Architecture internalisation in BIP

Simon Bliudze, Joseph Sifakis, Marius Bozga, Mohamad Jaber

Architecture Internalisation in BIP

Simon Bliudze, Joseph Sifakis
EPFL, Station 14, CH-1015 Lausanne, Switzerland

Marius Bozga
UJF-Grenoble 1 / CNRS, VERIMAG, Grenoble, France

Mohamad Jaber
American University of Beirut, Lebanon

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 parametrizable 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 internalisation 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 internalisation 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 analysing 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.

Keywords BIP; interaction expressions; connectors; data transfer; architecture internalisation; Top/Bottom component model

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 and security architectures.

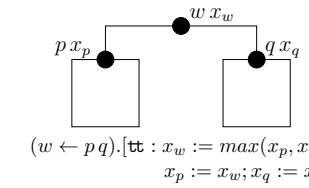
Using architectures allows shifting the focus of developers from low-level code to high-level structures ensuring coordination in a component-based system. These structures are constraints between the coordinated components expressed in terms of communication mechanisms such as multiparty interaction,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

There exists an abundant literature on software architectures. Most papers study Architecture Description Languages (ADLs) for representing and analyzing architectural designs [15]. ADLs provide both conceptual frameworks and 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 ADLs 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. The borders between the realm of the ADLs and that of programming languages are blurring.

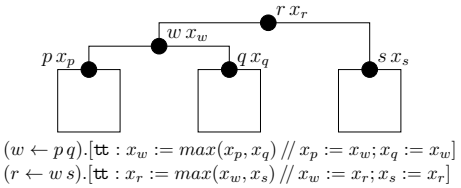
Despite the considerable diversity in the capabilities of different ADLs, they all share a common paradigm: software can be designed as the hierarchical composition of components by application of architectures. Components are computational elements characterized by their behavior and their interface. The latter defines points of interaction between a component 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 of each component are transformed upon interaction.

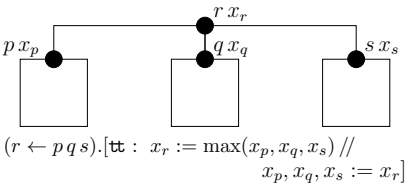
If the ADL is expressive enough, it is possible to describe architectures in a purely declarative manner. Architectures can be understood as constraints that adequately restrict the behavior of the coordinated components so as to achieve a desired coordination property. They are defined to a large extent, independently from the components that make up the system. We speak then of an *architecture-based* approach for building component-based systems. An-



(a) Simple BIP connector



(b) Hierarchical BIP connector



(c) Simple equivalent of Fig. 1b

Figure 1: BIP connectors

other approach considers irrelevant the distinction between atomic 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*.

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. On the contrary, CSP, proposes a parallel composition operator *parameterized* by a set of actions that must synchronize. The dichotomy illustrated by this example is further accentuated in practice. Architecture-based approaches adopted by ADLs are CSP-like. 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. 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 and by the various process algebras cloned from CCS.

In this paper, we study architecture internalization leading from an architecture-based system to an equivalent architecture-agnostic one. Two main reasons motivate this study. 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. 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 in their code?

We study the internalization problem for component-based systems described in BIP [5]. BIP (Behaviour, Interaction, Priority) is a component-based framework that 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. 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 connector is composed of two distinct parts:

Control-flow part specifies a relation between a set of bottom ports and a set of top ports. The interaction requires synchronization of all the ports. The top ports can be used to export the results of the interaction.

Data-flow part specifies the computation associated with the interaction. The computation can affect local variables and those associated with the ports.

