ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Verimag

1866 AUB American University of Beirut
الجامعة الأميركية في بيروت

# Architecture Internalisation in BIP

**EPFL IC IIF RiSD Technical Report**
**№ EPFL-REPORT-196997**
http://infoscience.epfl.ch/record/196997

**Simon Bliudze, Marius Bozga, Mohamad Jaber and Joseph Sifakis**

**February 20, 2014**

**Abstract:** We consider two approaches for building component-based systems, which we call respectively *architecture-based* and *architecture-agnostic*. The former consists in describing coordination constraints in a purely declarative manner through parameterizable glue operators; it provides higher abstraction level and, consequently, stronger correctness by construction. The latter uses simple fixed coordination primitives, which are spread across component behaviour; it is more error-prone, but allows performance optimisation. We study architecture internalization leading from an architecture-based system to an equivalent architecture-agnostic one, focusing, in particular, on component-based systems described in BIP.

BIP uses connectors for hierarchical composition of components. We study connector internalization in three steps. 1) We introduce and study the properties of interaction expressions, which represent the combined information about all the effects of an interaction. We show that they are a very powerful tool for specifying and analyzing structured interaction. 2) We formalize the connector semantics of BIP by using interaction expressions. The formalization proves to be mathematically rigorous and concise. 3) We introduce the T/B component model and provide a semantics preserving translation of BIP into this model. The translation is compositional that is, it preserves the structure of the source models.

The results are illustrated by simple examples. A Java implementation is evaluated on two case studies.

```
@TechReport{BBJS14-Internalisation-TR,
  author      = {Bliudze, Simon and
                 Bozga, Marius and
                 Jaber, Mohamad and
                 Sifakis, Joseph},
  title       = {Architecture Internalisation in {BIP}},
  institution = {EPFL IC IIF RiSD},
  month       = feb,
  year        = 2014,
  number      = {EPFL-REPORT-196997},
  note        = {Available at: \texttt{http://infoscience.epfl.ch/record/196997}}
}
```

# Architecture Internalisation in BIP

Simon Bliudze * Marius Bozga † Mohamad Jaber ‡ Joseph Sifakis *

**Abstract**

We consider two approaches for building component-based systems, which we call respectively *architecture-based* and *architecture-agnostic*. The former consists in describing coordination constraints in a purely declarative manner through parameterizable glue operators; it provides higher abstraction level and, consequently, stronger correctness by construction. The latter uses simple fixed coordination primitives, which are spread across component behaviour; it is more error-prone, but allows performance optimisation. We study architecture internalization leading from an architecture-based system to an equivalent architecture-agnostic one, focusing, in particular, on component-based systems described in BIP.

BIP uses connectors for hierarchical composition of components. We study connector internalization in three steps. 1) We introduce and study the properties of interaction expressions, which represent the combined information about all the effects of an interaction. We show that they are a very powerful tool for specifying and analyzing structured interaction. 2) We formalize the connector semantics of BIP by using interaction expressions. The formalization proves to be mathematically rigorous and concise. 3) We introduce the T/B component model and provide a semantics preserving translation of BIP into this model. The translation is compositional that is, it preserves the structure of the source models.

The results are illustrated by simple examples. A Java implementation is evaluated on two case studies.

## 1 Introduction

Architectures depict design principles, paradigms that can be understood by all, allow thinking on a higher plane and avoiding low-level mistakes. They are a means for ensuring correctness by construction by enforcing global properties characterizing the coordination between components.

Using architectures largely accounts for our ability to master complexity and develop systems cost-effectively. System developers extensively use libraries of reference architectures ensuring both functional and non-functional properties, for example fault-tolerant architectures, architectures for resource management and QoS control, time-triggered architectures, security architectures and adaptive architectures.

Using architectures allows shifting the focus of developers from lines-of-code to high level structures ensuring coordination in a component-based system. These structures are constraints between the coordinated components expressed in terms of communication mechanisms such as multiparty interaction, message passing, broadcast etc. Formally they can be understood as the assembly of coordination mechanisms which applied to the coordinated components restrict their behavior so as to satisfy a global characteristic property.

*EPFL, Rigorous System Design Laboratory, Station 14, 1015 Lausanne, Switzerland; first-name.lastname@epfl.ch

†UJF-Grenoble 1 / CNRS, Verimag UMR 5104, F-38041 Grenoble, France; marius.bozga@imag.fr

‡American University of Beirut, Lebanon, mj54@aub.edu.lb

There exists an abundant literature on software architectures. Most papers study Architecture Description Languages (ADLs) for representing and analyzing architectural designs [14]. ADLs provide both conceptual frameworks and a concrete syntax for characterizing software architectures. They also provide tools for parsing, compiling, analyzing, or simulating architectural descriptions written in their associated language. While all of these languages are concerned with architectural design, there is no agreement on what is an ADL, what aspects of architecture should be modeled in an ADL, and which of several possible ADLs is best suited for a particular problem. Furthermore, the borders between the realm of the ADLs and that of programming languages are blurring.

Despite the considerable diversity in the capabilities of different ADLs, they all share a common paradigm that determines a set of concepts and concerns for architectural description. This paradigm considers that software can be designed as the hierarchical composition of components by application of architectures. Components are computational elements characterized by their behavior and their interface. The latter defines points of interaction between a components and its environment.

ADLs specify the "glue" of architectural designs, usually expressed as the combination of connections between components. Connections may denote simple interaction mechanisms such as rendezvous, broadcast and function call. But they also may represent more complex ones such as protocols, buses and schedulers. In both cases, they are intended to specify two main aspects of interaction: 1) control-flow that is synchronization constraints; 2) data-flow that is how data provided by each component are transformed when interactions take place.

Adhering to the architecture paradigm confers numerous advantages. One comes from the fact that architectures usually enforce by construction some characteristic property characterizing the coordination between components. If they are sufficiently well-formalized, they can be reused and this allows correctness for free. Nonetheless, the main advantage is abstraction and separation of concerns. The designer can focus on aspects of coordination mechanisms by abstracting from irrelevant details of the behavior of components. These include compatibility of the interfaces, topology and connectivity, analysis of the overall system throughput and latency based on performance estimates of the integrated components. Furthermore, if the chosen ADL is expressive enough, it is possible to describe architectures in a purely declarative manner. Here lies the main distinction between architecture-based and architecture-agnostic description. Architectures can be understood as constraints that adequately restrict the behavior of the coordinated components so as to achieve a desired coordination property. They are defined to a large extent, independently from the components that make up the system. An alternative approach for building component-based systems is to consider as irrelevant the distinction between basic components and their associated coordination mechanisms. A system consists of a set of components—some providing basic functionality and some ensuring coordination. Dependencies between components are explicitly described by their behavior (code) via import clauses, function calls and read/write instructions. We call this approach architecture-agnostic.

In this paper we study architecture internalization as the process leading from an architecture-based system into an equivalent one that is architecture-agnostic. The latter is obtained as a set of interacting components by generating additional components ensuring the coordination intended by the architecture. Two main reasons motivate the study of architecture internalization. One is the exploration of relationships between architecture-based and architecture-agnostic approaches. The other is more practically oriented and deals with the possibility to compile declarative-style architectural constraints into executable code. Figure 1 illustrates the idea.

Architectures can be formally defined as behavior transformers that is, operators transforming the behavior of their arguments (sets of components) into a new behavior. Is it possible to generate from an architecture-based system an equivalent architecture-agnostic one, where architecture glue is cast in dependencies between components explicitly described on their code? This can

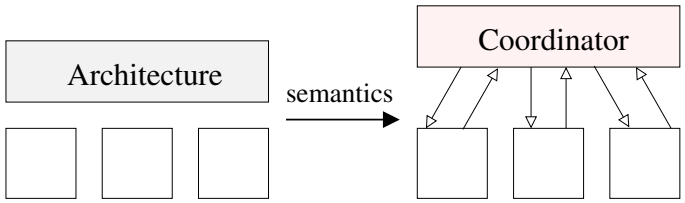



Figure 1: Arichitecture internalisation

be considered as a code generation problem, provided the communication primitives are readily executable.

The distinction between architecture-based and architecture-agnostic approaches appears very early in process algebra theories. CCS proposes a single parallel composition operator based on matching between input and output actions occurring in the description of processes. For instance, the semantics of parallel composition between two processes $?a.P$ and $!a.Q$ is defined by the rule: $?a.P|!a.Q = \tau.(P|Q)+?a.(P|!a.Q)+!a.(?a.P|Q)$. That is, interactions from a state are determined by the actions enabled and the rule that input actions may match output actions. On the contrary, CSP, proposes a parallel composition operator parameterized by a set of actions that must synchronize. For instance, the semantics of applying the parallel composition operator $|\{a\}|$ to two processes $a.P$ and $a.Q$ is defined by the rule: $a.P|\{a\}|a.Q = a.(P|\{a\}|Q)$. In this case, coordination constraints are specified externally and are applied independently of the process evolution.

The dichotomy illustrated by this example is further accentuated in practice. Architecture-based approaches are CSP-like and are adopted by ADLs. They consider coordination as external and independent from the evolution of components. Architectures are to a large extent, entities distinct from behavior. They are combinations of operators parameterized by the allowed interactions. On the contrary, architecture-agnostic designs are based on a single composition operator. Coordination is described in terms of communication primitives appearing in their code. This approach is taken by most programming languages as well as by the various process algebras cloned from CCS.

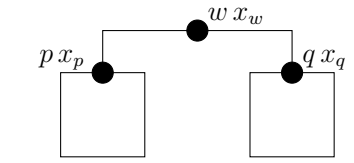
We study the internalization problem for component-based systems described in BIP. BIP allows hierarchical composition of components by using connectors. Components can be considered as transition systems. A component interface consists of ports that label its transitions with associated exported variables. Figure 2a depicts two components with ports $p$, $q$ and associated variables $x_p$, $x_q$. From some state, a port $p$ can participate in an interaction if the component has a transition labeled by $p$ which is enabled at this state.

In BIP, a composite component is an expression of the form $\{\alpha_1, \ldots, \alpha_m\}(B_1, \ldots, B_n)$, where $\{\alpha_1, \ldots, \alpha_m\}$ is s set of hierarchically structured connectors applied to a set of components described by their behavior. A connector is an interaction expression composed of two distinct parts:

1. A control-flow part specifying a relation between a set of bottom ports and a set of top ports. The interaction requires strong synchronization of all the ports. The top ports can be used to export the results of the interaction.

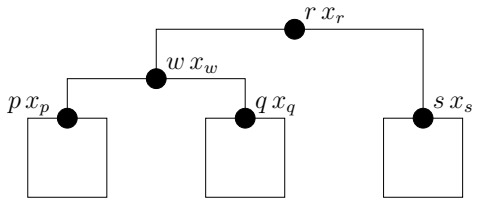
2. A data-flow part specifying the computation associated with the interaction. The computation can affect variables associated with the ports as well as local variables.

