HAL
open science

Programming Dynamic Reconfigurable Systems

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

To cite this version:

Rim El Ballouli, Saddek Bensalem, Marius Bozga, Joseph Sifakis. Programming Dynamic Reconfigurable Systems. Formal Aspects of Component Software - 15th International Conference, FACS 2018, Oct 2018, Pohang, South Korea. pp.118-136, 10.1007/978-3-030-02146-7_6 . hal-01888550

HAL Id: hal-01888550

https://hal.science/hal-01888550v1

Submitted on 5 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Programming Dynamic Reconfigurable Systems*

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

Univ. Grenoble Alpes, CNRS, Grenoble INP***, 38000 Grenoble, France

Abstract. DR-BIP is an extension of the BIP component framework intended for programming reconfigurable systems encompassing various aspects of dynamism. It 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 few examples and discusses avenues for dynamic reconfigurable system design.

Keywords: architectural motifs, components, reconfigurable systems

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, 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 [7] for a survey). The other approach is based on endogenous

*** Institute of Engineering Univ. Grenoble Alpes

* The research leading to these results has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement no. 700665 CITADEL (Critical Infrastructure Protection using Adaptive MILS)

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 in particularly adopted by formalisms based on π-calculus and process algebra, such as [1, 9–11]. 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 [14, 15, 18, 7, 19, 8]. 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 aspects. The first aspect 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 aspect 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 aspect is currently the most challenging. It meets in particular, the vision of "fluid architectures" [19] which allows components/services 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 [3, 2] which encompasses all these three aspects of dynamism. DR-BIP represents one step further in the research work which lead previously to DyBIP [6] for BIP with dynamic interactions and more recently to FunctionalBIP [12] and JavaBIP [17] for BIP with dynamic components and interactions. As such, DR-BIP follows an exogenous approach respecting the strict separation between behavior and architecture. It directly encompasses mul-

tiparty interaction [4] and is rooted in formal operational semantics allowing a rigorous implementation. DR-BIP privileges an 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 adhoc dynamism, it directly encompasses all kinds of dynamism at run time [7]: programmed dynamism and in addition adaptive dynamism, and self-organizing dynamism. It provides support for component creation and removal at run time. Moreover, DR-BIP directly supports component migration from one motif to another. It supports programmed reconfiguration and triggered reconfiguration in particular [8]. The big advantage from using motifs is that when a component is 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 and major design principles. Section 3 briefly recalls the key concepts of BIP and its operational semantics. Section 4 introduces the motif concept and its semantics. Section 5 introduces motif-based systems. Section 6 presents an example with results using the DR-BIP implementation. Finally, section 7 presents conclusions and future work directions.

2 DR-BIP Overview

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.

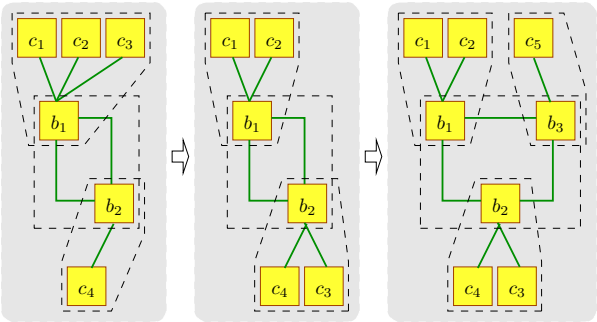

Fig. 1: An example : system reconfigurations

Fig. 1 provides an overall view of the structure and evolution of a motif-based system. The initial configuration consists of six interacting components organized in three motifs (indicated with dashed 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 up-

per 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 their 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 captures specific architectural/functional properties by design.