In Fig. 1a, 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 $\{w\}$ is the set of top ports. The data-flow part consists of an upward computation followed by a downward 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. Fig. 1b 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 [10], hierarchical connectors can be flattened into equivalent connectors. Fig. 1c 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 (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. Fig. 2 illustrates the principle of connector internalization on a simple example. The corresponding coordinator is a stateless automaton that

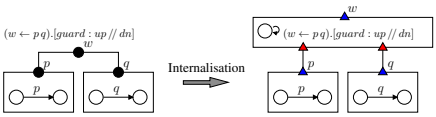


Figure 2: A BIP and the corresponding T/B component

can perform a transition labeled by the interaction expression. Top and bottom ports are shown by blue outward- and red inward-facing triangles, respectively.

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 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. The translation is compositional: it preserves the structure of the source models.

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. Furthermore, it can be used for the execution of general T/B models.

The paper is structured as follows. In Sect. 2, we introduce the notion of interaction expressions, shared by all the models in the subsequent sections. In Sect. 3, we provide a formalization of connectors in BIP and present their properties. In Sect. 4, we present the T/B component model, study its properties and present a structured encoding of BIP models. In Sect. 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 only if $\nexists i \in I \cap J : u_i \neq v_i$, by putting, for $\mathbf{u} \in \mathsf{D}[I], \mathbf{v} \in \mathsf{D}[J]$, $\mathbf{u} \sqcup \mathbf{v} \triangleq (w_i)_{i\in I\cup J}$, with $\forall i \in I$, $w_i = u_i$ and $\forall j \in J$, $w_j = v_j$.

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:

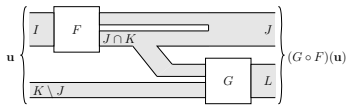


Figure 3: Composition of structured functions

$F_{\overline{J'}} \triangleq F_{J \setminus J'}$. 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 $G \circ F : \mathsf{D}[I \cup (K \setminus J)] \to \mathsf{D}[J \cup L]$ defined, when all sub-terms are defined, by putting $(G \circ F)(\mathbf{u}) \triangleq F(\mathbf{u}_I) \sqcup G(F_{K \cap J}(\mathbf{u}_I) \sqcup \mathbf{u}_{K \setminus J})$ (see Fig. 3).

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

For any $I \subseteq \mathcal{I}$, let $X_I = \{x_i : \mathsf{D}_i \mid i \in I\}$ be a set of typed variables x_i with corresponding domains 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 $F[\![X_L]\!] : \mathsf{D}[I \setminus L] \to \mathsf{D}[J \setminus L]$.

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

Lemma 2.2 *Let* $F_1[\![X_{L_1}]\!]$ *and* $F_2[\![X_{L_2}]\!]$ *be two structured partial functions and assume* $L_1 \cap L_2 = X_{L_1} \cap X_{L_2} = \emptyset$. *Then holds* $F_1[\![X_{L_1}]\!] \circ F_2[\![X_{L_2}]\!] = \big(F_1 \circ F_2\big)[\![X_{L_1} \cup X_{L_2}]\!]$.

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 $e[\![X_L]\!]$.

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

Definition 2.3 *An* interaction *is an expression of the form* $\alpha(X_L) = (P \leftarrow Q).[g(X_Q, X_L) : \ (X_P, X_L) := up(X_Q, X_L) \ // \ (X_Q, X_L) := dn(X_P, X_L)]$*, where* $P, Q \subseteq \mathcal{P}$ *are the* top *and* bottom *sets of ports;* $X_L : \mathsf{D}[L]$ *is the set of* local *variables;* $g(X_Q, X_L)$ *is the boolean* guard*;* $up(X_Q, X_L)$ *and* $dn(X_P, X_L)$ *are respectively the* up- *and* downward data transfer *expressions.*

For an interaction expression $\alpha(X_L)$ *as above, we denote by* $top(\alpha) \triangleq P$*,* $bot(\alpha) \triangleq Q$ *and* $supp(\alpha) \triangleq P \cup Q$ *the sets of top, bottom and by all ports in* α*, respectively. We denote* g_α*,* up_α *and* dn_α *the corresponding expressions in* α*.*

The first part, $(P \leftarrow Q)$, of an interaction expression 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 $g(X_Q, X_L)$ gives the dependency between the two parts: 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 *dn*. 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 *dn* 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.

