_i\}_{i\in I} \subseteq up_{a^t}(\sigma_0) \quad \gamma = (P, p_0, A) \in \Gamma \quad a = \{p_i\}_{i\in I} \in A \quad g_a((v_i)_{i\in I}) = true \quad a\uparrow (v'_0, (v_i)_{i\in I})}{(p_0 \mapsto v'_0) \in up_{a^t}(\sigma_0)}$$

In a dual manner, we define the downward valuation $dn_{a^t}(\sigma, v)$ obtained by transforming a given valuation σ on a^t ports according to downward transfer predicates and an initial value v on the top port:

$$[D_1]\ \frac{p = top(a^t)}{(p \mapsto v) \in dn_{a^t}(\sigma, v)} \qquad [D_2]\ \frac{(p_0 \mapsto v_0) \in dn_{a^t}(\sigma, v) \quad \{p_i \mapsto v_i\}_{i\in I} \subseteq \sigma \quad \gamma = (P, p_0, A) \in \Gamma \quad a = \{p_i\}_{i\in I} \in A \quad a\downarrow ((v'_i)_{i\in I}, v_0, (v_i)_{i\in I})}{\forall i \in I.\ (p_i \mapsto v'_i) \in dn_{a^t}(\sigma, v)}$$

Example 2.6.

The figure 2.6 gives an example of computing port valuations on the maximal interaction tree of the hierarchical connector from figure 2.4(b). Let consider an initial valuation $x_1 \mapsto 8, x_2 \mapsto 13, x_3 \mapsto 17$ *for the bottom ports. During the upward transfer, the value 17 is propagated to* x_4 *and* x_0*, following the upward predicates* $p_2p_3\uparrow$ *and then* $p_1p_4\uparrow$*. Then, during the downward transfer the value 17 gets propagated downwards to* x_1 *and* x_3 *following* $p_1p_4\downarrow$ *and then* $p_2p_3\downarrow$*.*

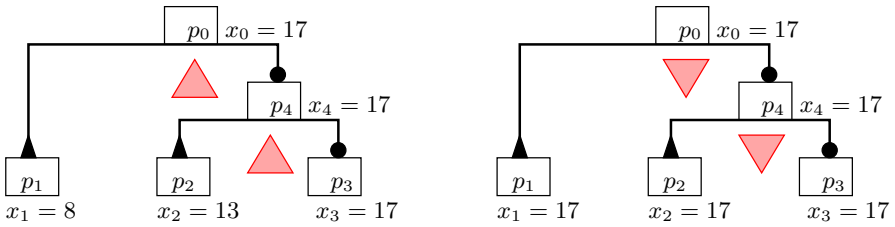


Figure 2.6: Example of data transfer on interaction trees

The formal abstract foundation of the chosen interaction models in BIP is the Algebra of Connectors $\mathcal{AC}(P)$, introduced in [BS07]. This algebra provides a compact notation for algebraic representation and manipulation of connectors. It extends the notion of connectors to algebraic 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 2.7(a). It is also possible to structure connectors hierarchically, as shown in figure 2.7(b) where terms p_1p_4 and p_3p_4 are typed and then fused to obtains a new connector.

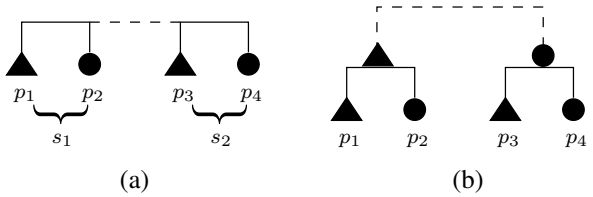


Figure 2.7: Fusion (a) and structuring (b) of connectors in $\mathcal{AC}(P)$

2.2.4 Priority Models

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 priorities may change dynamically depending on the system state, they can advantageously overcome the static restrictions of other execution models.

A priority model is a memoryless controller defined by a fixed set of dynamic priority rules on interaction trees. It filters the possible interaction trees from the interaction model, based on (an abstraction of) the current global state of the system.

Definition 2.8 (priority model)**.**

Let A^t be a set of interaction trees and X a set of variables.

A dynamic priority model Π *across interaction trees A^t and dependent on data X is defined by a set of priority rules π of the form (g, a^t, b^t) where*

- *a^t, b^t are distinct interaction trees in A,*
- *g is the condition of the priority, that is a predicate on X,*

We note priority rules $\pi = (g, a^t, b^t)$ as $a^t \prec_g b^t$.

A priority model is saturated *iff, forall distinct interaction trees a^t, b^t, c^t it holds*

$$a^t \prec_{g_1} b^t \ \wedge \ b^t \prec_{g_2} c^t \quad \Rightarrow \quad a^t \prec_{g_1 \wedge g_2} c^t$$

Let us notice that any priority model can be statically saturated.

A particular priority model, 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 interaction trees, of the form $(a_i)_{i\in I} \prec (a_i b_i)_{i\in I}$, where a_i and $a_i b_i$ are interactions of the same connector. As an example, maximal progress is necessary to model correctly a broadcast. Maximal progress is implicitly assumed in connectors for their compact and natural representation.

Example 2.7.

The maximal progress for the interaction trees of the interaction model given in figure 2.5 is defined by the priority rules: $p_0p_1 \prec_{true} p_0p_1p_4p_2 \prec_{true} p_0p_1p_4p_2p_3$.

2.2.5 Composite Components

Composite components are defined recursively by composition from atomic behavior or other composite components using glue consisting of interaction and priority models. We should notice the following key issues about the proposed composition operation:

- the interface of the new component is a subset of top level ports of the interaction model. That is, only the chosen top level ports remain visible outside for future interaction, whereas the others are hidden and become internal to the component;
- similarly, the data available from the composite component is a subset of data available from its subcomponents;
- finally, it is required that the priority model filters only maximal interaction trees completed within the component, that is, interaction trees having top level ports internal to the component.

Definition 2.9 (component)**.**

Components C are recursively defined as

- *atomic components, defined by an atomic behavior $B = (P, X, N)$ or*

- *composite components, defined as tuples* $(P, X, gl, \{C_j\}_{j\in J})$ *where*
  - $\{C_j\}_{j\in J}$ *are the sub-components, either atomic or composite, with disjoint interfaces* $\{P_j\}_{j\in J}$ *and available data* $\{X_j\}_{j\in J}$*,*
  - $gl = \langle \Pi, \Gamma \rangle$ *is the composition glue:*
    - Γ *is an interaction model for the set of interfaces* $\{P_j\}_{j\in J}$*,*
    - Π *is a saturated priority model across* $A^t(\Gamma, P^{(loc)})$*, the subset of maximal interaction trees which remain local, that is with top ports from* $P^{(loc)} = P(\Gamma^\top) \setminus P$*, and dependent on data available from subcomponents* $\cup_{j\in J} X_j$*,*
  - $P \subseteq P(\Gamma^\top)$ *is the interface of the component, that is a subset of the exported ports of the top level connectors,*
  - $X \subseteq \cup_{j\in J} X_j$ *is the available data of the component, that is a subset of data available from its subcomponents.*

Example 2.8.

The figure 2.8 shows a composite component which implements a simple distributed sorting algorithm. Every atomic component holds two variables, x_i *and* y_i*, such that* $x_i < y_i$*. These variables are available on ports* min_i *and* max_i*. Their values are exchanged over local connectors relating adjacent components, in order to obtain a globally sorted list of values. The global minimum* (x_1) *and maximum* (y_3) *values are made available on the interface of the component, for further composition.*

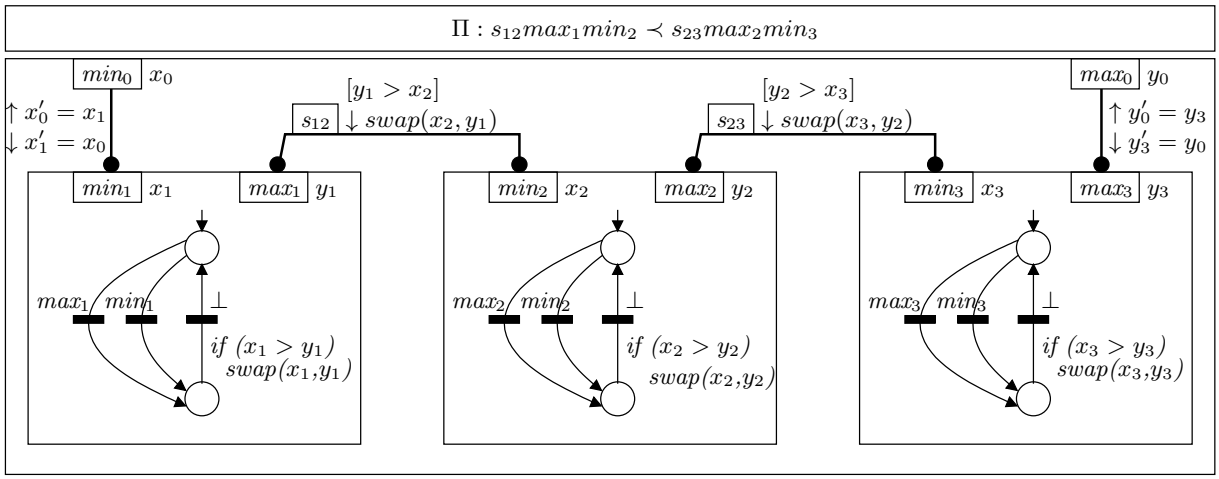


Figure 2.8: Example of a composite component

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

Definition 2.10 (component semantics)**.** *The semantics of component* C *is a labeled transition system* $\mathcal{S}_C = (Q_C, \Sigma_C, \xrightarrow[C]{})$ *defined inductively on the structure of* C *as follows:*

1. *C is an atomic component, defined by an atomic behavior $B = (P, X, N)$.*
   Then, $\mathcal{S}_C = \mathcal{S}_B$.

2. *C is a composite component defined as the $(P, X, gl = \langle \Pi, \Gamma \rangle, \{C_j\}_{j \in J})$*
   Let $\mathcal{S}_{C_j} = (Q_{C_j}, \Sigma_{C_j}, \xrightarrow[C_j]{})_{j \in J}$ be the semantics of its sub-components. The labeled transition system $\mathcal{S}_C = (Q_C, \Sigma_C, \xrightarrow[C]{})$ is defined as:

   - $Q_C = \bigotimes_{j \in J} Q_{C_j}$ *is the set of states, the Cartesian product of sets of states of sub-components,*
   - $\Sigma_C = P \times \mathcal{D}^2 \cup A^t(\Gamma, P^{(loc)}) \cup \{\beta\}$ *is the set of labels,*
   - $\xrightarrow[C]{} \subseteq Q_C \times \Sigma_C \times Q_C$ *is the transition relation, defined by the following rules:*

$$[C_1] \quad \frac{q_i \xrightarrow[C_i]{\beta} q_i' \qquad \forall j \in J \setminus \{i\}.q_j = q_j'}{((q_j)_{j \in J}) \xrightarrow[C]{\beta} ((q_j')_{j \in J})}$$

$$[C_2] \quad \frac{q_i \xrightarrow[C_i]{a_i^t} q_i' \qquad \forall j \in J \setminus \{i\}.q_j = q_j'}{((q_j)_{j \in J}) \xrightarrow[C]{\beta} ((q_j')_{j \in J})}$$

$$[C_3] \quad \frac{\begin{array}{c} \mathit{fireable}_C(q, a^t, q') \\ \mathit{stable}_C(q) \\ \forall (a^t \prec_g b^t) \in \Pi. \left(\begin{array}{c} g(q) = \mathit{true} \\ \Rightarrow \\ \neg \mathit{fireable}_C(q, b^t, -) \end{array} \right) \end{array}}{q \xrightarrow[C]{a^t} q'}$$

$$[C_4] \quad \frac{\begin{array}{c} \mathit{fireable}_C(q, p_0(v^{up}/v^{dn}), q') \\ \mathit{stable}_C(q) \\ \neg \exists a^t.\ \mathit{fireable}_C(q, a^t, -) \end{array}}{q \xrightarrow[C]{p_0(v^{up}/v^{dn})} q'}$$

where the $\mathit{fireable}_C$ and stable_C predicates are defined as follows:

$$\mathit{fireable}_C((q_j)_{j\in J}, \ell, (q'_j)_{j\in J}) =$$

$$\exists I \subseteq J.\ \exists \sigma_0, \sigma_{up}, \sigma_{dn}.\ \left(\begin{array}{c} \forall i \in I.\ [q_i \xrightarrow[C_i]{p_i(v_i^{up}/v_i^{dn})} q'_i] \quad \forall j \in J \setminus I.\ q_j = q'_j \\ a^t \in A^t(\Gamma) \quad p_0 = top(a^t) \\ \sigma_0 = \{p_i \mapsto v_i^{up}\}_{i\in I} \\ \sigma_{up} = up_{a^t}(\sigma_0) \qquad (p_0 \mapsto v_0^{up}) \in \sigma_{up} \\ \sigma_{dn} = \left\{ \begin{array}{ll} dn_{a^t}(\sigma_{up}, v_0^{dn}) & \text{if } p_0 \in P \\ dn_{a^t}(\sigma_{up}, v_0^{up}) & \text{if } p_0 \in P^{(loc)} \end{array} \right. \\ \sigma_{dn} \supseteq (p_i \mapsto v_i^{dn})_{i\in I} \\ \ell = \left\{ \begin{array}{ll} p_0(v_0^{up}/v_0^{dn}) & \text{if } p_0 \in P \\ a^t & \text{if } p_0 \in P^{(loc)} \end{array} \right. \end{array} \right)$$

$\mathit{stable}_C(q) = \neg\exists q'.\ q \xrightarrow[C]{\beta} q'$.

The semantics of composite components distinguish three types of transitions:

- *internal transitions*, defined by rules C_1 and C_2 and labeled by β, correspond to internal transitions and interaction transitions taking place in subcomponents;
- *interaction transitions*, defined by rule C_3 and labeled by maximal interaction trees a^t, correspond to complete interactions taking place within the component. They are defined according to interaction and priority models of the component, as follows. First, for an interaction to take place, the involved subcomponents must be ready to interact, and moreover the ports offered and their associated data must enable a^t and fulfill its associated data transfer. Furthermore, the interaction must be completed within the component, that means, its top port belongs to a local connector and therefore no extra participants are foreseen for interaction. Finally, this interaction tree must be also maximal according to the priority model amongst all other interaction transitions enabled;
- *visible transitions*, defined by the rule C_4 and labeled by a data exchange through an interface port, correspond to partial interactions taking place within the component. They are similar to interaction transitions, except that they involve an extra data exchange on the top port of the interaction tree, which belongs to the interface of the component.

Finally, the three types of transitions are implicitly prioritized such that, internal transitions have higher priority than interaction transitions, which in turn have higher priority than visible transitions. Intuitively, this implies that interactions take place on globally stable states.

Example 2.9.

The three types of transitions can be clearly identified on figure 2.9, which shows a fragment of the underlying semantics of the composite component presented in figure 2.8. In every state we highlight the values of variables x_i and y_i.

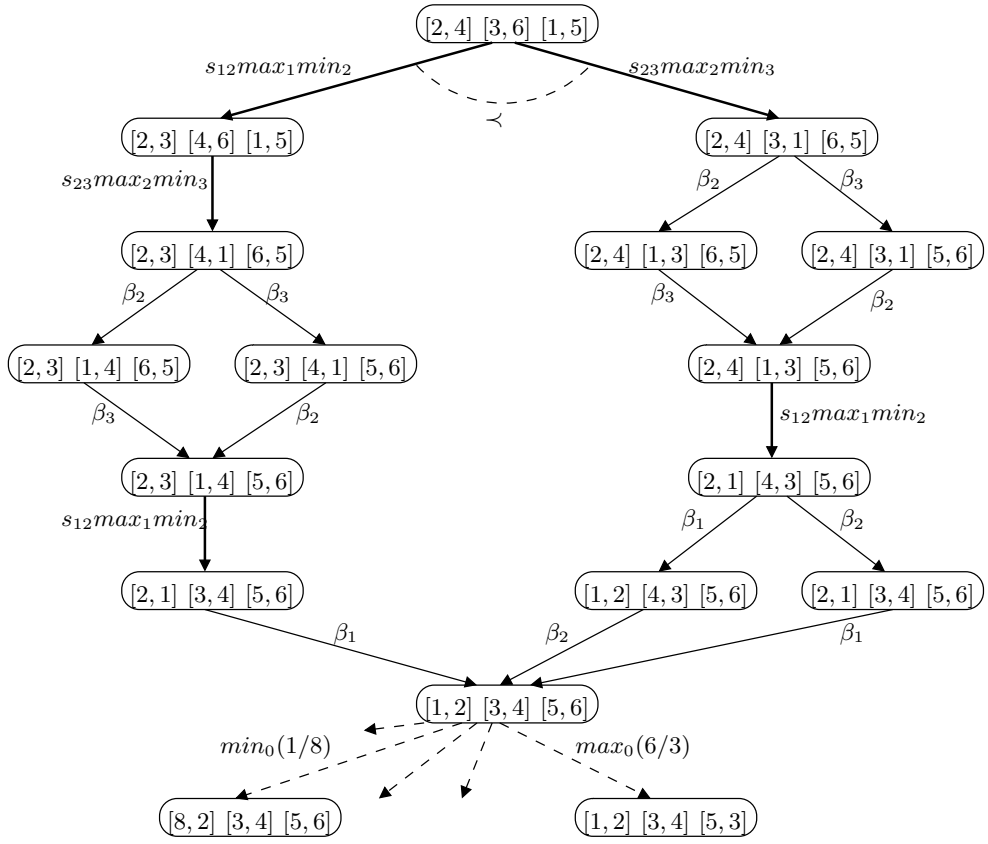


Figure 2.9: Semantics of a composite component (fragment)

2.3 The BIP Language

For representing system models in the BIP framework we developed the BIP language.

It is a user-friendly textual language which provides syntactic constructs for describing systems conforming to the formal framework presented in Section 2.2. The BIP language leverages on C style variables and data type declarations, expressions and statements, and provides additional structural syntactic constructs for defining component behavior, specifying the coordination through connectors, and describing the priorities. Moreover, it provides additional constructs for dealing with parametric descriptions (i.e., where the same component occur replicated in many places) as well as for expressing timing constraints associated with behavior.

The principal constructs are:

1. *atom*: to specify atomic 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 coordinations between the ports of components, their interactions and the associated guards and up/down transfer.
3. *priority*: to restrict the possible interactions, based on conditions depending on the state of the integrated components.
4. *composite*: to specify components hierarchically, from atomic behavior or composite components, 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.
6. *package*: to specify a set of related components and connectors;

The language allows defining types for ports, atoms, connectors, and composite components, defining priorities, and instantiating objects of the defined types.

Example 2.10.

The example below provides the concrete textual representation for the sorting example in BIP.

```
model sorting

/* definition of port types */
port type MinPort(int x)
port type MaxPort(int y)
port type InternalPort

/* definition of swapping connectors */
connector type Swap
      (MaxPort max, MinPort min)
   define max min
   on max min
      provided max.y > min.x
      down {# swap(max.x, min.y); #}
end

/* definition of singleton connectors */
connector type MinCopy(MinPort min)
   define min
   data int x0
   on min up x0 = min.x;
      down min.x = x0;
   export port MinPort min0(x0)
end

connector type MaxCopy(MaxPort max)
   define max
   data int y0
   on max up y0 = max.y;
      down max.y = y0;
   export port MaxPort max0(y0)
end

/* definition of atomic components */
atomic type Element(int px, int py)
   data int x = px
   data int y = py

   export port MinPort min(x)
   export port MaxPort max(y)
   port InternalPort update

   place sorted
   place unsorted = initial

   on min from sorted to unsorted
   on max from sorted to unsorted
   on update from unsorted to sorted
      do # if (x > y) swap(x, y); #
end

/* definition of the composition */
compound type Network
   component Element e1(2,4)
   component Element e2(3,6)
   component Element e3(1,5)

   connector MinCopy min0(e1.min)
   connector Swap s12(e1.max, e2.min)
   connector Swap s23(e2.max, e3.min)
   connector MaxCopy max0(e3.max)

   priority pi s12 < s23

   export port MinPort min0 is min0
   export port MaxPort max0 is max0
end

/* the top level component */
component Network n

end
```

2.4 Discussion

The BIP framework shares features with other existing frameworks proposed 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.

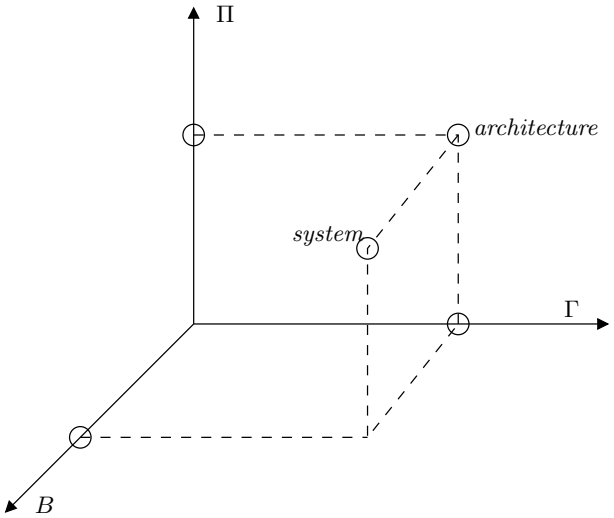


Figure 2.10: The BIP Construction Space

BIP characterizes systems as points in a three-dimensional space *Behavior* × *Interaction* × *Priority*, as represented in figure 2.10. Elements of this 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, asynchronous/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 later in section 3.1, timed systems can be obtained from an untimed systems by 1) refinement of its untimed behavior (adding continuous variables 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-freedom and state invariance.

Chapter 3

Language Factory

We show hereafter how BIP can be used to model several classes of systems as well as to represent several commonly used programming models.

First of all, we provide the general principles for modeling of timed systems using discrete time. Then, we provide a general representation for the synchronous programming model. We show that we can meaningfully represent synchronous components e.g., Lustre nodes, using (compositions of) atomic components with a particular cyclic behaviour. Moreover, for this particular class of components we provide sufficient syntactic conditions to ensure deadlock freedom and confluence of computations. Finally, we will show how BIP can be used to represent domain specific programming models used for wireless sensor networks and autonomous robotic systems.

3.1 Timed Systems

Timed systems are faithfully modeled as timed automata with urgency [BST98, BS00]. In timed automata models, the execution of a transition is an instantaneous action defining a discrete change in the state, whereas time progresses continuously in states. Urgency is expressed by means of an urgency attribute on transitions. This attribute can take the values *eager*, *lazy* or *delayable*. *Eager* transitions must executed at a point of time at which they become enabled, unless they are not disabled by executing another transition. *Lazy* transitions do not have any urgency constraints, i.e., if not executed they can be disabled by time progress. *Delayable* transitions are a composite type very useful in practice. Delayable transitions cannot be disabled by time progress. They are considered to be lazy at some state unless they are disabled at the next time unit; otherwise, they are considered eager.

As basis for the description of timed systems, we introduce atomic timed behavior which is a slight extension of the atomic behavior defined in section 2.2. The definition of atomic behavior is inspired from [AGS02]. It consists in adding the following restrictions/extensions on the atomic behavior model:

- the control flow is restricted to a simple automaton,
- the set of variables is partitioned into *continuous* (evolving continuously with time

progress) and *discrete* (unchanged over time progress) variables,

- for every location l and continuous variable c, there is an *evolution function* $\Phi_l^c : \mathcal{D}(c) \times \mathbb{R}^+ \to \mathcal{D}(c)$ describing the continuous evolution of c over time. Evolution function Φ_l^c must satisfy:
  1. $\Phi_l^c(v, 0) = v$, forall $v \in \mathcal{D}(c)$
  2. $\Phi_l^c(v, t_1 + t_2) = \Phi_l^c(\Phi_l^c(v, t_1), t_2)$, forall $v \in \mathcal{D}(c), t_1, t_2 \in \mathbb{R}^+$
- every transition τ has an *urgency* type attribute u_τ.