In Figure 2a, we provide the specification of the connector describing the interaction between two ports $p$ and $q$. The control flow part is described by the relation $w \leftarrow pq$ meaning that for the interaction $w$ to take place both $p$ and $q$ should participate—$\{p, q\}$ is the set of bottom ports and



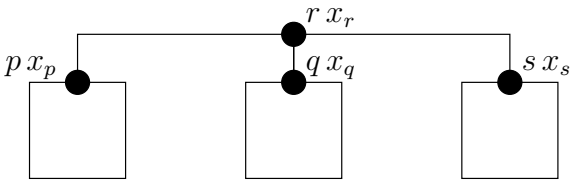

$(w \leftarrow p\,q).[\mathtt{tt} : x_w := max(x_p, x_q) \,//\, x_p := x_w; x_q := x_w]$

(a) Example of a simple BIP connector




$(w \leftarrow p\,q).[\mathtt{tt} : x_w := max(x_p, x_q) \,//\, x_p := x_w; x_q := x_w]$
$(r \leftarrow w\,s).[\mathtt{tt} : x_r := max(x_w, x_s) \,//\, x_w := x_r; x_s := x_r]$

(b) Example of a hierarchical BIP connector




$(r \leftarrow p\,q\,s).[\mathtt{tt} : x_r := max(x_p, x_q, x_s) \,//\, x_p := x_r; x_q := x_r; x_s := x_r]$

(c) Simple connector equivalent to that in Figure 2b

Figure 2: BIP connectors

$\{w\}$ is the set of top ports. The data-flow part consists of an upstream computation followed by a downstream computation separated by " // ". The execution of an interaction is atomic. For the considered example, the interaction between $p$ and $q$ consists in computing $\max(x_p, x_q)$ and assigning this value to the variables $x_p$ and $x_q$. Figure 2b depicts a hierarchical connector $r$ enforcing the interaction between ports $p$, $q$ and $s$. The execution of this interaction results in an upward computation of $\max(\max(x_p, x_q), x_s)$ followed by a downward computation assigning this value to the port variables of the atomic components. As shown in [9], hierarchical connectors can be flattened into equivalent connectors. Figure 2c shows a connector equivalent to the hierarchical connector by eliminating the interaction $w$.

Internalization of connectors in BIP models, consists in replacing them by a set of coordinators that directly implement their semantics. Coordinators play the role of an Engine that handles each interaction atomically. The internalized BIP model is the plain composition of the atomic BIP components with a set of coordinators, in bijection with the BIP connectors. To describe coordinators, we extend the BIP component model. The behavior of the components of the extended model is a set of transitions labeled with interaction expressions. Their interface is composed of sets of top and bottom ports and associated variables. As all the interaction capabilities of components are specified in their behavior, they can be composed without any additional external information. We show that component composition in the new model, called Top/Bottom model (T/B model), can be expressed by using a single associative partial operator $\|$.

The correspondence between a connector and the associated coordinator is straightforward. The latter is a T/B component that has the same interface as the connector (same set of top and bottom ports and associated variables). It exhibits a cyclic behavior by computing the data transfer functions of the connector.

Figure 3 illustrates the principle of connector internalization on a simple example. The corresponding coordinator is a stateless automaton that can perform a transition labeled by the interaction expression. Black arrows show the bottom-top flow.

We study the connector internalization problem for BIP in three steps. First, we study interaction expressions and their properties. We show in particular that they are a very powerful tool

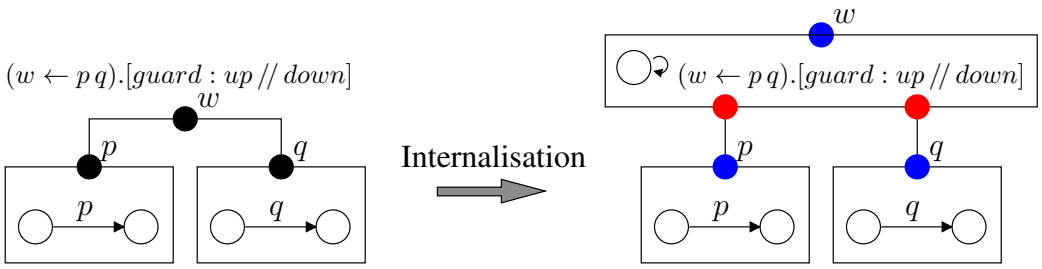



Figure 3: A BIP and the corresponding T/B component

for specifying and analyzing structured interaction. Second, we formalize the connector semantics of BIP by using interaction expressions. The formalization proves to be mathematically rigorous and concise. It treats on an equal footing control and data flow aspects. It differs from previous formalizations that were focusing mainly on control flow. Third, we introduce the T/B component model and provide a semantics preserving translation of BIP into this model. Furthermore, the translation is compositional that is, it preserves the structure of the source models. It consists in compiling a BIP composite component of the form $\{\alpha_1, \ldots, \alpha_m\}(B_1, \ldots, B_n)$ into a behaviourally equivalent T/B component $B_1 \parallel \cdots \parallel B_n \parallel K_1 \parallel \cdots \parallel K_m \parallel A$, where components $K_j$ are coordinators corresponding to $\alpha_j$ and $A$ is an arbitration component for conflict resolution.

We discuss an implementation of the T/B components and provide an algorithm for their execution. The implementation can be used for the execution of BIP components after internalization of their connectors. The advantage of this implementation is its rigorousness as it fully respects the formal semantics. Furthermore, it can be used for the execution of general T/B models.

The paper is structured as follows. In Section 2, we introduce the notion of interaction expressions, shared by all the models in the subsequent sections. In Section 3, we provide a formalization of connectors in BIP and present their properties. In Section 4, we present the T/B component model, study its properties and present a structured encoding of BIP models. In Section 5, we provide experimental results about a Java-based implementation.

# 2 Interactions

## 2.1 Structured Partial Functions

Let $(\mathsf{D}_i)_{i\in\mathcal{I}}$ be data domains, $\mathcal{I}$ a universal index set. For each $I \subseteq \mathcal{I}$, denote by $\mathsf{D}[I] \triangleq \prod_{i\in I} \mathsf{D}_i$ the set of unordered tuples $\mathbf{u} = (u_i)_{i\in I}$, such that $u_i \in \mathsf{D}_i$, for all $i \in I$. For $I = \emptyset$, we have $\mathsf{D}[\emptyset] = \mathbf{1} \triangleq \{*\}$. For each $\mathbf{u} = (u_i)_{i\in I} \in \mathsf{D}[I]$ and $I' \subseteq I$ define the *projection* $\mathbf{u}_{I'} = (u_i)_{i\in I'} \in \mathsf{D}[I']$. We also denote $\mathbf{u}_{\overline{I'}} \triangleq \mathbf{u}_{I\setminus I'}$ the complementary projection. For $I, J \subseteq \mathcal{I}$, the *merge* of tuples is the partial operation $\sqcup : \mathsf{D}[I] \times \mathsf{D}[J] \to \mathsf{D}[I \cup J]$ defined by putting, for $\mathbf{u} \in \mathsf{D}[I], \mathbf{v} \in \mathsf{D}[J]$,

$$\mathbf{u} \sqcup \mathbf{v} \triangleq \begin{cases} \bot\,, & \text{if } \exists i \in I \cap J : u_i \neq v_i\,, \\ (w_i)_{i\in I\cup J}\,, & \text{with } \forall i \in I,\, w_i = u_i \text{ and } \forall j \in J,\, w_j = v_j\,, \text{ otherwise.} \end{cases}$$

Consider structured partial functions $F : \mathsf{D}[I] \to \mathsf{D}[J]$. For each $J' \subseteq J$ the *projection* $F_{J'} : \mathsf{D}[I] \to \mathsf{D}[J']$ is defined by putting $F_{J'}(\mathbf{u}) \triangleq F(\mathbf{u})_{J'}$, for all $\mathbf{u} \in \mathsf{D}[I]$. The complementary projection notation is extended analogously: $F_{\overline{J'}} \triangleq F_{J\setminus J'}$. The *composition* of two structured partial functions $F : \mathsf{D}[I] \to \mathsf{D}[J]$ and $G : \mathsf{D}[K] \to \mathsf{D}[L]$ is the structured partial function

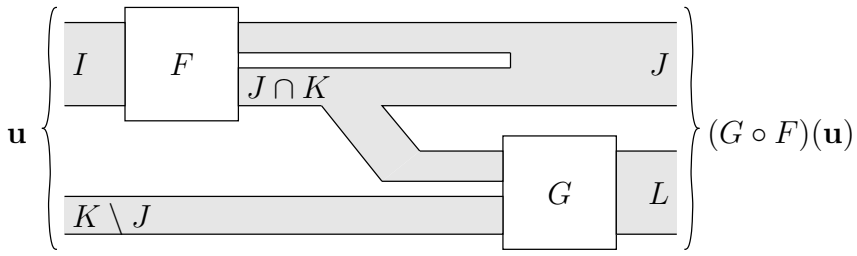



Figure 4: Composition of structured functions

$G \circ F : \mathsf{D}[I \cup (K \setminus J)] \to \mathsf{D}[J \cup L]$ defined by putting

$$(G \circ F)(\mathbf{u}) \triangleq \begin{cases} F(\mathbf{u}_I) \sqcup G(F_{K\cap J}(\mathbf{u}_I) \sqcup \mathbf{u}_{K\setminus J})\,, & \text{when all sub-terms are defined,} \\ \bot\,, & \text{otherwise.} \end{cases}$$

The composition of structured functions is graphically illustrated in Figure 4.

**Proposition 2.1.** *Composition of structured partial functions is associative. It is moreover commutative whenever $J \cap K = I \cap L = \emptyset$.*

For any $I \subseteq \mathcal{I}$, let $X_I = \{x_i{:}\mathsf{D}_i \,|\, i \in I\}$ be a set of typed variables $x_i$ with corresponding domains $\mathsf{D}_i$. We write $X_I{:}\mathsf{D}[I]$ to denote the product domain of the variables in $X_I$.

Let $F : \mathsf{D}[I] \to \mathsf{D}[J]$ be a structured partial function, such that $I \cap J \neq \emptyset$ and consider a non-empty set of variables $X_L{:}\mathsf{D}[L]$ with $L \subseteq I \cap J$. The variables $X_L$ can be used with $F$ as *local* variables to compute values in $\mathsf{D}[J \setminus L]$ based on values in $\mathsf{D}[I \setminus L]$; the variables $X_L$ are updated by *side effect*. We write

$$F[\![X_L]\!] : \mathsf{D}[I \setminus L] \to \mathsf{D}[J \setminus L]\,. \tag{1}$$

Let $\mathbf{v} = (v_l)_{l\in L} \in \mathsf{D}[L]$ be the valuation of $X_L$ and let $\mathbf{u} = (u_i)_{i\in I\setminus L} \in \mathsf{D}[I \setminus L]$. If $F$ is defined on $(\mathbf{u}, \mathbf{v})$, an application of $F[\![X_L]\!]$ to $\mathbf{u}$ produces the values $\mathbf{w} = (w_j)_{j\in J\setminus L} \in \mathsf{D}[J \setminus L]$ and the new valuation $\mathbf{v}' = (v'_l)_{l\in L} \in \mathsf{D}[L]$ of $X_L$, such that

$$(\mathbf{w}, \mathbf{v}') = F(\mathbf{u}, \mathbf{v})\,. \tag{2}$$

**Lemma 2.2.** *Let $F_1[\![X_{L_1}]\!]$ and $F_2[\![X_{L_2}]\!]$ be two structured partial functions with local variables as above and assume $L_1 \cap L_2 = X_{L_1} \cap X_{L_2} = \emptyset$. Then holds*

$$F_1[\![X_{L_1}]\!] \circ F_2[\![X_{L_2}]\!] = \big(F_1 \circ F_2\big)[\![X_{L_1} \cup X_{L_2}]\!]\,. \tag{3}$$

*To simplify the notation, we will also write $\big(F_1 \circ F_2\big)[\![X_{L_1}, X_{L_2}]\!]$ for the expression in the right-hand side of (3).*

Structured partial functions with local variables are particularly useful for the definition of the semantics of *assignment expressions* of the form $(X_J, X_L) := e(X_I, X_L)$, where $e$ is an expression on variables $X_I$ and $X_L$. Indeed, the expression defines a structured partial function $e : \mathsf{D}[I \cup L] \to \mathsf{D}[J \cup L]$ and the fact that variables $X_L$ appear on both sides of the assignment is reflected by considering $e[\![X_L]\!]$ (see (5) for example).

## 2.2 Interaction Expressions

Interaction expressions defined below represent the combined information about all the effects of an interaction involving several ports. We show that they are the basic and general concept for expressing coordination in both architecture-based and architecture agnostic models.

