# Modeling Dynamic architectures using Dy-BIP

Marius Bozga, Mohamad Jaber, Nikolaos Maris, Joseph Sifakis

# Modeling Dynamic Architectures Using Dy-BIP[⋆]

Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis
Firstname.Lastname@imag.fr

UJF-Grenoble 1 / CNRS, VERIMAG UMR 5104, Grenoble, F-38041, France

**Abstract.** Dynamic architectures in which interactions between components can evolve during execution, are essential for modern computing systems such as web-based systems, reconfigurable middleware, wireless sensor networks and fault-tolerant systems. Currently, we lack rigorous frameworks for their modeling, development and implementation. We propose Dy-BIP a dynamic extension of the BIP component framework rooted in rigorous operational semantics and supporting a powerful and high-level set of primitives for describing dynamic interactions. These are expressed as symbolic constraints offered by interacting components and computed efficiently by an execution Engine. We present experimental results which validate the effectiveness of Dy-BIP and show significant advantages over using static architecture models.

## 1 Introduction

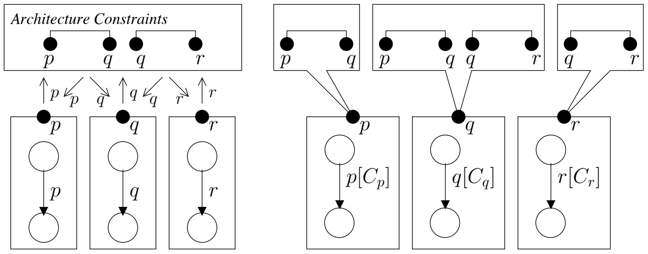
Architectures are essential for mastering the complexity of systems and facilitate their analysis and evolution. They allow a separation between detailed behavior of components and their overall coordination. Coordination is usually expressed by constraints that define possible interactions between components. There exists a large number of formalisms supporting a concept of architecture, including software component frameworks, systems description languages and hardware description languages. Despite an abundant literature and a considerable volume of research, there is no agreement on a common concept of architecture, while most definitions agree on the core e.g. diagrammatic representations by using connectors. This is due to two main reasons.

First, is the lack of rigorous operational semantics defining architectures as composition operators on components. That is the behavior of a composite component is inferred from the behavior of its constituent components by applying architectural constraints. For existing component frameworks, the definition of rigorous operational semantics runs into many technical difficulties. They fail to clearly separate between behavior of components and architecture. Connectors are not just memoryless switching elements. They can be considered as special types of components with memory e.g. fifo queues and specific behavior. Another difficulty stems from verbose architecture definitions e.g. by using ADLs [1], that do not rely on a minimal set of concepts. Such definitions are hardly amenable to formalization. Finally, some frameworks [2] use declarative languages e.g. first order logic to express global architecture constraints which are useful for checking correctness but as a rule do not provide a basis for defining operational semantics.

[⋆] This work is partially supported by the FP7 IP ASCENS.

The second is the distinction between static and dynamic architectures. Usually, hardware and system description languages rely on static architecture models. The relationships between components are known at design time and are explicitly specified as a set of connectors defining possible interactions. Dynamic architectures are needed for modeling reconfigurable systems or systems that adapt their behavior to changing environments. They are defined as the composition of dynamically changing architecture constraints offered by their constituent components. Filling the gap between static and dynamic architecture models raises a set of interesting problems. In principle, dynamic architecture models are more general: each configuration corresponds to a static architecture model. Is it possible to define a dynamic architecture modeling language as an extension of a static architecture modeling language? Furthermore, if we restrict to systems with a finite - although potentially large - set of possible configurations, any dynamic architecture model can be translated into a static architecture model. Such a translation can yield very complex static architecture models. As a rule, using dynamic architectures may lead to more concise models. However, static architecture models can be executed more efficiently thanks to the global and static knowledge of connectors [3].

We propose the Dy-BIP component framework based on rigorous operational semantics for modeling both static and dynamic architectures. Dy-BIP can be considered as an extension of the BIP language [4] for the construction of composite hierarchically structured components from atomic components. These are characterized by their behavior specified as automata extended with data and functions described in C. A transition of an automaton is labeled by a port name, a guard (boolean condition on local data) and an action (computation on local data). In BIP architectures are composition operators on components defining their interactions. An interaction is described as a set of ports from different components. It can be executed if there exists a set of enabled transitions labeled by its ports. The completion of an interaction is followed by the completion of the involved transitions: execution of the corresponding actions followed by a move to the target state. An operational semantics for BIP has been defined in [5]. It provides a basis for the implementation of an Engine that orchestrates component execution. The Engine knows the set of the interactions modeling the architecture. It executes cyclically and atomically the following three-step protocol: 1) from a state each component sends to the Engine the ports of its enabled transitions; 2) the Engine computes the set of feasible interactions (sets of received ports corresponding to some interaction); 3) the Engine chooses non-deterministically one interaction amongst the feasible interactions by sending back to the components the names of their ports involved in this interaction. Figure 1(a) shows a static architecture defined by interactions $pq$ and $qr$. Its consists of three components offering communications through ports $p$, $q$ and $r$. In contrast to BIP, the set of interactions characterizing architectures in Dy-BIP changes dynamically with states. A port $p$ has an associated architecture constraint $C_p$ which describes possible sets of interactions involving $p$. Feasible interactions from a state are computed as maximal solutions of constraints obtained as the conjunction of constraints offered by enabled transitions. Figure 1(b) illustrates a dynamic architecture with three components offering ports $p$, $q$ and $r$ with associated constraints $C_p$, $C_q$ and $C_r$. As for the static architecture, the possible interactions are $pq$ and $qr$.



(a) Static architecture (b) Dynamic architecture

**Fig. 1.** Static and dynamic architecture.

We provide operational semantics for Dy-BIP implemented by an Engine which as for BIP, orchestrates components by executing atomically a three-step protocol. The protocol differs in that components send not only port names of enabled transitions but also their associated architecture constraints.

Dy-BIP is an extension of BIP. A BIP model with a global architecture constraint $C$, can be represented as a Dy-BIP model such that the constraint $C_p$ associated with a port $p$ is the set of the interactions of $C$ involving $p$.