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

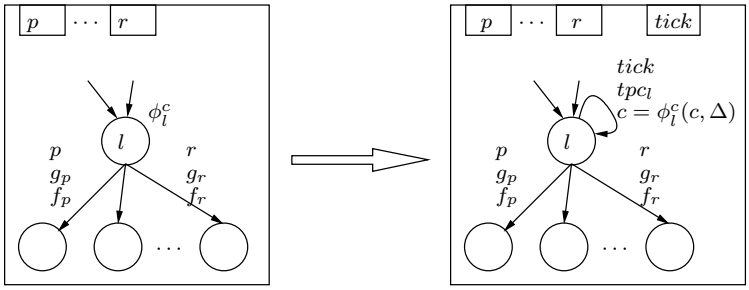


Figure 3.1: From timed atomic behavior to atomic behavior

Formally, the meaning of atomic timed behavior is defined by a simple structural mapping to regular atomic behavior. The principle of the mapping is illustrated in figure 3.1. It consists in discretizing time by a fixed step Δ and modelling explicitly time progress by loop transitions on every location. These transitions are labeled by a special fresh port, called *tick*, which will be further used for synchronization with other timed components. For a control location l, their guard and associated functions are defined as follows:

- the guard $tpc_l = \bigwedge\{\neg g_\tau \mid (l, \tau) \in F,\ u_\tau = \epsilon\}$, that means, time progress is enabled iff no eager transition outgoing from l is enabled,
- the function performs $c' = \phi_l^c(c, \Delta)$, for all continuous variables c, that means, all continuous variables take their value after Δ time units.

In composite components involving timed behavior, strong synchronization is necessary between all the *tick* ports as shown in the architecture of figure 3.2. That is, a connector implementing a rendez-vous between all the *tick* ports is needed in order to model synchronous time progress for all the interacting components. It is worth mentioning that such a construction is meaningful only if all atomic behavior has been discretized using the same time step Δ.

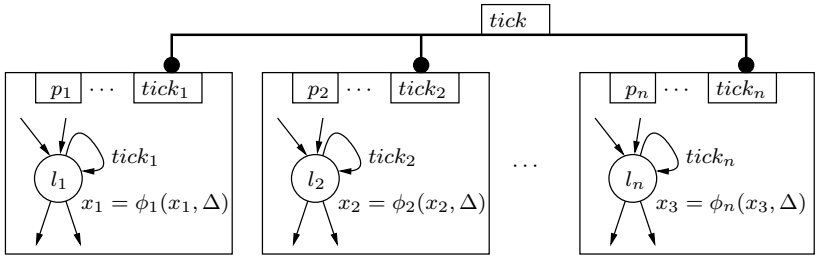


Figure 3.2: Composition of timed components

Case Study: Scheduling of Timed Tasks

This example taken from [WDE05], comes from a performance evaluation problem with timed tasks processing events from a bursty input generator. There are three tasks T_1, T_2 and T_3, connected serially to an input generator. A task is activated by the reception of an input event from its predecessor, and executes on dedicated CPU's. On completion, the task sends an output event to its successor. The block diagrams of two possible instantiations of the task system on several CPUs are shown in figure 3.3.

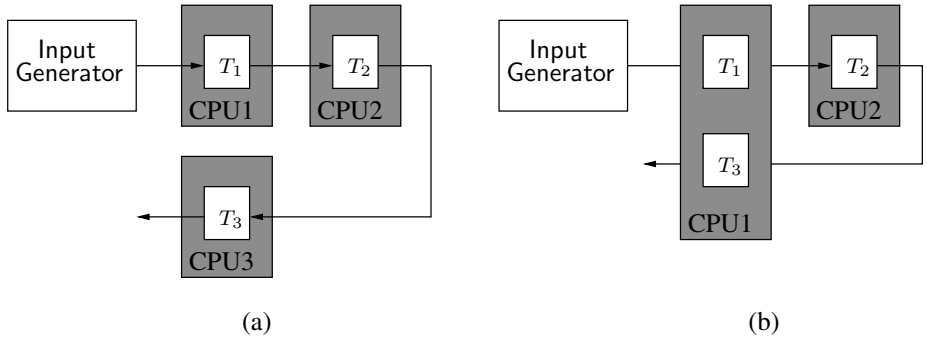


Figure 3.3: Scheduling of Timed Tasks

The basic components for the model are Task and InputGenerator. A generic timed model of Task is shown in figure 3.4, which can be used either as a simple task, or as a preemptable task. It uses an integer variable c to count the number of input events received and not yet handled. It uses a timed variable d to measure of the execution time. The evolution functions specifying the update of d with time are 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 3.4 means that the transition is lazy when $d < WCET$ and eager when $d = WCET$.

The ports and behavior of InputGenerator are shown in figure 3.5. This component is parameterized by the period T, the jitter J with $J > T$, and the minimum inter-arrival time D between successive input events being generated.

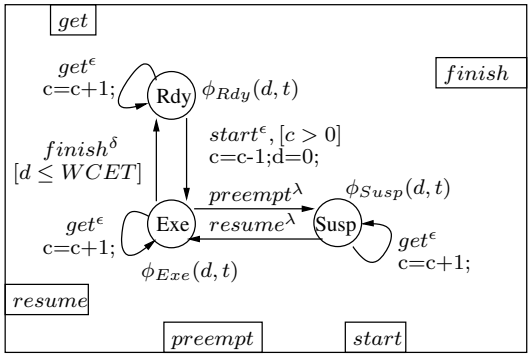


Figure 3.4: Task Component

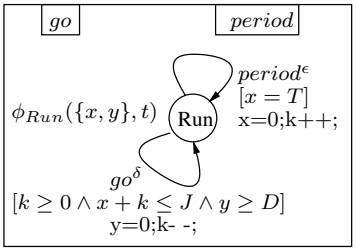


Figure 3.5: InputGenerator Component

The first system instance given in figure 3.3(a) is constructed in BIP as a serial composition of the InputGenerator, and three instances of Task, T_1, T_2 and T_3, as illustrated in figure 3.6. The transmission of the events, i.e., synchronization between the InputGenerator and Tasks are modeled by connectors. In this model, the tasks are non-preemptable, so the ports *preempt* and *resume* are not involved in any interaction.

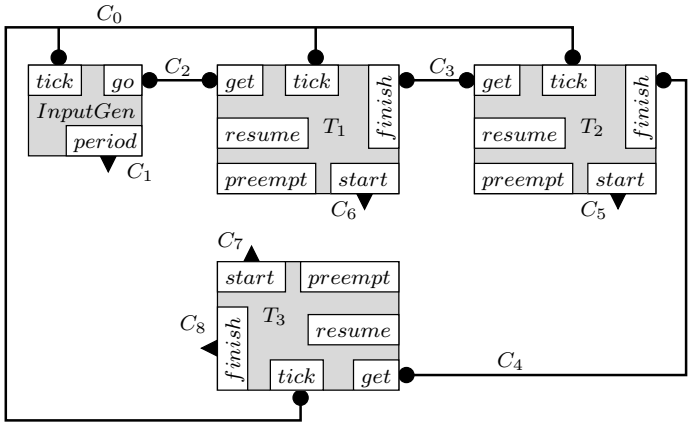


Figure 3.6: System model: first instance

The second system instance given in figure 3.3(b) where tasks T_1 and T_3 share CPU1 and T_1 can preempt T_3, whereas T_2 runs on CPU2, is constructed in BIP as illustrated in figure 3.7.

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.

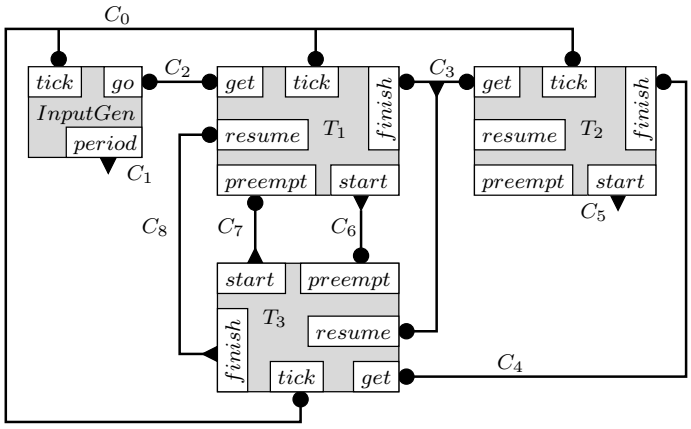


Figure 3.7: System model: second instance

Static priority between T_1 and T_3 is enforced by the priority rule

$$T_3.start \prec T_1.start$$

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

$$T_3.start\ T_1.preempt \prec \mathsf{InputGenerator}.tick\ T_1.tick\ T_2.tick\ T_3.tick$$

3.2 Synchronous Systems

The main principles of modeling synchronous systems in BIP have been introduced in [BSS09].

Synchronous systems are composed of strongly synchronized parallel components. Their global behavior is characterized by runs consisting of successive computation steps. In each step, all components perform some quantum of computation. This ensures a built-in fairness between components in sharing resources, usually enforced by using static scheduling policies. Synchronous computation models are particularly adequate for hardware, real-time systems and streaming systems. Their main advantage over asynchronous computation models is efficiency and predictability (determinacy), in particular thanks to lightweight analysis techniques for deciding deadlock-freedom and timeliness. Nonetheless, for general applications an adequate mix of synchronous and asynchronous computation is necessary for optimal use of resources e.g. GALS models [BCC+08].

3.2.1 Modal Flow Components

In this section, we show how the basic execution mechanisms underlying synchronous dataflow systems can be modeled in BIP. We define the class of *modal flow components*. They are a sub-class of BIP components where Petri nets are replaced by *modal flow graphs*. These correspond to a subclass of priority Petri nets for which deadlock-freedom and confluence can be decided at low cost. Modal flow graphs are structures expressing dependency relations

between events within one computation step. Similar structures such as [HM08, Now06, ZL08] have been proposed and used in different contexts. An important difference between modal flow graphs and related formalisms is the use of three different *modalities* characterizing dependency between events. For a given set of ports P, a modal flow graph is a directed acyclic graph with nodes P and edges representing the union of three binary relations. Each relation expresses a different kind of causal dependency (modality) between pairs of ports p and q:

- q *strongly depends* on p if the execution of p *must* be followed by the execution of q. That is, p and q cannot be executed independently, only the sequence pq is possible;
- q *weakly depends* on p if the execution of p *may* be followed by q. That is either p can be executed alone or the sequence pq;
- q *conditionally depends* on p if when both p and q are executed, then q must follow p. Conditional dependency requires that if p and q occur then only the sequence pq is possible; otherwise p or q may be independently executed.

The semantics of a modal flow component is defined by an atomic BIP component further restricted by a priority order on ports. The Petri net is derived from the modal flow graph as follows. We define a transition for every port in the modal flow graph. Moreover, we define an extra transition, labeled by a distinguished *sync* port, to delimit successive computation steps. Places of the net are defined for *minimal* ports in the modal flow graph as well as for every pair of dependent ports. The former are used to initialize the computation, whereas the latter are used to enforce the right order of execution between dependent ports. According to their definition, places can be tagged as initial, final, or both. The *sync* transition is enabled when only final places are marked. In this case, termination of a step consists in removing tokens from the final places and putting a token in each initial place. Finally, we consider a priority order on ports which ensures maximal progress in every computation step: first, all (regular) ports have higher priority than *sync* and second, every port has higher priority than all its dependent ports in the modal flow graph. This construction is illustrated in the following example.

Example 3.1.

In figure 3.8 we show a modal flow graph describing the treatment of an email in one computation step. Bold, simple and dashed arrows represent respectively strong, weak and conditional dependency relations. Depending on conditions [cond$_1$] and [cond$_2$], the possible execution sequences are : open read close, open read forward send$_1$ close, open read answer write send$_2$ close, open read (forward send$_1$ || answer write send$_2$) close where || is the interleaving of sequences.

Figure 3.9 shows the associated BIP behavior. Initial places are marked with a token; final places are grayed. As usual, transitions are enabled when their input places are marked and the associated condition is true. The priority order restricts choices amongst enabled transitions (ports). That is if both forward and close are enabled then forward is executed. Finally, the sync transition is not explicitly represented.

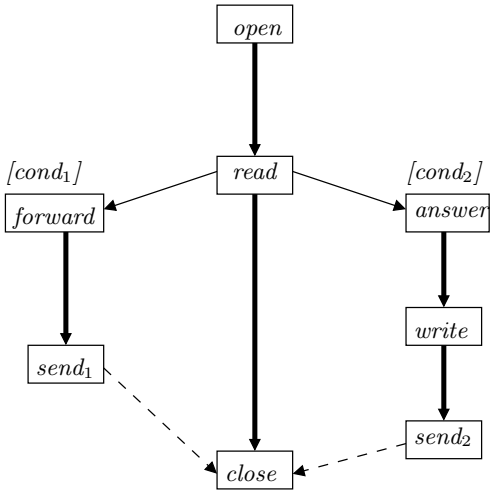


Figure 3.8: Modal flow graph

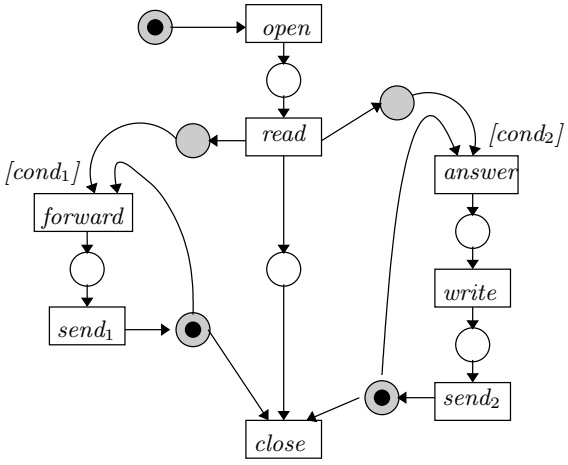


Figure 3.9: BIP behaviour

We show that modal flow graphs are deadlock-free if they are *well-triggered*. This property expresses consistency between the three types of dependency. It also guarantees confluence under some conditions of non interference of concurrent computations.

Well-triggered modal flow graphs can be decomposed as shown in Figure 3.10. The strong dependency relation defines a set of connected subgraphs involving all the ports of the component. Each one of these subgraphs has a single root which is the common cause for its ports. Weak dependencies express triggering of the root of a subgraph by some port of another subgraph. Finally, conditional dependencies may relate ports of different subgraphs provided the acyclicity property is not violated.

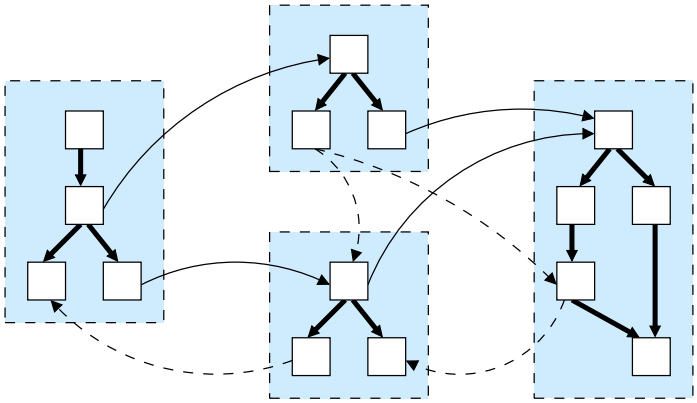


Figure 3.10: Well trigerred modal flow graphs

The following theorem taken from [BSS09] establishes the main results about modal flow

components.

Theorem 3.1.

1. *A well-triggered modal flow component is* deadlock-free *if every port with strong causes has its guard true.*
2. *A well-triggered modal flow component is confluent if for every independent ports, their associated guarded actions do not interfere.*

3.2.2 Modeling of Lustre programs

To illustrate the use of modal flow graphs for modeling synchronous systems, we provide a modular translation of Lustre [HCRP91] into modal flow graphs. Similar translations can be made for other synchronous languages or graphical formalisms [GGBM91, CCN06].

Lustre [HCRP91] is a dataflow synchronous language for programming reactive systems. Lustre programs operate on flows of values, that are infinite sequences $(x_0, x_1, \cdots, x_n, \cdots)$ of values at logical time instants $0, 1, \cdots, n$. An abstract syntax for Lustre programs is shown below. *In* (resp. *Out*) denotes the set of input (resp. output) flows of a program node. Symbols N, E, x, v, b denote respectively node names, expressions, flows, boolean flows and constant values.

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

A Lustre program is structured as a set of *nodes*. Each node computes output flows from input flows. Output flows are defined either directly by means of equations of the form $x = E$, meaning $x_n = E_n$ for any time instant $n \geq 0$ or, as the output of other (already defined) nodes instantiated with particular inputs $x, ... = N(E, ...)$.

The basic operators used in expressions E, are combinatorial operator (op), unit delay (**pre**), sampling (**when**) and interpolation (**current**). Combinatorial (memory-less) operators include usual boolean, arithmetic and relational operators. The unit delay **pre** operator gives access to the value of its argument at the previous time instant. For example, the expression $E' = \textbf{pre}(E, v)$ means $E'_0 = v$ and $E'_i = E_{i-1}$, for all $i \geq 1$.

In Lustre each flow (and expression) is associated with a logical clock. Implicitly, there always exists a unique, fastest, *basic clock* which defines the step (or basic clock cycle) of a synchronous program. Depending on this clock, other slower clocks can be defined as the sequences of time instants where boolean flow variables take the value *true*. In order to define and manipulate flows operating on slower clocks, Lustre provides two additional operators. The *sampling* operator **when**, samples a flow depending on a boolean flow. The expression $E' = E\ \textbf{when}\ b$, is the sequence of values E when the boolean flow b is *true*. The expression E and the boolean flow b have the same clock, while the expression E', operates on a slower clock defined by the instants at which b is true. The *interpolation* operator **current**, interpolates an expression on the clock which is immediately faster than its own clock. The expression

$E' =$ **current** E, takes the value of E at the last instant when b was true, where b is the boolean flow defining the slower clock of E.

We consider statically correct programs which satisfy the static semantics rules of Lustre [Hal98]. These rules exclude programs containing cyclic, dependent equations, recursive calls of nodes as well as combinatorial operators applied to expressions having different clocks.

We define modular operational semantics for Lustre, first for single-clock programs and then for multi-clock programs.

Single-clock synchronous programs

The single-clock subset of Lustre is generated by using only combinatorial and unit delay operators. All flows are sampled (indexed) by the basic clock.

The translation from Lustre to modal flow graphs is modular. Each node is represented by a well-triggered modal flow component with two kinds of ports: *act* control ports and *input* (in) or *output* (out) data ports. An *act* port is triggered by the basic clock and initiates the step of the node. The *in* (resp. *out*) data ports carry data input (resp. output) read (resp. produced) by the node. Additionally, modal flow graphs may contain internal ports and variables, depending on the specific computation carried by the node.

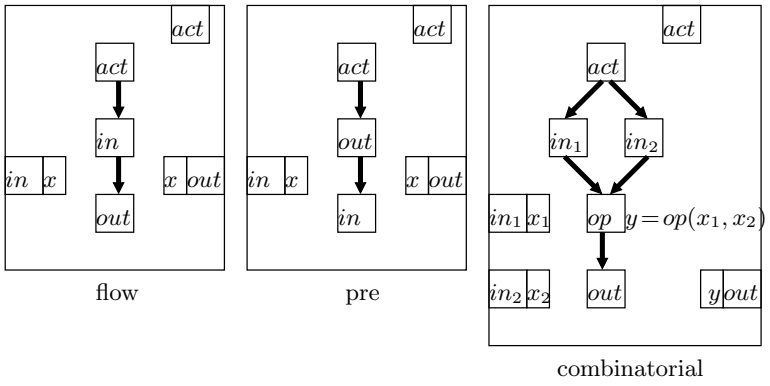


Figure 3.11: Single-clock operators

The modal flow components shown in Figure 3.11 correspond to basic Lustre elements: flow, pre operator and combinatorial operator. The flow component whenever activated through the *act* port, reads a value through the *in* port and outputs this value through the *out* port in the same step. The **pre** component has a local variable x. Whenever it is activated through *act*, it outputs the current value x, then it inputs and assigns a new value to x to be used in the next step. The combinatorial component starts a step when it is triggered through the *act* port. Then it reads input values in some arbitrary order, performs its specific computation, and finally, produces an output value.

The modal flow component representing a single-clock Lustre node is obtained by composing a set of components by using a set of interactions defined as follow:

- **components:** For each input and output flow declared in the node we add a *flow*

component. For each **pre** (resp. combinatorial) expression occuring within the equations, we add a *pre* (resp. *combinatorial*) component. Moreover, for each subnode called within equations we add its corresponding modal flow component.

- **interactions:** Interactions are of two types: *control flow* and *data flow*. A single control flow interaction realizes strong synchronization between all the *act* ports of all components. Data flow interactions synchronize one *out* port to one or more *in* ports. They are used to propagate data from input flow components to expression components and from expression components to output flow components or other expression components according to the syntactic structure of expressions and equations.

Example 3.2.

Figure 3.12 shows a discrete integrator written in Lustre and its corresponding synchronous network of operators.

```
node Integrator(i:  int)
     returns o: int;
let  o = i + pre(o,0); tel;
```

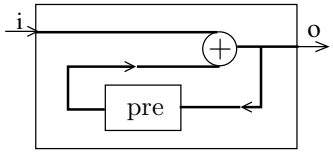


Figure 3.12: Integrator

The representation of this node as a composition of modal flow components is shown in Figure 3.13. The atomic modal flow components correspond to the **pre** *operator, the combinatorial + operator, the input flow and the output flow. In addition to the* act *interaction, there are three interactions for data transfer from outputs to inputs: 1) from the input flow component to the + component, 2) from the* **pre** *component to the + component and 3) from the + operator to the output flow component and back to the* **pre** *component. The result of the composition is shown in Figure 3.14.*

The following theorem is a consequence of modularity of translation and of the following facts: 1) the modal flow graphs corresponding to the basic constructs of Lustre are well-triggered; 2) for statically correct Lustre programs [HCRP91], composition of the basic modal flow graphs preserves well-triggeredness.

Theorem 3.2.