Formally, the *data transfer semantics* of α is defined by two parameterised structured partial functions $\alpha^{\uparrow}[\![X_L]\!] : \mathsf{D}[Q] \to \mathsf{D}[P]$ and $\alpha^{\downarrow}[\![X_L]\!] : \mathsf{D}[P] \to \mathsf{D}[Q]$:

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

where $\mathbf{v}$ is the current valuation of variables X_L. The *top-level semantics* of α is $\widehat{\alpha}\ [\![X_L]\!] : \mathsf{D}[Q] \to \mathsf{D}[Q]$, with $\widehat{\alpha}\ [\![X_L]\!] = \alpha^{\downarrow}[\![X_L]\!] \circ \alpha^{\uparrow}[\![X_L]\!]$.

Example 2.4 The interaction expression $\alpha_{io}(\emptyset) = (w \leftarrow out\ in_1\ in_2).[\mathtt{tt} : x_w := x_{out}\ //\ x_{in_1}, x_{in_2} := x_w]$ represents the coordination between an port *out* that delivers simultaneously its value to two ports in_1 and in_2. To avoid synchronization when the data at ports *in* have the same value as at *out*, we add a guard: $(w \leftarrow out\ in_1\ in_2).[(x_{out} \neq x_{in_1}) \vee (x_{out} \neq x_{in_2}) : \ x_w := x_{out}\ //\ x_{in_1}, x_{in_2} := x_w]$. 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 associated to these ports.

Definition 2.5 *The* composition *of interaction expressions is a partial operation ';' defined, for two interaction expressions α_1, α_2, with $X_{L_1} \cap X_{L_2} = \emptyset$, $X_L = X_{L_1} \cup X_{L_2}$ and $\alpha_i(X_{L_i}) = (P_i \leftarrow Q_i).[g_i(X_{Q_i}, X_{L_i}) : (X_{P_i}, X_{L_i}) := up_i(X_{Q_i}, X_{L_i}) \,//\, (X_{Q_i}, X_{L_i}) := dn_i(X_{P_i}, X_{L_i})]$, for $i = 1, 2$, by putting*

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

where $P = P_1 \cup P_2$ and $Q = Q_1 \cup Q_2$, $up(X_Q, X_L) = up_2(X_{Q_2}, X_{L_2}) \circ up_1(X_{Q_1}, X_{L_1})$, $dn(X_P, X_L) =$
$dn_1(X_{P_1}, X_{L_1}) \circ dn_2(X_{P_2}, X_{L_2})$, $g(X_Q, X_L) =$

$$g_1(X_{Q_1}, X_{L_1}) \wedge \Big[g_2(X_{Q_2}, X_{L_2}) \circ up_1(X_{Q_1}, X_{L_1})\Big]_{\overline{P_1 \cup L_1}}$$

(the projection in the second conjunct removes the outputs of up_1, keeping only the boolean value of g_2—cf. Fig. 3).

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

Example 2.6 We continue Ex. 2.4. The composition of two interaction expressions, $Max_1(\emptyset)$ and $Max_2(\emptyset)$, respectively

$$(w \leftarrow pqr).[\mathtt{tt} : \; x_w := \max(x_p, x_q, x_r) \,//\, x_p, x_q, x_r := x_w]$$
$$(z \leftarrow uvw).[\mathtt{tt} : \; x_z := \max(x_u, x_v, x_w) \,//\, x_u, x_v, x_w := x_z]$$

is the new interaction expression:

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

Proposition 2.7 *The operator ';' is associative. When $P_1 \cap Q_2 = P_2 \cap Q_1 = X_{L_1} \cap X_{L_2} = \emptyset$, it is also commutative.*

Under this disjointness condition, we write $\alpha_1 | \alpha_2 \triangleq \alpha_1; \alpha_2 = \alpha_2; \alpha_1$ and speak of interaction *synchronisation*.

3 Architecture-Based Model

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.