Dy-BIP allows modeling dynamic architectures as the composition of instances of component types. For the sake of simplicity, we assume that there is no dynamic creation/deletion of component instances. The main contributions are the following:

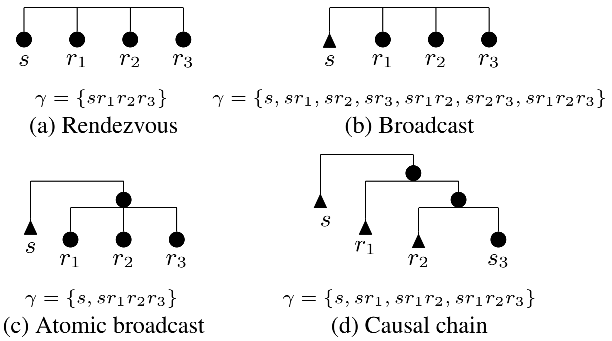
- Definition of a logic for the description of architecture constraints. The logic is expressive and amenable to analysis and execution. It encompasses quantification over instances of component types. Formulas involve port names used as logical variables and characterise sets of interactions. Given a formula, a feasible interaction is any set of ports assigned true by a valuation which satisfies the formula.
- Study of a semantic model and a modeling methodology for writing architecture constraints associated with ports. For a port $p$, the associated constraint is decomposed into three types of constraints characterizing interaction between ports [6]: "causal constraint", "acceptance constraint", "filter constraint". Causal constraint defines the ports required for interaction. Acceptance constraint defines optional ports for participation. Filter constraints are invariants used to discriminate undesirable configurations of a component's environment.
- Implementation principles for Engines handling symbolic architecture constraints. The proposed implementation is based on the resolution of architecture constraints on-the-fly. The Engines use efficient constraint resolution techniques based on BDDs. For a given model, quantifiers over components can be eliminated and formulas become boolean expressions on ports.
- Experimental results and benchmarks showing benefits from using dynamic architectures compared to static architectures. We consider several examples showing that compositional modeling of dynamic architectures allows enhanced conciseness and rigorousness. In particular, it is possible to master complexity of intricate dynamic interactions by compositional specification of interactions of individual components.

The paper is structured as follows. Section 2 presents related work. Section 3 describes the semantic model for Dy-BIP. Section 4 presents the dynamic architecture description language and the methodology for writing constraints. Section 5 is dedicated to examples and experimental results. Section 6 concludes and discusses future work directions.

## 2 Related work

In contrast to other frameworks [7, 8] Dy-BIP relies on a clear distinction between behavior and architecture as a set of stateless architecture constraints characterizing interactions. Architecture constraints are specified compositionally as the conjunction of individual architecture constraints of components. Existing frameworks usually describe dynamic architectures as a set of global transitions between configurations. Only process algebras adopt a compositional approach e.g. pi-calculus [9]. Nonetheless, they do not encompass a concept of architecture as behavior and composition operators are intermingled. Dy-BIP differs from other formalisms such as [10] in that it has rigorous operational semantics. In [2], a first order logic extended with architecture-specific predicates is used. However, there is no clear methodology on how to express synchronisation protocols (e.g., rendezvous, broadcast) whose combination is expressive enough to represent any kind of interaction and avoids the exhaustive enumeration of all possible interactions [11]. In [12], a dynamic architecture is defined as a set of global transitions between global configurations. These transitions are expressed in a first order logic extended with architecture-specific predicates. The same logic is used in [13, 14] but global configurations are computed at runtime from the local constraints of each component. Dy-BIP follows the same approach but constraints are stateless (they are based on the boolean representation of causal rules [6]) and take advantage of the stateful behavior of the components by eliminating some of the undesirable global configurations implicitly. [15] provides an operational semantics based on the composition of global configurations from local ones. These express three forms of dependencies between services (mandatory, optional and negative). Nonetheless, dynamism is supported only at the installation phase.

In BIP [4], coordination between components is modeled by using connectors [6]. A simple (or *flat*) connector is an expression of the form $p'_1 \ldots p'_k p_{k+1} \ldots p_n$ where primed $p'_i$ ports are triggers, and unprimed ports $p_j$ are synchrons. For a flat connector involving the set of ports $\{p_1, \ldots, p_n\}$, interaction semantics defines the set of its interactions $\gamma$ by the following rule: an interaction is any non-empty subset of $\{p_1, \ldots, p_n\}$ which contains some port that is trigger; otherwise (if all ports are synchrons), the only possible interaction is the maximal one, that is $p_1 \ldots p_n$. Connectors, representing these two protocols for a sender $s$ and receivers $r_1$; $r_2$; $r_3$, are shown in 2-(a,b). Triangles represent triggers, whereas bullets represent synchrons. Hierarchical connectors are expressions composed of types ports and/or typed sub-connectors. Figure 2-c shows a connector realizing an atomic broadcast from a port $s$ to ports $r_1, r_2, r_3$. The sending port $s$ is trigger, and the three receiving ports are strongly synchronized in a sub-connector itself typed as a synchron. The connector shown in Figure 2-d is a causal chain of interactions initiated by the port $s$, the possible interactions are $s, sr_1, sr_1r_2, sr_1r_2r_3$.



$\gamma = \{sr_1r_2r_3\}$
(a) Rendezvous

$\gamma = \{s, sr_1, sr_2, sr_3, sr_1r_2, sr_2r_3, sr_1r_2r_3\}$
(b) Broadcast

$\gamma = \{s, sr_1r_2r_3\}$
(c) Atomic broadcast

$\gamma = \{s, sr_1, sr_1r_2, sr_1r_2r_3\}$
(d) Causal chain

**Fig. 2.** Graphic representation of connectors

Connectors can be used to define easily any type of coordination between components. However, in this case connectors and their interactions are defined statically, and this leads to inefficiency for systems with dynamically changing architecture.

## 3 The Dynamic-BIP model

Dy-BIP offers primitives for dynamic architecture modeling. Atomic components are transition systems. Transitions are labeled with ports, that is, action names, and constraints for interaction with other components.

Ports are used to define interactions between atomic components. Henceforth $\mathcal{P}$ is a universal set of ports. An interaction is a non empty subset $a \subseteq \mathcal{P}$. To simplify notation, an interaction $a = \{p_1, p_2, \dots p_n\}$ is simply denoted by $p_1p_2 \dots p_n$.

### 3.1 Interaction constraints

To introduce dynamic architectures, we consider that each atomic component provides its own interaction constraints at each computation step. A global interaction is defined as a solution of the set of interaction constraints offered by components. As the interactions at some state may depend on interactions in the past, it is necessary to parametrize interaction constraints by *history variables* which keep track of the interactions already executed. For example, in a protocol, if A sends a message to B, then A can record the identity of B in a history variable to remember that an acknowledgement is expected from B. If A does not use history variables then it is necessary to encode the identity of the receiver in control locations and this may result in an explosion of the number of its control locations.

