International Journal on Software Tools for Technology Transfer (2021) 23:701–719
https://doi.org/10.1007/s10009-020-00596-7

GENERAL

Special Issue: MeTRID

Programming dynamic reconfigurable systems

Rim El Ballouli[1] · Saddek Bensalem[1] · Marius Bozga[1] · Joseph Sifakis[1]

Accepted: 1 December 2020 / Published online: 5 January 2021

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 inter-play 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; and (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 (ADLs) (see [11] 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 in particularly adopted by formalisms such as DYNAMIC WRIGHT [3], LEDA [14], PILAR [35], SCEL [17] to name just a few based on process algebra. The obvious advantage of endogenous coordination is that programmers do not have to build explicitly a global coordination model. The absence of such a model makes the validation of coordination mechanisms and the study of their underlying properties much harder. In contrast, 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 [29]. In this case, the coordination model is external to the behavior and can therefore be used to perform some analysis almost independently from the behavior.

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).

✉ Marius Bozga
Marius.Bozga@univ-grenoble-alpes.fr

1 Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), VERIMAG, 38000 Grenoble, France

There exists a huge literature on architecture modeling reviewed in detailed surveys classifying the various approaches and outlining new trends [24] and needs from an industrial perspective [29]. 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 inter-play 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" [38] 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 presents *dynamic reconfigurable* BIP (DR-BIP) component framework, an extension of BIP [4,5] which encompasses all these three aspects of dynamism. DR- BIP has been introduced in [22] and represents one further step in our research toward extending BIP with dynamic features. This research was initiated with DyBIP [9] for BIP with dynamic interactions and more recently continued with Functional-BIP [20] and Java-BIP [31] 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 multiparty interaction [7] 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 DR- BIP does not allow ad hoc dynamism, it directly encompasses several kinds of dynamism at run time, namely programmed dynamism and in addition adaptive dynamism, and self-organizing dynamism according to the classification in [11]. 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 as defined in [13]. 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" as in fluid architectures [38].

This paper is an extended version of two recent conference papers, namely [22] presenting the formal foundation and [21] introducing additional examples of DR- BIP. It was restructured to provide a comprehensive introduction and clarification of key DR- BIP concepts as well as to fully illustrate its modeling expressivity on a complete set of benchmarks. It justifies the proposed concepts, discusses their limitations and identifies potential improvements from a practical point of view. Furthermore, it provides an extended discussion of related work.

The paper is organized as follows. Section 2 provides a brief overview of DR- BIP and major design principles. Section 3 briefly recalls the key concepts of BIP and its operational semantics. Section 4 introduces formally the motif concept and its semantics, and Sect. 5 introduces formally motif-based systems. Section 6 presents several examples with benchmarks using the DR- BIP implementation as well as some lessons learned from these experiments. We discuss related work in Sect. 7. Finally, Sect. 8 presents conclusions and future work directions.

2 DR-BIP overview

The DR- BIP framework is designed to cover the practical needs for the design of dynamic systems and, therefore, fulfill specific requirements for rigorous design and analysis. It allows to:

- specify architectural constraints/styles, i.e., define architectures as parametric operators on components guaranteeing by design specific properties,
- describe systems with evolving architectures, i.e., define system architecture that can be updated at runtime using dedicated primitives,
- support separation of concerns, i.e., keeping separate the component behavior (functionality) from the system architecture to avoid blurring the behaviors with information about their execution context and/or reconfiguration needs,
- provide sound foundation for analysis and implementation, i.e., rely on a well-defined operational semantics,

leveraging on existing models for rigorous component-based design.

The following motivating example belongs to the category of dynamic systems we are interested to consider for DR-BIP. This example will be used along the paper to illustrate the newly proposed concepts.

Example 1 (Dynamic token ring) A *token ring* consists of two or more identical components interconnected using unidirectional communication links according to a ring topology. A number of *tokens* are circulating within the ring. A component is *busy* when it holds a token and *idle* otherwise. A component can do specific internal actions depending on its state, busy or idle. It can receive a token from the incoming link only its idle and send its token on the outgoing link only when its busy.