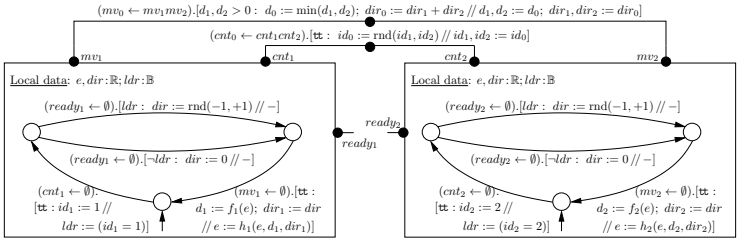


Figure 4: Leader/Follower example

3.1 Simple Connectors in BIP

In BIP, systems are build by composing *atomic components* with *interactions* defined using connectors. As in Sect. 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 *An atomic component B is a tuple $B = (\Sigma, P, X_L : \mathsf{D}[L], \to)$ where Σ is a finite set of control locations; $P \subseteq \mathcal{P}$ is a finite set of ports—the* interface *of B; $X_L : \mathsf{D}[L]$ is a set of local variables, with $X_L \cap X_P = \emptyset$; $\to \subseteq \Sigma \times \mathcal{E} \times \Sigma$ is a finite transition relation, with $\mathcal{E}$ the set of interaction expressions of the form $p(X) = (p \leftarrow \emptyset).[g(X) :\ x_p := up(X) \,//\, X := dn(x_p, X)]$, for $p \in P$ and $X \subseteq X_L$.*

Henceforth, we call interaction expressions of this form actions. *We use p for both the port and the action.*

Definition 3.2 *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 defined by the following rule:*

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

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

Example 3.3 The system shown in Fig. 4 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 its energy level and dir to store its opinion on the direction to follow, as well as a boolean variable ldr to remember whether it is a leader or not. In each operation cycle the i-th component performs the following three steps:

First, the component performs the *connect* action $cnt_i(ldr) = (cnt_i \leftarrow \emptyset).[\mathtt{tt} : id_i := i \,//\, ldr := (id_i = i)]$, where i is the constant component id (see Fig. 4)

and id_i is the variable associated to the port cnt_i. In the upward transfer, the component proposes itself as a candidate for the leadership. In the downward transfer, the updated value of id_i is compared to the component id. The result of this comparison is stored in the local variable ldr.

In the second step, the component performs its corresponding action $ready_i(dir, ldr)$. The leader randomly picks the direction and stores it in the local variable dir: $(ready_i \leftarrow \emptyset).[ldr : \ dir := \mathrm{rnd}(-1,+1) \,//\, -]$. The follower stores zero: $(ready_i \leftarrow \emptyset).[\neg ldr : \ dir := 0 \,//\, -]$. These actions do not have any downward data transfer, but only update the local data in the upward transfer.

In the last step, the leader and the direction of the movement are chosen. The component performs the action $move$: $(mv_i \leftarrow \emptyset).[\mathtt{tt} : \ d_i := f_i(e); \ dir_i := dir \,//\, e := h_i(e, d_i, dir_i)]$. In the upward transfer, 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 downward 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 *A* simple connector *is an interaction expression $\alpha(X_L)$, such that $top(\alpha) = \{w\}$ is a single port $w \in \mathcal{P}$, $bot(\alpha) = a \subseteq \mathcal{P}$, such that $w \notin a$, and both up and g expressions do not involve local variables, i.e. $\alpha(X_L) = (w \leftarrow a).[g(X_a) : \ (x_w, X_L) := up(X_a) \,//\, X_a := dn(x_w, X_L)]$.*

Example 3.5 Consider the connector (without local variables) shown in Fig. 4:

$$(cnt_0 \leftarrow cnt_1 cnt_2).[\mathtt{tt} : \ id_0 := \mathrm{rnd}(id_1, id_2) \,//\, id_1, id_2 := id_0].$$

On every cnt_i port 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 upward data transfer the connector randomly picks and propagates one of the proposed id's. At the downward data transfer, the updated value is communicated to both participating ports.