**Definition 1 (Interaction Constraint).** *Given a set of history variables $H$, an interaction constraint $C$ is defined by the following grammar:*

$$C ::= \texttt{true} \mid p{\in}h \mid p \mid \neg C \mid C \wedge C \qquad (1)$$

*where $p \in \mathcal{P}$ is a port, and $h \in H$ is a history variable.*

The syntax of interactions constraints is tacitly extended for all boolean operators e.g., `false`, $\Rightarrow$, $\vee$ in the usual way. To simplify notation, we overload the meaning of a port symbol p: within $p \in h$ it denotes the port itself whereas $p$ alone denotes a boolean variable. This ambiguity is removed in the two-step definition of semantics given below. We denote by $\mathcal{C}$ the set of all interaction constraints and by $\mathcal{C}^b$ the subset of boolean constraints, that is, without history variables.

Given a valuation of history variables $\mu : H \mapsto 2^{\mathcal{P}}$, an interaction constraint $C \in \mathcal{C}$ defines a boolean interaction constraint $[\![C]\!]_\mu$ by the following rules:

$$\begin{aligned}
[\![\mathtt{true}]\!]_\mu &= \mathtt{true} \\
[\![p \in h]\!]_\mu &= \begin{cases} \mathtt{true} & \text{if } p \in \mu(h) \\ \mathtt{false} & \text{otherwise} \end{cases} \\
[\![p]\!]_\mu &= p \\
[\![\neg C]\!]_\mu &= \neg [\![C]\!]_\mu \\
[\![C_1 \wedge C_2]\!]_\mu &= [\![C_1]\!]_\mu \wedge [\![C_2]\!]_\mu
\end{aligned} \tag{2}$$

That is, the terms of the form $p \in h$ are replaced by `true` or `false` as the case may be: `true` if port $p$ belongs to the stored interaction and `false` otherwise. All the other terms of the interaction constraint remain unchanged.

An interaction $a \subseteq \mathcal{P}$ satisfies a boolean constraint $C \in \mathcal{C}^b$ (denoted by $a \models C$), as defined by the following rules:

$$\begin{aligned}
&a \models \mathtt{true} \\
&a \models p && \Leftrightarrow p \in a \\
&a \models \neg C && \Leftrightarrow \neg(a \models C) \\
&a \models C_1 \wedge C_2 && \Leftrightarrow (a \models C_1) \wedge (a \models C_2)
\end{aligned} \tag{3}$$

We denote with $\mathcal{I}(C)$ and $\mathcal{I}^{max}(C)$ the set of interactions and respectively maximal interactions satisfying $C$, formally:

$$\begin{aligned}
\mathcal{I}(C) &= \{a \mid a \models C\} \\
\mathcal{I}^{max}(C) &= \{a \in \mathcal{I}(C) \mid \nexists a' \in \mathcal{I}(C) \,.\, a' \supset a\}
\end{aligned} \tag{4}$$

For example, we can specify interaction constraints for: (1) *rendez-vous* between $p_1, p_2, p_3$ by $C_1 = (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_3) \wedge (p_3 \Rightarrow p_1)$, the only possible interaction is the maximal one, that is $\mathcal{I}(C_1) = \mathcal{I}^{max}(C_1) = \{p_1p_2p_3\}$; (2) *broadcast* where $s$ is the sending port and $r_1, r_2, r_3$ are the receiving ports by $C_2 = (true \Rightarrow s) \wedge (r_1 \Rightarrow s) \wedge (r_2 \Rightarrow s) \wedge (r_3 \Rightarrow s)$, the possible interactions are $\mathcal{I}(C_2) = \{s, sr_1, sr_1r_2, sr_1r_2r_3\}$ and $\mathcal{I}^{max}(C_2) = \{sr_1r_2r_3\}$; (3) the constraint $p \Rightarrow$ `false` means absence of $p$ from any interaction; (4) $p \Rightarrow$ `true` allows inclusion of $p$ in any interaction.

### 3.2 Atomic components

An atomic component is an automaton extended with history variables. Transitions represent relations on control locations (local states). Each transition is labeled by a port, an interaction constraint and a set of history variables to be updated.

**Definition 2 (Atomic Component).** *An atomic component is a tuple* $B = (L, P, H, T)$*, where,*

- $L$ *is a finite set of control locations;*
- $P \subseteq \mathcal{P}$ *is a finite set of ports;*
- $H$ *is a finite set of history variables;*
- $T \subseteq L \times P \times \mathcal{C} \times 2^H \times L$ *is a finite set of transitions. Each transition* $(\ell, p, C, \mathbf{h}, \ell')$*, denoted by* $\ell \xrightarrow{p, C, \mathbf{h}} \ell'$ *is labeled with:*
  - $p \in P$*, the port offered for interaction*
  - $C \in \mathcal{C}$*, the interaction constraint on* $\mathcal{P}$ *and* $H$
  - $\mathbf{h} \subseteq H$*, the set of history variables to be updated*

Given $\mu : H \mapsto 2^{\mathcal{P}}$ a valuation of the history variables $H$, the state of an atomic component $B = (L, P, H, T)$ is a pair $(\ell, \mu)$, where $\ell \in L$ is a control location. $Q = L \times \boldsymbol{\mu}$ is the set of states ot the atomic component $B$, where $\boldsymbol{\mu}$ denotes the set of valuations on $H$. For each state $(\ell, \mu) \in Q$, we define its associated *state constraint* $SC(\ell, \mu)$ as follows:

$$SC(\ell, \mu) = \left[ \bigvee_{\ell \xrightarrow{p, C, \mathbf{h}} \ell'} \left( p \wedge [\![C]\!]_\mu \wedge \bigwedge_{p' \in P \setminus \{p\}} \neg p' \right) \right] \vee \bigwedge_{p \in P} \neg p \qquad (5)$$

The state constraint characterizes the set of possible contributions of the component to a global interaction at state $(\ell, \mu)$. Either the component executes some transition $\ell \xrightarrow{p, C, \mathbf{h}} \ell'$ and offers interactions (1) involving $p$ and excluding all other ports labeling transitions from this state, that is, $p \wedge \bigwedge_{p' \in P \setminus \{p\}} \neg p'$ holds and (2) involving ports which satisfy the constraint $C$ for the valuation $\mu$, that is, $[\![C]\!]_\mu$. Or, the component does not interact, if none of its ports is used, that is $\bigwedge_{p \in P} \neg p$.

### 3.3 Composition

A system $S = B_1 \| \ldots \| B_n$ is defined as the composition of a set of atomic components $B_i = (L_i, P_i, H_i, T_i)_{i=1,n}$. We assume that the sets of locations, ports and history variables are pairwise disjoint. The semantics of the composition is provided by the following definition.