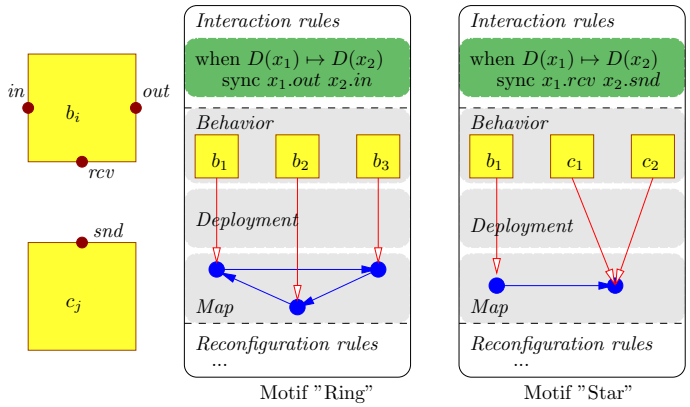

Fig. 2: An example : motifs definition

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

The "Ring" motif illustrated in Fig. 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 $x_2.in$. The rule *defines* 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 Fig. 1 (right). The "Star" motif illustrated in Fig. 2 (right) defines the second type. Here, three components are deployed into a two-position map. 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 $x_1.rcv$ and

$x_2.snd$. The rule *defines* two interactions, namely $b_1.rcv\ c_1.snd$ and $b_1.rcv\ c_2.snd$, also illustrated in Fig. 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 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. As illustrated in Fig. 3, the execution of interactions 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. Nonetheless, these changes have no impact on components, i.e. all running components preserve their state although components may be created/deleted.

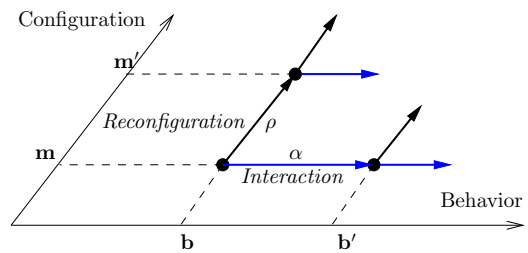

Fig. 3: Reconfiguration vs Interaction Steps

3 Component-Based Systems

BIP [3, 2] 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. We briefly recall the key BIP concepts and their operational semantics.

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 $T \subseteq L \times P \times \mathcal{G}(V) \times \mathcal{F}(V) \times L$ is a finite set of labeled transitions, where $\mathcal{G}(V)$ and $\mathcal{F}(V)$ are respectively Boolean guards and update

functions defined over variables V. Every transition $\tau = (\ell, p, g, f, \ell') \in T$ is equivalently denoted as $\tau = \ell \xrightarrow{p\ g\ f} \ell' \in T$. For every port $p \in P$, we associate a subset of variables $V_p \subseteq V$ exported and available for interaction through p.

For a component type $B^t = (L, P, V, T)$, its set of states is $Q = L \times \mathbf{V}$ where $\mathbf{V}$ is the set of all valuations defined on V. A valuation of a set of variables V is a function $\mathbf{v} : V \to \mathcal{D}$, where $\mathcal{D}$ is an underlying domain of data values. The semantics of a component type B^t is defined as the labeled transition system $[\![B^t]\!] = (Q, \Sigma, \to)$ where the set of labels $\Sigma = \{p(\mathbf{v}_p) \mid \mathbf{v}_p \in \mathbf{V}_p\}$ and transitions $\to \subseteq Q \times \Sigma \times Q$ are defined by the rule:

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

That is, $(\ell', \mathbf{v}')$ is a successor of $(\ell, \mathbf{v})$ labeled by $p(\mathbf{v}''_p)$ iff (1) $\tau = \ell \xrightarrow{p\ g\ f} \ell'$ is a transition of T, (2) the guard g holds on the current state valuation $\mathbf{v}$, (3) $\mathbf{v}''_p$ is a valuation of exported variables V_p and (4) $\mathbf{v}' = f(\mathbf{v}[\mathbf{v}''_p/V_p])$ that is, the next-state valuation $\mathbf{v}'$ is obtained by applying f 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 1. Fig. 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 control location from *idle* to *busy* while performing the computation $v := u + w$.

