DR-BIP - Programming Dynamic Reconfigurable Systems

Rim El Ballouli, Saddek Bensalem, Marius Bozga, Joseph Sifakis

Verimag Research Report n^o TR-2018-3

February 27, 2018

Reports are downloadable at the following address
http://www-verimag.imag.fr

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UGA
Bâtiment IMAG
Université Grenoble Alpes
700, avenue centrale
38401 Saint Martin d'Hères
France
tel : +33 4 57 42 22 42
fax : +33 4 57 42 22 22
http://www-verimag.imag.fr/

DR-BIP - Programming Dynamic Reconfigurable Systems

Rim El Ballouli, Saddek Bensalem, Marius Bozga, Joseph Sifakis

University Grenoble Alpes

February 27, 2018

Abstract

The paper introduces DR-BIP, a formal framework for programming dynamic reconfigurable systems. DR-BIP relies on architectural motifs to structure the architecture of a system and to coordinate its reconfiguration at runtime. An architectural motif defines a set of interacting components that evolve according to reconfiguration rules. With DR-BIP, the dynamism can be captured as the interplay of dynamic changes in three independent directions 1) the organization of interactions between instances of components in a given configuration; 2) the reconfiguration mechanisms allowing creation/deletion of components and management of their interaction according to a given architectural motif; 3) the migration of components between predefined architectural motifs which characterizes dynamic execution environments. The paper lays down the formal foundation of DR-BIP, illustrates its expressiveness on several examples and discusses avenues for dynamic reconfigurable system design.

Keywords: architectural motifs, components, operational semantics, BIP

Reviewers: Marius Bozga

How to cite this report:

```
@techreport {TR-2018-3,
  title = {DR-BIP - Programming Dynamic Reconfigurable Systems},
  author = {Rim El Ballouli, Saddek Bensalem, Marius Bozga, Joseph Sifakis},
  institution = {{Verimag} Research Report},
  number = {TR-2018-3},
  year = {}
}
```

Contents

1 Introduction

Modern computing systems exhibit dynamic and reconfigurable behavior. They evolve in uncertain environments and have to continuously adapt to changing internal or external conditions. This is essential to efficiently use system resources e.g. reconfiguring the way resources are accessed and released in order to adapt the system behavior in case of mishaps such as faults or threats, and to provide the adequate functionality when the external environment changes dynamically as in mobile systems. In particular, mobile systems are becoming important in many application areas including transport, telecommunications and robotics.

There exist two complementary approaches for the expression of dynamic coordination rules. One respects a strict separation between component behavior and its coordination. Coordination is exogenous in the form of an architecture that describes global coordination rules between the coordinated components. This approach is adopted by numerous Architecture Description Languages (ADL) (see [8] for a survey).

The other approach is based on endogenous coordination by using explicitly primitives in the code describing the behavior of components. Most programming models use internalized coordination mechanisms. Components usually have interfaces that specify their capabilities to coordinate with other components. Composing components boils down to composing interfaces. This approach is usually adopted with formalisms based on π-calculus and process algebra, such as [1, 10, 12, 13].

The obvious advantage of endogenous coordination is that programmers do not have to build explicitly a global coordination model. Consequently, the absence of such a model makes the validation of coordination mechanisms and the study of their underlying properties much harder. Exogenous coordination is advocated for enabling the study of the coordination mechanisms and their properties. It motivated numerous publications and the development of 100+ ADLs [15].

There exists a huge literature on architecture modeling reviewed in detailed surveys classifying the various approaches and outlining new trends and needs [8, 9, 14, 15, 17–19]. Despite the impressive amount of work on this topic there is no clear understanding about how different aspects of architecture dynamism can be characterized.

We consider that the degree of dynamism of a system can be captured as the interplay of dynamic change in three independent directions. The first direction requires the ability to describe parametric system coordination for arbitrary number of instances component types. For example, systems with m Producers and n Consumers or Rings formed from n identical components.

The second direction requires the ability to add/delete components and manage their interaction rules depending on dynamically changing conditions. This is needed for a reconfigurable ring of n components e.g. removing a component which self-detects a failure and adding the removed component after recovery. So adding/deleting components implies the dynamic application of specific interaction rules depending on their type. This is also needed for mobile components which are subject to dynamic interaction rules depending on the state of their neighborhood.

The third direction is currently the most challenging. It meets in particular, the vision of "fluid architectures" or "fluid software" [19] which entails a virtual but personal computing experience allowing the users to seamlessly roam and continue their activities on any available device or computer. Applications and objects live in an environment which is conceptually an architecture motif. They can be dynamically transported from one motif to another.

Supporting dynamic migration of components allows a disciplined and easy-to-implement management of dynamically changing coordination rules. For instance, self-organizing systems may adopt different coordination motifs to adapt their behavior so as to meet a global property.

The paper proposes *Dynamic Reconfigurable* BIP (DR-BIP) component framework, an extension of BIP [2, 3] and Dy-BIP [7] frameworks, which encompasses all these three aspects of dynamism. As such, DR-BIP follows an exogenous approach respecting the strict separation between behavior and architecture. It directly encompasses multiparty interaction [6] and is rooted in formal operational semantics allowing a rigorous implementation. It privileges the imperative and exogenous style characterizing dynamic architecture as a set of interaction rules implemented by connectors and a set of configuration rules.

Although it does not allow ad hoc dynamism, it directly encompasses all kinds of dynamism at run time [8]: programmed dynamism and in addition adaptive dynamism, and self-organizing dynamism. It provides support for component creation and removal at run time. In addition to these operations at motif level, our

formalism directly supports component migration from one motif to another. It supports programmed reconfiguration and triggered reconfiguration in particular [9].

The big advantage from using motifs is that when a component is deleted or created its type defines the interaction with other components. So, a motif is a "world" where components live and from which they can migrate to join other "worlds" [19].

The paper is organized as follows. Section 2 provides a brief overview of the DR-BIP objectives and major design principles. Section 3 briefly recalls the foundational component-based model used. Section 4 introduces the motif concept and its semantics. Section 5 introduces motif-based systems. Section 6 presents several examples. Section 7 presents DR-BIP framework implementation. Section 8 presents conclusions and future work directions.

2 Design Principles

We are seeking for a general framework, covering as much as possible the current practical needs for the design of dynamic systems, and therefore fulfilling specific requirements for rigorous modeling and analysis:

- enforce architectural constraints / styles, that is, allow the definition of architectures as parametric operators on components guaranteeing by construction specific properties,
- describe systems with evolving architecture, that is, the system architecture must be a living concept of the description, that can be updated at runtime using dedicated primitives,
- support separation of concerns, namely keep separate the system behavior (functionality) from the system architecture and so avoid as much as possible blurring the behavior of components with information about their execution context and/or reconfiguration needs,
- provide sound foundation for analysis and implementation, that is, rely on a well-defined operational semantics, leveraging on existing models for rigorous component-based design,

The DR-BIP framework relies on the key concept of *architectural motif* as the elementary unit of description of dynamic architectures. A motif encapsulates (i) behavior, as a set of components, (ii) interaction rules between components and (iii) reconfiguration rules about creating/deleting or moving components. Systems are constructed as a superposition of several motifs, possibly sharing their components, and evolving altogether.

Figure 1 provides an overall view on the structure and evolution of a motif-based system. The initial configuration (left) consists of six interacting components organized using three motifs (indicated with dotted lines). The central motif contains components b_1 and b_2 connected in a ring. The upper motif contains components b_1, c_1, c_2, c_3, with b_1 being connected to all others. The lower motif contains connected components b_2, c_4. The second system configuration (in the middle) shows the evolution following a reconfiguration step. Component c_3 *migrated* from the upper motif to the lower motif, by disconnecting from b_1 and connecting to b_2. The central motif is not impacted by the move. The third system configuration (right) shows one more reconfiguration step. Two new components have been created b_3 and c_5. The central motif now contains one additional component b_3, interconnected along b_1 and b_2 forming a larger ring. In addition a new motif is created containing b_3 and c_5.

The example above contains actually two types of motifs: ring motif and star motif. Types of motifs may be defined separately by giving the types of hosted components and parametric interactions and reconfiguration rules. Then, systems are described by superposing a number of such motifs on a set of components. In this manner, the overall system architecture capture/highlights specific architectural/functional properties by design.