**Definition 3 (Composition).** *The behavior of a system* $S = B_1 \| \ldots \| B_n$ *is a labeled transition system* $(Q, \Sigma, \rightarrow)$*, where,*

- $Q = \prod_{i=1}^{n} Q_i$ *is the set of states, where* $Q_i$ *is the set of states of component* $B_i$*;*
- $\Sigma = 2^P$ *is the set of interactions, where* $P = \cup_{i=1}^{n} P_i$*;*
- $\rightarrow \subseteq Q \times \Sigma \times Q$ *is the set of transitions, defined by the following rule:*

$$\frac{\begin{array}{c}\{\mu_i : H_i \mapsto P\}_{i=1}^n \quad a = \{p_j\}_{j\in J} \in \mathcal{I}^{max}(\bigwedge_{i=1}^n SC(\ell_i,\mu_i)) \\ \forall j \in J.\quad \ell_j \xrightarrow{p_j, C_j, \mathbf{h}_j} \ell'_j \quad \bigwedge_{j\in J} [\![C_j]\!]_{\mu_j} \quad \mu'_j = \mu_j[a/\mathbf{h}_j] \\ \forall j \notin J.\quad \ell'_j = \ell_j \quad \mu'_j = \mu_j\end{array}}{((\ell_1,\mu_1),\ldots,(\ell_n,\mu_n)) \xrightarrow{a} ((\ell'_1,\mu'_1),\ldots,(\ell'_n,\mu'_n))}$$

Intuitively, system transitions are taken for maximal interactions $a$ that satisfy *all* state constraints defined by atomic components $\bigwedge_{i=1}^n SC(\ell_i, \mu_i)$. The components participating in the interaction change their location according to the selected transition and update their local valuation of history variables. The components which do not participate keep their state unchanged.

The composition semantics has been implemented by using a centralized execution Engine. The Engine gathers current state interaction constraints from all atomic components. Then, it builds the overall system of constraints and solves it, that is, finds the set of maximally satisfying interactions. One maximal interaction is then selected and executed atomically by all involved components. This operation is repeated by the Engine on-the-fly, at every state reached at execution. More formally, atomic components interact and coordinate their execution through the Engine according to the following protocol:

1. Each component $B_i$, computes its state constraint $SC(\ell_i, \mu_i)$ and sends it to the Engine.
2. The Engine computes the global constraint $GC$ as the conjunction of state constraints defined by atomic components $GC = \bigwedge_{i=1}^n SC(\ell_i, \mu_i)$.
3. The Engine picks some maximal interaction $a$ on $\mathcal{P}$, such that $a \models GC$. That is, $a \in \mathcal{I}^{max}(GC)$.
4. The Engine notifies the selected interaction $a$ to all participating components, that is, all $B_j$ such that $a \cap P_j \neq \emptyset$.
5. Each notified component $B_j$ executes its local transition labeled by $p_j = a \cap P_j$ and updates its history variables.

*Example 1.* Figure 3 shows an example of a composite component consisting of three atomic components $B_1$, $B_2$, and $B_3$. Component $B_1$ requests synchronization with either $B_2$ or $B_3$, and then performs some computation with the synchronized component. To do so, we model component $B_1$ as follows. It has two control locations $l_1$, $l_2$, two ports $s$, $r$, and a history variable $h$. From control location $l_1$, the transition labeled by the port $s$ requests synchronisation with ports $b_1$ or $b_2$ ($s \Rightarrow b_1 \vee b_2$), forbids the synchronisation with $b_1$ and $b_2$ at the same time ($\neg(b_1 \wedge b_2)$), and forbids the synchronisation with $f_1$ and $f_2$ ($\neg f_1 \wedge \neg f_2$). After executing this transition, the executed interaction is stored in the history variable $h$ (**Update** $h$), to store the identity of the synchronized component. From control location $l_2$, the transition labeled by the port $r$, depending on the value of the history variable $h$, it either (1) requests synchronisation with port $f_1$ and forbids the synchronisation with port $f_2$; or (2) requests synchronisation with port $f_2$ and forbids the synchronisation with port $f_1$. In both cases, it forbids the synchronisation with ports $b_1$ and $b_2$ ($\neg b_1 \wedge \neg b_2$).

From location $l_1$ the state constraint is $SC(l_1, \mu_1) = (s \wedge C_s \wedge \neg r) \vee (\neg s \wedge \neg r)$ regardless of the value of $\mu_1$. This is due to the fact that constraint $C_s$ does not depend on

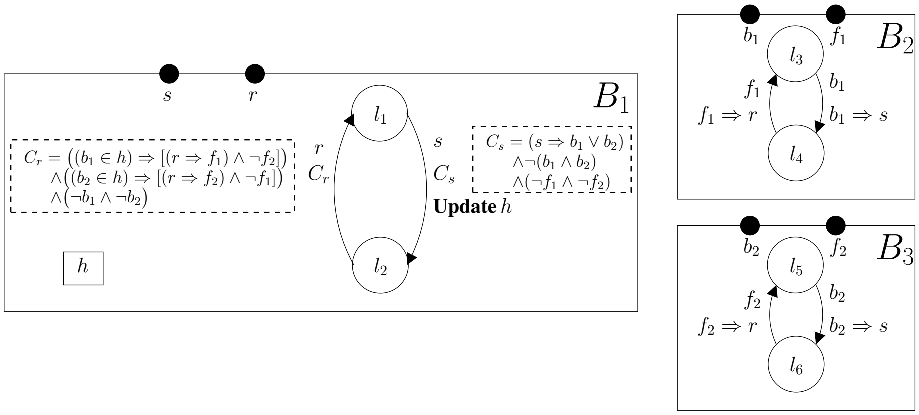

$s$
$r$
$B_1$
$l_1$
$l_2$
$r$
$C_r$
$s$
$C_s$
**Update** $h$
$C_r = \big((b_1 \in h) \Rightarrow [(r \Rightarrow f_1) \wedge \neg f_2]\big)$
$\wedge\big((b_2 \in h) \Rightarrow [(r \Rightarrow f_2) \wedge \neg f_1]\big)$
$\wedge\big(\neg b_1 \wedge \neg b_2\big)$
$C_s = (s \Rightarrow b_1 \vee b_2)$
$\wedge\neg(b_1 \wedge b_2)$
$\wedge(\neg f_1 \wedge \neg f_2)$
$h$
$b_1$
$f_1$
$B_2$
$l_3$
$l_4$
$f_1$
$f_1 \Rightarrow r$
$b_1$
$b_1 \Rightarrow s$
$b_2$
$f_2$
$B_3$
$l_5$
$l_6$
$f_2$
$f_2 \Rightarrow r$
$b_2$
$b_2 \Rightarrow s$