Let $\mathcal{P}$ be a set of ports, and assume $\mathcal{P} \subseteq \mathcal{I}$. For each $p \in \mathcal{P}$, let $x_p : \mathsf{D}_p$ be a typed variable associated to the port $p$. The interaction expressions defined below represent the combined information about all the effects of an *interaction* involving several ports.

**Definition 2.3** (Interaction expressions)**.** An *interaction* is an expression

$$\alpha(X_L) = (P \leftarrow Q).[g(X_Q, X_L) : \ (X_P, X_L) := up(X_Q, X_L) \,//\, (X_Q, X_L) := down(X_P, X_L)] \,, \quad (4)$$

where

- $P, Q \subseteq \mathcal{P}$ are, respectively the *top* and *bottom* sets of ports ;
- $X_L : \mathsf{D}[L]$ is the set of *local memory* variables ;
- $g(X_Q, X_L)$ is the boolean *guard* expression ;
- $up(X_Q, X_L)$ is the *upstream data transfer* expression ;
- $down(X_P, X_L)$ is the *downstream data transfer* expression .

For an interaction expression $\alpha(X_L)$ as above, we denote by $top(\alpha) \triangleq P$ the set of the top ports; $bot(\alpha) \triangleq Q$ the set of the bottom ports and by $support(\alpha) \triangleq P \cup Q$ the set of all ports in $\alpha$. Furthermore, we denote $g_\alpha$, $up_\alpha$ and $down_\alpha$ the corresponding expressions involved in $\alpha$.

An interaction expression (4) combines two parts: the first one $(P \leftarrow Q)$ describes the control flow, that is the dependency relation between the bottom and the top ports. The expression in the brackets describes the data flow. The guard $g(X_Q, X_L)$ materialises the dependency between the two parts: the interaction is only enabled when the values of the local variables together with those of variables associated to the bottom ports satisfy a boolean condition. As a side effect, the firing of an interaction expression can modify the local variables $X_L$.

Notice that an interaction expression can be understood as a generalized synchronous function call involving a set of callees $P$ and a set of callers $Q$. When the callers $Q$ are enabled, they offer a set of parameter values $X_Q$ that are used to compute sequentially the two functions *up* and *down*. The computation is possible only if the guard $g$ is true depending on the values of the exported parameters and the local variables. The *up* function updates the variables of the callees and the local variables. The returned values of the caller variables are computed by the *down* function that also updates the local variables. As explained in Section 3, when interactions are structured hierarchically, the callees at one level may become callers for the upper levels.

More formally, the *data transfer semantics* of $\alpha$ is defined by the pair $(\alpha^\uparrow[\![X_L]\!], \alpha^\downarrow[\![X_L]\!])$ of parameterised structured partial functions:

$$\begin{aligned} \alpha^\uparrow[\![X_L]\!] : &\quad \mathsf{D}[Q] \to \mathsf{D}[P] \\ \alpha^\downarrow[\![X_L]\!] : &\quad \mathsf{D}[P] \to \mathsf{D}[Q] \end{aligned} \quad (5)$$

defined by (see Figure 5a)

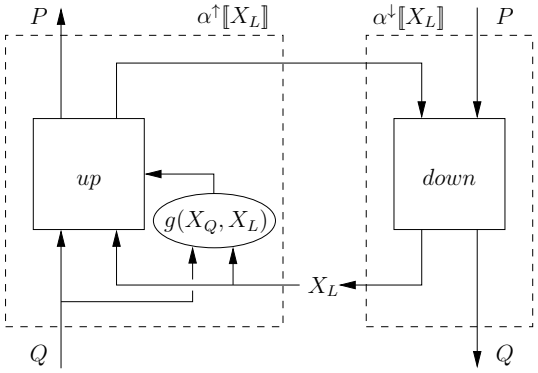
$$\alpha^\uparrow[\![X_L]\!](\mathbf{u}) = \begin{cases} up[\![X_L]\!](\mathbf{u}) \,, & \text{if } g(\mathbf{u}, \mathbf{v}) = \mathtt{tt} \\ \bot \,, & \text{otherwise} \end{cases} \,, \text{ for all } \mathbf{u} \in \mathsf{D}[Q] \,,$$

where $\mathbf{v}$ is the current valuation of variables $X_L$.

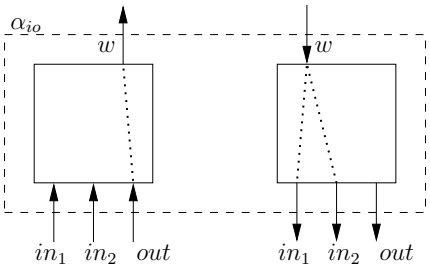
$$\alpha^\downarrow[\![X_L]\!](\mathbf{u}) = down[\![X_L]\!](\mathbf{u}) \,, \text{ for all } \mathbf{u} \in \mathsf{D}[P] \,.$$

The *top-level semantics* of $\alpha$ is defined by $\widehat{\alpha}\ [\![X_L]\!] : \mathsf{D}[Q] \to \mathsf{D}[Q]$, where

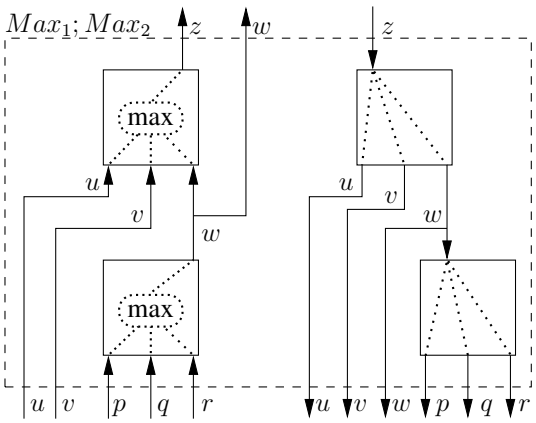
$$\widehat{\alpha}\ [\![X_L]\!] = \alpha^\downarrow[\![X_L]\!] \circ \alpha^\uparrow[\![X_L]\!] \,. \quad (6)$$




(a) Data transfer semantics of an interaction expression




(b) Graphical illustration of the interaction expression $\alpha_{io}$ (Example 2.4)




(c) Graphical illustration of the composition of interaction expressions (Example 2.6)

Figure 5: Data transfer for interaction expressions

**Example 2.4.** Consider the interaction expression

$$\alpha_{io}(\emptyset) = (w \leftarrow out\ in_1\ in_2).[\mathtt{tt} :\ x_w := x_{out} \,//\, x_{in_1}, x_{in_2} := x_w]$$

It does not have any local variables and represents the coordination between an output port *out* that delivers simultaneously its value to two input ports $in_1$ and $in_2$ (see Figure 5b). This interaction can be controlled by a guard to avoid synchronization when the input ports have the same value as the output:

$$(w \leftarrow out\ in_1\ in_2).[(x_{out} \neq x_{in_1}) \vee (x_{out} \neq x_{in_2}) :\ x_w := x_{out} \,//\, x_{in_1}, x_{in_2} := x_w]$$

The interaction expression $Max(\emptyset) = (w \leftarrow pqr).[\mathtt{tt} :\ x_w := \max(x_p, x_q, x_r) \,//\, x_p, x_q, x_r := x_w]$ allows the synchronization between ports $p$, $q$ and $r$ and returns the maximum of the values of the variables associated with these ports.

**Definition 2.5** (Composition of interactions)**.** The *composition* of interaction expressions is a partial operation ';' defined by putting, for two interaction expressions $\alpha_1$, $\alpha_2$, with, for $i = 1, 2$,

$$\alpha_i(X_{L_i}) = (P_i \leftarrow Q_i).[g_i(X_{Q_i}, X_{L_i}) :\ (X_{P_i}, X_{L_i}) := up_i(X_{Q_i}, X_{L_i}) \,//\, (X_{Q_i}, X_{L_i}) := down_i(X_{P_i}, X_{L_i})]\,,$$

where $L_1 \cap L_2 = \emptyset$, $L = L_1 \cup L_2$, $X_{L_1} \cap X_{L_2} = \emptyset$, and $X_L = X_{L_1} \cup X_{L_2}$,

$$(\alpha_1;\alpha_2)(X_L) \triangleq (P \leftarrow Q).[g(X_Q, X_L) :\ (X_P, X_L) := up(X_Q, X_L) \,//\, (X_Q, X_L) := down(X_P, X_L)]\,, \quad (7)$$

where

- $P = P_1 \cup P_2$ and $Q = Q_1 \cup Q_2$,
- $g(X_Q, X_L) = g_1(X_{Q_1}, X_{L_1}) \wedge \Big[g_2(X_{Q_2}, X_{L_2}) \circ up_1(X_{Q_1}, X_{L_1})\Big]_{\overline{P_1 \cup L_1}}$ (the projection in the second conjunct removes the outputs of $up_1$ to keep only the boolean value computed by $g_2$—cf. Figure 4),
- $up(X_Q, X_L) = up_2(X_{Q_2}, X_{L_2}) \circ up_1(X_{Q_1}, X_{L_1})$,
- $down(X_P, X_L) = down_1(X_{P_1}, X_{L_1}) \circ down_2(X_{P_2}, X_{L_2})$.

Notice that three expressions $g(X_Q, X_L)$, $up(X_Q, X_L)$ and $down(X_P, X_L)$ do not involve variables in $X_{P_1 \cap Q_2}$.

**Example 2.6.** We continue Example 2.4. The composition of two *Max* interactions, respectively

$$\begin{array}{rcl} Max_1(\emptyset) & = & (w \leftarrow pqr).[\mathtt{tt} :\ x_w := \max(x_p, x_q, x_r) \,//\, x_p, x_q, x_r := x_w] \\ Max_2(\emptyset) & = & (z \leftarrow uvw).[\mathtt{tt} :\ x_z := \max(x_u, x_v, x_w) \,//\, x_u, x_v, x_w := x_z] \end{array}$$

is the new interaction expression:

$$(Max_1; Max_2)(\emptyset) = (wz \leftarrow pqruvw).[\mathtt{tt} : x_w := \max(x_p, x_q, x_r), x_z := \max(x_u, x_v, \max(x_p, x_q, x_r) \,//\, x_p, x_q, x_r, x_u, x_v, x_w := x_z]$$

This composition is illustrated in Figure 5c.

**Proposition 2.7.** *The composition of interaction expressions is associative. Furthermore, when $P_1 \cap Q_2 = P_2 \cap Q_1 = X_{L_1} \cap X_{L_2} = \emptyset$, it is also commutative.*

*Proof.* This is an immediate consequence of Proposition 2.1. □

Under the disjointness condition in Proposition 2.7, we write

$$\alpha_1 | \alpha_2 \triangleq \alpha_1;\alpha_2 = \alpha_2;\alpha_1\,. \quad (8)$$

In this case, we will speak of interaction *synchronisation*. Notice that | is a partial associative and commutative operator over interaction expressions.

# 3 Architecture-Based Model: Connectors in BIP

This section provides a brief overview of BIP and a formalisation for simple and hierarchical connectors in BIP. The latter formalisation comprises abstract syntax and denotational semantics in terms of partial functions operating on structured domains. In addition, it formalises the flattening as a rewriting rule on hierarchical connectors and proves its soundness as a semantics-preserving transformation.

### 3.1 Atomic Components and Simple Connectors in BIP

In BIP, systems are build by composing *atomic components* with *interactions* defined using connectors. As in Section 2, let $\mathcal{P} \subseteq \mathcal{I}$ be a set of ports and assume that a variable $x_p : \mathsf{D}_p$ is associated with each port $p \in \mathcal{P}$.