Figure 2 depicts the principle of motif definition in DR-BIP. Motifs are structurally organized as the deployment of component instances on a logical map. Maps are arbitrary graph-like structures consisting of interconnected positions. Deployments relate components to positions on the map. The definition of the motif is completed by two sets of rules, defining respectively interactions and reconfiguration actions. Both sets of rules are interpreted on the current motif configuration. The first defines a set of interactions

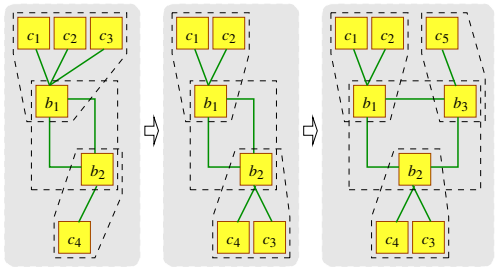


Figure 1: An example : system reconfigurations

between components. The second defines reconfiguration actions to update the content of the motif, that is, the components, the map and/or the deployment.

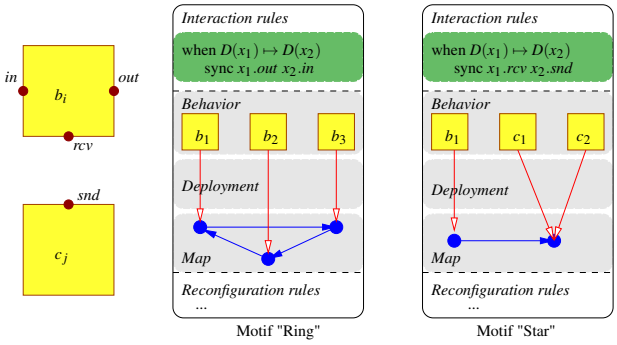


Figure 2: An example : motifs definition

The "Ring" motif illustrated in Figure 2 (left) defines the first type of motif used in the previous example. Three components b_1, b_2, b_3 are deployed into a three-position circular map. Given some deployment function D, the interaction rule reads as follows: for components x_1, x_2 deployed on adjacent nodes $D(x_1) \mapsto D(x_2)$ connect their ports $x_1.out$ and $x_2.in$. These rules *define* three interactions between the b's components namely $b_1.out$ $b_3.in$, $b_3.out$ $b_2.in$, $b_2.out$ $b_1.in$ that correspond to the ring shown in Figure 1 (right). The "Star" motif illustrated in Figure 2 (right) defines the second type. Here, three components are deployed into a two-position map. The rule reads as follows: for components x_1, x_2 deployed on adjacent nodes $D(x_1) \mapsto D(x_2)$ connect their ports $x_1.rcv$ and $x_2.snd$. In that motif configuration, the rule *defines* two interactions, namely $b_1.rcv$ $c_1.snd$ and $b_1.rcv$ $c_2.snd$, also illustrated in Figure 1 (middle, right).

The reasons for choosing maps and deployments as a mean for structuring motifs are their simplicity. On one hand, maps and deployments are common concepts, easy to understand, manipulate and formalize. On the other hand, they adequately support the definition of arbitrarily complex sets of interactions over components by relating them to connectivity properties (neighborhood, reachability, etc). Moreover, maps and deployments are orthogonal to the behavior. Therefore they can be manipulated/updated independently and provide also a very convenient way to express various forms of reconfiguration.

Finally, the operational semantics of motif-based systems is defined in a compositional manner. Every motif defines its own set of interactions based on its local structure. This set of interactions and the involved components remain unchanged as long as the motif does not execute a reconfiguration action. Hence in absence of reconfigurations, the system keeps a fixed static architecture and behaves like an ordinary BIP system. The execution of interaction has no effect on the architecture. In contrast to interactions, system and/or motif reconfigurations rules are used to define explicit changes to the architecture, however, these changes have no impact on components, that is, all running components preserve their state although components may be created/deleted. This independence between execution steps is illustrated in Figure 3.

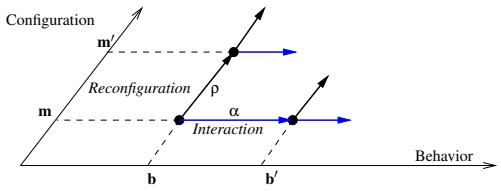


Figure 3: Reconfiguration vs Interaction Steps

3 Component-Based Systems

BIP [2, 3] is the underlying component-based framework for programming dynamic systems (DR-BIP). In BIP, systems are constructed from atomic components, which are finite state automata, extended with data and ports. Communication between components is by multiparty interactions with data transfer. BIP systems are static in the sense that components and interactions are fixed at design time and do not change during system execution.

In this section we briefly recall the key BIP concepts and their operational semantics. Let $\mathcal{V}$ be an universal domain of data values.

3.1 Component types and instances

A component type B^t is an extended labeled transition system (L,P,V,T), where L is a finite set of control locations, P is a finite set of ports, V is a finite set of data variables and $T \subseteq Q \times P \times Q$ is a finite set of port labeled transitions.

For every port $p \in P$, we denote by V_p the subset of variables exported and available for interaction through p. For every transition $\tau \in T$, we denote by g_τ its guard, that is, a Boolean expression defined on V and by f_τ its update function, that is, a sequence of assignments $v_1 := e_1, v_2 := e_2, \ldots$ to variables in V.

A valuation of a set of variables V is a function $\mathbf{v} : V \to \mathbf{V}$. We denote by $\mathbf{V}$ the set of all valuations defined on V.

The semantics of a component type B^t is defined as the labeled transition system $[\![B^t]\!] = (Q,\Sigma,\to)$ where the set of states $Q = L \times \mathbf{V}$, the set of labels $\Sigma = \{p(\mathbf{v}_p) \mid \mathbf{v}_p \in \mathbf{V}_p\}$ and transitions $\to$ are defined by the rule:

$$(\textsc{Comp})\ \frac{\tau = \ell \xrightarrow{p} \ell' \in T \qquad g_\tau(\mathbf{v}) \qquad \mathbf{v}''_p \in \mathbf{V}_p \qquad \mathbf{v}' = f_\tau(\mathbf{v}[\mathbf{v}''_p/V_p])}{B^t \ :\ (\ell,\mathbf{v}) \xrightarrow{p(\mathbf{v}''_p)} (\ell',\mathbf{v}')}$$

That is, $(\ell',\mathbf{v}')$ is a successor of $(\ell,\mathbf{v})$ labeled by $p(\mathbf{v}''_p)$ iff (1) $\tau = \ell \xrightarrow{p} \ell'$ is a transition of T, (2) the guard g_τ holds on the current state valuation $\mathbf{v}$, (3) $\mathbf{v}''_p$ is a valuation of exported variables V_p and (4) $\mathbf{v}' = f_\tau(\mathbf{v}[\mathbf{v}''_p/V_p])$ that is, the next-state valuation $\mathbf{v}'$ is obtained by applying f_τ on $\mathbf{v}$ previously updated according to $\mathbf{v}''_p$. Whenever a p-labeled successor exists in a state, we say that p is *enabled* in that state.

We consider a finite set of component types, fixed a priori. A component instance b is a couple (B^t,k) for some $k \in \mathbb{N}$. We denote respectively by $ports(b)$, $states(b)$, $labels(b)$ the set of ports, states and labels associated to the instance b according to its type.

Example 3.1 *Figure 4 (left) illustrates graphically a component type. The component has three ports (in, out, rcv) attached with variables (respectively u, v, w). It has two control locations (idle, busy) and three transitions labeled by the ports. For example, the transition labeled by in changes the control location from idle to busy while performing the computation* $v := f(u,w)$.

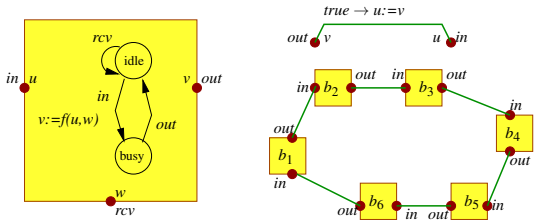


Figure 4: Component types, interactions and systems in BIP

3.2 Systems of components

Systems of components $\Gamma(B)$ are obtained by composing a finite set of component instances $B = \{b_1, ..., b_n\}$ using a finite set of multiparty interactions Γ.

A multiparty *interaction* a is a triple (P_a, G_a, F_a), where $P_a \subseteq \bigcup_{i=1}^{n} ports(b_i)$ is a set of ports, G_a is a guard, and F_a is a data transfer function. By definition, P_a must use at most one port of every component in B, that is, $|P_i \cap P_a| \leq 1$ for all $i \in \{1..n\}$. Therefore, we simply denote $P_a = \{b_i.p_i\}_{i \in I}$, where $I \subseteq \{1..n\}$ contains the indices of the components involved in a and for all $i \in I, p_i \in ports(b_i)$. G_a and F_a are both defined on the variables exported by the ports in P_a (i.e., $\bigcup_{p \in P_a} V_p$).