**Fig. 3.** An example of a composite component in Dy-BIP.

history variables. For this constraint, the possible interactions are $\mathcal{I}^{max}(SC(l_1, \mu)) = \{sb_1, sb_2\}$. From location $l_2$, if the history variable $h$ contains the interaction $sb_2$, then the state constraint is $SC(l_2, \mu_1) = (r \wedge [\![C_r]\!]_{\mu_1} \wedge \neg s) \vee (\neg s \wedge \neg r)$, where $[\![C_r]\!]_{\mu_1} = \big(\texttt{false} \Rightarrow [(r \Rightarrow f_1) \wedge \neg f_2]\big) \wedge \big(\texttt{true} \Rightarrow [(r \Rightarrow f_2) \wedge \neg f_1]\big) \wedge \big(\neg b_1 \wedge \neg b_2\big)$. For this constraint, the only possible interaction is $\mathcal{I}^{max}(SC(l_2, \mu_1)) = \{rf_2\}$.

Initially, components $B_1$, $B_2$, and $B_3$ are in locations $l_1$, $l_3$, and $l_5$, respectively. Starting from these locations, the Engine coordinates execution of these components as follows: (1) Components $B_1$, $B_2$, and $B_3$ compute their state constraints $SC(l_1, \mu_1)$, $SC(l_3, \mu_2)$, and $SC(l_5, \mu_3)$, respectively, where, $SC(l_1, \mu_1) = (s \wedge C_s \wedge \neg r) \vee (\neg s \wedge \neg r)$, $SC(l_3, \mu_2) = (b_1 \wedge (b_1 \Rightarrow s) \wedge \neg f_1) \vee (\neg b_1 \wedge \neg f_1)$, and $SC(l_5, \mu_3) = (b_2 \wedge (b_2 \Rightarrow s) \wedge \neg f_2) \vee (\neg b_2 \wedge \neg f_2)$; (2) The Engine picks any interaction from $\mathcal{I}^{max}(SC(l_1, \mu_1) \wedge SC(l_3, \mu_2) \wedge SC(l_5, \mu_3)) = \{sb_1, sb_2\}$; (3) If the Engine selects the interaction $sb_2$, it notifies components $B_1$ and $B_3$; (4) Components $B_1$ and $B_3$ execute their transitions labeled by $s$ and $b_2$, respectively. Moreover, component $B_1$ sets its history variable $h$ to $sb_2$.

## 4 Methodology for writing interaction constraints

Writing interaction constraints associated with transitions of atomic components, in the proposed declarative language may be error-prone or may lead to incomplete specifications. We provide a methodology based on a classification of constraints and on a set of macro-notations for enhancing soundness and completeness. The classification distinguishes between interactions in which a port p must, may or must not be involved. It allows a systematic analysis of interaction capabilities of components to make sure that no essential properties are omitted. Macro-notations allow a compact and high-level expression of the most commonly used constraints, thus avoiding specification errors.

We extend the interaction constraint language towards a first order logic with quantification over component instances. This extension is useful because in practice, systems are built from multiple, replicated instances of components of different types. The formulas of the logic interaction constraints are therefore defined as follows:

$$C ::= \texttt{true} \mid x.p \in h \mid x.p \mid x = y \mid x = \texttt{self} \mid \neg C \mid C \wedge C \mid \forall x{:}T.C(x) \qquad (6)$$

In this definition, $T$ denotes a component type. Each component type represents a set of component instances with identical interfaces and behavior. The variables $x, y$ range over component instances. These variables must occur in the scope of a quantifier, e.g. $\forall x : T.C(x)$. They are strongly typed and moreover, they can be tested for equality. Additionally, `self` represents a fixed component instance, that is, the (context) component where the constraint belongs. The remaining syntactic constructs are directly lifted from the propositional case: $h \in H$ denotes a history variable and $x.p$ denote the port $p$ belonging to component instance $x$. As previously, we consider the standard extension of this logic for all boolean operators and existential quantification.

In the sequel, we consider systems consisting of finitely many instances for each component type. Under this restriction, an interaction constraint written in the logic above boils down into a propositional interaction constraint by (1) substituting `self` by the current component instance, (2) elimination of universal quantifiers and (3) evaluation of equality constraints. For example, for a component type $T$ with instances $t_1, \ldots t_k$, universal quantifiers of the form $\forall x{:}T$ can be eliminated:

$$\forall x{:}T.C(x) \equiv C(t_1) \wedge \ldots \wedge C(t_k) \qquad (7)$$

In the rest of this section, we provide guidelines for writing interaction constraints in this logic. Consider a fixed transition in some component (type) which is labeled by a port $p$. The associated interaction constraint $C_p$ is a conjunction of three types of constraints, respectively *causal*, *acceptance* and *filtering*, as explained below.

### 4.1 Causal constraints

These constraints are used to specify the ports required for interactions of $p$. At propositional level, they can be reduced to the one of the two following forms, either $\texttt{true} \Rightarrow p$ or $p \Rightarrow C$, where $C$ is a boolean interaction constraint without negated ports. To express such constraints in practice, we provide hereafter few useful abbreviations:

- **Trigger** $\equiv \texttt{true} \Rightarrow \texttt{self}.p$. This constraint specifies that port $p$ is a trigger, that is, the transition labeled by **Trigger** does not require synchronization with transitions of other components to occur.
- **Require** $T.q \equiv \exists x : T.(\texttt{self}.p \Rightarrow x.q)$. This constraint specifies that an arbitrary instance of component type $T$ must participate with the port $q$ in the interaction involving $p$.
- **Require** $x_1.q_1 ... x_n.q_n[x_1 : T_1 \ldots x_n : T_n | C_1(x_1, ..., x_n)] \equiv \exists x_1 : T_1 ... \exists x_n : T_n. \big(C_1(x_1, ..., x_n) \wedge \texttt{self}.p \Rightarrow (x_1.q_1 \wedge \ldots \wedge x_n.q_n)\big)$. This is the most general type of require constraint. It specifies that a set of component instances $x_1, \ldots x_n$ satisfying the constraint $C_1(x_1 \ldots x_n)$ must jointly participate with respectively ports $q_1, \ldots q_n$ in the interaction involving $p$. Usually, the constraint $C_1$ is used to check previous participation of $x_1 \ldots x_n$ in interactions recorded into history variables.