Definition 3.6 *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 Γ be a set of simple connectors such that, for every $\alpha \in \Gamma$, $top(\alpha) \notin \bigcup_{i=1}^n P_i$, $bot(\alpha) \subseteq \bigcup_{i=1}^n P_i$ and $|supp(\alpha) \cap P_i| \leq 1$ for all $i \in [1, n]$. The operational semantics of the* parallel composition $\Gamma(\mathcal{B})$ *is defined as the LTS $(\Sigma, P, \rightarrow)$ 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 defined by the rule*

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

where $|a$ is the synchronisation of all $p_i(X_i)$ with $p_i \in a$.

Notice that the involved interaction expressions 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 upward transfer is computed first using the exposed values offered by the participating components. Then, the downward transfer modifies back all the port variables followed by execution of the update functions associated to component actions.

Example 3.7 The first synchronisation among the atomic components of Ex. 3.3 is performed through the connector $(cnt_0 \leftarrow cnt_1 cnt_2).[\mathtt{tt} : \ id_0 := \mathrm{rnd}(id_1, id_2) \,//\, id_1, id_2 := id_0]$. The id of the leader is randomly selected in the connector and transferred downward through both participating ports. In the next step each component idependently performs its corresponding step $ready_i$ (see Ex. 3.3). Finally, components synchronise again through the connector

$$(mv_0 \leftarrow mv_1 mv_2).[d_1, d_2 > 0 : \\ d_0 := \min(d_1, d_2);\ dir_0 := dir_1 + dir_2 \,// \\ d_1, d_2 := d_0;\ dir_1, dir_2 := dir_0)]\,.$$

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 *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 α denotes an arbitrary simple connector. We extend the top(), bot() and supp() to hierarchical connectors:*

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

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

We tacitly restrict ourselves to valid hierarchical connectors. Their data transfer semantics is defined structurally:

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

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

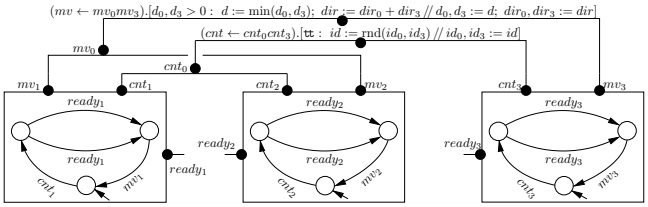


Figure 5: Leader/follower example with three atomic components

Example 3.9 We continue the running example of this section. Consider a system shown in Fig. 5, combining that of Fig. 4 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 mv and cnt. Composing the interaction expressions for simple connectors cnt_0 and cnt gives $(cnt_0\, cnt \leftarrow cnt_0\, cnt_1\, cnt_2\, cnt_3).[\mathtt{tt} :\ id_0 := rnd(id_1, id_2);\ id := rnd(id_1, id_2, id_3) \,//\, id_0, id_1, id_2, id_3 := id]$. Notice that, by discarding the port cnt_0 and the associated variable id_0, we obtain an equivalent simple connector $(cnt \leftarrow cnt_1\, cnt_2\, cnt_3).[\mathtt{tt} :\ id := rnd(id_1, id_2, id_3) \,//\, id_1, id_2,\ id_3 := id]$.

Similarly, any hierarchical connector can be *flattened* into a simple one [10], allowing us to extend BIP operational semantics (Def. 3.6) to hierarchical connectors. This is done formally in the extended version of this paper [6].

4 Architecture Agnostic Model

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. As above, 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 *A* T/B component *is a tuple* $T = (\Sigma, P^{bot}, P^{top}, X_L : \mathsf{D}[L], \rightarrow)$, *where* Σ *is a set of states,* $P^{bot}, P^{top} \subseteq \mathcal{P}$ *are finite sets of* 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* $\big(q, \alpha(X), q'\big) \in \rightarrow$.