The semantics of a system $S = \Gamma(B)$ is defined as the labeled transition system $[\![S]\!] = (Q, \Sigma, \rightarrow)$ where the set of states $Q = \langle b \mapsto q \mid b \in B,\ q \in states(b) \rangle$, the set of labels $\Sigma \subseteq \mathcal{P}(ports(B) \times \mathcal{P}(\mathbf{V}))$ contains the ports and sets of values exchanged during interactions and transitions $\rightarrow$ are defined by the rule:

$$\text{(SYS)}\ \frac{\begin{array}{c} a = (\{b_i.p_i\}_{i\in I}, G_a, F_a) \in \Gamma \qquad G_a(\{\mathbf{v}_{p_i}\}_{i\in I}) \qquad \{\mathbf{v}''_{p_i}\}_{i\in I} = F_a(\{\mathbf{v}_{p_i}\}_{i\in I}) \\ \forall i \in I.\ \left(B_i^t\ :\ (\ell_i, \mathbf{v}_i) \xrightarrow{p_i(\mathbf{v}''_{p_i})} (\ell'_i, \mathbf{v}'_i)\right) \qquad \forall i \notin I.\ (\ell_i, \mathbf{v}_i) = (\ell'_i, \mathbf{v}'_i) \end{array}}{\Gamma(B)\ :\ \langle (\ell_1, \mathbf{v}_1)/b_1, \ldots, (\ell_n, \mathbf{v}_n)/b_n \rangle \xrightarrow{\{b_i.p_i(\mathbf{v}''_{p_i})\}_{i\in I}} \langle (\ell'_1, \mathbf{v}'_1)/b_1, \ldots, (\ell'_n, \mathbf{v}'_n)/b_n \rangle}$$

For each $i \in I$, $\mathbf{v}_{p_i}$ above denotes the valuation $\mathbf{v}_i$ restricted to variables of V_{p_i}. The rule expresses that S can execute an interaction $a \in \Gamma$ *enabled* in state $((\ell_1, \mathbf{v}_1), \ldots, (\ell_n, \mathbf{v}_n))$, iff (1) for each $p_i \in P_a$, the corresponding component instance b_i can execute a transition labeled by p_i, and (2) the guard G_a of the interaction holds on the current valuation $\mathbf{v}_{p_i}$ of exported variables on ports in a. Execution of a triggers first the data transfer function F_a which modifies exported variables V_{p_i}. The new values obtained, encoded in the valuation $\mathbf{v}''_{p_i}$, are then used by the components' transitions. The states of components that do not participate in the interaction remain unchanged.

Example 3.2 *Figure 4 (right) illustrates a system obtained by composing six b_i instances with six out in interactions in a ring structure. It shows a binary interaction between two ports out, in, having guard true and data transfer $u := v$. That is, whenever the interaction is executed, the data is transferred from the out port to the in port.*

4 Motifs for Dynamic Architectures

Motifs are dynamic structures of interacting components. Their structure is expressed as a combination of three concepts namely, behavior, map and deployment. The behavior consists of a set of components. The map is an underlying logical structure (backbone) used to organize the interaction of components. The deployment provides the association between the components and the map. The components within a motif run in parallel and synchronize using multiparty interactions. The set of multiparty interactions is defined by interaction rules evaluated on the structure of the motif. Finally, the motif structure is also evolving. Any of the three layers can be modified i.e., components can be added/removed to/from the motif, the map and/or the deployment can change. The motif evolution is expressed using reconfiguration rules, which evaluate and update the motif structure accordingly. The following section introduce formally all the motif-related concepts.

4.1 Maps and deployments

Maps and deployments are abstract concepts used to organize the motifs. Maps denote arbitrary dynamic collections of inter-connected nodes (positions). They are defined as particular instances of generic map types H^t characterized by

- an underlying domain $N(H^t)$ of nodes
- a set of primitives $\Omega(H^t)$ to update/access the map content
- a logic $\mathcal{L}(H^t)$ to express constraints on the map content

We use maps as dynamic data structures (objects). For any primitive $op \in \Omega(H^t)$ we will use the dotted notation $H.op(\cdots)$ to denote the update and/or access to the map H according to op. Moreover, for any $\psi \in \mathcal{L}(H^t)$ we will use $H \models \psi$ to denote that the constraint ψ is satisfied on H.

Example 4.1 *Maps can be graphs (V,E) where vertices $V \subset \mathbb{N}$ denote the position (here assumed to be natural numbers) and edges $E \subseteq V \times V$ expressing the connectivity between these positions. Such maps can be manipulated explicitly using primitives such as* addVertex, remVertex, addEdge, remEdge. *Constraints might include predicates such as edge constraints $\cdot \rightarrow \cdot$, path constraints $\cdot \rightarrow^* \cdot$), etc, with an usual meaning.*

Example 4.2 *For the "Ring" example, the map is restricted to a cyclic graph. In this case, we consider specific primitives* initialize, add, remove *to respectively initialize, extend by one new position and remove one position from the map. Also, we consider the predicate $\cdot \mapsto \cdot$ to denote the edge relations.*

Deployments are mappings of a set B of components instances to the nodes of a map H, formally $D : B \rightarrow dom(H)^{\perp}$.

As for maps, we use deployments as dynamic data structures (objects). We consider a set of primitives $\Omega(D^t)$ to update and/or access the deployment content as well as a logic $\mathcal{L}(D^t)$ to express constraints on the deployment.

Practically, both maps and deployments are implemented as dynamic collections of objects, with specific interfaces, in a similar way to standard collections libraries available for regular programming languages (Java, C++, Python, etc).

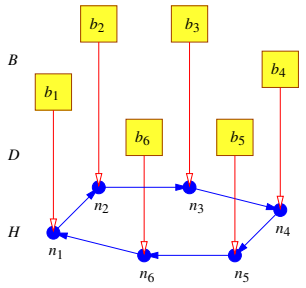
4.2 Motif types

Definition 4.1 (motif type) *A motif type M^t is a tuple $((\mathcal{B},\mathcal{H},\mathcal{D}), I\mathcal{R}, \mathcal{RR})$ where:*

- *the triple $(\mathcal{B},\mathcal{H},\mathcal{D})$ are motif meta-variables used to maintain respectively the set of component instances, the support map and the deployment of component instances to the map,*
- *$I\mathcal{R}$ is a set of motif interaction rules of the form $(\mathcal{Z}, \Psi, P_I, G_I, F_I)$ where $\mathcal{Z}$ is a set of rule parameters (context), Ψ is a rule constraint, and (P_I,G_I,F_I) is the interaction specification, namely the set of ports, the guard and the data transfer.*
- *$\mathcal{RR}$ is a set of motif reconfiguration rules of the form $(\mathcal{Z}, \Psi, G_R, \mathcal{Z}_L, A_R)$ where as before $\mathcal{Z}$ is a set of rule parameters (context), Ψ is a rule constraint, G_R is a reconfiguration guard, $\mathcal{Z}_L$ are local rule parameters, and A_R is a (sequence of) reconfiguration action(s).*

The motif configuration/content is defined by a consistent valuation of meta-variables $\mathcal{B}$, $\mathcal{H}$, $\mathcal{D}$ respectively as B, a set of components instances, H a map, and $D : B \rightarrow dom(H)^{\perp}$ a deployment.

Example 4.3 *Figure 5 illustrates the definition of the motif type for the "Ring" example. The motif configuration is defined by the set of six component instances $B = \{b_i\}_{i=1,6}$, the map H defined as the cyclic graph of six nodes $\{n_i\}_{i=1,6}$, and the deployment $D = \{b_i \mapsto n_i\}_{i=1,6}$. The motif type contains one interaction rule denoted as* sync-inout *and three reconfiguration rules denoted respectively* do-init, do-insert *and* do-remove. *The meaning of the rules is explained in the next subsections.*



sync-inout(x_1: C, x_2 : C) ≡
  when D(x_1) ↦ D(x_2)
  sync x_1.out x_2.in / true → x_2.u := x_1.v

do-init() ≡ when B = ∅
  do H.empty(), x_1 := B.create(C, busy),
    n_1 := H.add(), D(x_1) := n_1
    x_2 := B.create(C, idle),
    n_2 := H.add(), D(x_2) := n_2
do-insert() ≡
  do x := B.create(C, idle),
    n := H.add(), D(x) := n
do-remove(x : C) ≡ when |B| ≥ 3 ∧ x.idle
  do n := D(x), B.delete(x), H.remove(n)

Figure 5: The "Ring" motif type

The motif behavior is defined by interaction and reconfiguration rules. Rule parameters $\mathcal{Z}$ include symbols denoting (sets of) component instances (resp. map nodes) and interpreted as (subsets) elements of B (resp. $dom(H)$). Rule constraints Ψ are boolean combinations of equality, map, deployment constraints built using parameters in $\mathcal{Z}$ and meta-variables $\mathcal{B}$, $\mathcal{H}$, $\mathcal{D}$:

$$\Psi ::= \psi^0 \mid \psi^{\mathcal{H}} \mid \psi^{\mathcal{D}} \mid \Psi_1 \wedge \Psi_2 \mid \neg\Psi$$

In the above, Ψ^0 denotes any constraint using equality and set operators, $\Psi^{\mathcal{H}}$ denotes a logical constraint on the map (conforming to the map logic $\mathcal{L}(H^t)$)) and $\Psi^{\mathcal{H}}$ denotes a logical constraint on the deployment (conforming to the deployment logic $\mathcal{L}(D^t)$)).