**Definition 3.1** (Atomic component)**.** An atomic component $B$ is a tuple $B = (\Sigma, P, X_L : \mathsf{D}[L], \to)$ where

- $\Sigma$ is a finite set of control locations;
- $P \subseteq \mathcal{P}$ is a finite set of ports, called the *interface* of $B$;
- $X_L : \mathsf{D}[L]$ is a set of local variables indexed by $L$ such that $X_L \cap X_P = \emptyset$;
- $\to \subseteq \Sigma \times \mathcal{E} \times \Sigma$ is a finite transition relation, with $\mathcal{E}$ being the set of interaction expressions of the following form, for $p \in P$ and $X \subseteq X_L$,

$$p(X) = (p \leftarrow \emptyset).[g(X) : \ x_p := up(X) \,//\, X := down(x_p, X)] \,.$$

  Henceforth, we call interaction expressions of this form *actions*. Notice that we overload the notation and use $p$ for both the port and the action. Furthermore, since we do not impose any specific constraints on expressions *up* and *down*, several distinct actions can be used for transitions associated to the same port $p$.

We use the notation $q \xrightarrow{p(X)} q'$ as usual.

**Definition 3.2** (Operational semantics of atomic components)**.** The operational semantics of an atomic component $B = (\Sigma, P, X_L : \mathsf{D}[L], \to)$ is given by an LTS $\sigma(B) = \big(\Sigma \times \mathsf{D}[L], 2^P \times (\bigcup_{p \in P} \mathsf{D}_p)^2, \to\big)$, where a state $(q, v)$ consists of a control state of $B$ and the valuation $v \in \mathsf{D}[L]$ of local variables; $\to$ is the minimal transition relation inductively defined by the following rule:

$$\frac{\begin{array}{c} p(X) = (p \leftarrow \emptyset).[g(X) : \ x_p := up(X) \,//\, X := down(x_p, X))] \\ q \xrightarrow{p(X)} q' \quad g(v) = \mathtt{tt} \quad v^p_{up} = up_p(v) \quad v' = down\big(v^p_{down}, up_X(v)\big) \end{array}}{(q, v) \xrightarrow[v^p_{up} : v^p_{down}]{p} (q', v')} \,, \tag{9}$$

where $up_p$ and $up_X$ are the corresponding components of the *up* expression; $v^p_{up}, v^p_{down} \in \mathsf{D}_p$ are the data values associated to the port $p$ at the upward and downward data transfer phases respectively.

**Example 3.3.** The system shown in Figure 6 consists of two identical atomic components that can toghether move in one of two opposite directions. They have to agree on the distance, based on their respective energy levels. Each component has two real local variables: $e$ to store the energy level within the component and *dir* to store the components opinion on the direction to follow, as well as a boolean variable *leader* to remember whether it is a leader or not. In each operation cycle the $i$-th component performs the following three steps:

1. In the first step, the component performs the action

$$connect_i(leader) = (connect_i \leftarrow \emptyset).[\mathtt{tt} : \ id_i := i \,//\, leader := (id_i = i)] \,, \tag{10}$$

   where $i$ is the constant component id (see Figure 6) and $id_i$ is the variable associated to the port $connect_i$. In the upstream transfer of (10), the component proposes itself as a candidate for the leadership. In the downstream transfer, the updated value of $id_i$ is compared to the component id. The result of this comparison is stored in the local variable *leader*.

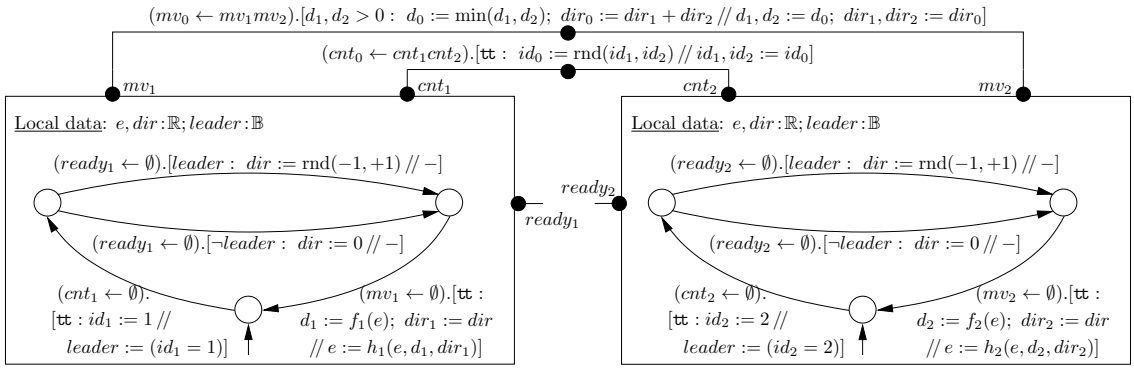



Figure 6: Leader/Follower example

2. In the second step, the component performs its corresponding action $ready_i(dir, leader)$. The leader randomly picks the direction and stores it in the local variable $dir$. The follower stores zero:

$$\text{Leader: } (ready_i \leftarrow \emptyset).[leader :\ dir := \mathrm{rnd}(-1, +1) \,//\, -]\,,$$
$$\text{Follower: } (ready_i \leftarrow \emptyset).[\neg leader :\ dir := 0 \,//\, -]\,.$$

   These actions do not have any downstream data transfer, but only update the local data in the upstream transfer.

3. In the last step, the leader and the direction of the movement are chosen. The component performs the action

$$(move_i \leftarrow \emptyset).[\mathtt{tt} :\ d_i := f_i(e);\ dir_i := dir \,//\, e := h_i(e, d_i, dir_i)]\,. \tag{11}$$

   In the upstream transfer of (11), the component exposes the distance it can cover based on its available energy stored in the local variable $e$, as well as its direction suggestion stored in the local variable $dir$ from the previous step. In the downstream transfer, the move is materialised by updating the energy level of the component, based on the new values of the direction and distance of the move.

**Definition 3.4** (Simple connector)**.** A simple connector is an interaction expression $\alpha(X_L)$, such that $top(\alpha) = \{w\}$ is a single port $w \in \mathcal{P}$, $bot(\alpha) = a \subseteq \mathcal{P}$ is an interaction, such that $w \notin a$, and both $up$ and $g$ expressions do not involve local variables, i.e.

$$\alpha(X_L) = (w \leftarrow a).[g(X_a) :\ (x_w, X_L) := up(X_a) \,//\, X_a := down(x_w, X_L)]\,.$$

**Example 3.5.** Consider the connector (without local variables) shown in Figure 6:

$$(connect_0 \leftarrow connect_1 connect_2).[\mathtt{tt} :\ id_0 := \mathrm{rnd}(id_1, id_2) \,//\, id_1, id_2 := id_0]\,.$$

On every $connect_i$ port ($i = 1, 2$) the value $id_i$ represents the id of a component interacting through this port. The guard of the interaction expression is a constant true, hence no additional restrictions are imposed on the interaction. As part of the upstream data transfer the connector randomly picks and propagates one of the proposed id's. During the downstream data transfer, the updated value is communicated to both participating ports.

**Definition 3.6** (operational semantics)**.** Let $\mathcal{B} = \{B_1, \ldots, B_n\}$ be a finite set of atomic components with $B_i = (\Sigma_i, P_i, X_{L_i} : \mathsf{D}[L_i], \rightarrow)$ such that their respective sets of ports and variables are pairwise disjoint. Let $\Gamma$ be a set of simple connectors such that for every $\alpha \in \Gamma$ hold

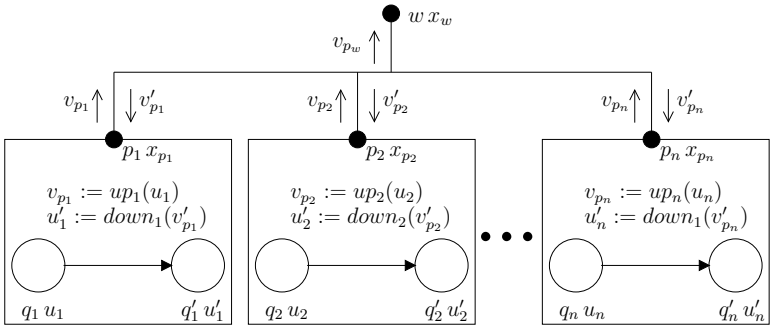



Figure 7: Composition with synchronisation and data transfer

1. $top(\alpha) \notin \bigcup_{i=1}^{n} P_i$,
2. $bot(\alpha) \subseteq \bigcup_{i=1}^{n} P_i$,
3. $|support(\alpha) \cap P_i| \leq 1$ for all $i \in [1, n]$.

Under the conditions above, the semantics of the parallel composition $\Gamma(\mathcal{B})$ is defined as the labelled transition system $(\Sigma, P, \rightarrow)$ where

- $\Sigma = \prod_{i=1}^{n} (\Sigma_i \times \mathsf{D}[L_i])$
- $P = \{top(\alpha) \,|\, \alpha \in \Gamma\}$
- $\rightarrow$ is the minimal transition relation iductively defined by the rule

$$\frac{\begin{array}{c}\alpha(X_L) \in \Gamma \quad top(\alpha) = w \quad bot(\alpha) = a = \{p_i \,|\, i \in I\} \quad \alpha^\star = (|a); \alpha \\ \forall i \in I, q_i \xrightarrow{p_i(X_i)} q'_i \quad \forall i \notin I, (q_i = q'_i \wedge \mathbf{u}_i = \mathbf{u}'_i) \quad (\mathbf{u}'_i)_{i \in I} = \Big[\, \widehat{\alpha^\star}\, [\![X_L]\!]\big((\mathbf{u}_i)_{i \in I}\big)\Big]_{\bigcup_{i \in I} L_i}\end{array}}{(q_1, \mathbf{u}_1), \ldots, (q_n, \mathbf{u}_n) \xrightarrow{w} (q'_1, \mathbf{u}'_1), \ldots, (q'_n, \mathbf{u}'_n)}\ , \tag{12}$$

where $|a$ denotes the synchronisation of all actions $p_i(X_i)$ with $p_i \in a$ (cf. (8)).

Notice that the interaction expressions involved in (12) are partial. Hence, for instance, when the guard of one of the actions is not satisfied, the values $(\mathbf{u}'_i)_{i \in I}$ are undefined and, thus, the rule is not applicable.

Intuitively, an interaction can be fired only if its guard and all guards associated to the corresponding component actions are true. When an interaction is fired, its upstream transfer is computed first using the exposed values offered by the participating components. Then, the downstream transfer modifies back all the port variables followed by execution of the update functions associated to component actions. This semantic is illustrated in Figure 7.

**Example 3.7.** We continue the Example 3.3. The first synchronisation among the atomic components is performed through the connector

$$(connect_0 \leftarrow connect_1 connect_2).[\mathtt{tt} :\ id_0 := \mathrm{rnd}(id_1, id_2) \,//\, id_1, id_2 := id_0] \tag{13}$$

to the actions (10), for $i = 1, 2$. The id of the leader is randomly selected in the connector and transferred downstream through both participating ports.

In the next step each component idependently performs its corresponding step $ready_i$ (see Example 3.3).

In the final step of the cycle, the components synchronise again by applying the connector

$$(move_0 \leftarrow move_1 move_2).[d_1, d_2 > 0 : \ d_0 := \min(d_1, d_2);\ dir_0 := dir_1 + dir_2\ //\\ d_1, d_2 := d_0;\ dir_1, dir_2 := dir_0)] \quad (14)$$

to the actions (11), for $i = 1, 2$. The distances each component can cover and their direction suggestions are combined in the connector to compute the global distance and direction (variables $d_0$ and $dir_0$), which are propagated further, updated and then distributed down to components.