A T/B component $(\Sigma, P^{bot}, P^{top}, X_L : \mathsf{D}[L], \rightarrow)$ *is an* atomic 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 *(obtained by hierarchically composing atomic components and coordinators).*

Definition 4.2 *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 defined by the following rule:*

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

where $a = a^{top} \cup a^{bot}$*;* $up_{a^{top}}$ *and* up_X *are the corresponding components of the up expression;* $\mathbf{v}_{up}, \mathbf{v}_{dn} \in \mathsf{D}[P]$ *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).*

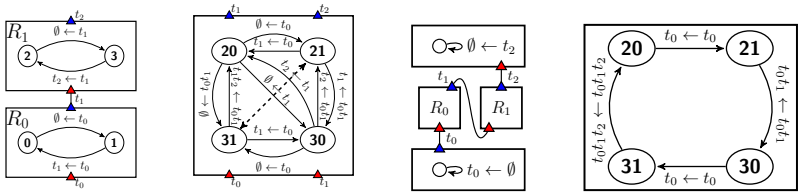
Notice that T/B components and their operational semantics generalise atomic BIP components (Def. 3.2). In particular, all components in the examples of Sect. 3 are T/B components without bottom ports.

Note 4.3 Notice that the values $\mathbf{v}_{up}$ and $\mathbf{v}_{dn}$ do not directly correspond to inputs and outputs. Indeed, in terms of the transferred data, the component *input* is the pair $(\mathbf{v}^{bot}_{up}, \mathbf{v}^{top}_{dn})$, whereas its *output* is the pair $(\mathbf{v}^{top}_{up}, \mathbf{v}^{bot}_{dn})$.

Recall the generalised function call metaphor (see the discussion after Def. 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 $\mathbf{v}^{top}_{dn}$, which are, finally, used by the function dn to compute $\mathbf{v}^{bot}_{dn}$.

4.2 Systems and Composition

Definition 4.4 *Let* $S = \{(\Sigma_i, P^{bot}_i, P^{top}_i, X_{L_i} : \mathsf{D}[L_i], \rightarrow)\}_{i=1}^n$ *be a finite set of T/B components and denote* $P^{bot} \triangleq \bigcup_{i=1}^n P^{bot}_i$ *and* $P^{top} \triangleq \bigcup_{i=1}^n P^{top}_i$*. 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.* $\forall i \neq j,\ X_i \cap X_j = P^{top}_i \cap P^{top}_j = \emptyset$*. A system is* closed *if* $P^{bot} = P^{top}$*.*



(a) (b) (c) (d)

Figure 6: T/B component model for the Mod-4 Counter

Definition 4.5 *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. Def. 4.8 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 defined by the following rules (*$i \neq j \in \{1, 2\}$*):*

$$\frac{q_i \xrightarrow{\alpha_i(X_i)} q_i'}{q_i q_j \xrightarrow{\alpha_i(X_i)} q_i' q_j} \quad , \qquad \frac{q_1 \xrightarrow{\alpha_1(X_1)} q_1' \quad q_2 \xrightarrow{\alpha_2(X_2)} q_2'}{q_1 q_2 \xrightarrow{(\alpha_1;\alpha_2)(X)} q_1' q_2'} \quad .$$

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 $\alpha_1 \parallel \alpha_2$, 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 [11, 14]) and the composition is undefined.

Example 4.6 Fig. 6a 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 models a Mod-2 counter, which produces one event on its top port (shown by blue outward-facing triangles) for every second event on its bottom port (shown by red inward-facing triangles). R_0 and R_1 share port t_1. Fig. 6b shows the T/B component $R_0 \parallel R_1$ (for clarity we omit two transitions indicated by the dotted green arrow).

Proposition 4.7 *Composition operator* $\parallel$ *is associative.*

Definition 4.8 *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* $E = \{(T_i, T_j) \mid P_i^{top} \cap P_j^{bot} \neq \emptyset\}$*. 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 Def. 4.5. Thus, by Prop. 4.7, the composed T/B component $\|S$ is well-defined.