For fixed motif content in terms of B,H,D, for given interpretation ζ of parameters, the constraint satisfaction $B,H,D,\zeta \models \Psi$ is defined recursively on the structure of Ψ as follows:

$$\begin{array}{rll}
B,H,D,\zeta \models \psi^0 & \text{iff} & \zeta \cup [B/\mathcal{B},H/\mathcal{H},D/\mathcal{D}] \models \psi^0 \\
B,H,D,\zeta \models \psi^{\mathcal{H}} & \text{iff} & H,\zeta \cup [B/\mathcal{B},D/\mathcal{D}] \models \psi^{\mathcal{H}} \\
 & & \text{conform to } \mathcal{L}(H^t) \\
B,H,D,\zeta \models \psi^{\mathcal{D}} & \text{iff} & D,\zeta \cup [B/\mathcal{B},H/\mathcal{H}] \models \psi^{\mathcal{D}} \\
 & & \text{conform to } \mathcal{L}(D^t) \\
B,H,D,\zeta \models \Psi_1 \wedge \Psi_2 & \text{iff} & B,H,D,\zeta \models \Psi_1 \text{ and } B,H,D,\zeta \models \Psi_2 \\
B,H,D,\zeta \models \neg\Psi & \text{iff} & B,H,D,\zeta \not\models \Psi
\end{array}$$

That means, equality constraints are evaluated in the usual way on the context ζ extended with the current valuation for meta-variables $\mathcal{B}$, $\mathcal{H}$, $\mathcal{D}$. Map constraints are evaluated as defined by their underlying logic $\mathcal{L}(H^t)$on the map H and the context ζ extended with the valuation for meta-variables $\mathcal{B}$, $\mathcal{D}$. The evaluation of deployment constraints is similar.

4.3 Interactions rules

Interaction rules are used to define multiparty interactions on the components instances within the motif. The syntax of the interaction specification part is presented below:

$$\begin{array}{rrcl}
\text{ports:} & P_I & ::= & x.p \mid X.p \mid P_I\, P_I \\
\text{guard:} & G_I & ::= & \mathbf{t} \mid e_I \mid G_I \wedge G_I \mid \neg G_I \\
\text{action:} & F_I & ::= & \varepsilon \mid x.v := e_I \mid X.v := e_I \mid a_I, a_I \\
\text{expression:} & e_I & ::= & x.v \mid X.v \mid op(e_I, \cdots, e_I)
\end{array}$$

The symbols x, X are rule parameters denoting respectively component instances or sets of component instances. Moreover, p is a component port, v is a component (exported) data variable and op is an operation on data values. A rule is syntactically well-formed iff all parameter names used in expressions (part of the

guard or data transfer) are also used as part of the interacting port specification. That is, the guard and data transfer involve only component data participating in the interaction.

For given B, H and D in a motif, the set of multiparty interactions $\Gamma(r)$ corresponding to an interaction rule $r = (\mathcal{Z}, \Psi, P_I, G_I, F_I)$ is defined as:

$$\Gamma(r) = \left\{ (P_a, G_a, F_a) \;\middle|\; \begin{array}{l} B, H, D, \zeta \models \Psi \\ P_a = P_I(\zeta),\ G_a = G_I(\zeta),\ F_a = F_I(\zeta) \\ (P_a, G_a, F_a) \textit{ well formed} \end{array} \right\}$$

The triple P_a, G_a, F_a is considered well formed iff it conforms to the definition of multiparty interactions, namely if P_a does not contain replicated or multiple ports of the same components, as well as if G_a and F_a use and update only variables exported on ports in P_a.

Example 4.4 *The ring motif illustrated in Figure 5 has a unique interaction rule denoted sync-inout. The rule connects the out port of a component x_1 to the in port of the component x_2 deployed next to it on the map. The resulting interactions are depicted in the right part of Figure 4.*

4.4 Reconfiguration rules

Reconfiguration rules are used to define actions impacting the content/organization of the motif. These actions essentially include creating/deleting component instances, updating the map structure and/or the deployment of component instances to the map. They are expressed as specific updates on the corresponding $\mathcal{B}$, $\mathcal{H}$, $\mathcal{D}$ meta-variables. For enhanced expressiveness, reconfiguration rules might use additional local parameters (that is, the local context $\mathcal{Z}_L$) with arbitrary types (data, component instances, map nodes, etc). The local context is updated using standard assignments. The syntax of reconfiguration guards and actions is presented below:

$$\begin{array}{lrcl} \text{guard:} & G_R & ::= & G_I \\ \text{action:} & A_R & ::= & \varepsilon \mid x := \mathcal{B}.create(B^t, q) \mid \mathcal{B}.delete(x) \mid \\ & & & \mathcal{H}.op_1(...) \mid \mathcal{D}.op_2(...) \mid z := e \mid A_R, A_R \end{array}$$

The symbol x denotes a rule parameter interpreted as component instance, z is an arbitrary local rule parameter and e is an arbitrary expression built on parameters and available operators. The intuitive meaning of reconfiguration actions is as follows. The action ε denotes an empty action with no effect. The action $x := \mathcal{B}.create(B^t, q)$ denotes the creation of a new component instance of type B^t. The newly created instance is x and is added to the set of components instances B. The parameter q denotes the initial state for the instance. The action $\mathcal{B}.delete(x)$ denotes the deletion of the component x from the motif, that is, the removal of the component instance x from the set B. The action $\mathcal{H}.op_1(...)$ denotes an update of the map according to an operator op_1 from $\Omega(H^t)$ and specific parameters. Similarly, the action $\mathcal{D}.op_2(...)$ denotes denotes an update of the deployment according to an operator op_2 from $\Omega(D^t)$. Finally, the action $z := e$ denotes an update of a rule parameter according to the expression e.

Formally, the semantics $[\![A_R]\!]$ of a reconfiguration action A_R is defined as a function[1] updating the motif

[1] up to the choice of fresh component instance

content (B, H, D), the set of component configurations ($\mathbf{b}$) and the parameter context (ζ):

$$
\begin{array}{l}
[\![\varepsilon]\!](B,H,D,\mathbf{b},\zeta) = (B,H,D,\mathbf{b},\zeta) \\
[\![x := \mathcal{B}.create(B^t,q)]\!](B,H,D,\mathbf{b},\zeta) = \\
\qquad\qquad = (B \cup \{b\},H,D[\bot/b],\mathbf{b}[q/b],\zeta[b/x]) \\
\qquad \text{where } b = (B^t,k) \textit{ fresh} \\
[\![\mathcal{B}.delete(x)]\!](B,H,D,\mathbf{b},\zeta) = (B \setminus \{b\},H,D_{|B\setminus\{b\}},\mathbf{b},\zeta) \\
\qquad \text{where } b = \zeta(x) \in B \\
[\![\mathcal{H}.op_1(...)]\!](B,H,D,\mathbf{b},\zeta) = (U,H',D_{|H'},\mathbf{b},\zeta) \\
\qquad \text{where } H' = H.op_1(...) \\
[\![\mathcal{D}.op_2(...)]\!](B,H,D,\mathbf{b},\zeta) = (B,H,D',\mathbf{b},\zeta) \\
\qquad \text{where } D' = D.op_2(...) \\
[\![z := e]\!](B,H,D,\mathbf{b},\zeta) = \\
\qquad\qquad = (B,H,D,\mathbf{b},\zeta[z \mapsto e(\zeta \cup (B/\mathcal{B},H/\mathcal{H},D/\mathcal{D}))]) \\
[\![A_{R1},A_{R2}]\!](B,H,D,\mathbf{b},\zeta) = ([\![A_{R2}]\!] \circ [\![A_{R1}]\!])(B,H,D,\mathbf{b},\zeta)
\end{array}
$$

These rules formalize the intuitive description of the reconfiguration actions. Their interpretation is straightforward, except maybe few subtle inter-dependencies of updates for component deletion and map update. First, the deletion of a component instance removes it from the component set B and therefore restricts the deployment domain. Nevertheless, the component configuration is not removed from the set of configurations $\mathbf{b}$ because components might be shared among several motifs. Second, a map update might have an impact on the deployment, therefore, the deployment co-domain must be restricted accordingly (e.g., the deployment of component instances mapped to positions removed from the map will be undefined).