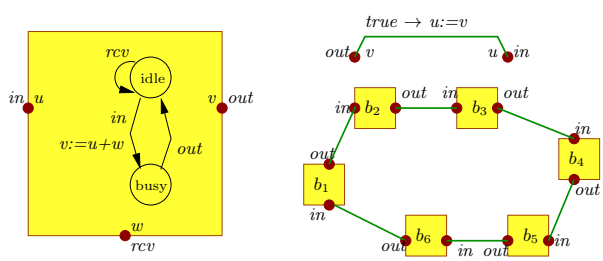

Fig. 4: Component types, interactions and systems in BIP

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 Boolean guard, and F_a is an update 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 $i \in I, p_i \in \mathit{ports}(b_i)$. G_a and F_a are defined on the variables exported by 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 $Q = \langle b \mapsto q \mid b \in B,\ q \in \mathit{states}(b)\rangle$, the set of labels $\Sigma \subseteq \mathcal{P}(\mathit{ports}(B) \times \mathcal{P}(\mathbf{V}))$ contains the ports and sets of values exchanged on interactions and transitions $\rightarrow$ are defined by the rule:

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

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 update 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 2. Fig. 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 update function $u := v$. 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 composed 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 constituents 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 introduces 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 (i) an underlying domain $N(H^t)$ of nodes, (ii) a set of primitives $\Omega(H^t)$ to update/access the map content and (iii) a logic $\mathcal{L}(H^t)$ to express constraints on the map content.

We use maps as dynamic data structures (objects). For a map H, its set of nodes is denoted by $dom(H)$. 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 3. Map types can be directed graphs (V, E) where vertices V denote the positions and edges $E \subseteq V \times V$ expressing the connectivity between these positions. Such a map type (i) has the domain V, (ii) can be manipulated explicitly using primitives such as addVertex, remVertex, addEdge, remEdge and (iii) has predicate constraints such as edge constraints $\cdot \mapsto \cdot$, path constraints $\cdot \mapsto^* \cdot$), etc, with the usual meaning.

Example 4. In the "Ring" example from Fig. 5 the map type is a specific type of graph, that is, a cyclic graph, whose (i) vertices compose the domain and (ii) primitives include initialize, extend, remove to respectively initialize, extend by one vertex and remove one vertex from it.

Deployments are partial mappings of a set B of component instances to the nodes of a map H, formally $D : B \to dom(H) \cup \{\perp\}$. As for maps, deployments are dynamic data structures defined as particular instances of a generic deployment types D^t. We consider a set of primitives $\Omega(D^t)$ to update and/or access the deployment as well as a logic $\mathcal{L}(D^t)$ to express constraints on it.

4.2 Motif types

Definition 1. *A motif type M^t is a tuple $((\mathcal{B}, \mathcal{H}, \mathcal{D}), \mathcal{IR}, \mathcal{RR})$ where:*

- *the triple $(\mathcal{B}, \mathcal{H}, \mathcal{D})$ are motif meta-variables used to maintain respectively the set of component instances, the map and the deployment of component instances to the map,*
- *$\mathcal{IR}$ 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, Ψ is a rule constraint, and (P_I, G_I, F_I) is the interaction specification, namely the set of ports of involved components, 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, 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 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 $D : B \to dom(H) \cup \{\bot\}$ a deployment. The configuration can dynamically change as the meta-variables are being updated when reconfiguration rules are executed. The meaning of the rules is explained in the next subsections.