As in process calculi like CCS [16], 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 Def. 4.5 with the second rule). The meaning of a complete system is defined as the largest closed sub-system obtained by pruning out all the non-matching transitions; thus the following definition.

Definition 4.9 *Let* $S = \{(\Sigma_i, P_i^{bot}, P_i^{top}, X_{L_i} : \mathsf{D}[L_i], \rightarrow)\}_{i=1}^n$ *be a closed composable system and let* $\|S = \big(\Sigma, P^{bot}, P^{top}, X_L : \mathsf{D}[L], \underset{par}{\longrightarrow}\big)$. *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 defined by the rule*

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

The second premise means that, for every bottom (resp. top) port, α must also contain the corresponding top (resp. bottom) port. Restriction, in our context, is the generalisation of the CCS restriction operator.

Example 4.10 Fig. 6c shows a system comprising T/B components R_0 and R_1 as in Ex. 4.6 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 Fig. 6d is, indeed, a Mod-4 counter.

Lemma 4.11 *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.*

4.3 T/B Component Encoding of BIP Models

Any atomic BIP component $B = (\Sigma, P, X_L : \mathsf{D}[L], \rightarrow)$ can be trivially encoded as a T/B component by making all ports of B top ports, i.e. $\tau(B) = (\Sigma, \emptyset, P, X_L : \mathsf{D}[L], \rightarrow)$. Thus, we only have to provide the encoding for connectors. Let $\alpha(X) = (w \leftarrow a).[g(X_a) : \ (x_w, X) := up(X_a) \,//\, X_a := dn(x_w, X)]$ be a simple connector with a set of local variables $X : \mathsf{D}$. The T/B component encoding of α is given by $\tau(\alpha) \triangleq \big(\{*\}, P, \{w\}, X : \mathsf{D}, \{* \xrightarrow{\alpha(X)} *\}\big)$.

Hierarchical connectors are encoded component-wise:
$\tau\big(\alpha\langle h\alpha_1, \ldots, h\alpha_n\rangle\big) \triangleq \big\{\tau(\alpha)\big\} \cup \bigcup_{i=1}^n \tau(h\alpha_i)$.

In the BIP operational semantics Def. 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. To enforce BIP semantics, for a set of connectors Γ, we add an *arbiter*: $\tau(\Gamma) = \big(\{*\}, P_\Gamma, \emptyset, \{y_w : \mathsf{D}_w \,|\, w \in P_\Gamma\}\{* \xrightarrow{\tilde{\alpha}} * \,|\, \alpha \in \Gamma\}\big)$,

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 α in the upward data transfer is reinjected back into the downward data transfer by $\tilde{\alpha}$.

Theorem 4.12 *Let $\mathcal{B}$ be a set of atomic BIP components and Γ be a set of hierarchical connectors and put $S = \{\tau(\Gamma)\} \cup \bigcup_{B\in\mathcal{B}}\{\tau(B)\} \cup \bigcup_{\alpha\in\Gamma}\tau(\alpha)$. 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. Def. 4.4). In [6], we provide the Java implementation of the Mod-4 counter from Ex. 4.6.

At runtime, we create a Java thread for each atomic component and a thread that plays the role of an *arbiter* for all the coordinators. The implementation of the execution engine can be drastically optimized in case where the coordinators are deterministic, i.e. 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 threads cyclically execute the following protocol: 1) Notify the top ports of the current outgoing transitions, whereof the guards are satisfied; 2) Notify the arbiter thread; 3) Wait for a notification from the arbiter; 4) Upon the notification from the arbiter, execute the action that corresponds to the received top port; 5) Modify the current state according to transition labeled by the received top port. Below is the algorithm of the atomic component 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. It is propagated upward by the atomic component thread until it reaches a top-level coordinator component (i.e. a coordinator whereof current outgoing transition does not have a top port).