Example 4.5 *The ring motif illustrated in Figure 5 contains three reconfiguration rules. The rule do-init initializes the motif with a ring of two components. The rule do-create creates a new component in the ring. The rule do-remove(x) removes an idle component x from the ring, provided it contains more than 3 components.*

4.5 Operational semantics

A motif evolves by performing two categories of steps, namely interactions and reconfigurations. Interactions are defined from interaction rules and are executed by motif components. Reconfiguration are defined by reconfiguration rules.

Formally, the semantics of a motif type $M^t = ((\mathcal{B},\mathcal{H},\mathcal{D}), I\mathcal{R}, \mathcal{RR})$ is defined as the labeled transition system $[\![M^t]\!] = (Q,\Sigma,\rightarrow)$ where

- the set of states Q correspond to motif configurations B, H, D consistently extended with configurations for all component instances $\mathbf{b} = \langle b \mapsto q \mid b \in B, q \in states(b)\rangle$,
- the set of labels Σ correspond to valid interactions α constructed on components and reconfiguration actions ρ,
- the set of transitions $\rightarrow = \underset{I}{\rightarrow} \cup \underset{R}{\rightarrow}$ correspond to execution of respectively multiparty interactions as defined by interaction rules ($\underset{I}{\rightarrow}$) and reconfiguration actions, as defined by reconfiguration rules ($\underset{R}{\rightarrow}$), formally

$$
\text{(MOT-I)}\ \frac{\Gamma = \cup_{r\in I\mathcal{R}}\Gamma(r) \qquad \Gamma(B)\ :\ \mathbf{b} \xrightarrow{\alpha} \mathbf{b}'}{M^t\ :\ (B,H,D,\mathbf{b}) \underset{I}{\xrightarrow{\alpha}} (B,H,D,\mathbf{b}')}
\qquad
\text{(MOT-R)}\ \frac{\begin{array}{c}(\mathcal{Z},\Psi,G_R,\mathcal{Z}_L,A_R) \in \mathcal{RR} \\ B,H,D,\zeta \models \Psi \qquad G_R(\zeta)(\mathbf{b}) = true \\ [\![A_R]\!](B,H,D,\mathbf{b},\zeta) = (B',H',D',\mathbf{b}',\zeta')\end{array}}{M^t\ :\ (B,H,D,\mathbf{b}) \underset{R}{\xrightarrow{\rho}} (B',H',D',\mathbf{b}')}
$$

The rule (MOT-I) says that the motif executes a multiparty interaction α and change the configurations of components instances from $\mathbf{b}$ to $\mathbf{b}'$ iff (1) α belongs to the set of valid interactions Γ defined from the interaction rules and (2) a valid step labeled by α is indeed allowed between $\mathbf{b}$ and $\mathbf{b}'$ according to the

component-based semantics. The rule (MOT-R) says that the motif executes a reconfiguration if (1) some reconfiguration rule is enabled at the current motif configuration, when both its constraint Ψ and guard G_R are satisfied and (2) the current and next motif configuration are related according to the semantics of the action A_R.

The distinction between interaction and reconfiguration steps ensures separation of concerns for execution within a motif. On one hand, execution of interactions does not have any impact on the structure of the motif, but only on component states. That is, the motif structure is fully preserved. On the other hand, while the execution of reconfiguration steps has an impact on the motif structure, it does not change component states. That is, reconfiguration may change the set of components instances, the map and the deployment, while it leaves unchanged components states. Yet, new instances can be created and added to the set.

5 Motif-based Systems

We consider systems defined as collections of motifs sharing a set of components. In such systems, every motif can evolve independently of the others, depending on its internal structure and associated rules. In addition, several motifs can also synchronize altogether and perform a joint reconfiguration over the system.

Two ways of coordination between motifs are therefore possible: implicit coordination, by means of shared components and explicit coordination, by means of inter-motif reconfiguration rules.

This section introduces formally inter-motif reconfiguration and defines the operational semantics of motif-based systems. We consider a finite set of motif types. A motif instance m is a couple (M^t, k) for some $k \in \mathbb{N}$.

5.1 Inter-motif reconfiguration rules

The rules for inter-motif reconfiguration allow joint reconfiguration of several motif instances. In addition to the application of local reconfiguration actions, these rules allow two additional types of actions, respectively creation and deletion of motif instances, and exchanging component instances between motifs.

Inter-motif reconfiguration rules are defined as tuples $(\mathcal{Z}^\star, \Psi^\star, G^\star, \mathcal{Z}^\star_L, A^\star_R)$ similar to local reconfiguration rules. The set of rule parameter $\mathcal{Z}^\star$ might include additional symbols denoting motif instances (y). The constraints Ψ^* are defined by the grammar:

$$\Psi^* ::= \Psi^{0*} \mid \langle y : \Psi \rangle \mid \Psi^*_1 \wedge \Psi^*_2 \mid \neg\Psi^*$$

In the above, Ψ^{0*} denotes some basic equality constraint expressed on context parameters, $\langle y : \Psi \rangle$ denotes a local constraint Ψ to be checked in the context of the motif instance y.

These constraints are evaluated on motif configurations extended with context parameters. Motif configurations are tuples $(M, \mathbf{m})$ where M is a set of motif instances and $\mathbf{m} = \langle m \mapsto (B, H, D) \mid m \in M \rangle$ provides the structure of these instances in terms of behavior, map and deployment. The constraints are evaluated as follows:

$$\begin{array}{rll} M, \mathbf{m}, \zeta \models \Psi^{0*} & \text{iff} & \zeta_{\mathbf{m}} \models \Psi^{0*} \\ M, \mathbf{m}, \zeta \models \langle y : \Psi \rangle & \text{iff} & B, H, D, \zeta_{\mathbf{m}} \models \Psi \\ & & \text{where } m \mapsto (B, H, D) \in \mathbf{m} \text{ and } \zeta(y) = m \\ M, \mathbf{m}, \zeta \models \Psi^*_1 \wedge \Psi^*_2 & \text{iff} & M, \mathbf{m}, \zeta \models \Psi^*_1 \text{ and } M, \mathbf{m}, \zeta \models \Psi^*_2 \\ M, \mathbf{m}, \zeta \models \neg\Psi^* & \text{iff} & M, \mathbf{m}, \zeta \not\models \Psi^* \end{array}$$

In the above, $\zeta_{\mathbf{m}}$ denotes an extended context, including valuations for all meta-variables $\mathcal{B}$, $\mathcal{H}$, $\mathcal{D}$ accessed using parameters y of ζ:

$$\zeta_{\mathbf{m}} = \zeta \cup \langle y.\mathcal{B} \mapsto B, y.\mathcal{H} \mapsto H, y.\mathcal{D} \mapsto D \mid \zeta(y) = m,\ m \mapsto (B, H, D) \in \mathbf{m} \rangle$$

Inter-motif reconfiguration guards and actions are defined by:

$$\begin{array}{rrcl} \text{guard:} & G^\star_R & ::= & G_I \\ \text{action:} & A^\star_R & ::= & \varepsilon \mid y := \mathcal{M}.create(M^t,(e_B,e_H,e_D)) \mid \\ & & & \mathcal{M}.delete(y) \mid y.\mathcal{B}.migrate(x) \mid \\ & & & \langle y : A_R \rangle \mid z := e \mid A^\star_R, A^\star_R \end{array}$$

That is, guards are the same as for interaction rules. The action $y := \mathcal{M}.create(M^t,(e_B,e_H,e_D))$ denotes the creation of a new motif instance y of type M^t, with initial structure defined by the valuation of e_B, e_H, e_D. The action $\mathcal{M}.delete(y)$ denotes the deletion of the motif instance y, that is, its removal from the set of motif instances. The action $y.\mathcal{B}.migrate(x)$ denotes the insertion of an existing component instance x within the set of component instances of the motif y. Finally, the action $\langle y : A_R \rangle$ denotes any local reconfiguration action to be executed in the context of the motif instance y.