Every statically correct single-clock Lustre node is represented by a well-triggered modal flow component such that:

1. *it has a unique root which is an* act *port;*
2. *all its dependencies are strong;*
3. *it is deadlock-free and confluent;*
4. *simulates the micro-step Lustre semantics [Hal98] of the node.*

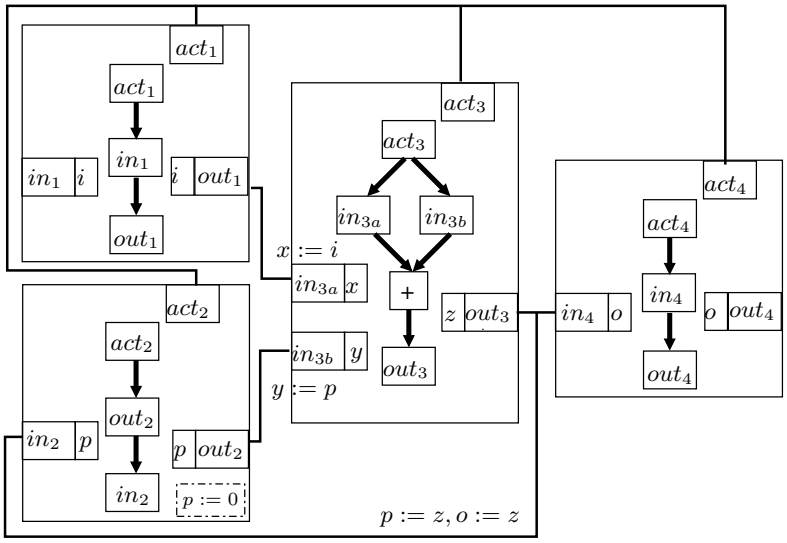


Figure 3.13: The integrator node as composition of elementary modal flow components

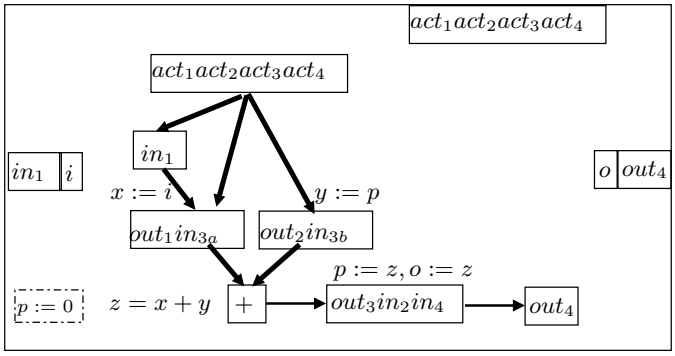


Figure 3.14: The integrator modal flow component

Multi-clock synchronous programs

In Figure 3.15, we provide two components modeling respectively the sampling and interpolation operators of Lustre. Both components have two control ports, act_i and act_o triggering respectively the input *in* and the output *out* data ports. For a sampling component, act_o depends weakly on act_i, and moreover, the output *out* depends conditionally on the input *in*. Thus an input is always read and whenever required, an output is produced with the most recent value of the input – which is precisely the interpretation of sampling. For the interpolation component, we have the opposite: act_i depends weakly on act_o but *out* conditionally depends on *in*. Thus the output is always produced with the most recent value of the input. The last modal flow graph in Figure 3.15 describes an additional component, the *derived clock* component corresponding to a boolean flow *b*. This component is used to initiate all the computations carried on the clock *b*. Intuitively, it triggers the slower *clock* port only after its base clock *act* has been triggered and if the value obtained through the data input *in* port is true.

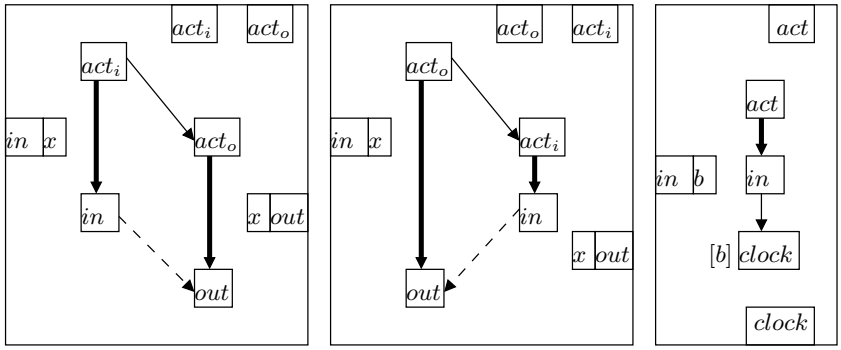


Figure 3.15: Multi-clock operators

We apply a similar modular construction method for building modal flow components for multi-clock nodes:

- **components:** First, we add a *derived clock* component for each clock (i.e, **when** b). Second, we add a *sampling* (resp. *interpolation*) component for each sampling (resp. interpolation) expression occuring within the equations of the node.
- **interactions:** The data flow interactions are the same as for the single-clock case, with the addition that data is also propagated to the input port of derived clocks. Regarding control flow interactions, we add one interaction which synchronizes all the *act* ports of flows and expressions sampled on the basic clock. In addition, for each derived clock component, we add an interaction which synchronizes its *clock* port with all *act* ports of flows and expressions sampled by that clock.

Example 3.3.

Consider the following Lustre program:

```
node input_handler(a: bool, x: int when a)
returns y: int;
let   y = if a then current x else pre(y, 0); tel ;

node output_handler(c: bool, y: int) returns z: int when c;
var   yc: int when c;
let   yc = y when c; z = yc * yc ; tel ;

node  input_output(a,c: bool, x: int when a)
returns z: int when c;
var   y: int;
let   y = input_handler(a, x); z = output_handler(c, y); tel;
```

Depending on an input value x triggered by an input clock a, the `input_output` *node produces a corresponding output value z triggered by an output clock c, by using the most recent available value of the input.*

The main node is the `input_output` *node which interconnects the two nodes,* `input_handler` *and* `output_handler`*. The* `input_handler` *node receives at every moment the boolean value a. An integer value x is received only when a is* true*. The output value y is an integer produced at every moment by interpolating the value of x. The* `output_handler` *node receives at every moment a boolean c and an integer variable y. It produces an output z by sampling y when c is* true*. Finally, the* `input_output` *top node connects the output of the* `input_handler` *to the input of the* `output_handler`*.*

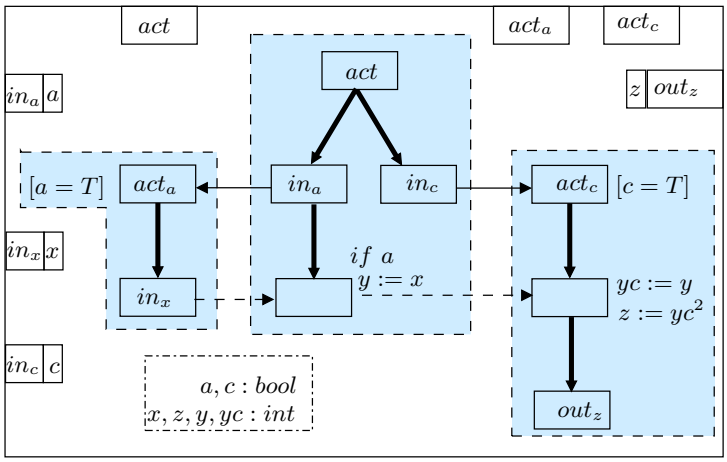


Figure 3.16: The input/output handler

Figure 3.16 shows the modal flow component representing the system. Its modal flow graph is obtained after composition and static simplification of the modal flow graphs of the **input_output** *node. It can be decomposed into three subgraphs with activation ports* act, act_a *and* act_c *corresponding respectively to the basic,* **when** `a`*, and* **when** `c` *clocks.*

The following theorem establishes the correctness of our translation.

Theorem 3.3.

Every statically correct multi-clock Lustre node is represented by a well-triggered modal flow component which:

1. *has multiple (control) root* act *ports, one for each clock in the Lustre program, and multiple data* in/out *ports;*
2. *the subgraphs defined by strong dependencies are connected through weak dependencies into a tree;*
3. *is deadlock-free and confluent;*
4. *simulates the micro-step Lustre semantics [Hal98] of the node.*

3.2.3 Experimental work

We have implemented a translator from Lustre to BIP synchronous components which directly generates Petri nets without using modal flow graphs. The translator is currently fully operational. It takes as input Lustre programs and produces full-fledged BIP systems, that can be simulated and analyzed using the BIP toolset.

This first approach has several important drawbacks. For the generated BIP programs it is not easy to verify properties guaranteed by construction for some synchronous programs e.g. deadlock-freedom and confluence. Moreover, the BIP compilation chain cannot easily recover the information that the system is indeed synchronous and consequently, it cannot produce optimized code. For example, experiments with concrete Lustre programs show an $600:1$ overhead of execution time between the C code produced by the BIP compiler and executed by the BIP engine, and the flat C code produced by the Lustre compiler. Although, this overhead can be diminished to $20:1$ by applying static composition of components in BIP, it still remains high.

Modal flow graphs allow coping with these drawbacks. We are now investigating the possibility to integrate directly modal flow components in BIP. Our results about confluence and deadlock-freedom of modal flow components provide syntactic conditions, easily implementable in an automatic tool. Moreover, modal flow components keep all the data-flow explicit and can be used to generate efficient code, monolithic or not, as synchronous language compilers do.

3.3 Architecture Analysis & Design Language

The SAE Architecture Analysis & Design Language (AADL) [FLV03] is a textual and graphical language proposed to design and analyze the software and hardware architectures of

performance-critical real-time systems. It plays actually a central role in several toolsuites such as OSATE [SEI06] and Ocarina[HZPK08].

A system modelled in AADL consists of application software mapped to an execution platform. Data, subprograms, threads, and processes collectively represent application software. They are called software components. Processor, memory, bus, and device collectively represent the execution platform. They are called execution platform components. Execution platform components support the execution of threads, the storage of data and code, and the communication between threads. Systems are called compositional components. They allow software and execution platform components to be organized into hierarchical structures with well-defined interfaces. Operating systems may be represented either as properties of the execution platform or can be modelled as software components.

Components may be hierarchical, i.e. they may contain other components. In fact, an AADL description is almost always hierarchical, with the topmost component being an AADL system that contains, for example, processes and processors, where the processes contain threads and data, and so on. Compared to other modeling languages, AADL defines low-level abstractions including hardware descriptions. These abstractions are more likely to help design a detailed model close to the final product.

The paper [CRBS08] presents a general methodology and an associated tool for the structural translation of AADL specifications into BIP. In this work, we define precise operational models for all AADL components in terms of BIP components. Moreover, we clearly define the AADL communication mechanisms using various types of ports in terms of BIP interaction models. This precise mapping enables simulation of systems specified in AADL as well as their analysis using formal verification techniques developed for BIP, e.g. deadlock detection.

Example 3.4.

An AADL thread represents a sequential flow of control that executes instructions within a process. The parallel execution of several threads inside a process is managed by a scheduler.

A thread type declaration specifies communication ports (including data ports, event ports, and event data ports), subprogram declarations, and property associations. A thread component implementation specifies local data, the definition of the behaviour either as a subprogram sequence call or as a state-machine, and thread property associations. Properties are used to represent attributes and other characteristics, such as the period, dispatch protocol, deadline, etc. For instance, the dispatch protocol defines how the thread is activated for execution. Four dispatch protocols are supported in AADL: periodic, aperiodic, sporadic, and background.

The representation of an AADL thread as an atomic BIP component is shown in figure 3.17. The initial state of the thread is halted. On an interaction through the load port the thread is initialized and moves to the init state. Then, it enters the ready state, if immediately available for execution, otherwise it enters the suspended state. When the thread is in the suspended state it cannot be dispatched for execution; the thread is waiting for an event and/or period depending on the dispatch protocol to wake up. In the ready state, a thread is waiting to be dispatched for execution through an interaction on the port exec. When dispatched, it enters the state compute to perform its specific computation. Upon successful completion, the thread goes to the outputs state and produces its outputs, if any. Then, it enters the finish state. Finally, the thread may be requested to enter its halted state through interactions on the stop port or abort port (at any time).

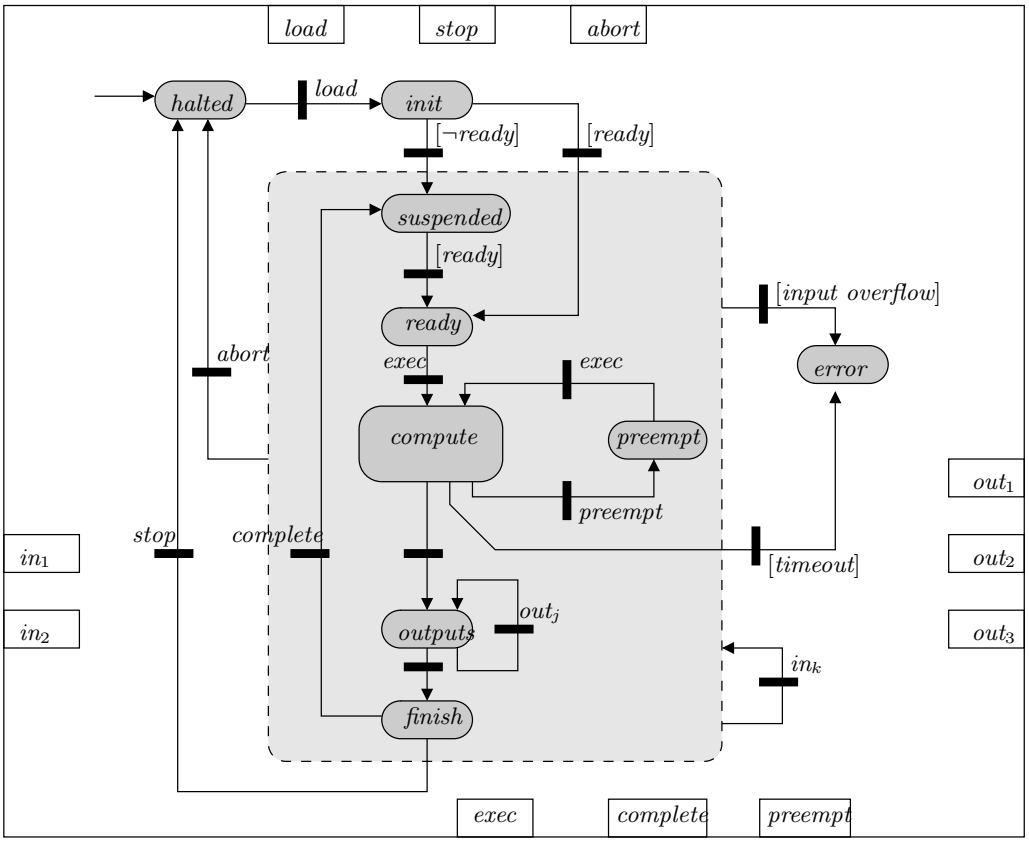


Figure 3.17: Generic representation of AADL threads in BIP

3.4 Domain Specific Languages

In the remainder of this chapter we will show how BIP can be used to represent domain specific programming models. We will present two concrete cases: the representation of nesC applications used to program motes in sensor networks and the representation of Genom applications used to program functional modules in autonomous robots. However, for sake of exhaustivity, let us mention that BIP has been also used for the representation of Business Process Execution Language BPEL [AAA+07], Distributed Operation Layer DOL [TBHH07] and FXML [YAD+08]

Wireless Sensor Networks

Wireless sensor networks are complex component-based systems with rich dynamics subject to strong extra-functional requirements. They have surprisingly many applications nowadays, however, their design is extremely complex because it involves the composition of a variety of hardware and software components developed with different methodologies and tools. Consequently, there is a limited understanding on how specific component features and applications impact the global behavior of such networks.

The main obstacle for a better design practice is the lack of modeling frameworks encompassing heterogeneity. Currently, there exists mainly simulation environments that use simulation software built in a more or less ad hoc manner, by integrating the application code in specific platforms e.g., Tossim [LLWC03] or EmTOS [GSR+04]. They can be useful for debugging purposes but they are not adequate for a more thorough exploration of a network's non-deterministic dynamics.

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. Following this principle, a model construction methodology using BIP has been developed for TinyOS [All] based networks. This methodology, presented in [BMP+07], 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 abstract models of TinyOS components. This opens the way for enhanced analysis and early error detection by using verifications techniques. The methodology 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. Some preliminary verification results have been reported in [BMP+07].

- Another important difference is incremental model construction of BIP models. Incrementality means that the global model is obtained by progressively composing its atomic components as shown earlier in section 2.2. This system construction methodology allows defining an architecture hierarchy, with the glue specifying the composition at an architectural level from its subordinate levels. Figure 3.18 shows the architecture of a sensor node (mote) consisting of nesC applications and TinyOS, with their internal contents. The 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.

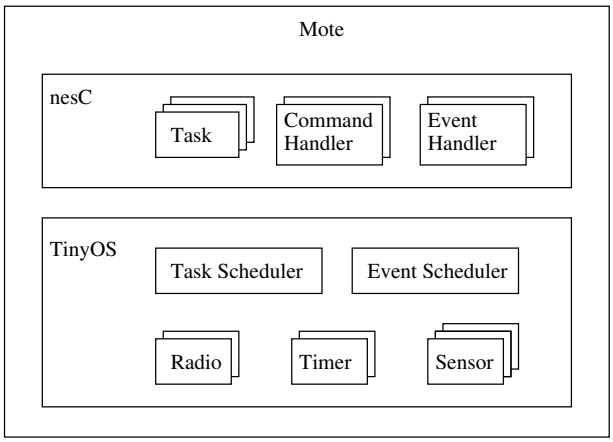


Figure 3.18: Architecture of a Mote

Autonomous Robotic Systems

Autonomous robots are complex systems that require the interaction/cooperation of numerous heterogeneous software and hardware components. Nowadays, robots are getting closer to humans and as such are becoming critical systems which must meet safety properties including in particular logical, temporal and real-time constraints.

The use of BIP for modeling, design and implementation of autonomous robots has been investigated in a joint project with LAAS[1], a leading laboratory in the robotics domain. At LAAS, researchers have developed a global software architecture, that enables the seamless integration of heterogeneous processes (e.g., with different functionalities and requirements) needed for driving robots. This architecture decomposes the robot software into three main levels:

- a *functional level*, it includes all the basic built-in robot action and perception capacities. These processing functions and control loops (e.g., image processing, obstacle avoidance,

[1]Laboratoire d'Analyse et d'Architecture des Systèmes, Toulouse

motion control, etc.) are encapsulated into controllable communicating modules developed using a dedicated software component framework, called GenoM [FHC97, MFB02]. Each module provides services which can be activated by the decisional level according to the current tasks, and exports posters containing data produced by the module and for others (modules or the decisional level) to use;

- a *decisional level*: this level includes the capacities of producing a task plan (using the IxTeT planner [LG95]) and supervising its execution, while being at the same time reactive to events from the functional level;
- at the interface between the decisional and the functional levels, lies an *execution control level* that controls the proper execution of the services according to safety constraints and rules, and prevents functional modules from unforeseen interactions leading to catastrophic outcomes. This level is usually programmed on the top of existing functional modules, however, it does not depend on the internal execution details of the modules themselves.

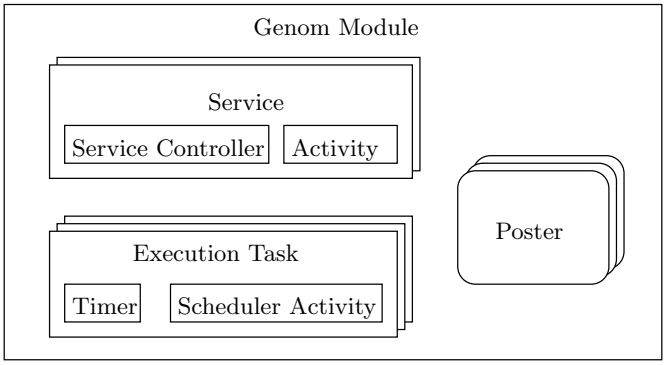


Figure 3.19: Architecture of a GenoM module

Our joint collaboration with LAAS reported in [BIS08, BGL+08] demonstrate that BIP can be seamlessly integrated within the preexisting design methodology, in particular:

- BIP has been used for (1) the incremental modeling of GenoM modules used to implement the functional level and (2) the modeling of their allowed interactions as specified by the control execution level of the robot;
- the resulting BIP model has been used to synthesize a controller for the overall execution of all the functional modules and to enforce by construction the constraints and the rules inside modules but also between the various functional modules;
- the BIP model has been also used to formally verify several safety-critical properties of the robot including ordering and synchronization constraints, data freshness properties, deadlock-freedom, etc.

The approach has been fully implemented and now there exists a GenoM/BIP controller for the navigation part of a functional level of the DALA robot [BGL+08], running in simulation

and on the real robot. This controller enforces online by construction the interactions model (intra-module and inter-module). The concrete runs on the robot show that the performance of the code generated from BIP is good enough for a simple yet complete robotics experiment. This work also shows that it is possible to use structural analysis techniques for deadlock detection and for verification of safety properties.

Chapter 4

System Implementation

Efficient implementation of component-based systems is a non-trivial task. Nowadays, it is widely admitted that modularity in component-based development incurs an additional non-negligible overhead for implementation because of extensive use of interfaces, wrappers and other implementation artifacts.

In this chapter, we present methods for efficient and modular implementation of BIP systems. By implementation, we mean producing one (or more) executable(s) running according to the operational semantics. First of all, we show that it is possible to eliminate the hierarchical definition of composite components and interaction models. That is, BIP models can be flatenned while preserving completely their operational semantics and without increasing their overall size. Second, we provide two implementation methods for flat composite components. The former is sequential and targets single-processor (or single-threaded) execution platforms, whereas the later is distributed and targets multi-processor (or multi-threaded) platforms. Third, we present static optimization allowing to reduce the overhead of componentization.

4.1 Flatenning

As argued in section 2.1, incrementality of a component-based development framework can be achieved through flatenning, that is, the ability to eliminate the hierarchical structure of composite components. For BIP, the flatenning of a composite component is fully defined through two steps:

1. *flattening of component hierarchy* which replaces the hierarchy on components by a hierarchical interaction model applied only on atomic components;
2. *flattening of connector hierarchy* which computes for each hierarchically structured connector an equivalent flat connector;

Example 4.1.

We illustrate the flatenning on the BIP model given in figure 4.1. It consists in the serial connection of two sorting networks of three elements each, identical to the one presented earlier in section 2.2. The internal behaviour of atomic components is not relevant for flatenning and therefore it is not presented.

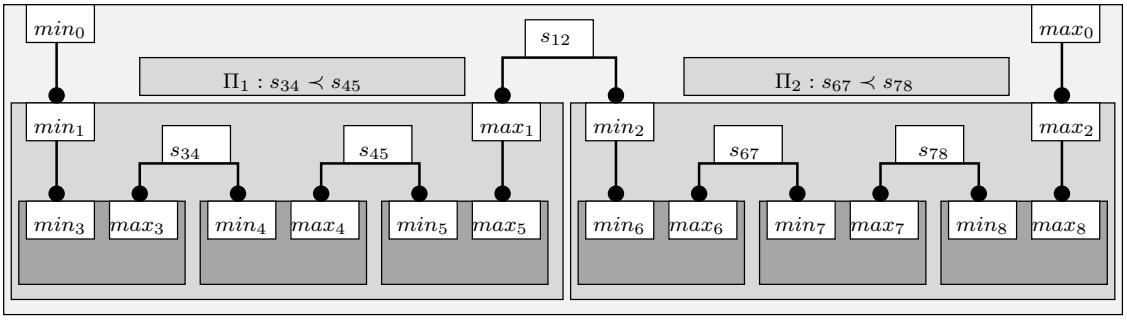


Figure 4.1: An example of hierarchical component

4.1.1 Flatenning of Component Hierarchy

The purpose of this transformation is to eliminate the hierarchical structure of composite components. This transformation is implemented as the *simultaneous inlining* of all the inner components, as defined below.

Consider an arbitrary composite component $C = (P, X, gl, (C_j)_{j\in J})$. The result of the simultaneous inlining of subcomponents within C is another composite component, denoted $\nabla C = (P_0, X_0, gl_0, (B_k)_{k\in K})$ where:

- $P_0 = P$, the set of interface ports is P,
- $X_0 = X$, the set of available data is X,
- $(B_k)_{k\in K}$, the set of all atomic components B_k contained recursively in C,
- $gl = \langle \Pi_0, \Gamma_0 \rangle$ where
  - $\Gamma_0 = \cup_{l\in L}\Gamma_l$, the union of all interaction models contained recursively in the composite subcomponents C_l of C,
  - $\Pi_0 = sat(\{a^t \prec_g b^t \mid \exists a^t_l \prec_g b^t_l \in \Pi_l.\ a^t_l \subseteq a^t,\ b^t_l \subseteq b^t\} \cup$
    $\{a^t_1 \prec_{true} a^t_2 \mid \exists C_1, C_2.\ top(a^t_1) \in P_1^{(loc)},\ top(a^t_2) \in P_2,\ C_1 \text{ contains } C_2\})$
    Priorities are obtained as the (saturated) union of two sets of priorities. First, there is the extension of priorities defined in sub components to complete interactions across the whole interaction model. Second, there are priorities derived from the implicit semantics rule that gives higher preference to interaction transition (i.e., complete interactions) over visible transitions (i.e., incomplete interactions) at every composite component.

Example 4.2.