## 3.2 Hierarchical Connectors in BIP

**Definition 3.8** (Hierarchical connector)**.** A hierarchical connector $h\alpha$ is a term generated by the grammar

$$h\alpha ::= \alpha \mid \alpha\langle h\alpha_1, \ldots, h\alpha_n\rangle\,,$$

where $\alpha$ denotes an arbitrary simple connector. We extend the $top()$, $bot()$ and $support()$ notations to hierarchical connectors by putting

$$\begin{aligned} top(\alpha\langle h\alpha_1, \ldots, h\alpha_n\rangle) &= top(\alpha)\,,\\ bot(\alpha\langle h\alpha_1, \ldots, h\alpha_n\rangle) &= \bigcup_{i=1}^{n} bot(\alpha_i)\,,\\ support(\alpha\langle h\alpha_1, \ldots, h\alpha_n\rangle) &= support(\alpha) \cup \bigcup_{i=1}^{n} support(h\alpha_i)\,. \end{aligned}$$

A hierarchical connector $h\alpha = \alpha\langle h\alpha_1, \ldots, h\alpha_n\rangle$ is *valid* iff

1. all sets $support(h\alpha_i)$, for $i \in [1, n]$, are pairwise disjoint;
2. for all $i \in [1, n]$, holds $support(h\alpha_i) \cap support(\alpha) = \{top(h\alpha_i)\}$ and $top(h\alpha_i) \in bot(\alpha)$;
3. all hierarchical sub-connectors $h\alpha_1, \ldots, h\alpha_n$ are valid.

From now on, we tacitly restrict ourselves to valid hierarchical connectors. Their data transfer semantics is defined structurally as follows:

$$\begin{aligned} \alpha\langle h\alpha_1, \ldots, h\alpha_n\rangle^{\uparrow} &= \alpha^{\uparrow} \circ (h\alpha_1{}^{\uparrow} \circ \cdots \circ h\alpha_n{}^{\uparrow})\\ \alpha\langle h\alpha_1, \ldots, h\alpha_n\rangle^{\downarrow} &= (h\alpha_1{}^{\downarrow} \circ \cdots \circ h\alpha_n{}^{\downarrow}) \circ \alpha^{\downarrow} \end{aligned}$$

Notice that the order of composition for sub-connector functions is irrelevant as they operate on disjoint sets of ports.

**Example 3.9.** We continue the running example of this section. Consider a system shown in Figure 8, combining that of Figure 6 with a third atomic component of exactly the same type as the other two. The behavour of the systems is generalised by a hierarchical application of the same (up to port renaming) connectors *move* and *connect*.

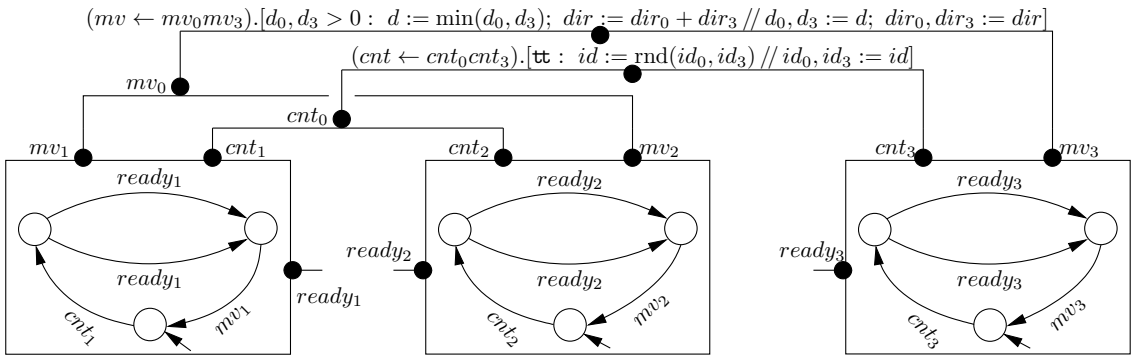



Figure 8: Leader/follower example with three atomic components

**Definition 3.10** (connector composition)**.** Let $\alpha_1$, $\alpha_2$ be two simple connectors

$$\alpha_1(X_{L_1}) = (w_1 \leftarrow a_1).[g_1(X_{a_1}) :\ (x_{w_1}, X_{L_1}) := up_1(X_{a_1}) \,//\, X_{a_1} := down_1(x_{w_1}, X_{L_1})]$$
$$\alpha_2(X_{L_2}) = (w_2 \leftarrow a_2).[g_2(X_{a_2}) :\ (x_{w_2}, X_{L_2}) := up_2(X_{a_2}) \,//\, X_{a_2} := down_2(x_{w_2}, X_{L_2})]$$

such that moreover $a_1 \cap a_2 = L_1 \cap L_2 = \emptyset$, $w_2 \in a_1$ and $w_1 \notin a_2$. We define $\alpha_1 \multimap \alpha_2$ as the *syntactic glueing* of $\alpha_1$ with $\alpha_2$, formally, the simple connector:

$$(\alpha_1 \multimap \alpha_2)(X_{L_1} \cup X_{L_2}) \triangleq (w_1 \leftarrow a_{12}).[g_{12}(X_{a_{12}}) :\ (x_{w_1}, X_{L_1}, X_{L_2}) := up_{12}(X_{a_{12}}) \,//\, X_{a_{12}} := down_{12}(x_{w_1}, X_{L_1}, X_{L_2})]\,,$$

where

$$a_{12} = \big(a_1 \setminus \{w_2\}\big) \cup a_2\,, \tag{15}$$
$$g_{12} = (g_1 \circ [up_2]_{\{w_2\}}) \wedge g_2\,, \tag{16}$$
$$up_{12} = [up_1 \circ up_2]_{\overline{\{w_2\}}}\,, \tag{17}$$
$$down_{12} = [down_2 \circ down_1]_{\overline{\{w_2\}}}\,. \tag{18}$$

Recall that composition of structured partial functions preserves the outputs of both operands (see Figure 4). Therefore, in (16)–(18), we have to take the corresponding projections: to compute the guard $g_1$ of connector $\alpha_1$, we need only the value provided by the $up_2$ function of the subconnector $\alpha_2$ through the port $w_2$, but we discard the values of the local variables $X_{L_2}$. Furthermore, since the port $w_2$ is removed, when glueing the two connectors, we discard the value of its associated variable during the upward and downward phases of the data transfer.

Connector glueing is used to define flattening on hierarchical connectors. Flattening is formalized as a term rewriting rule $\rightsquigarrow$ on hierarchical connectors, that is,

$$\alpha_1\langle\Gamma_1, \alpha_2\langle\Gamma_2\rangle, \Gamma_3\rangle \rightsquigarrow (\alpha_1 \multimap \alpha_2)\langle\Gamma_1, \Gamma_2, \Gamma_3\rangle\,,$$

for any simple connectors $\alpha_1, \alpha_2$, and arbitrary (potentially empty) lists of connectors $\Gamma_1, \Gamma_2, \Gamma_3$. The flattening transformation is graphically illustrated in Figure 9. It can be checked that, given a valid hierarchical connector as input (on the left), the resulting (hierarchical or simple) connector (on the right) is also valid. Moreover, flattening preserves the semantics of hierarchical connectors.

**Proposition 3.11** (Soundness of flattening)**.** *Given a hierarchical connector $h\alpha$, for any flattening $h\alpha \rightsquigarrow h\alpha'$ holds $h\alpha^{\uparrow} = h\alpha'^{\uparrow}$ and $h\alpha^{\downarrow} = h\alpha'^{\downarrow}$.*

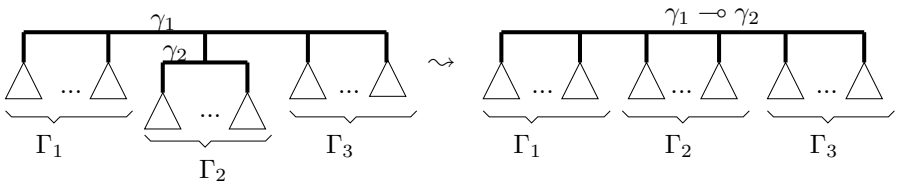



Figure 9: Flattening rule

**Example 3.12.** Hierarchical connectors provide a convenient way to construct arbitrary multiparty interactions by re-using a fixed number of simple connectors. A useful example is *time synchronization* in discrete-time systems. Every component $i$ contains a $tick_i$ port associated with an non-negative integer variable $t_i$ denoting the allowed time elapse. A global interaction amongst all components (i.e. all *tick* ports) is needed to mutually agree on the maximal time elapse at system level, that is, $\min(t_i)$, and to progress accordingly. The following simple connector defines the binary time synchronisation $sync^{(2)}$:

$$sync^{(2)}_{j,i_1,i_2}(\emptyset) = (tick_j \leftarrow tick_{i_1} tick_{i_2}).[t_{i_1}, t_{i_2} > 0 : \ t_j := \min(t_{i_1}, t_{i_2}) \,//\, t_{i_1}, t_{i_2} := t_j]$$

This binary connector can be used herarchically to obtain larger interactions e.g. a 4-ary time synchronisation $sync^{(4)}$:

$sync^{(4)}_{t_j,i_1,i_2,i_3,i_4} \equiv sync^{(2)}_{j,i_{12},i_{34}} \langle sync^{(2)}_{i_{12},i_1,i_2}, sync^{(2)}_{i_{34},i_3,i_4} \rangle$

The hierarchical connector $sync^{(4)}$ can be flattened successively as

$$\begin{aligned} sync^{(4)}_{j,i_1,i_2,i_3,i_4} &\rightsquigarrow (sync^{(2)}_{j,i_{12},i_{34}} \multimap sync^{(2)}_{i_{12},i_1,i_2}) \langle sync^{(2)}_{i_{34},i_3,i_4} \rangle \\ &\rightsquigarrow (sync^{(2)}_{j,i_{12},i_{34}} \multimap sync^{(2)}_{i_{12},i_1,i_2}) \multimap sync^{(2)}_{i_{34},i_3,i_4} \end{aligned}$$

and is semantically equivalent to the simple connector:

$$(tick_j \leftarrow tick_{i_1} tick_{i_2} tick_{i_3} tick_{i_4}).[t_{i_1}, t_{i_2}, t_{i_3}, t_{i_4} > 0 : \ t_j \leftarrow \min(t_{i_1}, t_{i_2}, t_{i_3}, t_{i_4}) \,//\, t_{i_1}, t_{i_2}, t_{i_3}, t_{i_4} \leftarrow t_j] \,.$$

# 4 Architecture Agnostic Model: T/B Components

## 4.1 T/B Component Model

Architecture-agnostic models are obtained from BIP models as the plain composition of *Top/Bottom (T/B) components*. In the translation, BIP connectors are replaced by T/B components that play the role of *coordinators*. These are extensions of the BIP components whose transitions are labeled with interaction expressions. The parallel composition mechanism relies on the matching between bottom and top ports (as for hierarchical connectors).

Interaction execution exhibits a cyclic pattern. In each cycle, the data of interacting atomic components are propagated upwards through top ports towards all relevant coordinators. At each stage, the computation can influence the decision as to what transitions of atomic components are enabled. Finally, once a global interaction has been choosen at the top level, the updated data is propagated back to atomic components. Below, we present this in a formal manner.

As above (cf. Section 2), we assume a universal set of ports $\mathcal{P}$ and, for each port $p \in \mathcal{P}$, a typed variable $x_p : \mathsf{D}_p$.

**Definition 4.1** (T/B components)**.** A *T/B component* is a tuple $T = (\Sigma, P^{bot}, P^{top}, X_L : \mathsf{D}[L], \rightarrow)$, where

- $\Sigma$ is a set of states,
- $P^{bot}, P^{top} \subseteq \mathcal{P}$ are finite sets of respectively *bottom* and *top* ports;
- $X_L : \mathsf{D}[L]$ is a set of *local data variables*;
- $\rightarrow \subseteq \Sigma \times \mathcal{E} \times \Sigma$ is a transition relation, with $\mathcal{E}$ being the set of action expressions $\alpha(X)$, such that $X \subseteq X_L$, $top(\alpha) \subseteq P^{top}$, $bot(\alpha) \subseteq P^{bot}$. We write $q \xrightarrow{\alpha(X)} q'$ for $(q, \alpha(X), q') \in \rightarrow$.

A T/B component $(\Sigma, P^{bot}, P^{top}, X_L : \mathsf{D}[L], \rightarrow)$ is a *basic component*, if $P^{bot} = \emptyset$; it is a *coordinator* if $P^{bot} \neq \emptyset$, but $P^{bot} \cap P^{top} = \emptyset$. Finally, if $P^{bot} \cap P^{top} \neq \emptyset$, the T/B component is *compound*.