Formally, the semantics $[\![A^*_R]\!]$ of inter-motif reconfiguration actions is defined as a function updating motif configurations $(M,\mathbf{m})$, component configurations $(B,\mathbf{b})$ and context parameters (ζ), as follows:

$$\begin{array}{l} [\![y := \mathcal{M}.create(M^t,(e_B,e_H,e_D))]\!](M,\mathbf{m},B,\mathbf{b},\zeta) = \\ \qquad = (M \cup \{m\}, \mathbf{m}', B, \mathbf{b}, \zeta[m/y]) \\ \qquad \text{where } m = (M^t,k) \text{ fresh}, \mathbf{m}' = \mathbf{m} \cup \langle m \mapsto (e_B,e_H,e_D)(\zeta_{\mathbf{m}}) \rangle \\ [\![\mathcal{M}.delete(y)]\!](M,\mathbf{m},B,\mathbf{b},\zeta) = (M \setminus \{m\}, \mathbf{m}_{|M\setminus\{m\}}, B, \mathbf{b}, \zeta) \\ \qquad \text{where } m = \zeta(y) \in M \\ [\![y.\mathcal{B}.migrate(x)]\!](M,\mathbf{m},B,\mathbf{b},\zeta) = (M,\mathbf{m}',B,\mathbf{b},\zeta) \\ \qquad \text{where } m = \zeta(y) \in M,\ m \mapsto (B_1,H,D) \in \mathbf{m}, \zeta(x) \mapsto b \in B \\ \qquad \mathbf{m}' = \mathbf{m}[m \mapsto (B_1 \cup \{b\}, H, D[b \mapsto \bot])] \\ [\![\langle y : A_R \rangle]\!](M,\mathbf{m},B,\mathbf{b},\zeta) = (M,\mathbf{m}',B',\mathbf{b}',\zeta') \\ \qquad \text{where } m = \zeta(y) \in M,\ m \mapsto (B_1,H,D) \in \mathbf{m}, \\ \qquad [\![A_R]\!](B_1,H,D,\mathbf{b},\zeta) = (B'_1,H',D',\mathbf{b}',\zeta') \\ \qquad \mathbf{m}' = \mathbf{m}[m \mapsto (B'_1,H',D')], B' = B \cup B'_1 \\ [\![z := e]\!](M,\mathbf{m},B,\mathbf{b},\zeta) = (M,\mathbf{m},B,\mathbf{b},\zeta[z \mapsto \zeta_{\mathbf{m}}(e)]) \\ [\![A^*_{R1}, A^*_{R2}]\!](M,\mathbf{m},B,\mathbf{b},\zeta) = ([\![A^*_{R2}]\!] \circ [\![A^*_{R1}]\!])(M,\mathbf{m},B,\mathbf{b},\zeta) \end{array}$$

These rules formalize the intuitive description provided earlier. We shall notice few inter-dependencies on various updates. In the case of motif deletion, the motif instance and its configuration are removed from respectively M and $\mathbf{m}$, however, this action has no effect on the set of component instances and their configurations $B,\mathbf{b}$. In the case of component migration, component configurations $\mathbf{b}$ remain also unchanged, that is, the component configuration is not replicated but gets shared among different motif instances.

Example 5.1 *Consider the inter-motif reconfiguration rule:*

do-merge(y_1, y_2 : Ring) ≡
  when y_1.B ∩ y_2.B = ∅ and $|y_1.B| + y_2.B| \leq 100$
  do B = y_1.B ∪ y_2.B, D = y_1.D ∪ y_2.D, H = merge-cycle(y_1.H, y_2.H),
    M.create(Ring, (B, H, D)), M.delete(y_1), M.delete(y_2)

The rule allows merging two Ring motif instances y_1, y_2 into a single one, whenever their sets of component instances are disjoint and altogether their number does not exceed 100. The new motif is created by taking the union of component instances, the union of deployments and the merging of the two underlying cyclic maps. The original motifs y_1 and y_2 are deleted.

5.2 Operational semantics

A motif-based system $\mathcal{S}$ is defined as a tuple $((B^t_i)_i, (M^t_j)_j, \mathcal{RR}^*))$ consisting of a set of component types $(B^t_i)_i$, a set of motif types $(M^t_j)_j$ and a set of inter-motif reconfiguration rules $\mathcal{RR}^*$.

A motif-based system evolves either by executing interactions and/or reconfiguration within any of the motifs, or by executing some inter-motif reconfiguration. Formally, the semantics of motif-based systems $\mathcal{S}$ is defined as the labeled transition system $[\![\mathcal{S}]\!] = (Q,\Sigma,\rightarrow)$ where:

- the set Q of system configuration contains tuples $(M,\mathbf{m},B,\mathbf{b})$ where $M=\{m_1,m_2,...\}$ is a set of motif instances, $\mathbf{m}=\langle m_j \mapsto (B_j,H_j,D_j) \mid m_j \in M,\ B_j \subseteq B\rangle$ are the motif configurations, B is the set of components instances, and $\mathbf{b}=\langle b \mapsto q \mid b \in B, q \in \mathit{states}(b)\rangle$ are the component configurations,

- the set of labels Σ correspond to valid interactions α on component instances, local reconfiguration actions ρ and inter-motif reconfiguration actions ρ^*,

- the set of transitions $\rightarrow = \underset{I}{\rightarrow} \cup \underset{R}{\rightarrow} \cup \underset{R^*}{\longrightarrow}$ correspond to execution of respectively multiparty interactions as defined by interaction rules ($\underset{I}{\rightarrow}$), local reconfiguration as defined by local reconfiguration rules ($\underset{R}{\rightarrow}$) and global reconfiguration actions ($\underset{R^*}{\longrightarrow}$), formally

$$\text{(MBS-I)}\ \frac{m_j \mapsto (B_j,H_j,D_j) \in \mathbf{m} \qquad M_j^t\ :\ (B_j,H_j,D_j,\mathbf{b}_j) \xrightarrow[I]{\alpha} (B_j,H_j,D_j,\mathbf{b}'_j) \qquad \mathbf{b}'=\mathbf{b}[\mathbf{b}'_j/B_j]}{\mathcal{S}\ :\ (M,\mathbf{m},B,\mathbf{b}) \xrightarrow[I]{\alpha} (M,\mathbf{m},B,\mathbf{b}')}$$

$$\text{(MBS-R1)}\ \frac{m_j \mapsto (B_j,H_j,D_j) \in \mathbf{m} \qquad M_j^t : (B_j,H_j,D_j,\mathbf{b}_j) \xrightarrow[R]{\rho} (B'_j,H'_j,D'_j,\mathbf{b}'_j) \qquad \mathbf{m}'=\mathbf{m}[(B'_j,H'_j,D'_j)/m_j] \qquad B'=B\cup B'_j \qquad \mathbf{b}'=\mathbf{b}[\mathbf{b}'_j/B'_j]}{\mathcal{S}\ :\ (M,\mathbf{m},B,\mathbf{b}) \xrightarrow[R]{\rho} (M,\mathbf{m}',B',\mathbf{b}')}$$

$$\text{(MBS-R2)}\ \frac{(Z^*,\Psi^*,G^*,Z_L^*,A_R^*) \in \mathcal{RR}^* \qquad M,\mathbf{m},\zeta \models \Psi^* \qquad G^*(\zeta)(\mathbf{b})=\mathit{true} \qquad [\![A_R^*]\!](M,\mathbf{m},B,\mathbf{b},\zeta)=(M',\mathbf{m}',B',\mathbf{b}',\zeta')}{\mathcal{S}\ :\ (M,\mathbf{m},B,\mathbf{b}) \xrightarrow[R^*]{\rho^*} (M',\mathbf{m}',B',\mathbf{b}')}$$

Rules (MBS-I) and (MBS-R1) lift the transitions (steps) allowed within the motifs at the level of the system, respectively for interactions and reconfigurations. The rule (MBS-R2) handles the execution of inter-motif reconfiguration. These actions are allowed if (1) some inter-motif reconfiguration rule is enabled and (2) the current and next system configurations are related by the semantics of A_R^*.

6 Examples

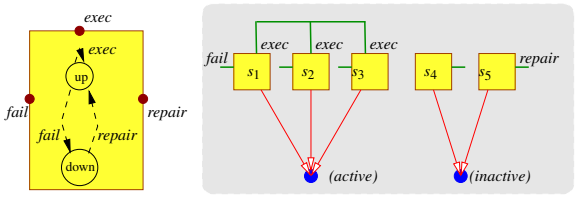
The following examples illustrate the expressiveness of DR-BIP.

6.1 Fault-Tolerant Servers

This example illustrates a model of a fault-tolerant server inspired by [1]. A set of active servers strongly synchronize to deliver some functionality. Any active server may crash and consequently quit the active set. Any crashed server may be repaired and join back the active set.

For this example, the map contains two positions for respectively, active and inactive servers. The connectivity is not relevant. The deployment assigns servers to one of the two positions.