Example 5. Fig. 5 (left) provides the formal definition of the "Ring" motif type presented in section 2. The motif type contains one interaction rule denoted as sync-inout and three reconfiguration rules denoted respectively do-init, do-insert and do-remove. Fig. 5 (right) provides one motif configuration 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 $D = \{b_i \mapsto n_i\}_{i=1,6}$.

sync-inout(x_1: C, x_2 : C) $\equiv$ when D(x_1) $\mapsto$ D(x_2)
  sync x_1.out x_2.in / true $\to$ x_2.u := x_1.v
do-init() $\equiv$ when B = $\emptyset$
  do x_1 := B.create(C, busy),
    x_2 := B.create(C, idle), H.init(),
    n_1 := H.extend(), D(x_1) := n_1
    n_2 := H.extend(), D(x_2) := n_2
do-insert() $\equiv$ do x := B.create(C, idle),
    n := H.extend(), D(x) := n
do-remove(x : C) $\equiv$ when |B| $\geq$ 3 $\wedge$ x.idle
  do n := D(x), B.delete(x), H.remove(n)

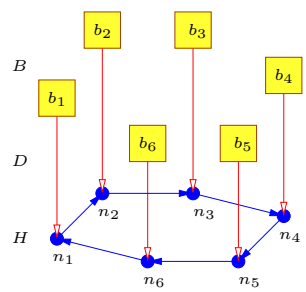

Fig. 5: The "Ring" motif type

4.3 Rule constraints

The motif behavior is defined by interaction and reconfiguration rules. Rule parameters $\mathcal{Z}$ include typed symbols denoting (sets of) component instances or map nodes and interpreted as (subsets) elements of B or $dom(H)$ respectively. Rule constraints Ψ are boolean combinations of map, deployment and basic 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 basic constraint using equality and/or cardinality constraints on parameters, $\Psi^{\mathcal{H}}$ denotes a constraint on the map (conforming to the map logic $\mathcal{L}(H^t)$) and $\Psi^{\mathcal{H}}$ denotes a 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:

$$\begin{aligned}
B, H, D, \zeta \models \psi^0 &\text{ iff } \zeta \cup [B/\mathcal{B}, H/\mathcal{H}, D/\mathcal{D}] \models \psi^0 \\
B, H, D, \zeta \models \psi^{\mathcal{H}} &\text{ iff } H, \zeta \cup [B/\mathcal{B}, D/\mathcal{D}] \models \psi^{\mathcal{H}} \\
B, H, D, \zeta \models \psi^{\mathcal{D}} &\text{ iff } D, \zeta \cup [B/\mathcal{B}, H/\mathcal{H}] \models \psi^{\mathcal{D}} \\
B, H, D, \zeta \models \Psi_1 \wedge \Psi_2 &\text{ iff } B, H, D, \zeta \models \Psi_1 \text{ and } B, H, D, \zeta \models \Psi_2 \\
B, H, D, \zeta \models \neg\Psi &\text{ iff } B, H, D, \zeta \not\models \Psi
\end{aligned}$$

That means, equality/inequality 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.

4.4 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 as follows:

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

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, only data from components participating in the interaction can be used.

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 6. The ring motif illustrated in Fig. 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 Fig. 4.

4.5 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 as follows:

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

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

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

Example 7. The ring motif illustrated in Fig. 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.

4.6 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.

[1] up to the choice of fresh component instance

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

- the states of set Q correspond to motif configurations B, H, D consistently extended with configurations for all component instances $\mathbf{b} = \langle b \mapsto q \mid b \in B, q \in \mathit{states}(b)\rangle$,
- the labels of Σ correspond to valid interactions α constructed on components and reconfiguration actions ρ,
- the 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

$$(\textsc{Mot-I})\ \frac{\Gamma = \cup_{r \in \mathcal{IR}} \Gamma(r) \qquad \Gamma(B)\ :\ \mathbf{b} \xrightarrow{\alpha} \mathbf{b}'}{M^t\ :\ (B, H, D, \mathbf{b}) \underset{I}{\xrightarrow{\alpha}} (B, H, D, \mathbf{b}')}$$

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

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 guards G_R are satisfied for the given interpretation of parameter ζ and configurations of component instances $\mathbf{b}$ and (2) the current and next motif configuration are related according to the semantics of the action A_R. The dichotomy between interaction and reconfiguration steps ensures separation of concerns for execution within a motif as previously discussed in section 2 and illustrated in Fig. 3.