Figure 4.2 presents the composite component obtained by flatenning the hierarchical composite component from figure 4.1. Intuitively, this transformation removes the inner composite boxes and moves the priorities to the top component. In this example, let us remark that priorities between inner interactions $s_{23} \prec s_{34}$ respectively $s_{67} \prec s_{78}$ are preserved as such. In addition, we have the rule $s_{12}max_1min_2 \prec s_{23}, s_{34}, s_{67}, s_{78}$ which gives priority to inner interactions with respect to outer ones.

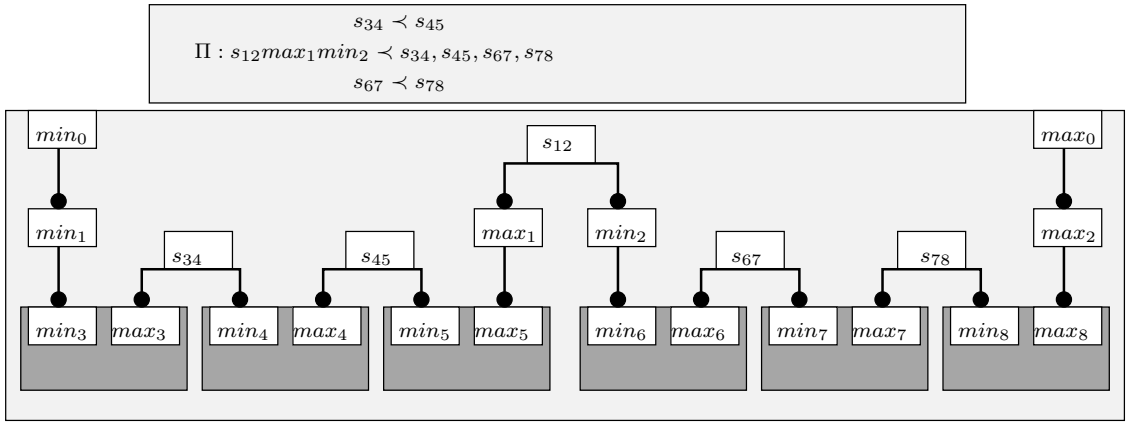


Figure 4.2: Flatenning of component hierarchy

Intuitively, by inlining we preserve the global interaction model. Therefore, the same interactions are potentially enabled and executed on the hierarchical model as well as on the flat model. Moreover, priorities on the flat model are defined in order to preserve both priorities from the hierarchical model and implicit priorities enforced by the semantics of parallel composition. Inner (lower-level) interactions have higher priority than outer (higher-level) interactions. Formally, the following proposition holds.

Proposition 4.1.

Structural inlining preserves semantics, up to hiding of inner interactions
$\mathcal{S}_C \simeq \mathcal{S}_{\nabla C}$

4.1.2 Flatenning of Connector Hierarchy

The purpose of this transformation is to eliminate the hierarchical structure of interaction models. This transformation relies on the composition (i.e., glueing) of two linked connectors, introduced below.

Definition 4.1 (composition of connectors)**.**

Let $\gamma_i = (P_i, p_{i,0}, A_i)_{1,2}$ be connectors such that $p_{2,0} \in P_1$, that is, γ_1 is hierarchically dependent on γ_2. The composition of γ_1 with γ_2 is a new connector denoted $\gamma_1 \circ \gamma_2 = (P_{12}, p_{12,0}, A_{12})$ where:

- $P_{12} = (P_1 \setminus p_{2,0}) \cup P_2$,
- $p_{12,0} = p_{1,0}$,
- $A_{12} = \{a_1 \mid a_1 \in A_1,\ p_{2,0} \notin a_1\} \ \cup$
  $\{a_1 \setminus \{p_{2,0}\} \cup a_2 \mid a_1 \in A_1,\ p_{2,0} \in a_1,\ a_2 \in A_2\}$

  In the first case, the guards and data transfer are inherited as such from γ_1. In the second case, the guard and the transfer are defined as follows:

  - $g_{a_{12}} = g_{a_2} \ \wedge \ \exists x_{p_{2,0}}.\exists x'_{p_{2,0}}.(a_2\uparrow \ \wedge \ g_{a_1} \ \wedge \ x_{p_{2,0}} = x'_{p_{2,0}})$

– $a_{12}\uparrow = \exists x_{p_{2,0}}.\exists x'_{p_{2,0}}.(a_1\uparrow \ \wedge \ a_2\uparrow \ \wedge \ x_{p_{2,0}} = x'_{p_{2,0}})$
– $a_{12}\downarrow = \exists x_{p_{2,0}}.\exists x'_{p_{2,0}}.(a_1\downarrow \ \wedge \ a_2\downarrow \ \wedge \ x_{p_{2,0}} = x'_{p_{2,0}})$

Intuitively, by composition, two linked connectors are glued together into a single connector. Their guards, respectively the upward and downward transfer predicates are composed according to the rules defined for propagation of port valuations in section 2.2. Consequently, any port valuation obtained by the successive application of the upward (resp. downward) transfer predicates of the two connectors is equally obtained by the application of the upward (resp. downward) transfer predicate of the composed connector.

Example 4.3.

Figure 4.3 illustrates the successive glueing of subconnectors for one of the hierarchical connectors occuring in figure 4.2. Roughly speaking, inner ports are eliminated and the effect of upward/downward transfer predicates is propagated to the upward/downward transfer of the parent connector.

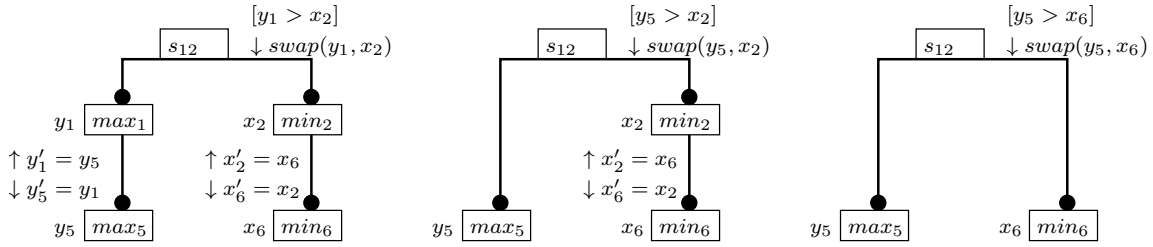


Figure 4.3: Example of connector glueing

The composition of connectors extends to hierarchical glues. Consider a hierarchical glue $gl = \langle \Pi, \Gamma \rangle$ and let $\gamma_2 = (P_2, p_{2,0}, A_2) \in \Gamma \setminus \Gamma^{\top}$ be one of its transient connectors. The connector γ_2 can be glued with all the dependent connectors in Γ and the result propagated over priorities. The result of the inlining will be a new glue, denoted $gl \triangle \gamma_2 = \langle \Pi', \Gamma' \rangle$ where:

- $\Gamma' = \{\gamma_1 \mid \gamma_1 = (P_1, p_{1,0}, A_1),\ p_{2,0} \notin P_1,\ \gamma_1 \neq \gamma_2\} \ \cup$
  $\{\gamma_1 \circ \gamma_2 \mid \gamma_1 = (P_1, p_{1,0}, A_1),\ p_{2,0} \in P_1\}$
- $\Pi' = \Pi[p_{2,0} \mapsto \bot] = \left\{(a^t \setminus \{p_{2,0}\}) \prec_g (b^t \setminus \{p_{2,0}\}) \mid a^t \prec_g b^t \in \Pi\right\}$

Example 4.4.

Figure 4.4 presents the composite component obtained by flatenning the interaction model from figure 4.1. Intuitively, all inner ports are removed and priority rules are updated accordingly.

Flatenning of hierarchical glue consists in iteratively glueing all linked connectors, until the final interaction model becomes flat e.g., it does not contain hierarchical connectors anymore. According to the following proposition, any composition step can be proven correct, that means, it preserves the semantics of the hierarchical model up to renaming of hierarchical interactions.

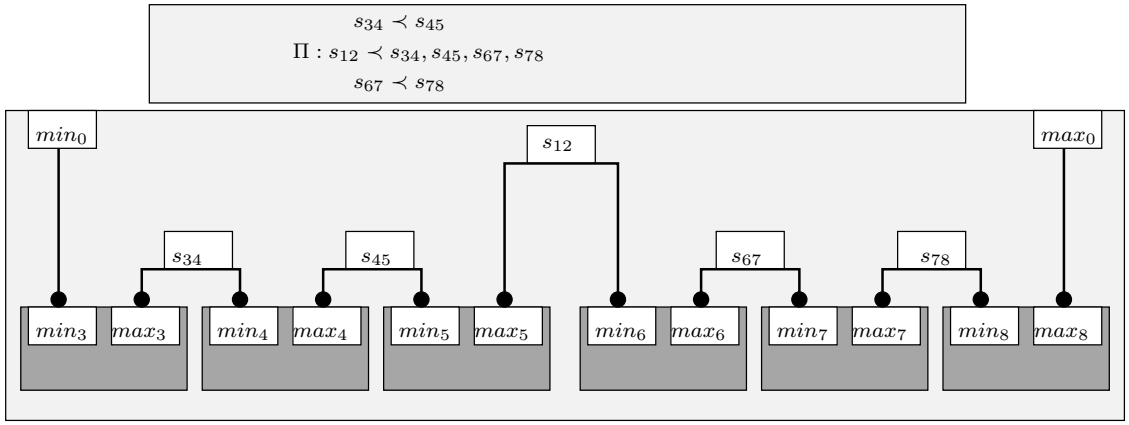


Figure 4.4: Flatenning of connector hierarchy

Proposition 4.2.

1. *inlining of transient connectors preserves semantics up to renaming* $\mathcal{S}_C \simeq \mathcal{S}_{C \triangle \gamma_2}$
2. *inlining of transient connectors is commutative* $(gl \triangle \gamma_1) \triangle \gamma_2 = (gl \triangle \gamma_2) \triangle \gamma_1$

4.2 Implementation

We present hereafter methods for implementing BIP components. Given a BIP (composite) component, we want to produce executable code that runs conforming to the operational semantics of the component.

Without loss of generality, we will consider only flat composite components. This is clearly not a restriction, as we have already seen in section 4.1 that any hierarchical component can be flatenned with no extra cost.

We are interested in producing modular implementations, where the code of atomic components is perfectly isolated from the glue and coordination code needed to play interactions and priorities. This way, we will be able to perform separate compilation as well as to include legacy components, for which the complete source code may not be available. For doing so, we consider a relatively simple interface for atomic components, consisting of two functions, *initialize* and *execute*:

- the *initialize* function is supposed to be called once in order to initialize the component and to execute its behaviour until the first stable state is reached. At that point, this function returns the set of ports on which the component is ready to interact, together with their associated (up) values;
- the *execute* function is supposed to be called iteratively, after *initialize*. Its argument is a port amongst the ones previously proposed for interaction together with the actual (down) value available on it. This function should then perform the quantum of

computation triggered by that port, starting from the current stable state and until the next stable state is reached. At that point, as the *initialize* function, it returns the set of ports ready for interaction at that state.

Let us mention that an actual implementation for this interface can be automatically generated from atomic BIP components.

Moreover, we distinguish between glue code and coordination code. In the first category, we consider the code needed for data transfer on connectors and for priority evaluation between enabled interactions. This code is also abstracted through simple interfaces and automatically generated from interaction and priority models. In the second category, we consider the code orchestrating the whole execution e.g., implementing the operational semantics of BIP models using atomic behavior's and glue implementation. This code, denoted further as the *engine*, is completely generic i.e., independent of BIP models running underneath, and developed manually once for all.

We are mainly interested in two categories of implementations: sequential and distributed. As the name suggests, in the first case, we are targeting sequential (or monothreaded) implementations where the code of atomic components, the glue and coordination code is running on a single processor. In the second case, we are targeting distributed (or multi-threaded) implementations where the code of atomic components and respectively, the glue and coordination code is split across different processors. Such distributed implementations can advantageously exploit the computing capabilities provided by e.g, multi-core execution platforms.

4.2.1 Sequential Implementation

The sequential implementation of BIP follows precisely the operational semantics described in section 2.2. From a BIP model, a compiler is used to generate the C++ code for atomic components and glue. This code is then orchestrated by a sequential engine that directly interprets the operational semantics rules.

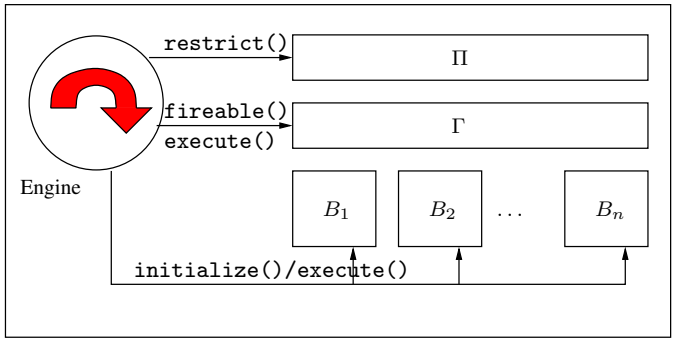


Figure 4.5: Architecture of sequential implementation

The main algorithm of the engine is given below. It starts by initializing and retrieving the set of enabled ports for every atomic components. Then, in the main loop, the engine computes from the set of the ports offered by individual components and the set of interactions, the set of the enabled interactions. Amongst these, it chooses a maximal one, according to priorities.

Then, for the chosen interaction, the engine executes the data transfer followed by the specific computations of every involved atomic components.

```
foreach j in 1,n do
   P_j := B_j.initialize();
do forever
   A := compute-fireable(Γ, P_1, ..., P_n);
   A^max := restrict-priorities(Π, A);
   if A^max is not empty then
      choose a = (p_i)_{i∈I} in A^max;
      execute-data-transfer(a);
      foreach i in I do
         P_i := B_i.execute(p_i);
   else
      deadlock();
      stop;
   fi
done
```

The centralized engine has run-time options for execution and enumerative state-space exploration.

In execution mode, 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.

In state space exploration mode, the engine generates the underlying labeled transition systems (LTS) of the model. The LTS can be minimized and compared using tools like Aldebaran [BFKM97] or further analyzed by model-checking. In particular, we have used the model-checker tool Evaluator [MS00] to perform verification of temporal logic properties. For example, properties related to specific order of execution of interactions have been verified for a robotic controller [BGL+08] and for a self stabilizing distributed reset algorithm [BBBS09].

A less costly alternative to temporal logic model checking is validation with observers. For a safety property Φ, we construct first an observer for Φ, i.e. an automaton which monitors the system behavior and reports an error on violation of Φ. Observers can be modeled in BIP as atomic components with extra annotations on locations, like *Error* and *Prune*. The validation consists of exploring the state-space of the system augmented with the observers. During exploration, if a global system state containing the *Error* location of the observer is reached, an error is reported. Additionally, the *Prune* locations are used to skip exploration of some specific paths of the state graph, and are useful in reducing the exploration time for big systems. This technique has been used in the verification of timing properties of modules of the robot controller [BGL+08].

4.2.2 Distributed Implementation

The operational semantics of BIP presented in section 2.2 is based on the notion of global stable states of the system, i.e., a complete stable state is required to trigger any interaction in a composite component. This is known as the global state semantics. In this section, we provide a distributed implementation method for systems in BIP, based on a partial state semantics where the assumption of global stability is relaxed. This implementation ensures a better usage of resources in a parallel execution environment while completely preserving the observational semantics of the system.

Partial State Semantics

The notion of partial state semantics as a basis for distributed implementation for BIP models has been introduced in [BBBS08].

The partial state semantics is based on a straightforward generalization of global state semantics where interactions are allowed to fire as soon as only the involved components are stable. Nonetheless, as the following example will show, this condition alone is too weak and the two models are not observationally equivalent in general.

Example 4.5.

Consider a composite component consisting of four atomic components A, B, C, D each one offering cyclically an interaction through ports a, b, c, d followed respectively by the internal execution of functions f_a, f_b, f_c, f_d (Figure 4.6). The glue consists of three rendez-vous connectors $\Gamma = \{ab, bc, cd\}$ and priorities $\Pi = \{ab \prec bc,\ cd \prec bc\}$.

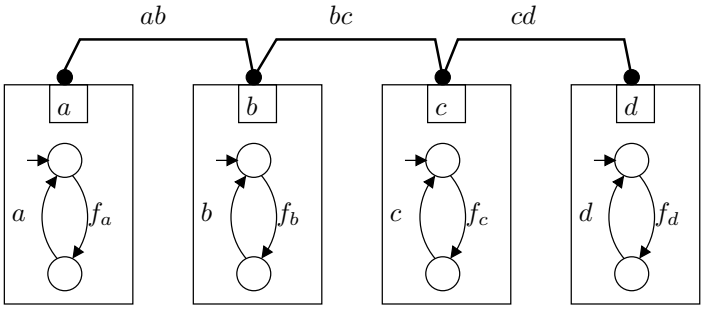


Figure 4.6: Global State vs Partial State Semantics

Using the global state semantics, this system executes forever the interaction bc. Consider now the corresponding partial state semantics where interactions are executed as soon as the involved components are stable. For this semantics, it is possible to execute the sequence $f_a.f_b.ab.(f_a.cd.f_c.f_b.ab.f_d)^\omega$ which goes through states never enabling the interaction bc.

The above example motivates the definition of partial state semantics based on global state semantics where the rule C_3 is replaced by rule C_3' given below. In addition to stability of the participating components, the premises of this rule include an oracle, a predicate parameterized by the priorities of the initial BIP model. The oracle characterizes the partial

states from which an interaction tree can be safely executed: if an interaction tree a^t can be inhibited by another interaction tree b^t, then a^t cannot be executed if the system has some internal evolution leading to a state enabling b^t. 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 stable components. The worst performing oracle is the one allowing interaction only when all the components are stable. For this oracle partial and global state semantics coincide.

$$[C_3'] \quad \frac{\begin{array}{c} \mathit{fireable}_C(q, a^t, q') \\ \forall p_i \in a^t \cap P_i.\ \ \mathit{stable}_C^{(i)}(q) \qquad \mathit{oracle}(q, a^t, \Pi) \\ \forall (a^t \prec_g b^t) \in \Pi.\ \left(\begin{array}{c} g(q) = \mathit{true} \\ \Rightarrow \\ \neg \mathit{fireable}_C(q, b^t, -) \end{array} \right) \end{array}}{q \xrightarrow[C]{a^t} q'}$$

where $\mathit{stable}_C^{(i)}((q_j)_{j\in J}) = \neg \left(\exists q_i'.\ q_i \xrightarrow[C_i]{\beta} q_i' \ \vee\ \exists a_i^t.\ q_i \xrightarrow[C_i]{a_i^t} q_i' \right)$

and $\mathit{oracle}(a^t, q, \Pi)$ is any predicate that implies, for each $a^t \prec_g b^t \in \Pi$ one of the following:

1. g evaluates to *false* on a stable subset of q:
   $\exists K \subseteq J.\ \ (\forall k \in K.\ \ \mathit{stable}_C^{(k)}(q)\ \wedge\ g(q_K) = \mathit{false})$
2. b^t is disabled by at least one stable component of q:
   $\exists p_i \in b^t \cap P_i.\ \left(\mathit{stable}_C^{(i)}(q) \wedge \neg(\exists q_i'.\ q_i \xrightarrow[C_i]{p_i(v_i^{up}/v_i^{dn})} q_i') \right)$
3. g evaluates to *true* on a stable subset of q and the full support of b^t is stable:
   $\exists K \subseteq J.\ \ (\forall k \in K.\ \ \mathit{stable}_C^{(k)}(q)\ \wedge\ g(q_K) = \mathit{true})\ \wedge\ \forall p_i \in b^t \cap P_i.\ \ \mathit{stable}_C^{(i)}(q)$

The following theorem proven in [BBBS08] provides 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.

Theorem 4.3.

If sub-components do not contain diverging internal computations, the global state semantics and the partial state semantics are observationally equivalent.

We will now define several oracles for the system 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.

dynamic oracle: The dynamic oracle implements a slightly strengthened version of conditions (1) and (3) from the oracle definition. In contrast to that conditions, it does not allow partial evaluation of guards. It requires the stability of all the components in the support set in order to evaluate the guard;

static oracle: The static oracle implements an even stronger condition. It does not allow either partial evaluation of guards nor partial evaluation of the non-fireability of interactions. That is, the condition (2) is also strengthened and requires that all the components involved in b^t to be stable, regardless their moves.

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 $oracle_{lazy}(q, a^t, \Pi) \iff stable_C(q)$

Example 4.6.

Consider the composite component presented in figure 4.7. There are five components A, B, C, D and E. Consider that the first two are stable, waiting for the interaction ab, whereas the last three are unstable, performing some internal computation.

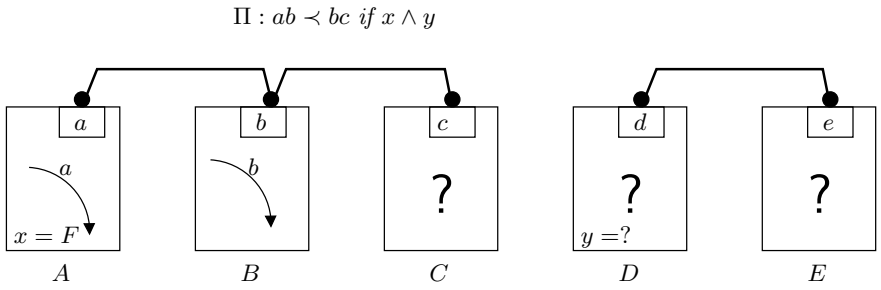


Figure 4.7: Example for oracles

According to the most liberal definition of oracle, the interaction ab is allowed to take place immediately. In fact, the priority guard $x \wedge y$ is false because x is known to be false in a stable component. However, the dynamic oracle does not authorize this interaction. It postpones it until either the D component become stable (and then, the guard can be completely evaluated to false) or the C component become stable and moreover, it disables the interaction bc. The static oracle requires that both C and D become stable, that is, complete stability of all components involved in the guard and conflicting interactions. Finally, the lazy oracles requires that all the components are stable e.g, including E.

Centralized Engine

The principle of distributed implementation with centralized engine is illustrated in figure 4.8.

The partial state semantics is enforced by a centralized engine which coordinates the parallel execution of the atomic components. Each atomic component is assigned to a different thread (or processor), the engine being assigned a thread as well. Each atomic component performs

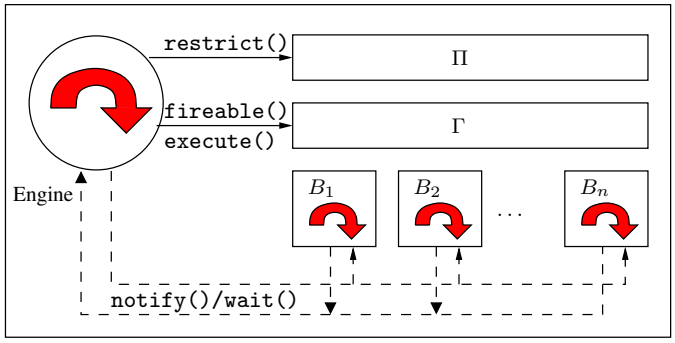


Figure 4.8: Architecture of distributed implementation

its computations locally and then, when it reaches a stable state, it notifies the engine about the ports on which it is willing to interact. The engine is parameterized by an oracle. Iteratively, the engine computes feasible interactions available on stable components. Then, if such interactions exist and the oracle allows them, the engine selects one for execution and notifies the involved components. The algorithms for respectively atomic components and engine are sketched below.

```
P_i := initialize();
do forever
   notify(E, P_i);
   wait(E, p_i);
   P_i := execute(p_i);
done
```

```
foreach j in 1,n do
   P_j := ⊥;
do forever
   wait(B_k, P_k);
   do forever
      A := compute-fireable(Γ, P_1, ..., P_n);
      A^max := restrict-priorities(Π, A, 𝒪);
      if A^max is not empty then
         choose a = (p_i)_{i∈I} in A^max;
         execute-data-transfer(a);
         foreach i in I do
            notify(B_i, p_i);
            P_i := ⊥;
      else
         break;
      fi
   done
   if forall j = 1,n.  P_j ≠ ⊥ then
      deadlock();
      stop;
   fi
done
```

A relevant measure of the performance of a distributed implementation is the degree of parallelism over time, that means, the number of simultaneously executing atomic components.