### 4.2 Acceptance constraints

These constraints define optional ports for participation used to define the boundary of interactions. They are expressed by excluding explicitly from interactions all the ports that are not optional. At propositional level, they are of the form $r \Rightarrow \texttt{false}$ where port $r$ is excluded from interaction. In practice, we use the following abbreviations:

- **Accept** $T.q \equiv \bigwedge_{(T',q')\neq(T,q)} \forall x : T'.\big(x.q' \neq \texttt{self}.p \Rightarrow (x.q' \Rightarrow \texttt{false})\big)$. This constraint accepts only ports $q$ of component instances of type $T$.
- **Accept** $x.q[x : T|x.r \in h] \equiv$ **Accept** $T.q \wedge \forall x : T.\big(x.r \notin h \Rightarrow (x.q \Rightarrow \texttt{false})\big)$. This constraint restricts participation to ports of component instances $x$ that had participated in the interaction stored in $h$.

### 4.3 Filtering constraints

These constraints are used to exclude some of interactions allowed by causal and acceptance constraints. At propositional level, filter constraints are of the form $p \Rightarrow C$ where each monomial in $C$ has at least one negated port. In practice, we are commonly requiring unicity constraints of the form:

- **Unique** $T.q \equiv \forall x : T.\forall y : T.\ (x.q \wedge y.q \Rightarrow x = y)$. This constraint forbids the participation of $p$ with more than one instance of component type $T$ with the port $q$.

## 5 Experimental results

The operational semantics of Dy-BIP has been implemented using the CUDD BDD package [1]. We compare execution times for Dy-BIP and BIP (where we define architecture statically) for a Master-Slave example. Two other non-trivial examples, Fault-Tolerant Servers, and Tracker and Peers are provided. Static architecture modeling for these examples in BIP leads to complex descriptions and may be error-prone. Dy-BIP models are concise and can be efficiently implemented. We present below the three examples and the simulation results. Execution times are provided for executing $1000$ interactions of the Engine running on PC Quad-Xeon 2.67HGz with 6GB RAM.

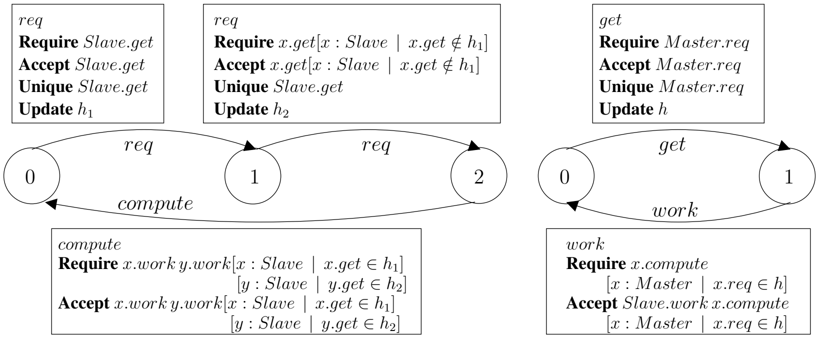
### 5.1 Masters and Slaves

In this example, we consider two scenarios. In the first scenario, is a system consisting of $M$ Masters and $S$ Slaves. Each Master sends requests sequentially to two Slaves, and then performs some computation involving both of them. The model of the Master component type is shown in Figure 4(a). Initially, from location $0$, a Master requests synchronization with some Slave. To do so, it must synchronize his $request$ port with the $get$ port of any Slave (**Require** $Slave.get$). As it requires one and only one Slave, we add a **Unique** constraint. When a Master instance $m_1$ synchronizes with

[1] CUDD: CU decision diagram package (`http://vlsi.colorado.edu/˜fabio/CUDD/`)

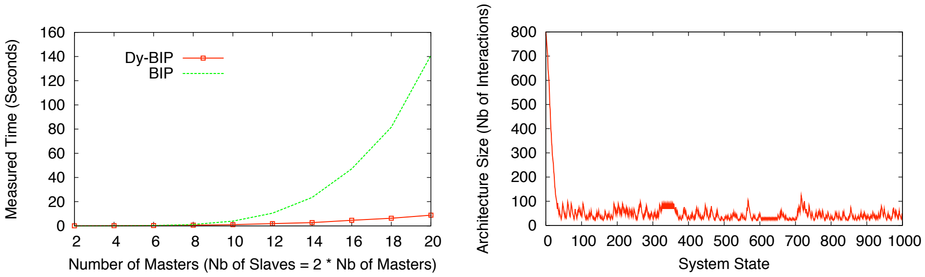
some Slave instance, say $s_1$, interaction $m_1.req\,s_1.get$ is chosen. Then, the Master $m_1$ has to keep track of the identity of the synchronized Slave ($s_1$) by saving the interaction $m_1.req\,s_1.get$ to the history variable $h_1$ (**Update** $h_1$). From location 1 the Master requires another Slave to synchronize with it, and keeps again track of its identity. Finally, from location 2, the Master establishes a ternary interaction with the two Slaves recorded in its history variables.

The Slave component type is shown if Figure 4(b). It accepts a request and provides a response. In order to allow participation in ternary interactions, the $Accept$ clause includes the port of one Master and the work port of another Slave. Notice that, the statically predefined connector structure (using BIP) needs $M \times S + M \times S \times (S-1)$ interactions. Figure 5(a) shows execution times of 1000 interactions for BIP and Dy-BIP, as a function of the number of components in the system. Dy-BIP considerably outperforms BIP. Figure 5(b) shows the number of the interactions created dynamically for Dy-BIP along the evolution of the system for 20 Masters and 40 Slaves. Notice that, for BIP the number of the interactions created is 32000 regardless of the system state.



(a) Master type (b) Slave type

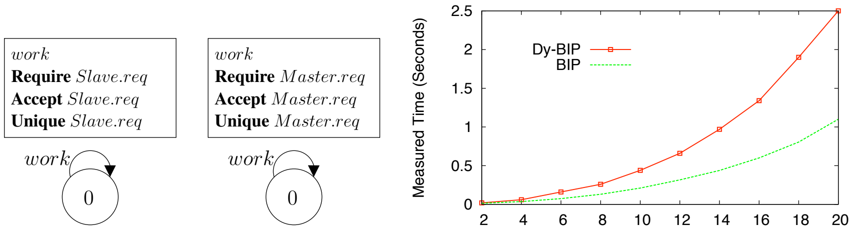
**Fig. 4.** Masters and Slaves in Dy-BIP.