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/inequality 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:

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

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:

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

That is, guards are the same as for interaction rules. The action $y := \mathcal{M}.\mathit{create}(M^t, (e_B, e_H, e_D))$ 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 $\mathcal{M}.\mathit{delete}(y)$ denotes the deletion of the motif instance y, that is, its removal from the set of motif instances. The action $y.\mathcal{B}.\mathit{migrate}(x)$ 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:

$$\begin{aligned}
&[\![y := \mathcal{M}.\mathit{create}(M^t, (e_B, e_H, e_D))]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M \cup \{m\}, \mathbf{m}', B, \mathbf{b}, \zeta') \\
&\quad \text{where } m = (M^t, k) \text{ fresh, } \ \mathbf{m}' = \mathbf{m} \cup \langle m \mapsto (e_B, e_H, e_D)(\zeta_{\mathbf{m}}) \rangle, \zeta' = \zeta[y \mapsto m]
\end{aligned}$$

$$
\begin{aligned}
&[\![\mathcal{M}.delete(y)]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M \setminus \{m\}, \mathbf{m}_{|M\setminus\{m\}}, B, \mathbf{b}, \zeta)\\
&\quad \text{where } m = \zeta(y) \in M\\
&[\![y.\mathcal{B}.migrate(x)]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M, \mathbf{m}', B, \mathbf{b}, \zeta)\\
&\quad \text{where } m = \zeta(y) \in M, m \mapsto (B_1, H, D) \in \mathbf{m},\ \zeta(x) \mapsto b \in B,\\
&\quad \mathbf{m}' = \mathbf{m}[m \mapsto (B_1 \cup \{b\}, H, D[b \mapsto \bot])]\\
&[\![\langle y : A_R\rangle]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M, \mathbf{m}', B', \mathbf{b}', \zeta')\\
&\quad \text{where } m = \zeta(y) \in M, m \mapsto (B_1, H, D) \in \mathbf{m},\\
&[\![A_R]\!](B_1, H, D, \mathbf{b}, \zeta) = (B_1', H', D', \mathbf{b}', \zeta')\\
&\quad \text{where } \mathbf{m}' = \mathbf{m}[m \mapsto (B_1', H', D')], B' = B \cup B_1'\\
&[\![z := e]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = (M, \mathbf{m}, B, \mathbf{b}, \zeta[z \mapsto \zeta_{\mathbf{m}}(e)])\\
&[\![A_{R1}^*, A_{R2}^*]\!](M, \mathbf{m}, B, \mathbf{b}, \zeta) = ([\![A_{R2}^*]\!] \circ [\![A_{R1}^*]\!])(M, \mathbf{m}, B, \mathbf{b}, \zeta)
\end{aligned}
$$

Example 8. Consider an inter-motif reconfiguration rule for two "Ring" motifs:

do-merge(y_1, y_2 : Ring) ≡
  <u>when</u> y_1.B ∩ y_2.B = ∅ and |y_1.B| + |y_2.B| ≤ 10
  <u>do</u> 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 10. 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 $((B_i^t)_i, (M_j^t)_j, \mathcal{RR}^*)$ consisting of a set of component types $(B_i^t)_i$, a set of motif types $(M_j^t)_j$ 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

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

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

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

Rules (M-I) and (M-R1) lift the transitions (steps) allowed within the motifs at the level of the system, respectively for interactions and reconfigurations. The rule (M-R2) handles inter-motif reconfiguration. These transitions 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 Implementation and Experiments

We have developed a prototype implementation of DR-BIP including 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).