We analyze the relationship between the degree of parallelism and parameters of the system. To simplify the analysis, consider an abstract system consisting of n atomic components always able to interact through their ports. We distinguish the following cases, illustrated in Figure 4.9:

- For a distributed 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. However, it is worth mentioning that such an implementation without priorities is sound in general iff there are no priorities applied in the system;

- 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.

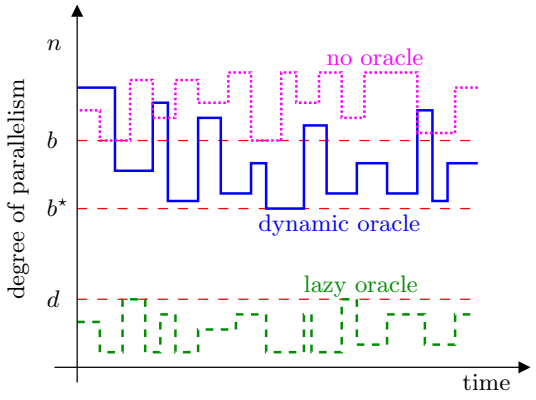


Figure 4.9: Performance analysis

Case Study: the Hypercube Adder

This case study 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 4.11 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 4.10. 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 node.

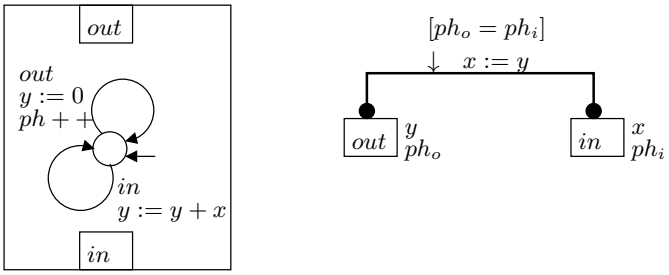


Figure 4.10: The adder node and out/in connectors

In pipelined execution, 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 4.12. 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.

Decentralized Engine

We have proposed a distributed implementation for BIP, but it still has a single centralized engine. As the previous experiments show, this implementation behaves very well in practice and most likely it outperforms the sequential implementation. Nevertheless, the centralized architecture is an important drawback and communication with one single engine for every interaction may lead to poor performance in many situations. Still, there is no magic solution for *breaking* the engine. The BIP semantics requires a global control to deal with priorities

Π : $\forall i, j, k.\ N_i.out \cdot N_j.in \prec N_k.out \cdot N_i.in$ if $N_i.ph = N_k.ph$

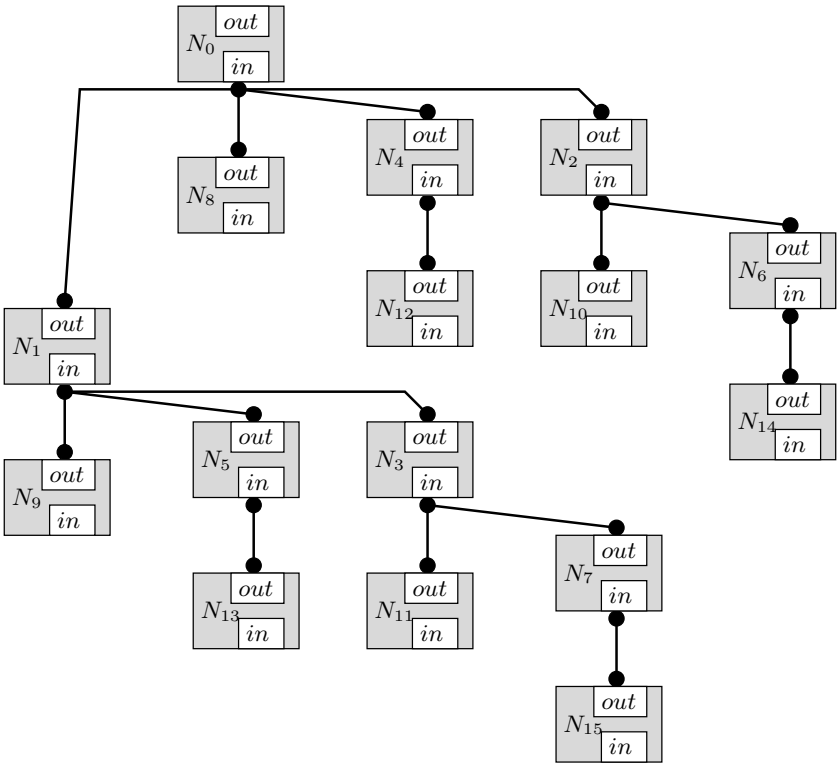


Figure 4.11: The hypercube architecture

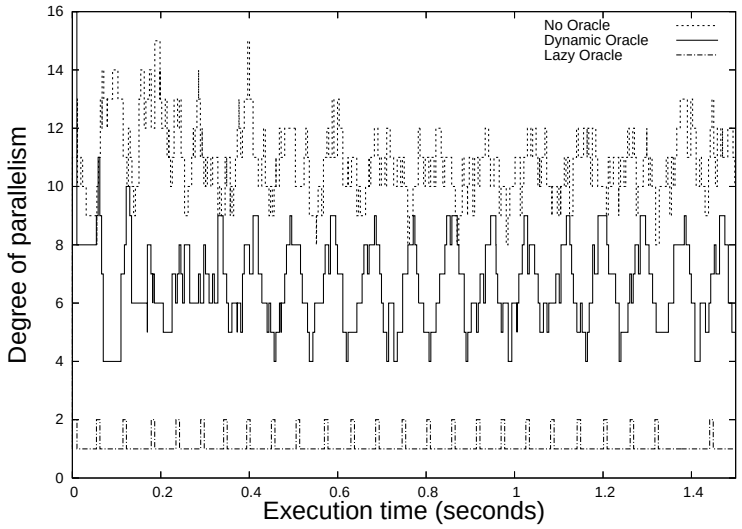


Figure 4.12: Degree of parallelism measured for the Parallel Adder

between the interactions, in addition to handling conflicts arising due to local choices offered by the components.

We start to investigate alternative solutions, with more or less centralized architectures. For sake of clarity, we define the architecture of the decentralized implementation in BIP as well, that is, by considering that engine(s) responsible of the execution of connectors are particular BIP components and the glue is specific to communication primitives available on the execution platform (e.g, asynchronous message passing).

The spectrum of possibilities for decentralized implementation is very large as shown in figure 4.13.

The first solution, illustrated in figure 4.13(a) correspond to the centralized engine detailed earlier.

The second solution, illustrated in figure 4.13(b) and detailed in [Qui09], consists in computing statically a partition of connectors according to their potential conflicts and then, implementing every conflicting class on a dedicated engine. This method works well as long as conflicts can be determined statically with enough precision in order to allow a non-trivial partitioning of the overall set.

Example 4.7.

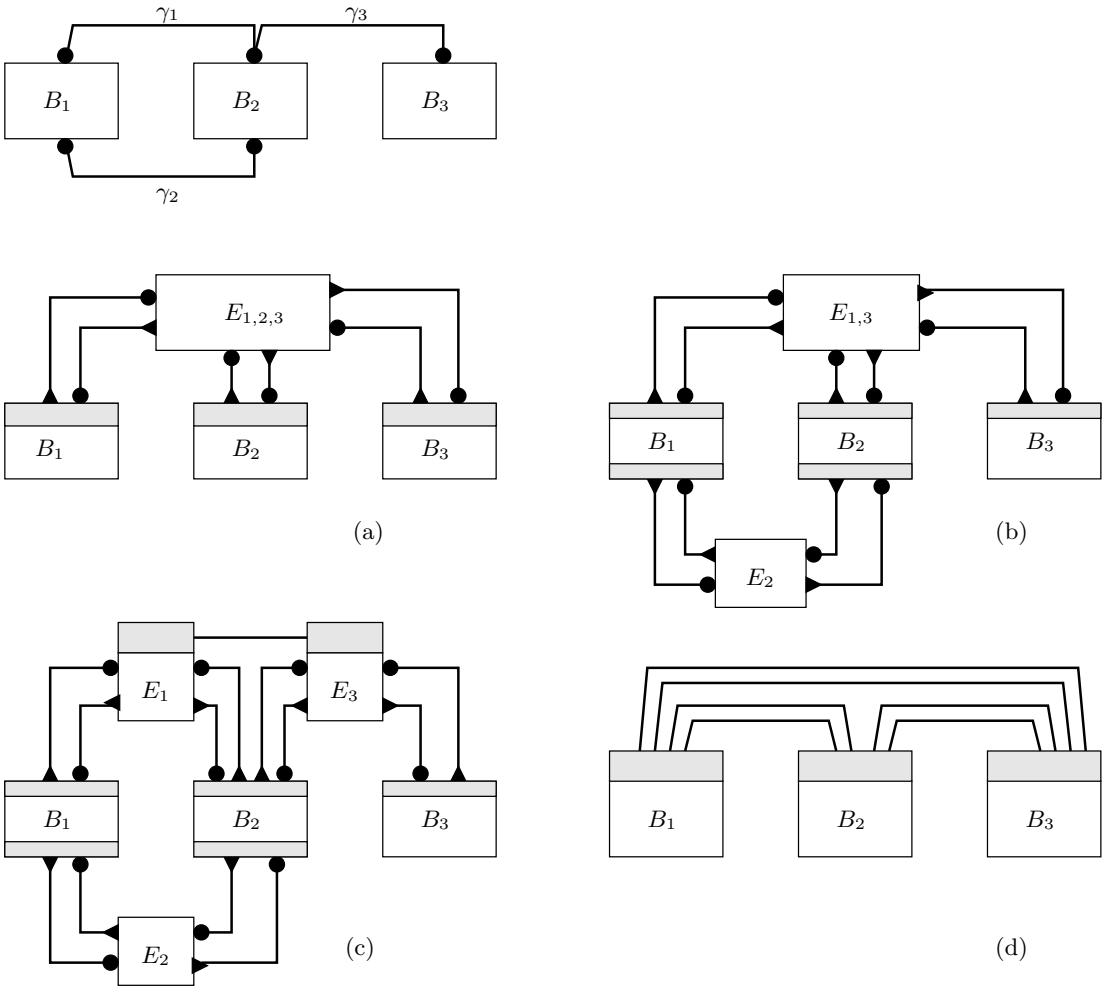


Figure 4.13: The spectrum of decentralized implementation

The parallel adder presented earlier has potential conflicts between all connectors. This happens because the atomic behavior of every node is defined by a single state automaton, as shown in figure 4.10, where both ports in and out are simultaneously enabled. However, this behavior can be re-modeled such that every node strictly alternates a fixed number of in and one out moves. In this case, there are no conflicts due to internal choices, but only because several connectors share the same in port. Every such group of connectors can be handled by a dedicated engine.

The third solution, illustrated in figure 4.13(c) takes a more radical approach to enforce distribution. It consists in implementing every connector on one distinct engine, and in addition, adding an extra coordination protocol to solve potential conflicts and to enforce dynamic priorities. Such coordination protocols have been investigated in the literature e.g., precisely for the distributed implementation of multi-party interactions [PCT04], or in general for solving graph related problems such as finding independents sets or maximal matchings. Nevertheless, they are not entirely satisfactory for BIP as such because, in particular, none of these approaches handle dynamic priorities amongst interactions.

Finally, an extreme solution would be to include engine specific code as well as the coordination protocols within atomic components, as shown in figure 4.13(d).

In all cases, an orthogonal problem is finding the best implementation for a given execution platform e.g., network on chip or multi-core architecture with specific characteristics. Computing the deployment which enables maximal performance (e.g., increased degree of parallelism) with reduced coordination overhead (e.g, number of exchanged coordination messages) is subject of future work.

4.3 Optimization

Another wellknown way to increase the performance of an implementation is to use compile-time optimizations. In this section, we will describe such an optimization that reduces the overhead of the engine code by reducing the overall number of atomic components and by moving some of glue and coordination code into atomic components. The basic idea is to statically compose several atomic components into a single atomic component using the corresponding glue. Under particular conditions, this transformation completely preserves the BIP semantics.

4.3.1 Composition of Atomic Behaviour

This optimization consists in the static composition of (subsets of) atomic components within composite components, under some restrictions on their behaviour and glue. The result is a new composite component, with fewer atomic components and simplified glue but with identical semantics with the initial composite component. This operation is an adaptation of the usual syntactic parallel composition with synchronization on Petri nets.

Formally, the transformation is defined as follows. Let $C = (P_C, X_C, gl, (B_j)_{j\in J} \cup (B_k)_{k\in K})$ be a flat composite component with glue $gl = \langle \Pi, \Gamma \rangle$. Let $B_j = (P_j, X_j, (L_j, T_j, F_j))_{j\in J}$, be a subset of the atomic subcomponents of C. The subcomponents $(B_j)_{j\in J}$ can be statically composed, while preserving the semantics of C, if the following restrictions are met:

- none of the atomic components $(B_j)_{j\in J}$ and $(B_k)_{k\in K}$ contains internal transitions;
- the interaction model Γ is flat and for every every connector $\gamma = (P, p_0, A)$ of Γ, either
  1. is local, that is $p_0 \notin P_C$, and uses exclusively ports of $(B_j)_{j\in J}$, that is $P \subseteq \cup_{j\in J} P_j$ or,
  2. is exported, that is $p_0 \in P_C$, and uses at most one port of $(B_j)_{j\in J}$, that is $|P \cap \cup_{j\in J} P_j| \leq 1$
- the priority model Π is empty.

Let us remark that the second restriction holds trivially for closed (system level) compositions, that is where the set P_C of exported ports is empty. Given the restrictions above, the set of internal connectors of C and the set of atomic components $(B_j)_{j\in J}$ can be replaced inside C by a single atomic component $B_0 = (P_0, X_0, N_0)$ defined as follows:

- $P_0 = \{p \in \cup_{j\in J} P_j \mid \exists \gamma = (P, p_0, A).\ p_0 \in P_C \ \wedge \ p \in P\}$ is the set of ports, that is, the ones used on interactions on exported connectors;
- $X_0 = \cup_{j\in J} X_j$ is the set of variables,
- the Petri net $N_0 = (L_0, T_0, F_0)$ is defined as follows:
  - $L_0 = \cup_{j\in J} L_j$ is the set of places,
  - $T_0 = \{\tau \mid \tau \in \cup_{j\in J} T_j,\ p_\tau \in P_0\} \ \cup$
    $\{\langle a, (\tau_i)_{i\in I}\rangle \mid \exists \gamma = (P, p_0, A) \in \Gamma.\ p_0 \notin P_C \ \wedge \ (p_{\tau_i})_{i\in I} = a \in A\}$
    The set of transitions contains (1) transitions from atomic components involving ports in P_0 and (2) transitions corresponding to sets of interacting transitions from atomic components on local connectors. For transitions of the first category, the associated guards and update functions are kept as they were in the local components. For every transition $\tau = \langle a, (\tau_i)_{i\in I}\rangle$ of the second category, the port p_τ is $\perp$ and the associated guard g_τ and function f_τ are defined by:
    - $g_\tau = \bigwedge_{i\in I} g_{\tau_i} \wedge g_a$
    - $f_\tau = (\exists x_{p_0}.\exists x'_{p_0}.\ x_{p_0} = x'_{p_0} \wedge a\uparrow \wedge a\downarrow) \circ \bigwedge_{i\in I} f_{\tau_i}$

    That means, the new guard is obtained as the conjunction of all local guards plus the guard of the interaction taken. Also, the new update function is obtained as the sequential composition of the interaction transfer (upward, downward) and the local update functions, in an arbitrary order. In fact, the order is completely irrelevant because the composed transitions come from different components and are working on disjoint sets of variables.
  - $F_0 = \{(l, \langle a, (\tau_i)_{i\in I}\rangle) \mid (l, \tau_i) \in F_i\} \ \cup \ \{(\langle a, (\tau_i)_{i\in I}\rangle, l) \mid (\tau_i, l) \in F_i\} \cup$
    $\{(l, \tau) \mid (l, \tau) \in \cup_{j\in J} F_j\} \ \cup \ \{(\tau, l) \mid (\tau, l) \in \cup_{j\in J} F_j\}$
    That is, the flow constraints are preserved as such for transitions of the first category and *composed* for interacting transitions.

The following proposition establishes formally the correctness of our transformation.

Proposition 4.4.

Let $C = (P_C, X_C, gl, (B_j)_{j\in J} \cup (B_k)_{k\in K})$ a composite component satisfying the restrictions above.

Let B_0 be the result of composition of $(B_j)_{j\in J}$ and let $C_0 = (P_C, X_C, gl_0, B_0 \cup (B_k)_{k\in K})$.

Then $\mathcal{S}_C = \mathcal{S}_{C_0}$.

Example 4.8.

The figure 4.14 illustrates the principle of the static composition for two simple atomic components. Iteratively, every atomic component interacts twice through the in_j port then once through the out_j port. If they are connected serially as shown in the figure 4.14 (left), their composition can be performed statically and leads to the atomic component shown in figure 4.14 (right). This component is fully equivalent to the composition: iteratively, it interacts four times on the in_1 port before interacting once on the out_2 port.

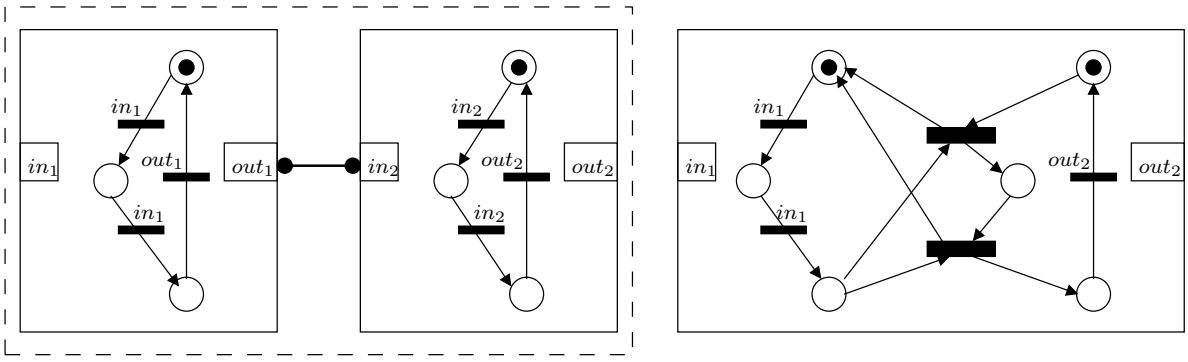


Figure 4.14: An example of atomic behaviour composition

In contrast to flatenning transformations described in section 4.1, component composition may lead to an exponential blowup of the number of transitions in the resulting Petri net. This situation may happen if the same interaction can be realized by combining different transitions from each one of the involved components. For instance, the interaction p_1p_2 can give rise to four transitions in the resulting Petri net if there are two transitions labeled by p_1 and p_2 in the synchronizing components. Nevertheless, in practice we are rarely faced to this situation, as in atomic components, each port occurs at most in one transition (as in examples shown hereafter). In this case, the resulting Petri net has as many transitions as interactions between the composed components.

Case Study: the Mpeg4 Encoder

In the context of an industrial project, we have componentized in BIP an Mpeg4 encoder written in C by an industrial partner. The aim of this work was to evaluate gains in scheduling and quality control of the componentized program. The results were quite positive regarding quality control [CFSS08, CFLS05] but the componentized program has been almost two times slower than the handwritten C program. To overcome this performance issue, we have used the optimization above and then we generate an equivalent C implementation from the final atomic BIP component.

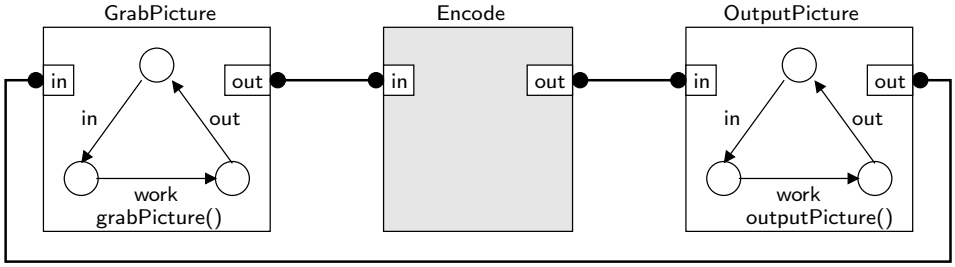


Figure 4.15: Mpeg4 encoder structure

The initial BIP model for the encoder is shown in figure 4.16. It consists of 11 atomic components, and 14 connectors. It uses the data and the functions of the initial handwritten C program. At top level, the model is composed of two atomic components and one composite component. The atomic component GrabFrame gets a frame and produces macroblocks (each frame is split into *Max* macroblocks of 256 pixels). The atomic component OutputFrame produces an encoded frame. The composite component Encode consists of 9 atomic components and the corresponding connectors. It encodes macroblocks produced by the component GrabFrame.

Figure 4.17 shows the execution times for the initial handwritten C code, for the componentized BIP model and for the single component BIP model. Notice that the latter and the handwritten C code have almost the same execution time. However, the advantages from the componentization of the handwritten code are multiple. In addition to increased clarity, the componentized BIP model has been rescheduled as shown in [CFSS08, CFLS05] so as to meet given timing requirements. Table 4.1 gives the size of the handwritten C code, the structured BIP model, as well of the generated C code from respectively, the structured BIP model $C^{(1)}$ and the single component BIP model $C^{(2)}$. The time taken by the BIP2BIP tool to flatten and to perform the static composition is negligible (less than 1 second).

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

Table 4.1: Code size in lines-of-code (loc) for Mpeg4 Encoder

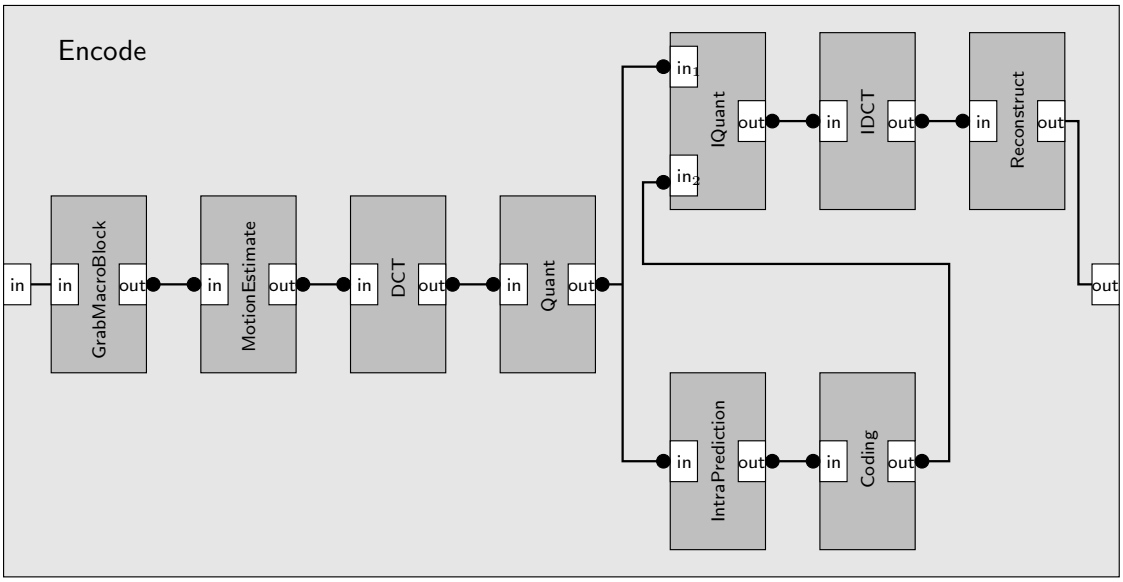


Figure 4.16: Encode component structure

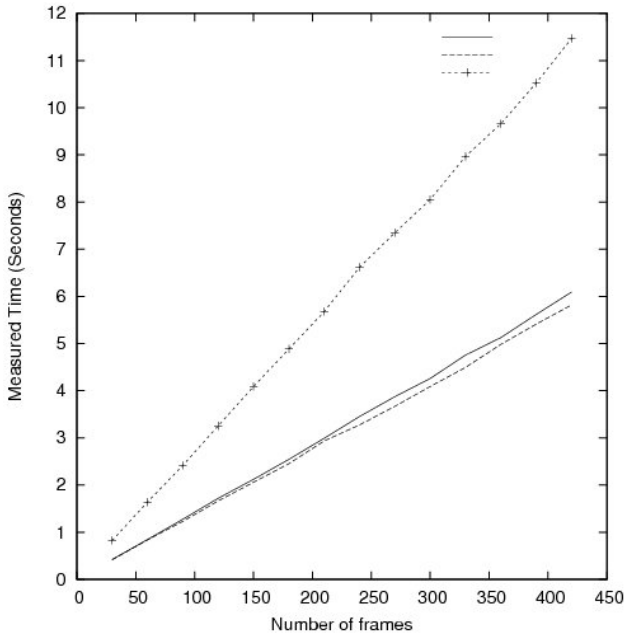


Figure 4.17: Execution time for the Mpeg4 Encoder

Chapter 5

System Validation

We present hereafter fully automatic methods for analysis and validation of BIP systems. We begin with an iterative and compositional method for generation of invariants. This method derives increasingly stronger invariants of a particular form, that is, conjunction of local invariants for atomic components and interaction invariants characterizing the composition glue. Such invariants are used to validate safety properties and in particular deadlock-freedom.