Compound components are obtained by hierarchically composing basic components and coordinators.

**Notation 4.2.** For a T/B component with the sets of input and output ports respectively $P^{bot}$ and $P^{top}$, we will always denote $P \triangleq P^{bot} \cup P^{top}$. Conversely, for any object $*$ that can be viewed as a collection indexed by a subset of ports of $P$, we denote by $*^{bot}$ and $*^{top}$ the respective restrictions of $*$ to $P^{bot}$ and $P^{top}$.

**Definition 4.3** (Operational semantics of T/B components)**.** The operational semantics of a T/B component $T = (\Sigma, P^{bot}, P^{top}, X_L : \mathsf{D}[L], \rightarrow)$ is given by an LTS $\sigma(T) = (\Sigma \times \mathsf{D}[L], 2^P \times \mathsf{D}[P]^2, \rightarrow)$, where a state $(q, v)$ consists of a control state of $T$ and the value $v \in \mathsf{D}[L]$; $\rightarrow$ is the minimal transition relation inductively defined by the following rule:

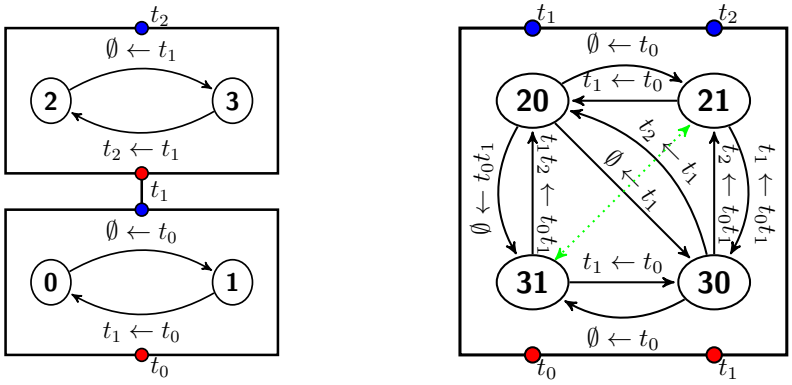
$$\frac{\begin{array}{l}\alpha(X) = (a^{top} \leftarrow a^{bot}).[g(X_{a^{bot}}, X) : \ (X_{a^{top}}, X) := up(X_{a^{bot}}, X) \,//\, (X_{a^{bot}}, X) := down(X_{a^{top}}, X)] \\ q \xrightarrow{\alpha(X)} q' \quad g(\mathbf{v}^{bot}_{up}, v) = \mathtt{tt} \quad \mathbf{v}^{top}_{up} = up_{a^{top}}(\mathbf{v}^{bot}_{up}, v) \qquad (\mathbf{v}^{bot}_{down}, v') = down\big(\mathbf{v}^{top}_{down}, up_X(\mathbf{v}^{bot}_{up}, v)\big)\end{array}}{(q, v) \xrightarrow[\mathbf{v}_{up} : \mathbf{v}_{down}]{a} (q', v')} \ , \tag{19}$$

where $a = a^{top} \cup a^{bot}$; $up_{a^{top}}$ and $up_X$ are the corresponding components of the *up* expression; $\mathbf{v}_{up}, \mathbf{v}_{down} \in \mathsf{D}[P]$ are *partial data valuations associated to ports at the upward and downward data transfer phases* respectively (the values of variables assocaited to ports that do not participate in the interaction are undefined).

Notice that T/B components and their operational semantics generalise atomic BIP components (Definition 3.2). In particular, all components in the examples of Section 3 are T/B components without bottom ports.

**Remark 4.4.** For the values $\mathbf{v}_{up}$ and $\mathbf{v}_{down}$, in (19), it is important to notice the difference with the input/output dichotomy. Indeed, in terms of the transferred data, the component *input* is the pair $(\mathbf{v}^{bot}_{up}, \mathbf{v}^{top}_{down})$, whereas its *output* is the pair $(\mathbf{v}^{top}_{up}, \mathbf{v}^{bot}_{down})$.

Recall the generalised function call metaphor (see the discussion after Definition 2.3). When a transition labelled by $\alpha(X)$ is called, it is provided the values $\mathbf{v}^{bot}_{up}$. If these values satisfy the guard $g$, they are used by the function *up* to compute the values $\mathbf{v}^{top}_{up}$, which are provided to the subsequent callees. In return, the latter provide the updated values $\mathbf{v}^{top}_{down}$, which are, finally, used by the function *down* to compute $\mathbf{v}^{bot}_{down}$.




(a) $\{R_0, R_1\}$ (b) $R_0 \parallel R_1$

Figure 10: T/B component model for the Mod-4 Counter

## 4.2 Systems and Composition

**Definition 4.5** (Systems)**.** Let $S = \{T_i = (\Sigma_i, P_i^{bot}, P_i^{top}, X_{L_i} : \mathsf{D}[L_i], \rightarrow) \mid i \in [1, n]\}$ be a finite set of T/B components and denote $P^{bot} \triangleq \bigcup_{i=1}^{n} P_i^{bot}$ and $P^{top} \triangleq \bigcup_{i=1}^{n} P_i^{top}$. Here and below, we skip the index on $\rightarrow$ since it is always clear from the context. $S$ is a *system* iff the sets of local variables and top ports of all the components are pairwise disjoint, i.e. $\forall i \neq j,\ X_i \cap X_j = P_i^{top} \cap P_j^{top} = \emptyset$.

A system is *closed* if $P^{bot} = P^{top}$; otherwise it is *open*. An open system is *bottom-closed* if $P^{bot} \subseteq P^{top}$.

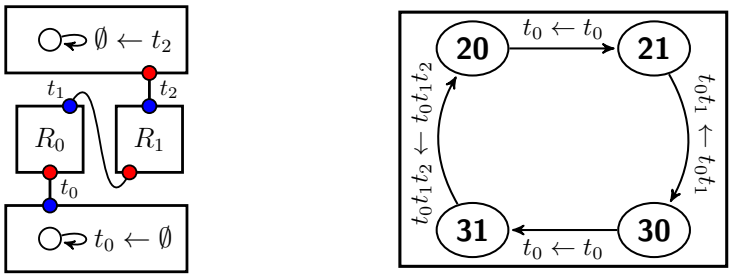
**Definition 4.6** (Composition of T/B components)**.** Let $T_i = (\Sigma_i, P_i^{bot}, P_i^{top}, X_{L_i} : \mathsf{D}[L_i], \rightarrow)$, for $i = 1, 2$, be two T/B components, such that $P_2^{top} \cap P_1^{bot} = \emptyset$ (cf. Definition 4.9 below). Their parallel composition is a compound T/B component $T_1 \parallel T_2 \triangleq (\Sigma, P^{bot}, P^{top}, X_L : \mathsf{D}[L], \rightarrow)$, where $\Sigma = \Sigma_1 \times \Sigma_2$, $P^{bot} = P_1^{bot} \cup P_2^{bot}$, $P^{top} = P_1^{top} \cup P_2^{top}$, $X_L = X_{L_1} \cup X_{L_2}$ and $\rightarrow$ is the minimal transition relation inductively defined by the following rules:

$$\frac{q_1 \xrightarrow{\alpha_1(X_1)} q_1'}{q_1 q_2 \xrightarrow{\alpha_1(X_1)} q_1' q_2}, \qquad \frac{q_1 \xrightarrow{\alpha_1(X_1)} q_1' \quad q_2 \xrightarrow{\alpha_2(X_2)} q_2'}{q_1 q_2 \xrightarrow{(\alpha_1;\alpha_2)(X)} q_1' q_2'}, \qquad \frac{q_2 \xrightarrow{\alpha_2(X_2)} q_2'}{q_1 q_2 \xrightarrow{\alpha_2(X_2)} q_1 q_2'}. \tag{20}$$

When $P_1^{top} \cap P_2^{bot} = \emptyset$, we put $T_1 \parallel T_2 \triangleq T_2 \parallel T_1$. Thus, $\parallel$ is a commutative partial operator defined when $P_2^{top} \cap P_1^{bot} = \emptyset$ or $P_1^{top} \cap P_2^{bot} = \emptyset$. When both equalities hold, the transition in the conclusion of the second rule is labelled by $\alpha_1 \parallel \alpha_2$ (cf. (8)), which is symmetric in the order of its operands. When both $P_2^{top} \cap P_1^{bot} \neq \emptyset$ and $P_1^{top} \cap P_2^{bot} \neq \emptyset$, this means that there is a data-flow causality loop among the two components (as in I/O models [10, 13]) and the composition is undefined.

**Example 4.7** (Mod-4 counter)**.** Figure 10a shows a simple model consisting of two T/B components $R_0$ and $R_1$ without data variables and transfer, identical up to port renaming. Each T/B component models a Mod-2 counter, which produces one event on its top port (shown by blue disks) for every second event on its bottom port (shown by red disks). $R_0$ and $R_1$ share port $t_1$. Figure 10b shows the T/B component $R_0 \parallel R_1$ (for clarity we omit two transitions indicated by the dotted green arrow).

**Proposition 4.8.** *Composition operator $\parallel$ in Definition 4.6 is associative.*




(a) (b)

Figure 11: T/B component model for the Mod-4 Counter

As a consequence of Proposition 4.8, Definition 4.6 can be extended to a larger class of systems.

**Definition 4.9** (Composable systems)**.** Let $S$ be a system and consider the directed graph $\tau(S) = (S, E)$, having the components of the system as vertices and the set of edges $E = \{(T_i, T_j) \,|\, P_i^{top} \cap P_j^{bot} \neq \emptyset\}$. In other words, there is an edge from $T_i$ to $T_j$ if some of the top ports of the former are bottom ports of the latter. $S$ is *composable* iff $\tau(S)$ is a directed acyclic graph.

In a composable system $S$, any pair of components can be ordered so as to satisfy the requirement of Definition 4.6. Hence, by Proposition 4.8, we can define the composed T/B component $\|S$.

As in process calculi like CCS [15], in order for the composition operator $\|$ to be associative, it must allow interleaving (i.e. independent firing) of transitions involving matchable ports (compare first and third rules in (20) with the second rule). When a complete system is built, its meaning is defined as the largest closed sub-system obtained by pruning out all the non-matching transitions; thus the following definition.

**Definition 4.10** (Restriction)**.** Let $S = \{T_i = (\Sigma_i, P_i^{bot}, P_i^{top}, X_{L_i} : \mathsf{D}[L_i], \rightarrow) \,|\, i \in [1, n]\}$ be a closed composable system and $\|S = \big(\Sigma, P^{bot}, P^{top}, X_L : \mathsf{D}[L], \underset{par}{\longrightarrow}\big)$ be the corresponding compound T/B component. The *restriction* of $S$ is given by a T/B component $\rho(S) = \big(\Sigma, P^{bot}, P^{top}, X_L : \mathsf{D}[L], \underset{pr}{\longrightarrow}\big)$, where $\underset{pr}{\longrightarrow}$ is the minimal transition relation inductively defined by the rule

$$\frac{q \xrightarrow[par]{\alpha(X)} q' \qquad bot(\alpha) = top(\alpha)}{q \xrightarrow[pr]{\alpha(X)} q'}. \tag{21}$$

In (21), the second premise means that, for every bottom (resp. top) port, $\alpha$ must also contain the corresponding top (resp. bottom) port. The third premise means that, for any top port, $\alpha$ must also contain some corresponding bottom port, if such bottom port exists. Pruning, in our context, is the equivalent of the CCS hiding operator.

**Example 4.11.** Figure 11a shows a system comprising T/B components $R_0$ and $R_1$ as in Example 4.7 and closed with two additional components: an atomic T/B component that generates events $t_0$ and a top-level T/B component that consumes $t_2$. One can easily see that the restriction of this system, shown in Figure 11b is, indeed, a Mod-4 counter.