We have effectively used DR-BIP for programming reconfigurable systems in different application domains [13]. For better illustration of DR-BIP concepts, we reconsider hereafter the exercise on dynamic task management for a multicore platform proposed in [13]. A *multicore task system* consists of a fixed $n \times n$ grid of interconnected homogeneous cores, each executing a finite number of tasks. Every task is either running or completed; running tasks may execute on the associated cores and get eventually completed. The load of a core is defined as the number of its associated tasks, both

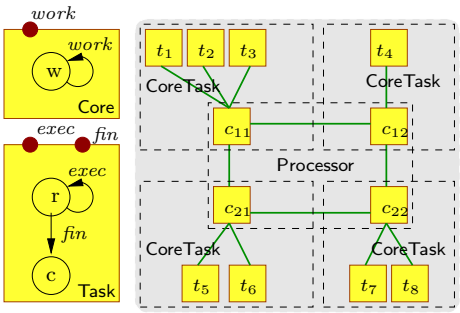

Fig. 6: Multicore Task System

[2] maps are restricted to simple graphs e.g., chain, cyclic, star

running and completed. A multicore task system is *dynamic* if the overall number of tasks and their allocation to cores may change over time. More specifically, new running tasks may enter the system at the core c_{11} and completed tasks may be withdrawn from the system at the core c_{nn}. Moreover, any task is allowed to migrate from its core to any of the neighboring cores (left, right, top or bottom) in the grid, provided the load of the receiving core is smaller (K).

Fig. 6 presents the overall structure of the motif-based system for four cores. We distinguish two types of atomic components, namely Task and Core. Multiple cores are interconnected together in a motif of type Processor. The interconnecting topology reflects the platform architecture (e.g., a 2×2 grid in the figure) and is enforced using a similar grid-like map and deployment. An additional CoreTask motif type is used to represent every core with its assigned tasks. The interactions in the system are defined within the CoreTask motif. The execution of a task by the core and resp. the task completion are represented by the rules:

$$\textsf{sync-coretask-exec}(x_1 : \textsf{Core},\ x_2 : \textsf{Task}) \equiv \underline{\textsf{sync}}\ x_1.\textsf{work}\ x_2.\textsf{exec}$$
$$\textsf{sync-coretask-fin}(x : \textsf{Task}) \equiv \underline{\textsf{sync}}\ x.\textsf{fin}$$

The migration of a task from one core to another is modeled using an inter-motif reconfiguration rule which involves three distinct motifs. A task x_3 migrates from motif y_1 (of type CoreTask) to motif y_2 (of type CoreTask) if the core x_1 of y_1 is connected to the core x_2 of y_2 (according to the processor motif Processor) and if the number of tasks in y_1 exceeds the number of tasks in y_2 by constant K:

$$\begin{aligned}
&\textsf{do-migrate}(y_1, y_2 : \textsf{CoreTask},\ y_3 : \textsf{Processor},\ x_1, x_2 : \textsf{Core},\ x_3 : \textsf{Task}) \equiv \\
&\quad \underline{\textsf{when}}\ \langle\ y_1 : x_1 \in \textsf{B}\ \rangle \wedge \langle\ y_2 : x_2 \in \textsf{B}\ \rangle \wedge \langle\ y_3 : \textsf{D}(x_1) \mapsto \textsf{D}(x_2)\ \rangle \wedge \\
&\qquad\qquad |y_1.\textsf{B}|\ >\ |y_2.\textsf{B}| + \textsf{K} \wedge x_3 \in y_1.\textsf{B} \\
&\quad \underline{\textsf{do}}\ y_2.\textsf{migrate}(x_3),\ y_1.\textsf{delete}(x_3)
\end{aligned}$$

Fig. 7 illustrates the execution of the dynamic multicore task system with 3×3 cores for 3000 steps. Each core is initialized with a random load between 1 and 20. The constant K is set to 3, hence tasks are allowed to migrate to neighboring cores (left, right, top or bottom) that differ in task load by at least 3 tasks. The cores c_{11}, and c_{33} are used to respectively create new tasks and withdraw completed tasks. These two cores retain the maximum and minimum load. As tasks migrate, the task load of cores converges and balances along the execution having at most a difference of 3 tasks between neighboring cores. For example, in core c_{21} the task load increased from 6 to 14. As expected the cores (c_{21}, and c_{12}) closest to c_{11} maintain a high load and as we move away from c_{11} the core's load gradually decreases.