A second validation method we experiment is explicit state exploration or model-checking. We recall here the main principles for building an efficient state-space exploration tool and we illustrate them on the IF toolbox. This toolbox handle successfully a significant fragment of BIP where the interaction models are restricted to asynchronous message passing and shared variables.

Finally, we provide some key idea about our ongoing work on compositional generation of abstractions for timed systems.

5.1 Compositional Generation of Invariants

Compositional verification techniques [CLM89, KV98] as well as assume-guarantee techniques [AH96, AL95, CJ88, GL94, McM97, Pnu85, Sta85] have been invented to cope with state explosion in concurrent systems. The general idea is to apply divide-and-conquer approaches to infer global properties of complex systems from properties of their components. Separate verification of components limits state explosion. Nonetheless, components mutually interact in a system and their behavior and properties are interrelated. This is a major difficulty in designing scalable compositional techniques.

We present hereafter the compositional method for verification of BIP systems introduced in [BBNS08]. This method is based on the use of two kinds of invariants: *component invariants* which are over-approximations of components' reachability sets and *interaction invariants* which are constraints on the states of components involved in interactions. Interaction invariants are obtained by computing traps and locks of finite-state abstractions of the verified system. The method is applied in particular for deadlock verification. It has been implemented in D-Finder [BBNS09], an interactive tool that takes as input BIP systems and applies proof strategies to eliminate potential deadlocks by computing increasingly stronger invariants.

We consider flat composite components $C = (P, X, gl, (B_1, \ldots, B_n))$ obtained by composing a set of atomic components $B_1, ..., B_n$ by using a flat glue $gl = \langle \Pi, \Gamma \rangle$.

To prove a global invariant Φ for C, we use the following rule:

$$[] \quad \frac{\{B_i \models \Box\Phi_i\}_{i=1,n} \quad \Psi \in \mathcal{II}(gl, \{\Phi_i\}_{i=1,n}) \quad (\bigwedge_{i=1}^{n} \Phi_i) \wedge \Psi \Rightarrow \Phi}{C \models \Box\Phi}$$

where

- $B_i \models \Box\Phi_i$ means that Φ_i is an invariant of the atomic component B_i,
- $\Psi \in \mathcal{II}(gl, \{\Phi_i\}_{i=1,n})$ means that Ψ is an *interaction invariant* of C computed from the glue gl and invariants Φ_i known for atomic components.

Let us notice that the rule above is an instance of the compositionality rule given in section 2.1.

There are two key issues for the implementation of this rule respectively, finding component invariants Φ_i and finding interaction invariants $\mathcal{II}$. We illustrate below effective algorithmic methods for solving both issues. First, we provide a method for computing atomic component invariants based on static analysis of their behavior. Then, we provide a general method for computing interaction invariants $\mathcal{II}(gl, (\Phi_i)_{i=1,n})$ from the glue gl and a given set of component invariants Φ_i.

Example 5.1.

As running example, we consider the following case study taken from [ACH+95]. This case study is about controlling the coolant temperature in a reactor tank by moving two independent refrigerating rods. The goal is to maintain the coolant between the temperatures θ_m and θ_M. When the temperature reaches its maximum value θ_M, the tank must be refrigerated with one of the rods. The temperature rises at a rate v_r and decreases at rate v_d. A rod can be moved again only if T time units have elapsed since the end of its previous movement. If the temperature of the coolant cannot decrease because there is no available rod, a complete shutdown is required.

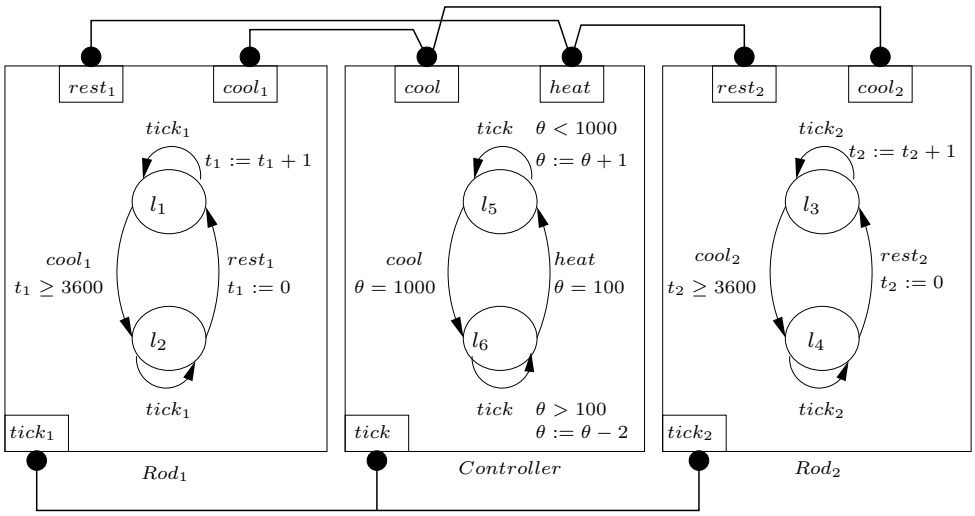


Figure 5.1: Temperature Control System

In figure 5.1 we provide a discretized model of the system in BIP. It is decomposed into three atomic components: the Controller and two components Rod_1, Rod_2 modeling the rods. We take $\theta_m = 100°$, $\theta_M = 1000°$, $T = 3600$ seconds. Furthermore, we assume that $v_r = 1°/s$ and $v_d = 2°/s$. The Controller has two control locations $\{l_5, l_6\}$, a variable θ, three ports $\{tick, cool, heat\}$ and four transitions: 2 loop transitions labeled by tick which increase or decrease the temperature as time progresses and 2 transitions triggering moves of the rods. The components Rod_1 and Rod_2 are identical, up to the renaming of states and ports. Each one has two control locations and four transitions: two loop transitions labeled by tick and two transitions synchronized with transitions of the Controller. The components are composed by using the following set of interactions, indicated by connectors in the figure: $\{tick, tick_1, tick_2\}$, $\{cool, cool_1\}$, $\{cool, cool_2\}$, $\{heat, rest_1\}$, $\{heat, rest_2\}$.

In this model, complete shutdown corresponds to a deadlock. We want to generate invariants in order to prove deadlock-freedom of the system when started in the ideal state where the temperature in the reactor is as low as possible, and the two rods are ready to be used:
$Init \;=\; at\_l_5 \wedge (\theta = 100) \wedge at\_l_1 \wedge (t_1 = 3600) \wedge at\_l_3 \wedge (t_2 = 3600)$

5.1.1 Invariants for Atomic Components

Let $B = (P, X, (L, T, F))$ be an atomic component, fixed.

For the sake of clarity, we made several simplifying assumptions about the behavior. First of all, we consider that behavior is defined using plain automata and not general Petri nets. That is, we exclude concurrency between transitions within atomic components. Second, we consider that no data is exchanged through ports i.e., we are considering only *pure* control interactions. Finally, we consider that there are no internal transitions, that is, all transitions trigger ports belonging to the interface of the component.

For a location l, we use the predicate at_l which is *true* iff the behavior is at location l. A

state predicate Φ is a boolean expression involving location predicates and predicates on X. Any state predicate can be put in the form $\bigvee_{l \in L} at\_l \wedge \varphi_l$. Notice that predicates on locations are disjoint and their disjunction is true.

We recall hereafter the definition of the *Post* predicate transformer allowing to compute successors of global states represented symbolically by state predicates. Given a state predicate $\Phi = \bigvee_{l \in L} at\_l \wedge \varphi_l$, we define

$$Post(\Phi) = \bigvee_{l \in L} (\bigvee_{(l,\tau),(\tau,l') \in F} at\_l' \wedge Post_\tau(\varphi_l))$$

where

$$Post_\tau(\varphi)(X) = (\exists X'.g_\tau(X') \wedge f_\tau(X', X) \wedge \varphi(X')).$$

Equivalently, we have that $Post(\Phi) = \bigvee_{l \in L} at\_l \wedge (\bigvee_{(l',\tau),(\tau,l) \in F} Post_\tau(\varphi_{l'}))$ which allows computing $Post(\Phi)$ by forward propagation of the assertions associated with control locations in Φ.

We define in a similar way the Pre_τ predicate transformer for a transition τ:

$$Pre_\tau(\varphi)(X) = \exists X'.g_\tau(X) \wedge f_\tau(X, X') \wedge \varphi(X')$$

Definition 5.1 (invariant)**.**

Let $\langle B, \Phi_{init} \rangle$ be an initialized atomic component. A predicate Φ is

- *an inductive invariant of $\langle B, \Phi_{init} \rangle$ iff $(\Phi_{init} \vee Post(\Phi)) \Rightarrow \Phi$,*
- *an invariant of $\langle B, \Phi_{init} \rangle$ iff there exists an inductive invariant Φ_0 such that $\Phi_0 \Rightarrow \Phi$.*

Notice that invariants are over-approximations of the set of the reachable states of B when started at Φ_{init}. Let us recall the obvious facts if Φ_1, Φ_2 are two invariants of an atomic component B then $\Phi_1 \wedge \Phi_2$, $\Phi_1 \vee \Phi_2$ are also invariants of B. We compute sequences of inductive invariants for atomic components by using the proposition below.

Proposition 5.1.

Given an initialized atomic component $\langle B, \Phi_{init} \rangle$, the following iteration defines a sequence of increasingly stronger inductive invariants:

$$\Phi_0 = true \quad \Phi_{i+1} = \Phi_{init} \vee Post(\Phi_i)$$

In our heuristic, we use different strategies for producing invariants. We usually iterate until we find *deadlock-free invariants*. Their use guarantees that global deadlocks are exclusively due to synchronization.

A key issue is the efficient computation of component invariants as the precise computation of *Post* requires quantifier elimination. An alternative to quantifier elimination is to compute over-approximations of *Post* based on syntactic analysis of the predicates. In this case, the obtained invariants may not be inductive.

We give here a very brief description of a syntactic technique used for approximating $Post_\tau$ for a fixed transition τ. A more detailed presentation, as well as much elaborated techniques for generating invariants for sequential behavior are given in [BL99].

Consider a transition τ of B. Assume that its guard is of the form $g_\tau(Y)$ and the associated update function f_τ is of the form $Z_1' = e_\tau(U) \wedge Z_2' = Z_2$ where $Y, Z_1, Z_2, U \subseteq X$ and $\{Z_1, Z_2\}$ is a partition of X.

For an arbitrary predicate φ find a decomposition $\varphi = \varphi_1(Y_1) \wedge \varphi_2(Y_2)$ such that $Y_2 \cap Z_1 = \emptyset$ i.e., which has a conjunct not affected by the update function f_τ. We apply the following rule to compute over-approximations $Post^a_\tau(\varphi)$ of $Post_\tau(\varphi)$

$$Post^a_\tau(\varphi) = \varphi_2(Y_2) \wedge \left\{ \begin{array}{cc} g_\tau(Y) & \text{if } Z_1 \cap Y = \emptyset \\ true & \text{otherwise} \end{array} \right\} \wedge \left\{ \begin{array}{cc} Z_1 = e_\tau(U) & \text{if } Z_1 \cap U = \emptyset \\ true & \text{otherwise} \end{array} \right\}$$

Proposition 5.2.

If τ and φ are respectively a transition and a state predicate as above, then $Post_\tau(\varphi) \Rightarrow Post^a_\tau(\varphi)$.

Example 5.2.

Using static analysis techniques we are able to automatically compute the invariants Φ_1, Φ_2, Φ_3 respectively for Rod_1, Rod_2 and Controller:

$$\begin{array}{rcl} \Phi_1 & = & (at\_l_1 \wedge t_1 \geq 0) \vee (at\_l_2 \wedge t_1 \geq 3600) \\ \Phi_2 & = & (at\_l_3 \wedge t_2 \geq 0) \vee (at\_l_4 \wedge t_2 \geq 3600) \\ \Phi_3 & = & (at\_l_5 \wedge 100 \leq \theta \leq 1000) \vee (at\_l_6 \wedge 100 \leq \theta \leq 1000) \end{array}$$

5.1.2 Invariants for Flat Interaction Models

The notion of invariant defined from atomic components is naturally lifted to composite components. As before, invariants are predicates over-approximating the set of reachable states of composite components.

Any invariant of an atomic component is an invariant of the composite component as well. However, in this section, we are mainly interested in finding *interaction invariants*, that are predicates relating control and data variables from different atomic components according to the composition glue.

For the sake of clarity, we made few simplifications about the glue. First of all, we consider only flat interaction models. This is not really an important restriction, since we have a systematic way to flatten any hierarchical glue. Second, as for atomic components, we did not consider data transfer but only pure control interactions. Finally, we are ignoring priorities. In principle, by doing so, we increase the non-determinism and the set of reachable states, for any model. Therefore, any invariant obtained on the relaxed model, is also an invariant for the model where priorities are applied.

We first show how to compute interaction invariants for composite components $C = (P, X, gl, (B_1, \ldots, B_n))$ without data, that is, where the atomic sub-components B_i are finite transition systems. Then, we show how to deal with general transition systems extended with data by using abstraction.

Atomic Components without Data

Let $C = (P, \emptyset, gl, (B_1, \ldots, B_n))$ be a composite component where the glue $gl = \langle \Pi, \Gamma \rangle$ is flat and atomic subcomponents $B_i = (P_i, \emptyset, (L_i, T_i, F_i))_{i=1,n}$ are transition systems without data.

Definition 5.2 (Forward/Backward Interaction Sets)**.**

We define for a set of locations $L \subseteq \bigcup_{i=1}^{n} L_i$ *its forward interaction set* $L^\bullet$*:*

$$L^\bullet = \left\{ \langle a, \{\tau_i\}_{i\in I}\rangle \;\middle|\; \begin{array}{l} \forall i \in I.\ (\tau_i \in T_i) \ \wedge\ \exists i \in I.\ ({}^\bullet\tau_i = l \in L) \ \wedge \\ \exists \gamma \in \Gamma.\ \{p_{\tau_i}\}_{i\in I} = a \in A(\gamma) \end{array} \right\}$$

We define in a similar manner, its backward interaction set ${}^\bullet L$*:*

$${}^\bullet L = \left\{ \langle a, \{\tau_i\}_{i\in I}\rangle \;\middle|\; \begin{array}{l} \forall i \in I.\ (\tau_i \in T_i) \ \wedge\ \exists i \in I.\ (\tau_i^\bullet = l \in L) \ \wedge \\ \exists \gamma \in \Gamma.\ \{p_{\tau_i}\}_{i\in I} = a \in A(\gamma) \end{array} \right\}$$

That is, $L^\bullet$ (resp. ${}^\bullet L$) consists of sets of sub-component transitions involved in some interaction of γ in which at least one transition τ_i issued from (resp. going into) a location in L can participate (see figure 5.2).

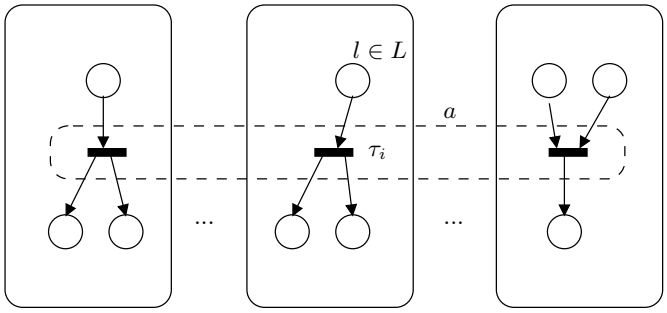


Figure 5.2: Forward interaction sets

The elements of ${}^\bullet L$ and $L^\bullet$ can also be viewed as the transitions of the Petri net obtained by static composition of C as defined in section 4.3. As for Petri nets, we can define the notions of traps and locks.

Definition 5.3 (Traps and Locks)**.**

A trap is a set L *of locations* $L \subseteq \bigcup_{i=1}^{n} L_i$ *such that* $L^\bullet \subseteq {}^\bullet L$

A lock is a set L *of locations* $L \subseteq \bigcup_{i=1}^{n} L_i$ *such that* ${}^\bullet L \subseteq L^\bullet$*.*

The following proposition expresses a well known characteristic property of traps and locks in Petri nets. First, if the initial state of C has some control location belonging to a trap then all its reachable states have some control location belonging to that trap. Second, if the initial state of of C has no control locations belonging to a lock, then none of its reachable states has a control location belonging to that lock.

Proposition 5.3.

Let $\langle C, \Phi_{init}\rangle$ *be an initialized composite component and let* $L \subseteq \bigcup_{i=1}^{n} L_i$ *be a set of locations.*

1. *if* L *is a trap which contains an initial location defined by* Φ_{init} *then* $\Phi = \bigvee_{l\in L} at\_l$ *is an interaction invariant of* $\langle C, \Phi_{init}\rangle$*,*

2. *if L is a lock which does not contain any initial location defined by Φ_{init} then $\Phi = \bigwedge_{l \in L} \neg at\_l$ is an interaction invariant of $\langle C, \Phi_{init} \rangle$.*

Traps and locks can be effectively computed using for instance the method of [Sif78] which characterizes them as solutions of a system of boolean implications.

Atomic Components with Data

We present hereafter a compositional method to compute interaction invariants for initialized composite components $\langle C, \Phi_{init} \rangle$ where $C = (P, X, gl, (B_1, \cdots, B_n))$ and $B_i = (P_i, X_i, N_i)$ are arbitrary atomic components with data. The key idea is to rely on data abstraction, as follows:

1. for every atomic component $B_i = (P_i, X_i, N_i)$ compute separately a finite state abstraction $B_i^\sharp = (P_i, \emptyset, N_i^\sharp)$. In principle, any predicate abstraction method can be used, however, we give a particular attention to the method of [BLO98a] and previously implemented in the InVeSt [BLO98b] tool because it offers several advantages. First, it allows to construct an abstract transition system, by abstracting each concrete transition separately. Second, it is parameterized by an abstraction function α_i, and therefore the abstraction can be made dependent on particular concrete predicates e.g., the atomic predicates occuring in the guards, or in invariants Φ_i of the concrete behavior;

2. compute the initial set of abstract states $\Phi_{init}^\sharp$ using the abstraction functions $(\alpha_i)_{i=1,n}$ chosen at previous step;

3. consider the abstract composite component $C^\sharp = (P_i, \emptyset, (B_1^\sharp, ..., B_n^\sharp))$. It is clear that $C^\sharp$ is a composite component without data, and moreover, it can be shown that $C^\sharp$ is an abstraction of C. Therefore, we can apply the previously described techniques to derive abstract interaction invariants $\Phi^\sharp$ for $C^\sharp$ given initial states $\Phi_{init}^\sharp$;

4. finally, obtain concrete interaction invariants Φ of C by concretizing the abstract invariants $\Phi^\sharp$. The concretization is indeed possible because of the particular abstraction method used at the first step. It amounts to rewriting back abstract variables (or abstract locations) as defined by the abstraction functions $(\alpha_i)_{i=1,n}$ chosen for atomic components.

This method is illustrated below on the running case study.

Example 5.3.

In order to compute interaction invariants for the Temperature Control System we need a finite-state abstraction. Figure 5.3 presents such an abstraction computed with respect to the local invariants Φ_1, Φ_2 and Φ_3 computed earlier. The abstraction function applied uses elementary predicates occuring in local invariants and is summarized in the table below.

$\phi_{11} = at\_l_1 \wedge t_1 = 0$	$\phi_{51} = at\_l_5 \wedge \theta = 100$	$\phi_{31} = at\_l_3 \wedge t_2 = 0$
$\phi_{12} = at\_l_1 \wedge t_1 \geq 1$	$\phi_{52} = at\_l_5 \wedge 101 \leq \theta \leq 1000$	$\phi_{32} = at\_l_3 \wedge t_2 \geq 1$
$\phi_{21} = at\_l_2 \wedge t_1 \geq 3600$	$\phi_{61} = at\_l_6 \wedge \theta = 1000$	$\phi_{41} = at\_l_4 \wedge t_2 \geq 3600$
$\phi_{22} = at\_l_2 \wedge t_1 < 3600$	$\phi_{62} = at\_l_6 \wedge 100 \leq \theta \leq 998$	$\phi_{42} = at\_l_4 \wedge t_2 < 3600$

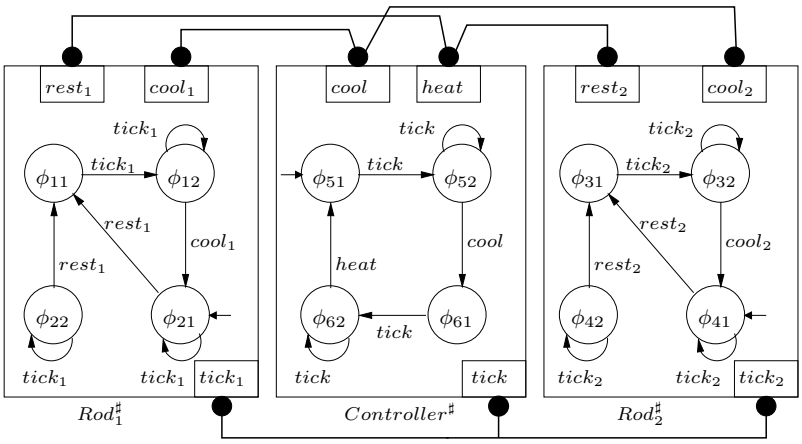


Figure 5.3: Abstract model for the Temperature Control System

The set of of minimal traps for the abstract system are:

$$
\begin{array}{lcl}
L_1 & = & \{\phi_{21}, \phi_{41}, \phi_{51}, \phi_{52}\} \\
L_2 & = & \{\phi_{11}, \phi_{12}, \phi_{21}, \phi_{31}, \phi_{32}, \phi_{41}\} \\
L_3 & = & \{\phi_{32}, \phi_{41}, \phi_{42}, \phi_{51}\} \\
L_4 & = & \{\phi_{11}, \phi_{12}, \phi_{31}, \phi_{32}, \phi_{61}, \phi_{62}\} \\
L_5 & = & \{\phi_{12}, \phi_{21}, \phi_{22}, \phi_{51}\}
\end{array}
$$

These traps lead to the following concrete interaction invariant:

$$
\begin{array}{lcl}
\Psi & = & ((at\_l_2 \wedge t_1 \geq 3600) \vee (at\_l_4 \wedge t_2 \geq 3600) \vee (at\_l_5 \wedge 100 \leq \theta \leq 1000)) \bigwedge \\
 & & ((at\_l_1 \wedge t_1 \geq 0) \vee (at\_l_2 \wedge t_1 \geq 3600) \vee (at\_l_3 \wedge t_2 \geq 0) \vee (at\_l_4 \wedge t_2 \geq 3600)) \bigwedge \\
 & & ((at\_l_3 \wedge t_2 \geq 1) \vee (at\_l_4) \vee (at\_l_5 \wedge \theta = 100)) \bigwedge \\
 & & ((at\_l_1 \wedge t_1 \geq 0) \vee (at\_l_3 \wedge t_2 \geq 0) \vee (at\_l_6 \wedge \theta = 1000) \vee (at\_l_6 \vee 100 \leq \theta \leq 998)) \bigwedge \\
 & & ((at\_l_1 \wedge t_1 \geq 1) \vee (at\_l_2) \vee (at\_l_5 \wedge \theta = 100))
\end{array}
$$

5.1.3 Application for Checking Deadlock-Freedom

We present an application of the method for checking deadlock-freedom of composite components.