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

1. Select non-deterministically an enabled top-level coordinator component, i.e. such that its the current outgoing transition has no top ports and all its bottom ports have been notified. (If such a component does not exist, a deadlock has occurred.)

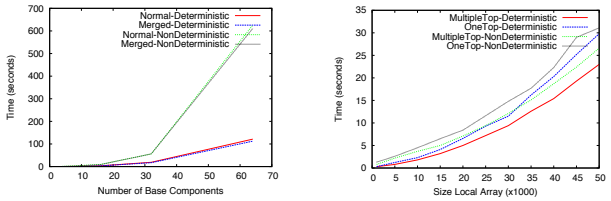
2. Execute the *dn* function of the selected transition and update the state of the coordinator accordingly.

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

4. When the downward 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 upward propagation without being modified during the downward propagation.

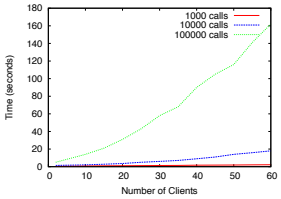
Notice that arbiter selects only one top-level coordinator even though there exits more than one top-level coordinator that are non conflicting. Two top-level coordinators are conflicting if the downward propagation will lead to notify the same atomic component but with two different top ports. Obviously, selecting two top-level coordinators that are conflicting will lead to the violation of the semantics presented in Sect. 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,



(a) Deterministic/Non-Deterministic

(b) One Top/Multiple-Top



(c) POTS

Figure 7: Performance (execution time in seconds) of the case-study examples: (a),(b) NSA; (c) POTS

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 Case Studies

5.2.1 Network Sorting Algorithm (NSA)

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 [10], we have provided a BIP application implementing this algorithm. In order to evaluate the results of the present paper, we have implemented an internalised version using T/B components. We have also implemented a modified version of this model, where we merge some coordinators, which might improve the performance. Detailed descriptions are available in [6].

Figures 7a and 7b provide benchmarks for NSA by considering the initial and merged models for the two implementations (deterministic and non-deterministic). Fig. 7a 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. Fig. 7b shows that the former implies slightly better performance.

5.2.2 Plain Old Telephone Service (POTS)

We have implemented a T/B component model for POTS [12], 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 coordinators that collect requests coming from the clients as follows: 1) $CallerAgregration$ collects dialing requests, 2) $CalleeAgregation$ 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 $CalleeAgregation$). $VoiceSync$ 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 $DiscAgregation_2$). More detailed description is available in [6]. The proposed model is very concise and can be modified incrementally, e.g. by adding new clients.

Fig. 7c shows the performance of POTS for three different values of the number of calls to be satisfied.

6 Related Work

Coordination [13] as a means to alleviate complexity in complex system design by distinguishing between a computing part comprising components involved in manipulating data and a coordination 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 [17], 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 [1, 3] 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 [18], 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 [14] and Statecharts [11]. Our model extends the interaction mechanism with data transfer. To avoid causality anomalies [14], 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 internalising 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, fully describing the control- and data-flow involved in an interaction. They are used both to specify connectors, i.e. architectural constraints, and 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 values computed when the interaction occurs. 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 implementations in the form of various execution engines [4]. 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. 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 [10] 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 internalisation 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 and component variables as in the Dy-BIP coordination language [9]. These could be used in the guards and affected by the *up* and *dn* functions, making 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 [7, 8]. 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 *ICSE*, 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. ArchJava: connecting software architecture to implementation. In *ICSE*, pages 187–197. ACM, 2002.

[4] 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.

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

[6] Simon Bliudze, Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Architecture internalisation in BIP. Technical Report EPFL-REPORT-196997, EPFL IC IIF RiSD, February 2014. Available at: `http://infoscience.epfl.ch/record/196997`.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

[15] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *SIGSOFT ESEC/FSE*, volume 1301 of *LNCS*, pages 60–76. Springer, 1997.

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

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

[18] 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.