**Lemma 4.12.** *For any transition in the restriction of a closed composable system, the data transfer coincides with the top-level semantics of the composition of the corresponding interaction expressions (see (6)).*

### 4.3 T/B Component Encoding of BIP Models

Any atomic BIP component $B = (\Sigma, P, X_L : \mathsf{D}[L], \to)$ can be trivially encoded as a T/B component by making all ports of $B$ top ports, i.e. $\tau(B) = (\Sigma, \emptyset, P, X_L : \mathsf{D}[L], \to)$. Thus, we only have to provide the encoding for connectors. Let

$$\alpha(X) = (w \leftarrow a).[g(X_a) : \ (x_w, X) := up(X_a) \,//\, X_a := down(x_w, X)]$$

be a simple connector with a set of local variables $X : \mathsf{D}$. The T/B component encoding of $\alpha$ is given by $\tau(\alpha) \triangleq \big(\{*\}, P, \{w\}, X : \mathsf{D}, \{* \xrightarrow{\alpha(X)} *\}\big)$.

The encoding of a hierarchical connector is a set of T/B components obtained by separately encoding each subconnector:

$$\tau\big(\alpha\langle h\alpha_1, \ldots, h\alpha_n\rangle\big) \triangleq \big\{\tau(\alpha)\big\} \cup \bigcup_{i=1}^{n} \tau(h\alpha_i)\,. \tag{22}$$

In the BIP operational semantics Definition 3.6, only one connector $\alpha \in \Gamma$ can be fired at a time. On the contrary, parallel composition of T/B components allows any number of component transitions to synchronise. In order to enforce BIP semantics, for a set of connectors *Gamma*, we add an *arbiter* T/B component

$$\tau(\Gamma) = \big(\{*\}, P_\Gamma, \emptyset, \{y_w : \mathsf{D}_w \,|\, w \in P_\Gamma\}\{* \xrightarrow{\tilde{\alpha}} * \,|\, \alpha \in \Gamma\}\big)\,,$$

where $P_\Gamma = \bigcup_{\alpha\in\Gamma} top(\alpha)$, $y_w$ are fresh variables and, for each $\alpha \in \Gamma$ and $\{w\} = top(\alpha)$, we put

$$\tilde{\alpha}(y_w) = (\emptyset \leftarrow w).[\mathtt{tt} : \ y_w := x_w \,//\, x_w := y_w]\,,$$

that is the data provided by $\alpha$ in the upstream data transfer is reinjected back into the downstream data transfer by $\tilde{\alpha}$.

**Theorem 4.13** (Encoding correctness)**.** *Let $\mathcal{B}$ be a set of atomic BIP components and $\Gamma$ be a set of hierarchical connectors and put*

$$S = \{\tau(\Gamma)\} \cup \bigcup_{B\in\mathcal{B}} \big\{\tau(B)\big\} \cup \bigcup_{\alpha\in\Gamma} \tau(\alpha)\,.$$

*The LTS $\sigma\,(\rho(S))$ and $\Gamma(\mathcal{B})$ are isomorphic: there exist agreeing bijections between their sets of states and transitions.*

# 5 Experimental Results

## 5.1 Java Implementation

The implementation consists mainly of: (1) atomic components; (2) coordinators; and (3) connections. Recall that, atomic components have no bottom ports. Connections connect top ports to bottom ports. For composable system they define a hierarchy on T/B components. We assume that a bottom port is connected to exactly one top port; a top port may be connected to more than one bottom port (cf. Definition 4.5). Figure 14 in the appendix shows the Java implementation of the Mod-4 counter from Example 4.7.

We create a Java thread for each atomic component and one thread that plays the role of an *arbiter* for all the coordinators. The implementation of the execution engine would be drastically optimized in case where the coordinators are deterministic. A coordinator is deterministic if from

any state: (1) there exists only one outgoing transition; or (2) the guards of all the outgoing transitions are mutually exclusive. Non-deterministic coordinators may contain a state with more than one outgoing transitions that could be enabled at the same time. That is, more than one up function may be executed. For the sake of clarity, we first provide the algorithm for deterministic coordinators. Atomic component's thread cyclically executes the following:

1. Notify the top ports of the current outgoing transitions that have guard true.
2. Notify the arbiter thread.
3. Wait for a notification from the arbiter.
4. Notification received from the arbiter to execute a top port.
5. Execute the action that corresponds to the received top port.
6. Modify the current state according to transition labeled by the received top port.

Below is the algorithm of the atomic component's thread.

```
// atomic component's thread
run() {
  while(true) {
    for all current outgoing transitions t {
       if guard of t is true {
          t.sendPort.notify();
    }  }
    notify arbiter thread;
    wait for arbiter thread;
    port = notification received from arbiter thread;
    performTransition(port);
  }
}
```

Notification of the top ports is executed by the threads of the atomic components. A notification will be propagated upward by the atomic component's thread until it reaches a top level coordinator component (i.e., a coordinator where its current outgoing transition does not have a top port). That is, notification of a top port is done as follows:

```
topPort.notify() {
  notify bottom ports that are connected to topPort;
  for each coordinator component c that has a
  bottom port notified {
    if exists a current outgoing transition t in c
    where all its bottom ports have been notified
    and its guard is true {
      store the values of the variables of c;
      execute its corresponding up function;
        if t is labeled by a top port {
          t.topPort.notify();
        }
      }
   }
}
```

Note that, upstream propagation is done in parallel by the atomic components' threads. Arbiter thread resumes its execution when the upstream propagation is completed by all the atomic components' threads. Arbiter's thread cyclically executes the following:

1. Select non-deterministically a top level coordinator component which is enabled (i.e., current outgoing transition has not top port and it all its bottom ports have been notified). If such a component does not exist, a deadlock has occurred. Otherwise, we execute the following.
2. Execute the down function of the selected transition and update the state of the coordinator accordingly.

3. Notify all the top ports that are connected to the bottom ports of the selected transition until we reach atomic components. Execute the down function of the transition that has a top port notified.

4. When the downstream propagation is completed, notify all the atomic components to execute their corresponding transitions. Moreover, recover the values of the variables of all the coordinators that have been notified during the upstream propagation and not being modified over the down propagation.

Notice that arbiter selects only one top level coordinator even though there exits more than one top level coordinator that are non conflicting. Two top level coordinators are conflicting if the downstream propagation will lead to notify the same atomic component but with two different top ports. Obviously, selecting two top level coordinators that are conflicting will lead to the violation of the semantics presented in Section 4. Thread arbiter is parameterized to support the two implementations (one top level selection, or multiple non-conflicting top level selection).

For non-deterministic coordinators, the upward propagation has to be modified as follows. First the up function does not modify the actual data of a coordinator but it creates a copy of its variables. If a transition has a top port, we notify that port with an index which represents the values of the data that make this transition enabled. Recall that, the guard of a given transition depends on the value of the variables of the coordinator and the variables of the top ports that are connected to the bottom ports of that transition. So that, before evaluating the guard of a given transition of a coordinator component, we should first set the indices of the bottom coordinators. As the upward propagation is done in parallel, we should also lock those bottom coordinators to avoid the evaluation of other guards that depend on those coordinators but with different indices.

## 5.2 Experiments - case studies

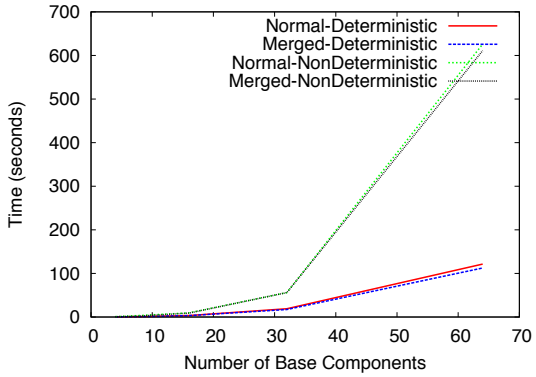
### 5.2.1 Network Sorting Algorithm

The Network Sorting Algorithm (NSA) [2] can be considered as the coordinated product of $2^n$ atomic components, each containing an array of $N$ items. The goal is to sort the items, so that all the items in the first component are smaller than those of the second component and so on. In [9], we have provided a BIP application implementing this algorithm. In order to evaluate the results of the present paper, we have implemented an internalized version using T/B components (see Figure 15 in the appendix for $n = 3$). We have also implemented a modified version of this model, where we merge some coordinators, which might improve the performance (see Figure 16 in the appendix).

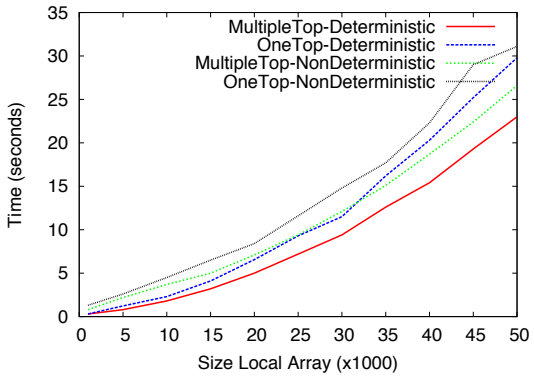
Figures 12a and 12b provide benchmarks for NSA by considering the initial and merged model for the two implementations (deterministic and non-deterministic). Figure 12a shows that the non-deterministic implementation introduces some overhead. We also study the efficiency of selecting all non-conflicting top level coordinators versus selecting only one top level coordinator. Figure 12b shows that selecting all non-conflicting top level coordinators slightly improves the performance.

### 5.2.2 Case Study: POTS

We have implemented a T/B component for the Plain Old Telephone Service (POTS) [11] example (see Figure 17 in the appendix), which provides voice connections between pairs of clients. We distinguish between clients and coordinators. Clients are atomic components with three states. Initially a client can start a new call by dialing the callee id, or it can receive a call from another caller. Then, a voice connection is established between the two clients. When a client hangs up the call is disconnected. We have two-level hierarchy of coordinators. The first level includes




(a) Performance (execution time in seconds) of NSA: Deterministic vs. Non-Deterministic




(b) Performance (execution time in seconds) of NSA: One Top vs. Multiple-Top

Figure 12: Performance (execution time in seconds) of NSA.

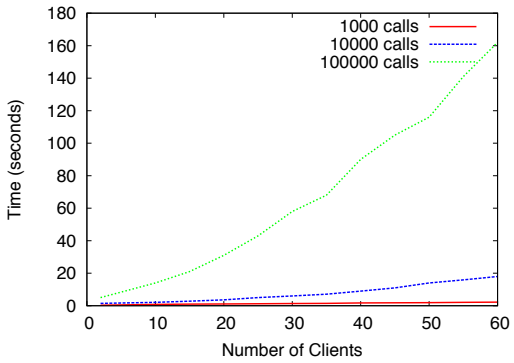



Figure 13: Performance (execution time in seconds) of POTS.

coordinators that collect requests coming from the clients as follows: (1) $CallerAgragration$ collects dialing requests, (2) $CalleAgregation$ collects waiting requests, (3) $VoiceAgregation_1$ and $VoiceAgregation_2$ collect voice requests, (4) $DiscAgregation_1$ and $DiscAgregation_2$ collect disconnect requests. The second level includes coordinators that synchronize requests of bottom coordinators. More precisely, $DialWaitSync$ synchronizes a dialing request (from $CallerAgregation$) with its corresponding waiting request (from $CalleAgregation$). $VoiceSync$ synchronizes voice request (from $VoiceAgregation_1$) with its corresponding voice request (from $VoiceAgregation_2$). $DiscSync$ synchronizes a disconnect request (from $DiscAgregation_1$) with its corresponding disconnect request (from $DiscAgregation_2$). Notice that, the proposed model is very concise and can be modified incrementally e.g. by adding new clients.

Figure 13 shows the performance of POTS for three different values of the number of calls to be satisfied.