(a) Master-Slave execution time (b) Architecture variation Dy-BIP

**Fig. 5.** Masters and Slaves execution time and variation of dynamic architecture.

In the second scenario, we developed a simplified version of the Master-Slave model presented above. Each Master performs some computation in tight synchronization with any Slave. The model of the Master and Slave component types are shown in Figures 6(a) and 6(b). As expected, Figure 6(c) shows that the execution time for BIP outperforms Dy-BIP. This is due to the fact that the number of the interactions created is $M \times N$ for BIP as well as for Dy-BIP regardless of the system state.



(a) Master type (b) Slave type (c) Simple Master-Slave execution time

**Fig. 6.** Simple Master-Slave in Dy-BIP.

### 5.2 Fault-Tolerant Servers

This example is inspired from the Fault-Tolerant Client-Server System presented in [3]. We consider a system consisting of a fixed number of available Servers and a pool of alternative Servers used in case of crash. When a Server crashes, another Server is turned on to preserve availability of the service, and so on. After successive crashes, we get a chain of Servers $s_1 \ldots s_t$ where $s_1 \ldots s_{t-1}$ are crashed and $s_t$ is available. Crashes in this example are software crashes like a memory error, so turning off and on a crashed Server is also a way of repairing it through an action called *softrepair*. Additionally, whenever a crashed Server $s_i$ gets repaired, then the running Server in the chain should turn off $s_t$ and all crashed Server that replaced the repaired Server $(s_j, i+1 < j < t)$ provide a *softrepair*. Figure 5.2 shows a possible scenario for an example with four Servers: (1) the first Server is turned on while all the others are turned off; (2) the first Server crashes and turns on the second Server; (3) the second Server crashes and turns on the third Server; (4) the first Server gets repaired. In this case, the third Server must be turned off and the crashed Server (the second one) has to turn off by executing *softrepair*. Notice that, the decision of turning off a running Server has to be propagated from the repaired Server. Also, Servers have to change state synchronously to keep constant the number of available Servers.

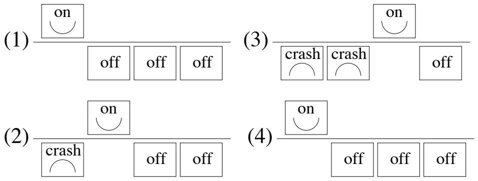


**Fig. 7.** Possible scenario for Fault-Tolerant Client-Server.

The model of a Server component is shown in Figure 8. Moreover, we use the plus symbol "+" to denote logical disjunction of require constraints involving several ports determined by the same expression. For instance, **Require** $x.repair + x.softrepair[x : Server \mid x.crash \in h_2] \equiv$ **Require** $x.repair[x : Server \mid x.crash \in h_2] \vee$ **Require** $x.softrepair[x : Server \mid x.crash \in h_2]$. A turnoff requires a repair or a *softrepair* from any Server that has crashed.

Notice that, statically predefined connector structure (using BIP) leads to more than $N^N$ interactions, where $N$ is the total number of Servers. For this reason, modeling statically this architecture is practically impossible even for a relatively small number of Servers. Figure 9(a) depicts execution time for Dy-BIP. For a total number of 27 Servers with 13 available Servers, executing 1000 interactions requires only 8 seconds. Notice that, using BIP we have to statically define more than $27^{27}$ interactions!

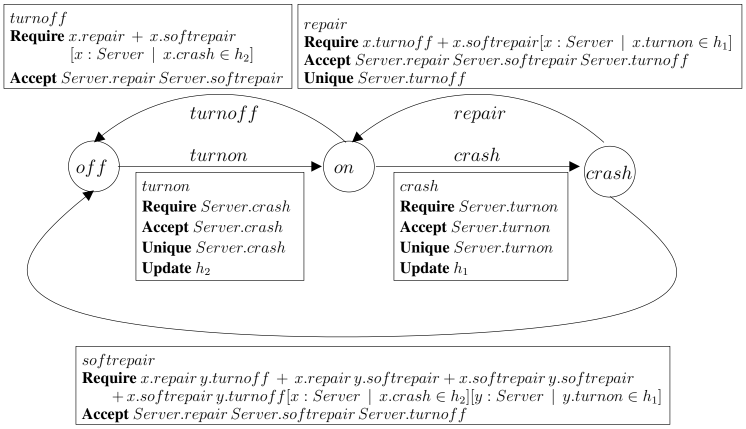
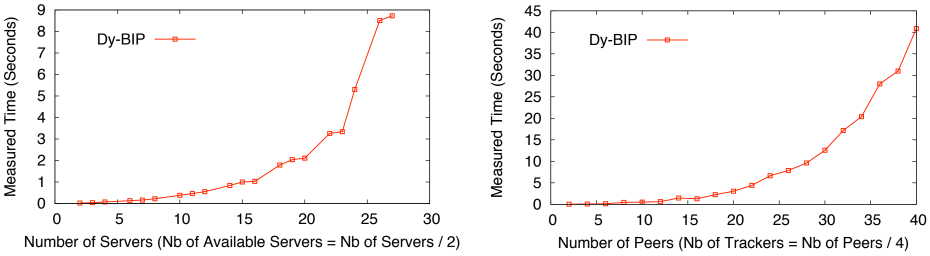

$turnoff$
**Require** $x.repair + x.softrepair$ $[x : Server \mid x.crash \in h_2]$
**Accept** $Server.repair\ Server.softrepair$
$repair$
**Require** $x.turnoff + x.softrepair[x : Server \mid x.turnon \in h_1]$
**Accept** $Server.repair\ Server.softrepair\ Server.turnoff$
**Unique** $Server.turnoff$
$turnoff$
$repair$
$turnon$
$crash$
$off$
$on$
$crash$
$turnon$
**Require** $Server.crash$
**Accept** $Server.crash$
**Unique** $Server.crash$
**Update** $h_2$
$crash$
**Require** $Server.turnon$
**Accept** $Server.turnon$
**Unique** $Server.turnon$
**Update** $h_1$
$softrepair$
**Require** $x.repair\ y.turnoff + x.repair\ y.softrepair + x.softrepair\ y.softrepair$ $+ x.softrepair\ y.turnoff[x : Server \mid x.crash \in h_2][y : Server \mid y.turnon \in h_1]$
**Accept** $Server.repair\ Server.softrepair\ Server.turnoff$