Definition 5.4 (Enabled states)**.**

Let $C = (P, X, gl, (B_1, ..., B_n))$ be a composition of atomic components $B_i = (P_i, X_i, (L_i, T_i, F_i))$. The predicate Enabled characterizes the set of the states of C from which interactions are enabled, formally:

$$
\begin{array}{rcl}
Enabled & = & \bigvee_{\gamma \in \Gamma} \bigvee_{a \in A(\gamma)} enabled(a) \\
enabled(a) & = & \bigwedge_{i=1}^{n} \bigwedge_{p \in a \cap P_i} \bigvee_{\tau \in T_i} at\_{}^{\bullet}\tau \wedge g_\tau
\end{array}
$$

$enabled(a)$ characterizes all the states from which interaction a can be executed.

Obviously, whenever started in a given set of initial states Φ_{init}, the component is deadlock-free iff the predicate *Enabled* is an invariant for $\langle C, \Phi_{init} \rangle$.

Example 5.4.

The set of deadlock-free states for the Temperature Control System is characterized by the following predicate, extracted automatically from the interaction model:

$$\begin{aligned} \mathit{Enabled} \;=\; & (at\_l_5 \wedge \theta < 1000) \bigvee (at\_l_6 \wedge \theta > 100)) \bigvee \\ & ((at\_l_5 \wedge \theta = 1000) \wedge (at\_l_3 \wedge t_2 \geq 3600)) \bigvee \\ & ((at\_l_5 \wedge \theta = 1000) \wedge (at\_l_1 \wedge t_1 \geq 3600)) \bigvee \\ & ((at\_l_6 \wedge \theta = 100) \wedge at\_l_2) \bigvee \\ & ((at\_l_6 \wedge \theta = 100) \wedge at\_l_4) \end{aligned}$$

Using the local and interaction invariants computed earlier, we are ready to prove that Enabled is also an invariant and the system is deadlock free if the following implication hold:

$$\Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Psi \;\Rightarrow\; \mathit{Enabled}$$

Unfortunately, the implication above does not hold. In fact, one obtain that $\Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Psi \wedge \neg \mathit{Enabled}$ *is satisfiable and it is the disjunction of the following terms:*

1. $(at\_l_1 \wedge 1 \leq t_1 < 3600) \wedge (at\_l_3 \wedge 1 \leq t_2 < 3600) \wedge (at\_l_5 \wedge \theta = 1000)$
2. $(at\_l_1 \wedge 1 \leq t_1 < 3600) \wedge (at\_l_4 \wedge t_2 \geq 3600) \wedge (at\_l_5 \wedge \theta = 1000)$
3. $(at\_l_2 \wedge t_1 \geq 3600) \wedge (at\_l_3 \wedge 1 \leq t_2 < 3600) \wedge (at\_l_5 \wedge \theta = 1000)$

Each one of the above terms represents a family of possible potential deadlocks. There are two possibilities: either there are real deadlock states (reachable from the initial state) and belonging to these sets or there are no such deadlocks, but our invariants are too weak to prove deadlock freedom. In this example, the first situation happens, the system indeed contains real deadlock states and so, it is not surprising that we cannot prove deadlock freedom.

5.1.4 The D-Finder Tool

The techniques presented below have been implemented in a tool called D-Finder designed as shown in figure 5.4.

This tool takes as input a BIP system and progressively find and eliminate potential deadlocks. It basically works as follows. First, it constructs the predicate characterizing the set of deadlock states ($\neg \mathit{Enabled}$ generation module). Second, iteratively, it constructs increasingly stronger local invariants of components (Φ_i generation module) and using them, finer finite state abstractions and increasingly stronger global interaction invariants (Abstraction and Ψ generation module). Third, it checks deadlock freedom by checking satisfiability of $\bigwedge \Phi_i \bigwedge \Psi \bigwedge \neg \mathit{Enabled}$ (satisfiability module). If it succeeds, the system is proven deadlock-free, else it may continue or gives up, according to the user choice. For doing all this, D-Finder is connected with several external tools. It uses Omega [Tea96] for quantifier elimination and Yices [DdM06] for checking satisfiability of predicates. It is also connected to the state space exploration tool of the BIP platform, for finer analysis when the heuristic fails to prove deadlock-freedom.

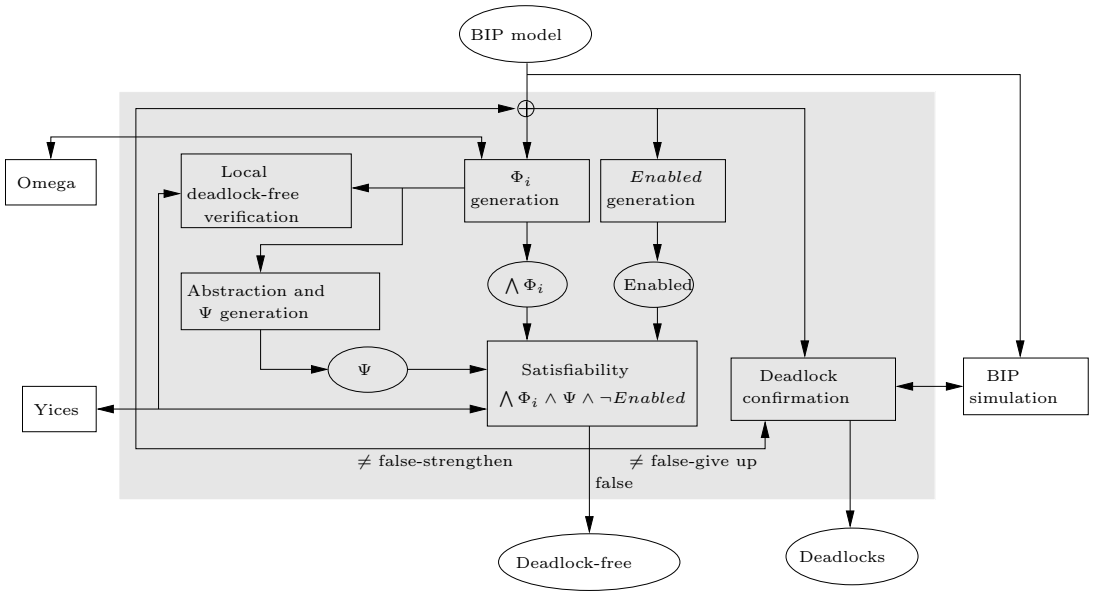


Figure 5.4: D-Finder Tool Architecture and Workflow

5.2 Model-Checking of Real-Time Systems

State-space exploration is one of the most successful techniques used for the analysis of concurrent systems and also the core component of any model-based validation tool (i.e, model-checker, test-generator, etc). Nevertheless, exploration is far from being trivial for heterogeneous systems that, use complex data, involve various communication mechanisms, mix descriptions of different levels of abstraction, and moreover, depend on time constraints. The solution we propose is an *open*, *modular* and *extensible* exploration platform designed to cope with the complexity and the heterogeneity of actual concurrent systems.

This section presents an overview on the IF toolset [BGM02, BGO+04, BGMO08] which is an environment for modelling and validation of asynchronous real-time systems. The toolset is built upon a rich formalism, the IF notation, allowing structured automata-based system representations. Moreover, the IF notation is expressive enough to support real-time primitives and extensions of high-level modelling languages such as SDL and UML by means of structure preserving mappings.

Although developed earlier, IF can also be seen as a BIP profile for asynchronous systems where interactions are restricted to asynchronous message passing and shared variables. Therefore, any BIP model fitting these particular restrictions can be equally interpreted as an IF model and analyzed with the IF toolset.

The core part of the IF toolset consists of a syntactic transformation component and an open exploration platform. The syntactic transformation component provides language level access to IF descriptions and has been used to implement static analysis and optimization techniques. The exploration platform gives access to the graph of possible executions. It has been connected to different state-of-the-art model-checking and test-case generation tools.

5.2.1 An Open and Modular Exploration Platform

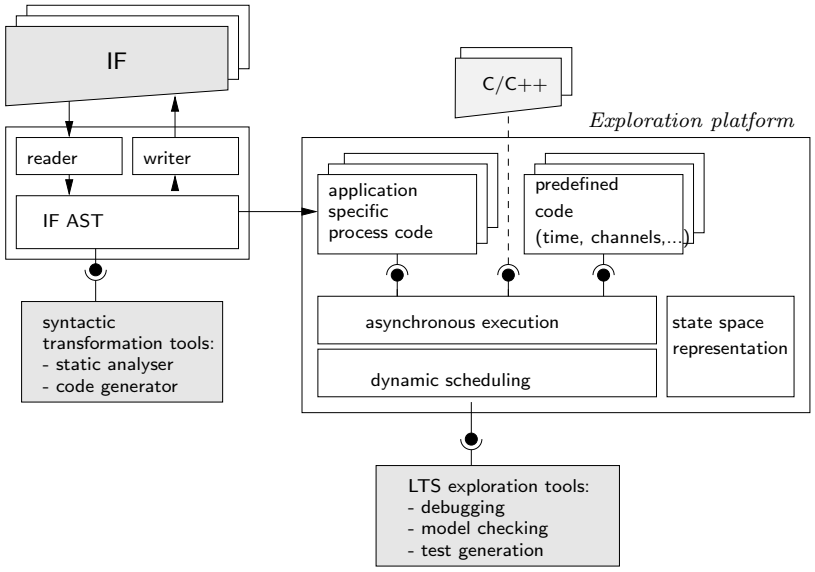


Figure 5.5: The core components of the IF toolbox

The exploration platform gives access to the operational semantics models corresponding to IF descriptions. This exploration platform can be seen as the specialization of the sequential BIP engine to exhaustive state-space exploration of IF models. It offers primitives for representing and accessing execution states and labels as well as basic primitives for traversing the state space: an *init* function which gives the initial state, and a *successor* function which computes the set of enabled transitions and successor states from a given state. These are the key primitives for implementing any on-the-fly forward enumerative exploration or validation algorithm.

Figure 5.5 shows the structure of the exploration platform. The main features of the platform are simulation of the process execution, non-determinism resolution, management of time and representation of the state space. The exploration platform can be seen as a tiny operating system where process instances are plugged-in and jointly executed. Process instances are either application specific (coming from IF descriptions) or generic (such as time or channel handling processes).

Simulation time is handled by a specialized process managing clock allocation/deallocation, computing time progress conditions and firing timed transitions. There are two implementations available, one for discrete time and one for dense time. For discrete time, clock values are explicitly represented by integers. Time progress is computed with respect to the next enabled deadline. For dense time, clock valuations are represented using variable-size Difference Bound Matrices (DBMs) as in tools dedicated to timed automata such as Kronos[Yov97] and Uppaal[LPY98].

The exploration platform composes all active processes and computes global states and the corresponding system behaviour. The exploration platform consists of two layers sharing a common state representation:

- *Asynchronous execution layer:* this layer implements the general interleaving execution of processes. The platform asks successively each process to execute its enabled steps. During a process execution, the platform manages all inter-process operations: message delivery, time constraints checking, dynamic creation and destruction, tracking of events. After a completion of a step by a process, the platform takes a snapshot of the performed step, stores it and delivers it to the second layer.
- *Dynamic scheduling layer:* this layer collects all the enabled steps. It uses a set of dynamic priority rules to filter them. The remaining ones, which are maximal with respect to priority rules, are delivered to the user application via the exploration API.

The exploration platform and its interface has been used as back-ends of debugging tools (interactive or random simulation), model checking (including exhaustive model generation, on the fly μ-calculus evaluation, model checking with observers), test case generation, and optimization (shortest path computation).

This architecture provides features for validating heterogeneous systems. Exploration is not a priori limited to IF descriptions: executable components with an adequate interface can be executed in parallel on the exploration platform. It is indeed possible to use native C/C++ code (either directly, or instrumented accordingly) of already implemented components.

Another advantage of the architecture is that it can be extended by adding new interaction primitives and exploration strategies. Presently, the exploration platform supports asynchronous (interleaved) execution and asynchronous point-to-point communication between processes. Different execution modes, like synchronous or run-to-completion, are obtained by using dynamic priorities.

Concerning the exploration strategies, reduction heuristics such as partial-order reduction or some form of symmetry reduction are already incorporated in the exploration platform. More specific heuristics may be added depending on a particular application domain.

5.2.2 Static Analysis for Model-Checking and Test Generation

The central problem arising in model-checking or model-based test generation is the well known state explosion. There are mainly two reasons: concurrency, which is usually flattened using an interleaving semantics and data, which are also evaluated to all possible, distinct values. Various solutions exist and have been implemented to deal with state explosion. For instance, on-the-fly techniques attempt to explore only a part of the model e.g., guided by the test purpose. Symbolic techniques rely on symbolic and compact representations of sets of states instead of individual states.

In this context, we proposed several complementary reduction techniques [BFG99, BFG00] relying on the use of *static slicing* of IF specifications depending on properties to be verified or test purposes before model generation. A first slicing takes into account the set of interactions between specification processes, starting from inputs enabled in the test purpose. We obtain a first reduction of the specification, consisting of the part which is statically reachable, given the inputs of the test purpose. A second slicing, computes variables and parameters which may be relevant to outputs observed by the test purpose. All other, irrelevant variables and associated actions are safely discarded. Finally, the specification is sliced with respect to

constraints on data attached to the test purpose. The constraints we consider are either constants (i.e, variable equals some value at a control point) or intervals (i.e, for numerical variables only, restriction to intervals of values).

All these optimizations transform IF specifications without loss of information with respect to the test purpose. Moreover, they are independent and can be implemented separately. Furthermore, they can be applied iteratively, in any order, until no more reduction is obtained – a fixpoint is always reached given that specifications are statically finite.

5.2.3 Applications and Case Studies

Case Study: Ariane 5 Flight Program

The objective of the Ariane 5 Flight Program is to control the launcher mission from lift-off to payload release. This software operates in a completely automatic mode and has to handle both the external disturbances and different hardware failures that may occur during the flight. This case study presents the most relevant points required for embedded application and focuses on the real time critical behavior. More detailed descriptions of this case study can be found in [GOO06].

In an earlier experience with this case study using SDL as a modeling language [BLM01], we modelled only the asynchronous part of the system, responsible for controlling the flight phases and the exception handling. We succeeded in verifying the correctness of this part of the software using several models of the environment (representing the synchronous part) obtained from a real flight, but we had no insurance that these environments represent an abstraction of the other parts of the software.

The Ariane-5 Flight Program is a composition of several synchronous and asynchronous modules with strong interactions. Synchronous, cyclically scheduled modules have a specific period and phase; they receive their inputs at the start of their periods and shall produce their outputs before their next execution. Non cyclic, asynchronous modules are activated on external events, and are synchronized or not with the cyclic synchronous processes depending of their required deadline; which is in some cases very short. Moreover, in order to verify the correctness of the real time software design, non-functional features such as the scheduling policy (task or thread definition, priorities between tasks) CPU consumption of each algorithmic function (associated to a task) and different timing constraints have to be expressed.

The design has been modelled in UML using Rational Rose and the Omega UML profile as a collection of objects communicating mostly through asynchronous signals, and whose behavior is described by state machines. Abstract operations are used to model the guidance, navigation and control tasks. For the modeling of timed behavior and timing properties, we used the Omega real-time features. The model obtained for the Ariane-5 Flight Program is relatively large: 23 classes, each one with several operations and a state machine. In addition, the UML model has been annotated with 12 observers, expressing all the relevant untimed and timed requirements or assumptions. The translation into IF is completely automatic using uml2if and gives around 7000 lines of IF code.

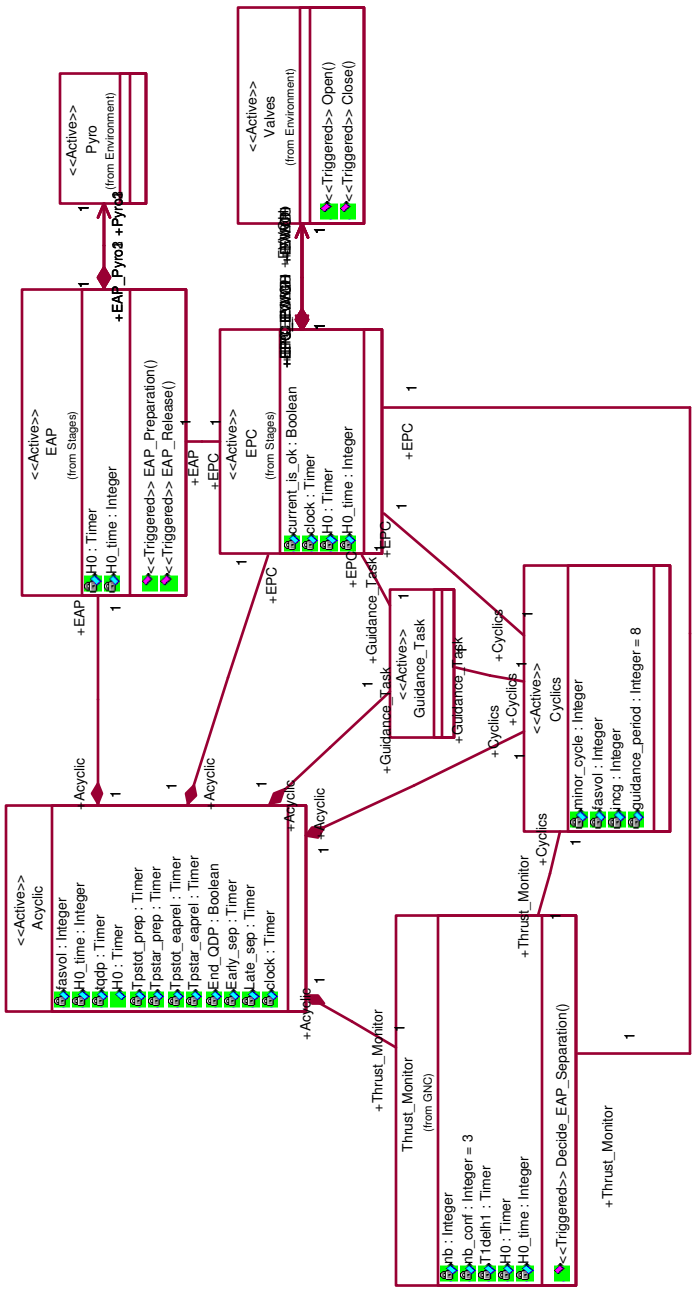


Figure 5.6: Fragment of the UML Specification

To cope with the complexity of the model, in addition to static analysis and partial order reduction, some application specific abstractions and reductions have been needed. In order to verify the correct cooperation between the cyclic and the acyclic behavior, we have used three approaches: we verified the cooperation between the concrete specification of the cyclic behavior with an abstraction of the acyclic behavior and the other way round. This gave satisfactory results concerning the properties of the asynchronous part handling the global flight phases. In addition, we considered the cooperation between the concrete specifications of both parts by scaling down the durations of some phases of the mission.

This last method allowed to verify the main schedulability related property of the basic cycle ensuring that its execution finishes before its deadline (i.e. before being requesting to restart a new execution cycle) has been naturally expressed by a timed observer and validated for several scheduling policies and timing assumptions. Notice that an RMA approach simply concluded that the system is not schedulable. On the reduced model, all properties have been validated through model checking. Intentionally added bugs have been discovered as well, and error scenarios provided for all of them.

Case Study: K9 Rover Executive

The NASA Ames K9 rover is an experimental platform for autonomous wheeled vehicles called rovers, targeted for the exploration of the Martian surface. K9 is specifically used to test out new autonomous software. The rover Executive provides a flexible means for commanding the rover through plans that control the movement, the experimental apparatus and other resources on board - also taking into account the possibility of failures. The correctness requirement states the conformance of the executive with respect to plan semantics, which means that the Executive indeed executes the plans according to their intended semantics.

The K9 rover Executive is a highly non-deterministic system: it is designed as a parallel composition of threads running in an open environment. For a fixed plan, it is therefore possible to obtain many executions, depending on the interleaving of threads, occurrence of failures and interactions with the external environment.

The validation experience of the K9 rover Executive is completely described in [ABBO04, BBKT04]. Our approach can be summarized as follows. First, we build an abstract IF model of the K9 rover Executive. This abstraction has been done manually on the source code, still in a systematic manner which could be automated. The obtained IF model consists of 20 parallel processes corresponding to threads and objects, totalizing around 1000 lines of IF code. Second, we precisely capture the plan semantics using timed IF observers: given a plan, we synthesize automatically a timed observer which encodes the correct executions for that plan. Third, checking the conformance with respect to any fixed plan amounts to the model-checking of the abstract model of the Executive against the corresponding plan observer.

5.3 Automatic Abstraction of Timed Components

The only way to master the complexity of large systems is to apply a hierarchical/compositional analysis methodology. The basic principle of such a methodology is that a

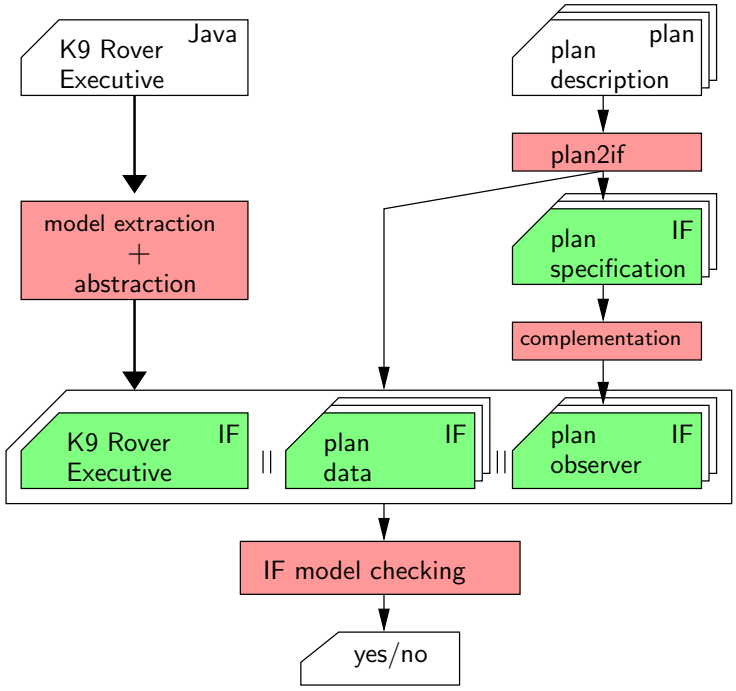


Figure 5.7: Approach for validation of the K9 rover executive

composite component C treated at one level of abstraction as a composition of some detail, is encapsulated as an atomic component $C^\sharp$ when a higher level is considered. The major difference between C and $C^\sharp$ is that the latter abstracts away from the internal details appearing in C and focuses on the interface behavior of the component which determine its high-level functionality.