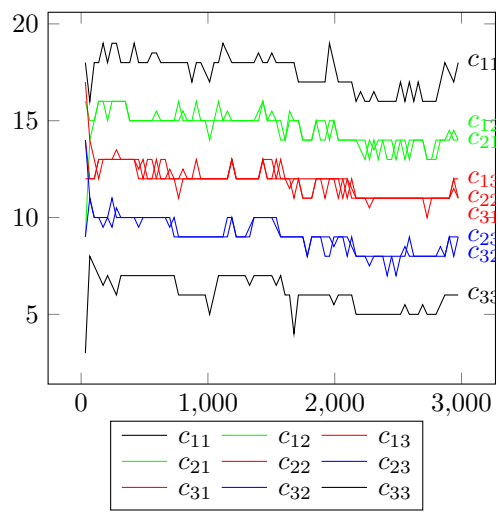

Fig. 7: Task load across 3000 steps

7 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 their 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. 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 have recently shown this approach can be extended for parametric systems [5].

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.

References

1. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: International Conference on Fundamental Approaches to Software Engineering. pp. 21–37. Springer (1998)

2. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software **28**(3), 41–48 (2011)
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time systems in BIP. In: SEFM'06 Proceedings. pp. 3–12. IEEE Computer Society Press (2006)
4. Bliudze, S., Sifakis, J.: The algebra of connectors structuring interaction in BIP. IEEE Transactions on Computers **57**(10), 1315–1330 (2008)
5. Bozga, M., Iosif, R., Sifakis, J.: Checking deadlock-freedom of parametric component-based systems. arXiv preprint arXiv:1805.10073 (2018)
6. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling dynamic architectures using Dy-BIP. In: International Conference on Software Composition. pp. 1–16. Springer (2012)
7. Bradbury, J.: Organizing definitions and formalisms for dynamic software architectures. Tech. Rep. 2004-477, Software Technology Laboratory, School of Computing, Queen's University (2004)
8. Butting, A., Heim, R., Kautz, O., Ringert, J.O., Rumpe, B., Wortmann, A.: 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)
9. Canal, C., Pimentel, E., Troya, J.: Specification and refinement of dynamic software architectures. In: Software Architecture, pp. 107–125. Springer (1999)
10. Cuesta, C., de la Fuente, P., Barrio-Solárzano, M.: Dynamic coordination architecture through the use of reflection. In: Proceedings of the 2001 ACM symposium on Applied computing. pp. 134–140. ACM (2001)
11. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. TAAS **9**(2), 7:1–7:29 (2014)
12. Edelmann, R., Bliudze, S., Sifakis, J.: Functional BIP: Embedding connectors in functional programming languages. Journal of Logical and Algebraic Methods in Programming **92**, 19–44 (2017)
13. El Ballouli, R., Bensalem, S., Bozga, M., Sifakis, J.: Four exercises in programming dynamic reconfigurable systems: Methodology and solution in DR-BIP. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 8th International Symposium, ISoLA 2018 (2018), to appear
14. Garlan, D.: Software architecture: A travelogue. In: Future of Software Engineering (FOSE'14). pp. 29–39. ACM (2014)
15. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: A survey. IEEE Trans. on Soft. Eng. **39**(6) (2006)
16. Mavridou, A., Baranov, E., Bliudze, S., Sifakis, J.: Configuration logics: Modeling architecture styles. J. Log. Algebr. Meth. Program. **86**(1), 2–29 (2017)
17. Mavridou, A., Rutz, V., Bliudze, S.: Coordination of dynamic software components with JavaBIP. In: International Conference on Formal Aspects of Component Software. pp. 39–57. Springer (2017)
18. Oreizy, P.: Issues in modeling and analyzing dynamic software architectures. In: International Workshop on the Role of Software Architecture in Testing and Analysis. pp. 54–57 (1998)
19. Taivalsaari, A., Mikkonen, T., Syst, K.: 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)