Interactions are defined by three rules. The rule *sync-active* synchronizes all the active servers deployed on the active position. The rule *sync-fail* models failures and the rule *sync-join* models repairs interactions, for every server. Reconfiguration rules are used to move the servers between the active and inactive position of the map, according to their configuration. Rule *do-leave* changes the deployment of a failed server from from active to inactive. Conversely, rule *do-join* changes the deployment of a repaired server from inactive to active.



sync-active(X : S) ≡ when X = D^{-1}(active) sync X.exec
sync-fail(x : S) ≡ when D(x) = active sync x.fail
sync-join(x : S) ≡ when D(x) = inactive sync x.join

do-init() ≡ do x := B.create(S); D(x) := active; ...
do-fail($x : S$) ≡ when x.down do D(x) := inactive
do-join($x : S$) ≡ when x.up do D(x) := active

Figure 6: Fault-tolerant server motif

6.2 Task Migration on Multicore Architectures

Consider task management for a multicore platform. Usually, tasks running on a multicore shall be evenly distributed amongst the cores so as to optimize the performance of the overall system. Dynamic task migration can be therefore seen as the result of a load balancing algorithm that aims at continuously improving the distribution of tasks amongst the cores.

To model this example we introduce two types of atomic components, namely Task (T) and Core (C). Multiple cores are interconnected together in a Processor (P) motif type. The interconnecting topology reflects the platform architecture e.g., a 2×2 grid in our example obtained using a similar grid-like map. Within this motif, any two adjacent cores may interact and exchange specific data according to the interaction rule:

sync-data-exchange($x_1 : C$, $x_2 : C$) ≡
when D(x_1) ↦ D(x_2) ∧ $x_1 \neq x_2$ sync x_1.exchange x_2.exchange

Moreover, we use an additional CoreTask (CT) motif type to represent each core with its assigned tasks. The supporting map of CT is composed of two positions, one deploying the core and the other deploying the tasks assigned to it. Within this motif we model the execution of a task by the interaction rule:

sync-execute($x_1 : C$, $x_2 : T$) ≡ sync x_1.work x_2.execute

The migration of a task from one core to another connected core is realized using an inter-motif reconfiguration rule which involves 3 distinct motifs. A task t migrates from motif y_1 (of type CT) to motif y_2 (of type CT) if the core x_1 of y_1 is connected to the core x_2 of y_2 (according to the processor motif P) and if the number of tasks in y_1 exceeds the number of tasks in y_2 by some constant K:

do-migrate($y_1 : CT$, $y_2 : CT$, $y_3 : P$, $x_1 : C$, $x_2 : C$, $t : T$) ≡
when ⟨ $y_1 : x_1 \in$ B ⟩ ∧ ⟨ $y_2 : x_2 \in$ B ⟩ ∧ ⟨ y_3 : D(x_1) ↦ D(x_2) ⟩ ∧
$|y_1.\mathsf{B}| > |y_2.\mathsf{B}| + \mathsf{K} \wedge t \in y_1.\mathsf{B}$
do y_2.migrate(t), y_1.delete(t)

Figure 7 illustrates the effect of migration rule applied for several cores and tasks - task t_2, t_4 migrated from core c_1 to respectively core c_4, c_2, task t_6 migrated from core c_3 to core c_4.

6.3 Automated Highway

In this example we illustrate the modeling of an automated highway inspired by [5]. In an automated highway cars can form form platoons to increase the capacity of roads. A platoon is a group of cars that are closely following a leader car in the same lane. Platoons may merge or split. A merge may take place if two platoons are close enough. A platoon may split when a car needs to leave the platoon in order to perform some specific maneuver.

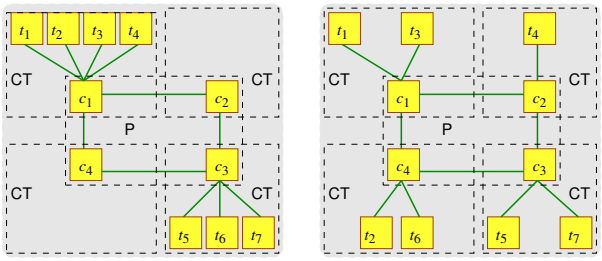


Figure 7: Inter-motif reconfiguration for task migration

We define an atomic component type, *Car* (C) to model the behavior of a car. Each car maintains its position *pos*. A *platoon* (P) motif is composed of a group of cars. The map of a platoon motif is an instance of linked list type. We consider primitives head, and tail which point to the position of the leader and tail of a platoon namely, the beginning and the end of the list. In addition, we consider the primitive append which appends and links two maps of type linked list together. finally, the primitive sublist and length creates a sublist from a linked list and returns the length of the list respectively. The deployment assigns the cars in a bijective manner. We introduce the primitive restrict which retains only the deployment of elements in a given map.

Interaction within a platoon motif is defined by the rule *sync-move*, which synchronizes the movement of all the cars in a platoon by connecting all the move ports of all cars in the motif. The reconfiguration rules, *do-merge*, and *do-split* handle the merging and the splitting of platoons respectively.

do-merge($y_1, y_2 : P, x_1, x_2 : C$) $\equiv$
  when $\langle y_1 : \mathsf{D}(x_1) = \mathsf{H.tail} \rangle \wedge \langle y_2 : \mathsf{D}(x_2) = \mathsf{H.head} \rangle \wedge |x_1.\mathsf{pos} - x_2.\mathsf{pos}| < K$
  do $\mathsf{B} := y_1.\mathsf{B} \cup y_2.\mathsf{B}$, $\mathsf{H} := \mathsf{append}(y_2.\mathsf{H}, y_1.\mathsf{H})$, $\mathsf{D} := y_1.\mathsf{D} \cup y_2.\mathsf{D}$,
    M.create(P, (B, H, D)), M.delete(y_1), M.delete(y_2)
do-split($y : P, x : C$) $\equiv$
  do $\langle y : H_1 := \mathsf{H.sublist}(0, \mathsf{D}(x)), \mathsf{B}_1 := \mathsf{D}^{-1}(\mathsf{H}_1), \mathsf{D}_1 := \mathsf{D.restrict}(\mathsf{H}_1),$
    $\mathsf{H}_2 := \mathsf{H.sublist}(\mathsf{D}(x), \mathsf{H.length}), \mathsf{B}_2 := \mathsf{D}^{-1}(\mathsf{H}_2), \mathsf{D}_2 := \mathsf{D.restrict}(\mathsf{H}_2) \rangle$,
    M.create(P, (B_1, H_1, D_1)), M.create(P, (B_2, H_2, D_2)), M.delete(y)

The effect of the split rule is illustrated in Figure 8. An initial 8-car platoon is split into two other platoons as requested by a leave maneuver of car c_4.

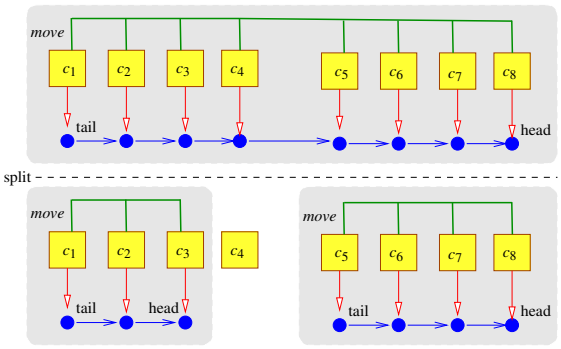


Figure 8: Split reconfiguration for platoons

7 Implementation

Our prototype implementation of DR-BIP includes a concrete language to describe motif-based systems and an interpreter (implemented in JAVA) for the operational semantics. The language provides syntactic

constructs for describing component and motif types, with some restrictions on the maps and deployments allowed[2]. The interpreter allows the computation of enabled interactions and (inter-motif)reconfiguration rules on system configurations, and their execution according to predefined policies (interactive, random, etc).