# 6 Related Work

Coordination [12] as a means to alleviate complexity in complex system design by distinguishing between a computing part comprising components involved in manipulating data and a coor-

dination part responsible for the harmonious cooperation between the components. The paper points out two main approaches to coordination and studies their relationship. The key concept relating the two approaches is internalisation meaning that external architectural constraints applied to a set of components are cast into their code. To the best of our knowledge, there is no work clearly addressing the problem. In [16], a survey of coordination models and languages is presented and their classification as either "data-driven" or "control-driven". Data-driven coordination languages offer coordination primitives which are mixed within the purely computational part of the code. In the control-driven category, there is a complete separation of coordination from computational concerns. The state of the computation at any moment in time is defined in terms of only the coordinated patterns that the components involved in some computation adhere to. There exists a broad literature on bridging the gap between the design level, as this is expressed by some ADL, and the implementation level, as this is realized by some computational model. Archjava [4, 3, 1] is a small, backwards-compatible extension to Java that smoothly integrates software architecture specifications into Java implementation code. It seamlessly unifies architectural structure and implementation in one language, allowing flexible implementation techniques, ensuring traceability between architecture and code, and supporting the co-evolution of architecture and implementation. In [17], is presented a methodology for mapping architectural representations written in ACME a generic language for describing software architectures, down to executable code. The mapping process involves the use of the coordination paradigm. All these works lack formal foundation and do not allow a deep understanding of the differences between architecture-based and architecture-agnostic approaches. The T/B-component model has some similarities with formalisms using an input/output interaction mechanism for the description of hierarchically structured automata such as Argos [13] and Statecharts [10]. Our model extends the interaction mechanism with data transfer. To avoid causality anomalies [13], we restrict composition to composable systems where hierarchical structure of interaction eliminates by construction cyclic dependencies.

# 7 Conclusion and Future Work

We study a formal framework bridging the gap between architecture description languages and their implementation. The framework clearly distinguishes between two main approaches for tackling the coordination paradigm. One approach is based on the separation between computational and coordination mechanisms; the latter are described as constraints that are independent from the internal behavior of the coordinated components. The other approach consists in internalizing the constraints by generating a set of coordinators that play the role of an execution engine. Formally relating the two approaches opens the way for consistent code generation and guarantees that important architectural properties are guaranteed to hold in the implementation.

Interaction expressions are a key concept that fully describes the control flow and data flow involved in an interaction. They are used both to specify connectors, i.e. architectural constraints as well as executable code in the coordinators. They directly express multiparty interactions and have features for hierarchical structuring. They can be assimilated to synchronous function calls from the bottom ports, that return the values computed when an interaction occurs. It is easy to see that the proposed coordination mechanism is general enough to directly encompass existing mechanisms. In particular it can express data-driven and event-driven interaction. Usually, ADLs use connectors that do not involve computation. For example, data-flow is defined by distinguishing between input and output ports. When an interaction occurs the value of an output is copied into possibly many inputs. For such languages, the expression of interactions involving computation requires the use of additional components.

We have already published formal operational semantics for BIP and developed implemen-

tations in the form of various execution engines [5]. Nonetheless, so far the relation between semantics and the corresponding implementation was not fully formalized. The proposed translation provides a full formalization of the execution engine as a set of interacting coordinators and an arbiter. Furthermore, it preserves the structure of the BIP models. Each connector is implemented by a coordinator. Furthermore, by applying the T/B component composition rule the executable model can be flattened in different possible ways. As shown in [9] flattening allows the generation of more efficient code.

The implementation of T/B component models can be used either for the execution of BIP models after internalization of their connectors or for the execution of such models written independently of BIP.

We see two main directions for future work. One is to study extensions of interaction expressions to encompass dynamic coordination. This can be achieved by including in the set of local variables $X_L$, port variables and component variables as for the Dynamic BIP coordination language [8]. These variables could be used in the guards and affected by the up and down functions, making thus possible dynamic configuration of a model.

The second direction is to study techniques for distributing the generated engine in the form of a T/B component model. So far, we have studied code generation techniques for BIP, that generate distributed implementations for flattened models [6, 7]. This limits the possibility of physically distributing coordinators by preserving the architecture hierarchy. The new techniques will allow full preservation of the coordination structure and enhanced freedom for discovering optimal implementations.

# References


[1] Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley R. Schmerl, Nagi H. Nahas, and Tony Tseng. Modeling and implementing software architecture with acme and ArchJava. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 676–677. ACM, 2005.

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[3] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In Boris Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 334–367. Springer, 2002.

[4] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In Will Tracz, Michal Young, and Jeff Magee, editors, *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 187–197. ACM, 2002.

[5] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *FORTE*, volume 5048 of *LNCS*, pages 116–133. Springer, 2008.

[6] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. In *EMSOFT*, pages 209–218, 2010.

[7] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 25(5):383–409, 2012.

[8] Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling dynamic architectures using Dy-BIP. In *Software Composition*, volume 7306 of *LNCS*, pages 1–16. Springer, 2012.

[9] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-source architecture transformation for performance optimization in BIP. *IEEE Trans. Industrial Informatics*, 6(4):708–718, 2010.

[10] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[11] Jonathan D. Hay and Joanne M. Atlee. Composing features and resolving interactions. In *SIGSOFT FSE*, pages 110–119. ACM, 2000.

[12] Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Comput. Surv.*, 26(1):87–119, 1994.

[13] Florence Maraninchi and Yann Rémond. Argos: an automaton-based synchronous language. *Comput. Lang.*, 27(1/3):61–92, 2001.

[14] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In Mehdi Jazayeri and Helmut Schauer, editors, *Software Engineering - ESEC/FSE '97, 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997, Proceedings*, volume 1301 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 1997.

[15] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[16] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in Computers*, 46:329–400, 1998.

[17] George A. Papadopoulos, Aristos Stavrou, and Odysseas Papapetrou. An implementation framework for software architectures based on the coordination paradigm. *Sci. Comput. Program.*, 60(1):27–67, March 2006.

# Appendix: Additional Figures

```
public class Generator extends AtomicComponent {
  private State s0 = new State(this);
  public TopPort t0 = new TopPort(this);
  public Generator(Compound compound) {
    super(compound);
    setInitial(s0);
    addTransition(new Transition(s0,s0,t0) {
      // default guard and action
    });
  }
}

public class Modulo2 extends Coordinator {
  private Sate s0 = new State(this);
  private State s1 = new State(this);
  public TopPort t1 = new TopPort(this);
  public BottomPort t0 = new BottomPort(this);

  public Modulo2(Compound compound) {
    super(compound);
    setInitial(s0);
    addTransition(new Transition(s0, s1, null, t0) { });
    addTransition(new Transition(s1, s0, t1, t0) { });
  }
}

public class Modulo4 extends Compound {
  public Modulo4() {
    // Behavior Component
    Generator g = new Generator(this);

    // Coordiantor Components
    Modulo2 R0 = new Modulo2(this, "R0");
    Modulo2 R1 = new Modulo2(this, "R1");

    // Connections
    R0.t0.connect(g.t0);
    // For R1, t0 instead of t1, since we reuse the same class Modulo2
    R1.t0.connect(R0.t1);
  }
}
```

Figure 14: Java implementation of the Mod-4 Counter (Example 4.7)

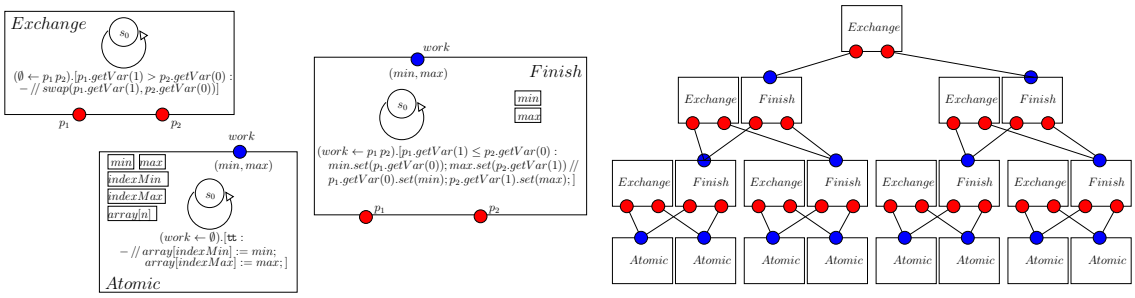



Figure 15: Network Sorting Algorithm in T/B component

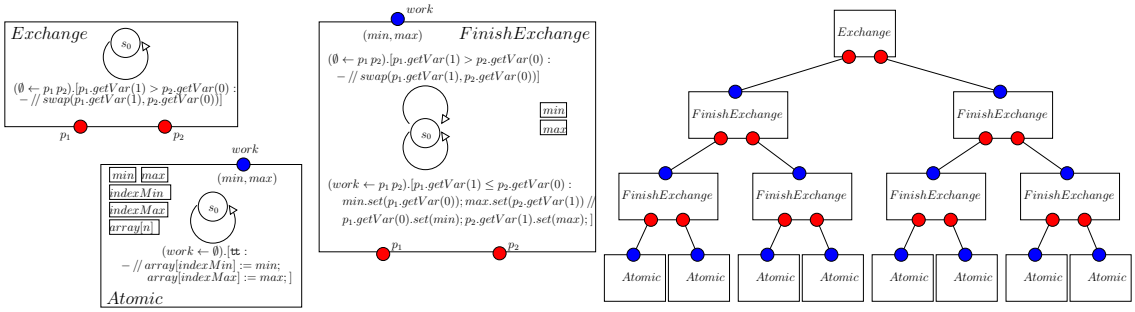



Figure 16: Network Sorting Algorithm in T/B component —Merging Exchange and Finish Coordinators

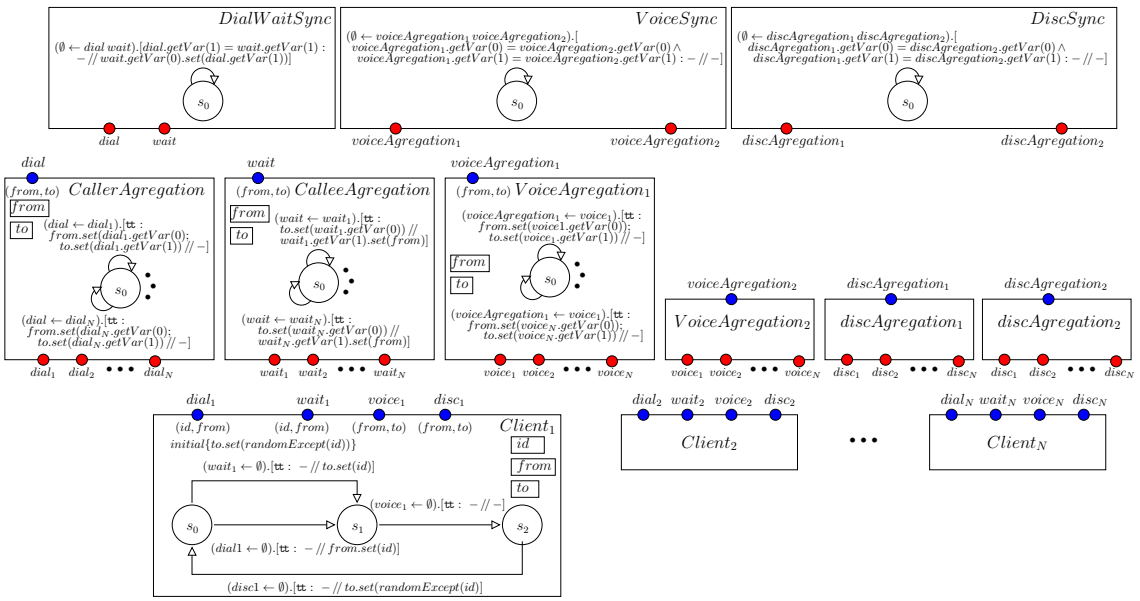



Figure 17: A T/B component for POTS