**Fig. 8.** Fault-Tolerant Servers in Dy-BIP.



(a) Fault-Tolerant Servers

(b) Trackers and Peers

**Fig. 9.** Execution times for Fault-Tolerant Servers and Trackers and Peers.

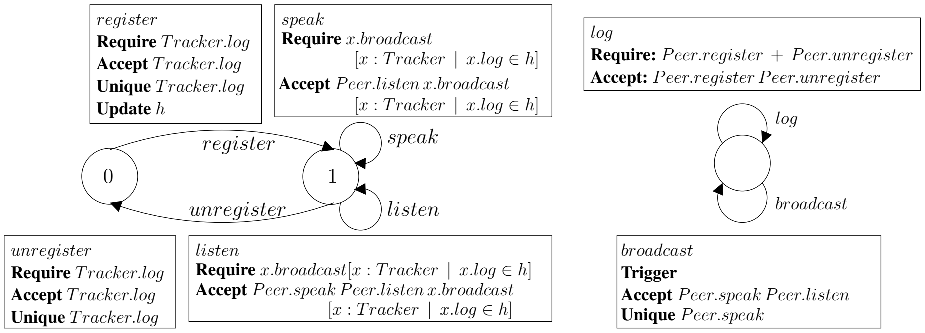
### 5.3 Trackers and Peers

As third benchmark, we consider a simplified wireless audio protocol for reliable multicast communication. The protocol allows an arbitrary numbers of participants (named

*Peers*) to dynamically connect and communicate along an arbitrary number of wireless communication channels (managed by dedicated *Trackers*).

In this protocol, Peers are allowed to use at most one communication channel at a time. The access to channels is subject to an explicit registration mechanism. Dynamically, every Peer selects and registers to the channel it wants to use. Once registered, Peers can either speak, that is, send data over the channel, or listen, that is, receive data sent by others. For every channel, its associated Tracker ensures for any communication that (1) exactly one registered peer (on that channel) is speaking and (2) all other registered peers are listening. That is, data is delivered atomically to all potential receivers. Finally, Peers can dynamically de-register, then register to other channels, etc.

The Dy-BIP model is depicted in Figure 10. As for the previous example, we use the "+" abbreviation for disjunction of require constraints applied to several ports. Notice that, a statically predefined connector structure is exponentially complex and the model explodes rapidly. We need $P \times T \times (3+2^{P-1})$ connectors, where $P$ is the total number of Peers and $T$ is the total number of Trackers.



register
Require Tracker.log
Accept Tracker.log
Unique Tracker.log
Update h
speak
Require x.broadcast [x : Tracker | x.log ∈ h]
Accept Peer.listen x.broadcast [x : Tracker | x.log ∈ h]
register
speak
0
1
unregister
listen
unregister
Require Tracker.log
Accept Tracker.log
Unique Tracker.log
listen
Require x.broadcast[x : Tracker | x.log ∈ h]
Accept Peer.speak Peer.listen x.broadcast [x : Tracker | x.log ∈ h]
log
Require: Peer.register + Peer.unregister
Accept: Peer.register Peer.unregister
log
broadcast
broadcast
Trigger
Accept Peer.speak Peer.listen
Unique Peer.speak

(a) Peer type

(b) Tracker type

**Fig. 10.** Trackers and Peers in Dy-BIP.

Figure 9(b) shows execution times for 1000 interactions for a Dy-BIP model where there are four times more Peers than Trackers.

## 6 Conclusion

The paper provides a simple modeling language for compositional description of dynamic architectures. The language as a straightforward extension of a static architecture modeling language, bridges the gap between static and dynamic description styles. It is simple but expressive enough as illustrated by non-trivial examples. Its associated modeling methodology leads to concise and intelligible models obtained as the composition of components. Global architecture constraints can be synthesized by composing architecture constraints of individual components. Using history variables in components avoids state explosion and duplication of ports.

This work contrasts with existing approaches for dynamic architectures which lack clear semantics and are not compositional. It proposes a rigorous methodology for writing architecture constraints of components. In particular, the distinction between different types of constraints provides guidance for their soundness and completeness. The choice between different implementations permits exploration efficiency trade-offs. On-the-fly computation seems more adequate for architectures with a large number of configurations while regular BIP execution is more advantageous for systems with a small number of configurations.

Further developments will focus on integrating data transfer, priorities, as well as creation and deletion of components.

## References

1. N. Medvidovic and R. N. Taylor, “A framework for classifying and comparing architecture description languages,” in *ESEC / SIGSOFT FSE*, 1997, pp. 60–76.
2. D. Garlan, R. T. Monroe, and D. Wile, “Acme: An architecture description interchange language,” in *Proceedings of CASCON’97*, Toronto, Ontario, November 1997, pp. 169–183.
3. R. Allen, R. Douence, and D. Garlan, “Specifying and analyzing dynamic software architectures,” in *Fundamental Approaches to Software Engineering*, Lisbon, Portugal, March 1998, pp. 21–37.
4. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, “Rigorous component-based system design using the bip framework,” *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.
5. A. Basu, P. Bidinger, M. Bozga, and J. Sifakis, “Distributed semantics and implementation for systems with interaction and priority,” in *FORTE*, 2008, pp. 116–133.
6. S. Bliudze and J. Sifakis, *Causal Semantics for the Algebra of Connectors*. Berlin, Heidelberg: Springer-Verlag, 2008.
7. P. Inverardi and A. L. Wolf, “Formal specification and analysis of software architectures using the chemical abstract machine model,” *IEEE Trans. Software Eng.*, vol. 21, no. 4, pp. 373–386, 1995.
8. D. L. Métayer, “Describing software architecture styles using graph grammars,” *IEEE Trans. Software Eng.*, vol. 24, no. 7, pp. 521–533, 1998.
9. R. Milner, *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.
10. J. Magee and J. Kramer, “Dynamic structure in software architectures,” *SIGSOFT Softw. Eng. Notes*, vol. 21, pp. 3–14, October 1996. [Online]. Available: http://doi.acm.org/10.1145/250707.239104
11. S. Bliudze and J. Sifakis, “The algebra of connectors: structuring interaction in bip,” in *EMSOFT*, 2007, pp. 11–20.
12. J. S. Kim and D. Garlan, “Analyzing architectural styles,” *Journal of Systems and Software*, vol. 83, no. 7, pp. 1216–1235, 2010.
13. M. H. Kacem, M. Jmaiel, A. H. Kacem, and K. Drira, “Evaluation and comparison of adl based approaches for the description of dynamic of software architectures,” in *ICEIS (3)*, 2005, pp. 189–195.
14. I. Georgiadis, J. Magee, and J. Kramer, “Self-organising software architectures for distributed systems,” in *WOSS*, 2002, pp. 33–38.
15. M. Belguidoum and F. Dagnat, “Dependency management in software component deployment,” *Electr. Notes Theor. Comput. Sci.*, vol. 182, pp. 17–32, 2007.