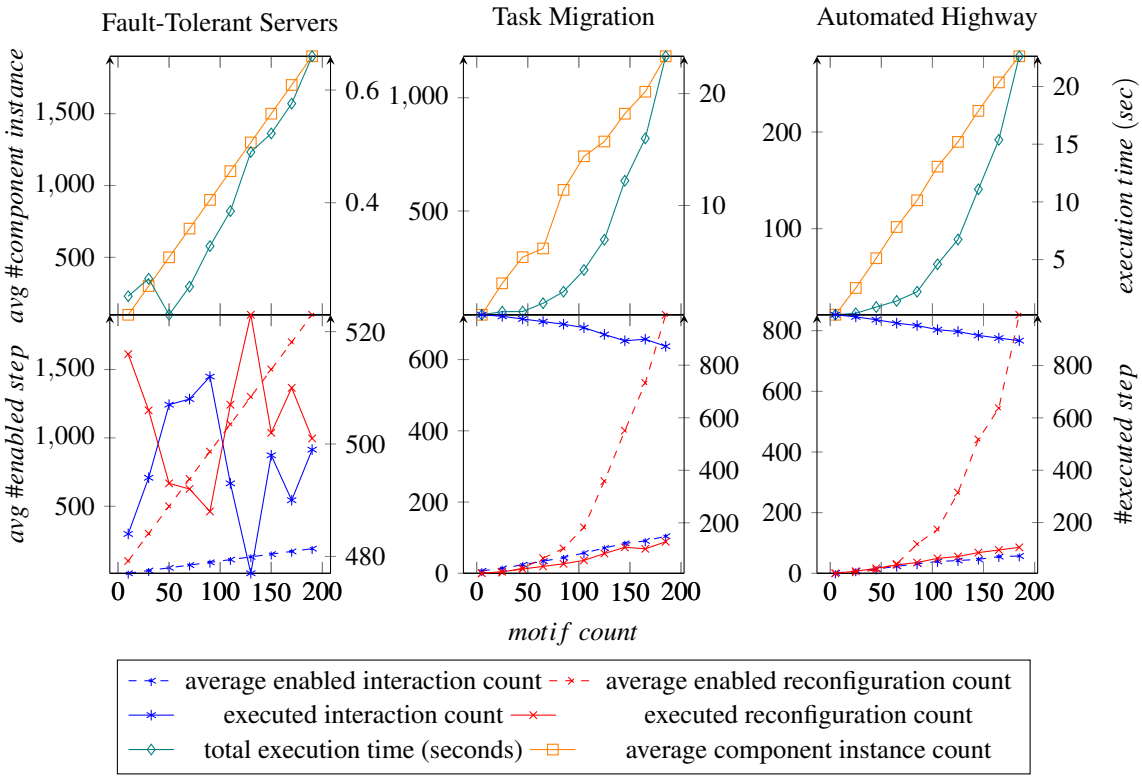


Figure 9: DR-BIP performance

Figure 9 provides measurements of the coordination overhead (at the top, execution time of interactions and reconfigurations, and average component instance count) and system's complexity (at the bottom, initial motif instance count/ average enabled interactions and reconfiguration count) for examples presented in section 6. Each example is modeled using the DR-BIP language and simulated for 1000 random steps by the interpreter. At each step, the interpreter executes randomly either an interaction or a reconfiguration (either within a motif or an inter-motif reconfiguration). To evaluate scalability, we measured the execution time for each example while varying the initial motif instance count between 5 to 190. Each motif instance is initialized with 10 component instances, hence we evaluated the performance on systems of initial size upto 1,900 components. The execution time varied between 0.2 and 23 seconds with the least execution times found in the fault-tolerant example which does not have inter-motif reconfigurations. In contrast, the multicore and automated highway examples include inter-motif reconfiguration rules whose evaluation causes the most time overhead. The above highlights the effectiveness of the prototype in handling systems of moderate size at an acceptable cost.

Regarding system's complexity, we count the average number of enabled steps including interactions and reconfigurations along with the total count of executed steps. On one hand, in the three examples, the average number of enabled reconfigurations is greater than the average of enabled interactions. For instance, in the fault-tolerant servers example, any server can fail/join at any time and this contributes to the high number of reconfigurations enabled. Similarly, any car in a platoon can request a split in the automated highway example and any task can migrate between cores in the multicore example. On the other hand, note that the number of executed steps is balanced between interactions and reconfigurations

[2]maps are restricted to linear graphs

in the fault-tolerant example. However, for the two other examples, the number of executed interactions is greater than the number of reconfigurations due to the fact that the reconfigurations are enabled only under specific conditions: platoons can merge only when they are close enough and tasks can migrate between cores only when there is a load imbalance.

8 Discussion

The DR-BIP framework for programming dynamic reconfigurable systems has been designed to encompass three complementary structuring aspects of component-based coordination. Architecture motifs are environments where live instances of components of predefined types subject to specific parametric interaction and reconfiguration rules. Reconfiguration within a motif supports in addition to creation/deletion of components, the dynamic change of maps and the mobility of components. Maps are a common reference structure that proves to be very useful for both the parametrization of interactions and the mobility of components. It is important to note that a map can have either a purely logical interpretation, or a geographical one or a combination of both. For instance, a purely logical map is needed to describe the functional organization of the coordination in a ring or a pipeline. To describe mobility rules of cars on a highway a map is needed representing at some abstraction level, a geographic map of its external environment e.g. the structure of the highway with fixed and mobile obstacles. Finally a map with both logical and geographic connectivity relations may be used for cars on a highway to express their coordination rules. These depend not only on the physical environment but also on the communication features available.

Structuring a system as a set of loosely coordinated motifs confers the advantage that when components are created or migrate, we do not need to specify associated coordination rules; depending on their type, components are subject to predefined coordination rules of motifs.

The proposed framework is sufficiently expressive and general to allow a complete comparison of existing types of coordination. A basic distinction is between local and shared memory coordination. In DR-BIP, components have disjoint state spaces so they can directly model local memory systems such as actors, data flow systems and distributed systems of any kind. Shared memory systems allow only local interactions between two types of components: objects and threads. Objects can be modeled as non-mobile components that form the shared memory structure. Threads are mobile components which represent the flow of computation and ultimately determine its degree of parallelism. It can be shown that any local memory interacting system can be simulated by a thread-based system.

Clearly these results are too recent and there are many open avenues to be explored. One is how we make sure that the modeled systems meet given properties. The proposed structuring principle allows a separation of concerns between interaction and reconfiguration aspects. To verify correctness of the parametric interacting system of a motif we extend the approach adopted for static BIP: assuming that dynamic connectors correctly enforce the sought coordination, it remains to show that restricting the behavior of deadlock-free components does not introduce deadlocks. We are currently studying an extension of the D-Finder approach [4] for parametric systems which requires the solution of parametric Boolean equations.

To verify the correctness of reconfiguration operations a different approach is taken. If we have already proven correctness of the parametric interacting system of a motif, it is enough to prove that its architecture style is preserved by statements changing the number of components, move components and modify maps and their connectivity. In other words the architecture style is an invariant of the coordination structure. This can be proven by structural induction. The architecture style of a motif can be characterized by a formula of configuration logic ϕ [16]. We have to prove that if a model m of the system satisfies ϕ then after the application of a reconfiguration operation the resulting model m' satisfies ϕ.

Finally the language and the supporting tools should be evaluated against real-life applications. These include mobile systems such as autonomous transport systems, swarm robotics, mobile telecommunication systems as well as various routing algorithms. This is the object of work in the framework of an ongoing European project [11].

References

[1] Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *International Conference on Fundamental Approaches to Software Engineering*, pages 21–37. Springer, 1998. 1, 6.1

[2] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time systems in BIP. In *SEFM'06 Proceedings*, pages 3–12. IEEE Computer Society Press, 2006. 1, 3

[3] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011. 1, 3

[4] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. Compositional verification for component-based systems and application. *IET Software*, 4(3):181–193, 2010. 8

[5] Carl Bergenhem. Approaches for facilities layer protocols for platooning. In *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on*, pages 1989–1994. IEEE, 2015. 6.3

[6] Simon Bliudze and Joseph Sifakis. The algebra of connectors—structuring interaction in bip. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008. 1

[7] Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling dynamic architectures using dy-bip. In *International Conference on Software Composition*, pages 1–16. Springer, 2012. 1

[8] Jeremy S Bradbury. Organizing definitions and formalisms for dynamic software architectures. *Technical Report*, 477, 2004. 1

[9] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A classification of dynamic reconfiguration in component and connector architecture description languages. In *4th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp'17)*, 2017. 1

[10] Calos Canal, Ernesto Pimentel, and José M Troya. Specification and refinement of dynamic software architectures. In *Software Architecture*, pages 107–125. Springer, 1999. 1

[11] CITADEL Consortium. Critical infrastructure protection using adaptive mils. http://www.citadel-project.org/, 2016. H2020/IA-700665. 8

[12] Carlos E Cuesta, Pablo de la Fuente, and Manuel Barrio-Solárzano. Dynamic coordination architecture through the use of reflection. In *Proceedings of the 2001 ACM symposium on Applied computing*, pages 134–140. ACM, 2001. 1

[13] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *TAAS*, 9(2):7:1–7:29, 2014. 1

[14] David Garlan. Software architecture: A travelogue. In *Future of Software Engineering (FOSE'14)*, pages 29–39. ACM, 2014. 1

[15] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 39(6), June 2006. 1

[16] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. Configuration logics: Modeling architecture styles. *J. Log. Algebr. Meth. Program.*, 86(1):2–29, 2017. 8

[17] Nenad Medvidovic, Eric M Dashofy, and Richard N Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1):12–31, 2007. 1

[18] Peyman Oreizy. Issues in modeling and analyzing dynamic software architectures. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 54–57. Citeseer, 1998.

[19] Antero Taivalsaari, Tommi Mikkonen, and Kari Systä. Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In *IEEE 38th Annual Computer Software and Applications Conference (COMPSAC'14)*, 2014. 1