The problem we tackle here is how to derive automatically $C^\sharp$ from C such that on one hand, the description of $C^\sharp$ is significantly less complex than the one of C and on the other hand, $C^\sharp$ is a valid abstraction, as precise as possible, of C. Moreover, we are looking for compositional generation methods, that means given $C = gl(C_1, ..., C_n)$ we want to compute $C^\sharp$ from $gl(C_1^\sharp, C_2^\sharp, ..., C_n^\sharp)$ instead of $gl(C_1, ..., C_n)$.

We experiment this compositional abstraction idea in several particular application domains. The following procedure taken from [SBM09, Sal07] summarizes the abstraction process for timed components C occuring in the modeling of asynchronous circuits:

1. From a composite component C we construct an extended composite component $C^{(X)}$ by adding fresh auxiliary input clocks that do not participate in transition guards or invariants, but only observe the dynamics of the component and measure the time elapsed since each input event. An input clock is discarded after a finite amount of time when the chain of reactions triggered by its event terminates.

2. We apply the symbolic forward reachability algorithm to the extended component $C^{(X)}$ to obtain its semantic model $\mathcal{S}_{C^{(X)}}$ in form of a zone graph [Tri98]. This model is identical to the semantic model $\mathcal{S}_C$ of C, however, it is annotated with additional

(redundant) timing constraints involving clocks in X.

3. We relax the timing constraints of $\mathcal{S}_{C^{(X)}}$ by projecting them on clocks in X. To be more precise, each transition guard is projected on the clock associated with the input event that has triggered it.

4. We project the symbolic graph on the interface ports to obtain $\mathcal{S}^{vis}_{C^{(X)}}$, thus making some internal transitions silent.

5. We then reduce the discrete state space of $\mathcal{S}^{vis}_{C^{(X)}}$ by merging states which are equivalent in terms of the untimed behaviors they admit and after merging transitions guards we obtain the reduced atomic component $C^{\sharp}$.

It has been proven in [Sal07] that this abstraction method preserves completely the untimed (qualitative) semantics but may relax the timing constraints, that is, exact temporal correlation between input and output events is lost. This loss of precision is due to steps 3 and 5 above which over-approximate some of the temporal constraints of the initial behaviour.

As reported in [SBM09], using this compositional method we were able to compute abstractions and to verify properties on several complex asynchronous circuits. Some of these circuits are far too complex for being analyzed using traditional monolithic model-checking.

We are currently investigating the possibility to extend the abstraction methods to other categories of timed systems, and we consider in particular, resource allocation and scheduling problems in multi-processor systems.

Chapter 6

Conclusion

System Construction

We have presented the BIP component framework, that shares features with other existing frameworks 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.

It has been shown that the BIP glue, consisting of the interactions and priority rules, is as expressive as the universal glue [BS08]. 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.

Language Factory

We have provided evidence through examples treated in BIP, that the combination of interactions and priorities allow enhanced modularity and direct modeling of useful programming

models.

We present a general approach for modeling synchronous component-based systems [BSS09]. These are systems of synchronous components strongly synchronized by a common action that initiates execution steps of each component. Steps can be described using modal flow graphs, a particular class of Petri nets for which deadlock-freedom and confluence are met by construction provided some easy-to-check conditions hold. This result is the generalization of existing results for classes of Petri nets without conflicts. As an example, we applied our construction to the Lustre synchronous language [HCRP91] and provide a semantics preserving mapping of Lustre into BIP. This translation shows the interplay between data flow and control flow and allows understanding how strict synchrony can be weakened to get more less synchronous computation models.

We have provided a system construction methodology, leading to componentization of non trivial systems defined using domain specific languages [CFSS08, BGL+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. For example, in [BMP+07] we have described how a global model of a sensor network 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.

Another successful application of the BIP framework is in the construction and verification of a robotic system [BGL+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.

System Implementation

We elaborate a design flow methodology and we develop the associated software infrastructure based on the BIP component framework. A concrete language, defined as an extension of C, has been proposed for describing systems in BIP. The associated toolbox provides a frontend parser for generation of BIP models, a model transformation tool and code generation facilities for execution or enumerative exploration.

We have defined and implemented several useful architectural transformations on BIP systems [BJS09]. These transformations include flatenning of hierarchical compositions and hierarchical connectors and also static composition of atomic behavior. We show that these transformations are semantic preserving and moreover, when used in the implementation flow, they can increase the time performance of the final implementation with few orders of magnitude.

We have developed two implementation methods for BIP, sequential and distributed, which target respectively single-processor or multi-processor execution platforms. In both cases we rely on the use of a centralized controller, the engine, which coordinates the interactions and the execution of code within atomic components. For the distributed implementation we have

considered a relaxed operational semantics, called partial state semantics, which allows for a better exploitation of parallel execution resources. We provide sufficient conditions to ensure that the (global) state and the partial state semantics are equivalent and we show how the later can be implemented using asynchronous send/receive primitives [BBBS08].

We are now pursuing the work on distributed implementation using specific model transformations. Our aim is to transform progressively the application model towards a distributed implementation model, where execution platform constraints, such as the support for parallel execution and the communication primitives, are taken into account. We plan to study the spectrum of distributed architectures, from fully centralized to fully decentralized ones. In particular, we foreseen the use of existing distributed algorithms for multiparty interaction and conflict resolution e.g. maximal matching algorithms, as a basis for the distributed implementation methods. The adequacy of every solution will be established with respect to two criteria, respectively the degree of parallelism and the overhead for coordination.

Another work direction planned for implementation concerns the integration of memory management policies. BIP adopts a private memory model which is safe for programming but may lead to inefficient implementations. The aim is to study memory transformation from private to shared memory and conversely. We are also interested in transformations leading to mixed solutions combining private and shared memory and determining tradeoffs.

System Validation

We have discussed that constructivity is necessary for building correct systems from components and glue with known properties. As example, we provide correct-by-construction results for synchronous systems developed using modal-flow components and associated composition operators [BSS09].

Alternatively, we have developed a general compositional deadlock-detection method for BIP [BBNS08] and we have implemented it in the D-Finder tool [BBNS09]. This method is based on simple techniques for automatic generation of two categories of invariants respectively, atomic component invariants and interaction invariants. It scales well to large systems. For classical benchmarks, D-Finder outperforms traditional validation methods based on exhaustive exploration, regardless the type of representation used for the state space e.g., symbolic or explicit.

This compositional method is now being extended in two directions. First extension concerns interaction models with data transfer. Actually, the inability to handle data transfer is a rather severe limitation. We are now investigating how to produce sound abstractions without data from general BIP systems, which are as precise as possible and where data transfer has been taken into account. Second extension concerns incrementality of the method. Actually, the method analyzes the BIP system at once. Any structural or behavioral change requires to restart the analysis from the beginning. To improve this situation, an incremental method would re-analyze only the part of the system that has been impacted by the change. For example, it can be shown that by adding one connector, all the already known interaction invariants are preserved and usually, only a few newer interaction invariants are established. The tool must therefore focus on discovering these new invariants instead of re-discovering the previous ones.

Finally, a significant subset of BIP can be simulated and model checked using the IF toolset [VER]. Although we do not advocate traditional model-checking as a major technique for system validation, it still provides a valuable help in system-level debugging or as part of compositional methods requiring finer analysis of smaller parts of the system.

Bibliography

[AAA+07] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guizar, N. Kartha, C. K. Liu, R. Khalaf, D. Konig, M. Marin, V. Mehta, S. Thatte, D. v. d. Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0, Committee Draft, 25 January, 2007, 2007.

[ABBO04] A. Akhavan, M. Bozga, S. Bensalem, and E. Orfanidou. Experiment on Verification of a Planetary Rover Controller. In *Proceedings of the 4th Intl. Workshop on Planning and Scheduling for Space, IWPSS'04, Darmstadt, Germany*, June 2004.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, Pei-Hsin Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[AD94] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AGS02] K. Altisen, G. Gößler, and J. Sifakis. Scheduler Modeling Based on the Controller Synthesis Paradigm. *Real-Time Systems*, 23(1-2):55–84, 2002.

[AH96] R. Alur and T.A. Henzinger. Reactive Modules. In *Proceedings of the 11th Annual Symposium on LICS*, pages 207–208. IEEE Computer Society Press, 1996.

[AL95] M. Abadi and L. Lamport. Conjoining Specification. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[All] TinyOS Alliance. http://www.tinyos.net/.

[Arb05] F. Arbab. Abstract Behavior Types: a foundation model for components and their composition. *Science of Computer Programming*, 55(1-3):3–52, 2005.

[AVCL02] R. Allen, S. Vestal, D. Cornhill, and B. Lewis. Using an architecture description language for quantitative analysis of real-time systems. In *Proceedings of the 3rd International Workshop on Software and Performance WOSP'02*, pages 203–210, New York, NY, USA, 2002. ACM.

[Bas08] A. Basu. *Component-based Modeling of Heterogeneous Real-Time Systems in BIP*. PhD thesis, Université Joseph Fourier, December 2008.

[BBBS08] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed Semantics and Implementation for Systems with Interaction and Priority. In *Proceedings of FORTE'08*, volume 5048 of *LNCS*, pages 116–133. Springer, June 2008.

[BBBS09] A. Basu, B. Borzoo, M. Bozga, and J. Sifakis. Brief Announcement: Incremental Component-Based Specification, Verification, and Performance Evaluation of Distributed Reset. In *Proceedings of DISC'09 - 23rd International Symposium on Distributed Computing, Elche/Elx, Spain*, October 2009.

[BBKT04] S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis. Testing conformance of real-time software by automatic generaton of observers. In *Proceedings of Runtime Verification Workshop, RV'04*, April 2004.

[BBNS08] S. Bensalem, M. Bozga, T. Nguyen, and J. Sifakis. Compositional Verification for Component-based Systems and Application. In *Proceedings of ATVA 2008, Seoul, Korea*, volume 5311 of *LNCS*, pages 64–79. Springer, 10 2008.

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

[BBS06] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Systems in BIP. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06), Pune, India*, pages 3–12, September 2006. invited talk.

[BC85] G. Berry and L. Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, 1985. Springer-Verlag.

[BCC+08] A. Benveniste, B. Caillaud, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Composing heterogeneous reactive systems. *ACM-TECS*, 7(4), 2008.

[BCF02] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C#. In *Proceedings of the 16th European Conference on Object-Oriented Programming ECOOP'02*, pages 415–440, London, UK, 2002. Springer-Verlag.

[BCS04] E. Bruneton, T. Coupaye, and J.B. Stefani. *The Fractal Component Model*. The Object Web Consortium, 2004.

[BFG99] M. Bozga, J. Cl. Fernandez, and L. Ghirvu. State Space Reduction based on Live Variables Analysis. In A. Cortesi and G. Filé, editors, *Proceedings of SAS'99 (Venice, Italy)*, volume 1694 of *LNCS*, pages 164–178. Springer, September 1999.

[BFG00] M. Bozga, J. Cl. Fernandez, and L. Ghirvu. Using Static Analysis to Improve Automatic Test Generation. In S. Graf and M. Schwartzbach, editors, *Proceedings of TACAS'00 (Berlin, Germany)*, LNCS, pages 235–250. Springer, March 2000.

[BFKM97] M. Bozga, J. Cl. Fernandez, A. Kerbrat, and L. Mounier. Protocol Verification with the Aldebaran Toolset. *Software Tools for Technology Transfer*, 1(1+2):166–183, December 1997.

[BFLL04] R. Bruni, J. L. Fiadeiro, I. Lanese, and A. Lopes. New insights on architectural connectors. In *Proceedings of IFIP TCS 2004, 3rd IFIP International Conference on Theoretical Computer Science*, pages 367–379. Kluwer Academics, 2004.

[BGK+06] K. Balasubramanian, A.S. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing Applications Using Model-Driven Design Environments. *IEEE Computer*, 39(2):33–40, 2006.

[BGL+08] A. Basu, M. Gallien, C. Lesire, T. Nguyen, S. Bensalem, F. Ingrand, and J. Sifakis. Incremental Component-Based Construction and Verification of a Robotic System. In *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, volume 178 of *FAIA*, pages 631–635, 2008.

[BGM02] M. Bozga, S. Graf, and L. Mounier. If-2.0: A validation environment for component-based real-time systems. In K.G. Larsen Ed Brinksma, editor, *Proceedings of CAV'02 (Copenhagen, Denmark)*, volume 2404 of *LNCS*, pages 343–348. Springer, July 2002.

[BGMO08] M. Bozga, S. Graf, L. Mounier, and I. Ober. *Real Time Systems 1: Modeling and verification techniques*, chapter Modeling and Verification of Real Time Systems Using the IF Toolbox, pages 319–352. Wiley, 2008.

[BGO+04] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF toolset. In *The SFM'04 School (Bertinoro, Italy)*, volume 3185 of *LNCS*, pages 237–267. Springer, September 2004.

[BIS08] S. Bensalem, F. Ingrand, and J. Sifakis. Autonomous Robot Software Design Challenge. In *2008 International Advanced Robotics Program (IARP). International Workshop on Technical Challenges for Dependable Robots in Human Environnements, May 16-17, Pasadena, USA*, 2008.

[BJS09] M. Bozga, M. Jaber, and J. Sifakis. Source-to-Source Architecture Transformation for Performance Optimization in BIP. In *Proceedings of SIES'09 - IEEE Symposium on Industrial Embedded Systems - Lausanne, Switzerland*, July 2009.

[BL99] S. Bensalem and Y. Lakhnech. Automatic Generation of Invariants. *Formal Methods in System Design*, 15(1):75–92, July 1999.

[BLM01] M. Bozga, D. Lesens, and L. Mounier. Model-Checking Ariane-5 Flight Program. In *Proceedings of FMICS'01 (Paris, France)*, pages 211–227. INRIA, 2001.

[BLO98a] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In A. Hu and M. Vardi, editors, *Proceedings of CAV'98 (Vancouver, Canada)*, volume 1427 of *LNCS*, pages 319–331. Springer, June 1998.

[BLO98b] S. Bensalem, Y. Lakhnech, and S. Owre. Invest: A tool for the verification of invariants. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of CAV '98, Vancouver, BC, Canada*, volume 1427 of *Lecture Notes in Computer Science*, pages 505–510. Springer, 1998.

[BMP+07] A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis. Using BIP for Modeling and Verification of Networked Systems – A Case Study on TinyOS-based Networks. In *Proceedings of NCA'07*, pages 257–260, 2007.

[BS00] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163(1):172–202, 2000.

[BS07] S. Bliudze and J. Sifakis. The Algebra of Connectors — Structuring Interaction in BIP. In *Proceeding of the EMSOFT'07*, pages 11–20, Salzburg, Austria, October 2007. ACM SigBED.

[BS08] S. Bliudze and J. Sifakis. A Notion of Glue Expressiveness for Component-Based Systems. In Franck van Breugel and Marsha Chechik, editors, *Proceedings of CONCUR 2008*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008.

[BSS09] M. Bozga, V. Sfyrla, and J. Sifakis. Modeling Synchronous Systems in BIP. In *Proceedings of EMSOFT'09, Grenoble, France*, October 2009.

[BST98] S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 103–129, London, UK, 1998. Springer-Verlag.

[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(4):45–52, 2003.

[CCN06] S. L. Campbell, J. Chancelier, and R. Nikoukhah. *Modeling and Simulation in Scilab/Scicos*. Springer, 2006.

[CFLS05] J. Combaz, J.C. Fernandez, T. Lepley, and J. Sifakis. Fine Grain QoS Control for Multimedia Application Software. In *Proceedings of DATE'05*, pages 1038–1043, 2005.

[CFSS08] J. Combaz, J.C. Fernandez, J. Sifakis, and L. Strus. Symbolic quality control for multimedia applications. *Real-Time Systems*, 40(1):1–43, 2008.

[CJ88] K. M. Chandy and J.Misra. *Parallel program design: a foundation*. Addison-Wesley Publishing Company, 1988.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional Model Checking. In *Proceedings of the 4th Annual Symposium on LICS*, pages 353–362. IEEE Computer Society Press, 1989.

[CRBS08] Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Translating AADL into BIP - Application to the Verification of Real-Time Systems. In *ACES MB'08 Workshop*, 2008.

[DdM06] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of CAV'06*, volume 4144 of *LNCS*, pages 81–94, 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(1):127–144, 2003.

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *Javaspaces: Principles, Patterns, Practice*. Sun Microsystems, 1999.

[FHC97] S. Fleury, M. Herrb, and R. Chatila. GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. In *International Conference on Intelligent Robots and Systems, Grenoble, France*, pages 842–848, 1997.

[FLN+03] M. Fontoura, T. Lehman, D. Nelson, T. Truong, and Y. Xiong. TSpaces Services Suite: Automating the Development and Management of Web Services. In *Proceedings of The 12th International World Wide Web Conference, Budapest, Hungary*, 2003. http://www.almaden.ibm.com/cs/tspaces/.

[FLV03] P. H. Feiler, B. Lewis, and S. Vestal. The SAE Architecture Analysis and Design Language (AADL) Standard: A basis for model-based architecture-driven embedded systems engineering. In *Proceedings of the RTAS Workshop on Model-driven Embedded Systems*, pages 1–10, 2003.

[GGBM91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real time applications with Signal. *Proceedings of IEEE*, 79(9):1321–1336, 1991.

[GL94] O. Grumberg and D. E. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[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. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[GOO06] S. Graf, I. Ober, and I. Ober. Validating Timed UML models by simulation and verification. *STTT, Software Tools for Technology Transfer*, 8(2), 2006.

[GS04] D. Garlan and B. R. Schmerl. Using Architectural Models at Runtime: Research Challenges. In *European Workshop on Software Architecture*, pages 200–205, 2004.

[GS05] G. Gößler and J. Sifakis. Composition for Component-based Modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005.

[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. In *2nd International Conference on Embedded Networked Sensor Systems*. ACM Press, 2004.

[Hal98] N. Halbwachs. About synchronous programming and abstract interpretation. *Science of Computer Programming*, 31(1):75–89, 1998.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of IEEE*, 79(9):1305–1320, 1991.

[Hen96] T. A. Henzinger. The Theory of Hybrid Automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science LICS'96*, page 278, Washington, DC, USA, 1996. IEEE Computer Society.

[HM08] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling*, 7(2):237–252, 2008.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), August 1978.

[HS01] F. Huber and B. Schätz. Integrated Development of Embedded Systems with AutoFOCUS. Technical Report TUM-I0107, Fakultät für Informatik, TU München, 2001.

[HZPK08] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the Ocarina AADL tool suite. *ACM Transactions on Embedded Computer Systems*, 7(4):1–25, 2008.

[ITU99] ITU. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union – Standardization Sector, Genève, November 1999.

[KV98] O. Kupferman and M. Y. Vardi. Modular model checking. *LNCS*, 1536:381–401, 1998.

[LG95] P. Laborie and M. Ghallab. IxTeT: an Integrated Approach for Plan Generation and Scheduling. In *Proceedings ETFA'95, Paris, France*, pages 485–495, 1995.

[LLWC03] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *SenSys '03: 1st International Conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.

[LPY98] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Journal on Software Tools for Technology Transfer*, 1:134–152, 1998.

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

[MB07] F. Maraninchi and T. Bouhadiba. 42: programmable models of computation for a component-based approach to heterogeneous embedded systems. In *Proceedings of the 6th international conference on Generative programming and component engineering GPCE'07*, pages 53–62, New York, NY, USA, 2007. ACM.

[McM97] K. L. McMillan. A Compositional Rule for Hardware Design Refinement. In *Proceedings of CAV '97, Haifa, Israel*, pages 24–35. Springer-Verlag, 1997.

[MFB02] A. Mallet, S. Fleury, and H. Bruyninckx. A specification of generic robotics software components: future evolutions of GenoM in the Orocos context. In *International Conference on Intelligent Robotics and Systems, Lausanne, Switzerland*, 2002.

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

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

[MS00] R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. Technical Report 3899, INRIA Rhône-Alpes, France, 2000.

[Now06] D. Nowak. Synchronous structures. *Information and Computation*, 204(8):1295–1324, 2006.

[OMG08a] OMG. *OMG Systems Modeling Language SysML (OMG SysML)*. Object Management Group, 2008.

[OMG08b] OMG. *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems*. Object Management Group, 2008.

[OMG09] OMG. *Unified Modeling Language UML*. Object Management Group, 2009.

[Pan01] P. R. Panda. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. In *Proceedings of the International Symposium on Systems Synthesis (ISSS)*, pages 75–80. ACM, 2001.

[PCT04] J. A. Pérez, R. Corchuelo, and M. Toro. An order-based algorithm for implementing multiparty synchronization: Research articles. *Concurrency and Computation : Practice & Experience*, 16(12):1173–1206, 2004.

[Pnu85] A. Pnueli. *Logics and models of concurrent systems*, chapter In transition from global to modular temporal reasoning about programs, pages 123–144. Springer-Verlag, Inc., New York, USA, 1985.

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

[Qui09] J. Quilbeuf. Etude d'une implantation distribuée du langage bip. Master's thesis, Université Grenoble I, 2009.

[RC03] A. Ray and R. Cleaveland. Architectural Interaction Diagrams: AIDs for system modeling. In *Proceedings of the 25th International Conference on Software Engineering ICSE'03*, pages 396–406, Washington, DC, USA, 2003. IEEE Computer Society.

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

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, second edition, 2004.

[Sal07] R. Ben Salah. *On Timing Analysis of Large Systems*. PhD thesis, Institut National Politechnique de Grenoble, October 2007.

[SBM09] R. Ben Salah, M. Bozga, and O. Maler. Compositional Timing Analysis. In *Proceedings of EMSOFT'09, Grenoble, France*, October 2009.

[SEI06] SEI Software Engineering Institute, CMU. *An Extensible Open Source AADL Environment (OSATE)*, 2006.

[Sif78] J. Sifakis. Structural properties of petri nets. In *Proceedings of MFCS'78*, volume 64 of *LNCS*, pages 474–483, 1978.

[Sif05] Joseph Sifakis. A Framework for Component-based Construction. In 3^{rd} *IEEE International Conference on Software Engineering and Formal Methods (SEFM05)*, pages 293–300, September 2005. Keynote talk.

[Sta85] E. W. Stark. A proof technique for rely/guarantee properties. In *FSTTCS: proceedings of the 5th conference*, volume 206, pages 369–391. Springer-Verlag, 1985.

[TBHH07] L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proceedings of the 7th International Conference on Application of Concurrency to System Design ACSD'07*, pages 29–40, Washington, DC, USA, 2007. IEEE Computer Society.

[Tea96] Omega Team. The omega library. Version 1.1.0, November 1996.

[Tri98] S. Tripakis. *L'Analyse Formelle de Systèmes Temporisés en Pratique*. PhD thesis, Université Joseph Fourier, Grenoble, France, December 1998.

[VER] VERIMAG/DCS. If web page. http://www.verimag.imag.fr/~async/IF.

[VPL99] J. Vera, L. Perrochon, and D. C. Luckham. Event-Based Execution Architectures for Dynamic Software Systems. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture WICSA 1*, pages 303–318. Kluwer, B.V., 1999.

[WDE05] Workshop on Distributed Embedded Systems, Lorentz Center, Leiden, 2005. http://www.tik.ee.ethz.ch/ leiden05.

[YAD+08] S. Yovine, I. Assayad, F. Defaut, M. Zanconi, and A. Basu. A formal approach to derivation of concurrent implementations in software product lines. In *Process Algebra for Parallel and Distributed Processing: Algebraic Languages in Specification-Based Software Development*, page 11. CRC Press-Taylor and Francis Group, LLC, 04 2008.

[Yov97] S. Yovine. KRONOS: A Verification Tool for Real-Time Systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, December 1997.

[ZL08] Y. Zhou and E. A. Lee. Causality interfaces for actor networks. *ACM-TECS*, 7(3), 2008.