A token ring is *dynamic* if idle components are allowed to leave the ring at any time as long as at least two components remains in the ring. New idle components are allowed to enter the ring at any time (as long as the maximal allowed ring size is not reached). A *token ring system* consists of one or more, pairwise disjoint, token rings. A token ring system is *dynamic* if every ring is dynamic, and moreover, two rings are allowed to *merge* into a single one provided their overall size is not exceeding the maximal allowed ring size.

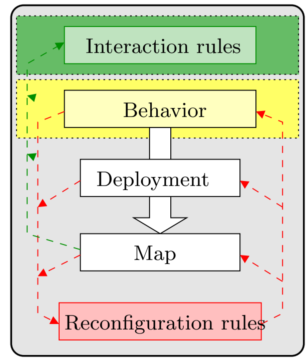

Fig. 1 Motif concept

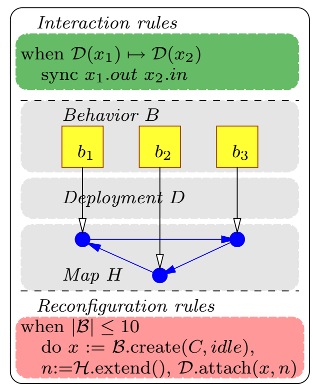

Fig. 2 Motif example

2.1 Motifs for Dynamic Architectures

In DR- BIP, a *motif* is the elementary unit used to describe dynamic architectures. A motif encapsulates (i) behavior, as a set of components, (ii) interaction rules dictating multiparty interaction between components and (iii) reconfiguration rules dictating the allowed modifications to the configuration of a motif including the creation/deletion/migration of components. Motifs are structurally organized as the deployment of component instances on a logical map as illustrated in Fig. 1. Maps are arbitrary graph-like structures consisting of interconnected positions. Deployments relate components to positions on the map.

Example 2 (Motif structure) Figure 2 (middle) illustrates the proposed motif concept for describing the dynamic token ring system introduced in Example 1. In the depicted configuration, three component instances b_1, b_2, b_3 define the behavior B. They are deployed into a three-position cycle map denoted by H. The deployment is denoted by D.

The definition of the motif is completed by two sets of rules, defining interactions and reconfiguration actions of the following generic forms:

interaction-rule ::=
  sync-*rule-name*(*formal-params*) ≡
    when *rule-constraint*
    sync *interaction-ports*
      [[*guard* →] *interaction-action*$^+$]

reconfiguration-rule ::=
  do-*rule-name*(*formal-params*) ≡
    when *rule-constraint*
    do [*guard* →] *reconfiguration-action*$^+$

Both sets of rules are interpreted on the current motif configuration. *Formal-params* denotes (sets of) component instances and defines the scope of the rule. *Rule-constraint* defines the conditions under which the rule is applicable. Constraints are essentially Boolean combinations on deployment and map constraints built from *formal-args*. An interaction rule also defines the set of interacting ports (*interaction-ports*), the interaction guard (*guard*) and the associated interaction actions (*interaction-action*). The guard and the action define, respectively, a triggering condition and an update of the data of components participating in the interaction. Finally, a reconfiguration rule defines a reconfiguration guard (*guard*) and a number of reconfiguration actions (*reconfiguration-action*) to update the content of the motif. Such actions include creation/deletion of component instances, and change of their deployment on the map as well as change of the map itself, i.e., adding/removing map positions and their interconnection. Notice that rule constraints and guards deal with complementary aspects. The former are constraints on motif configuration (map and deployment), whereas the later are constraints on component data only. In a similar way, reconfiguration actions update motif configurations, whereas interaction actions update component data only.

Example 3 (Motif rules) The interaction rule given in Fig. 2 (top) reads as follows: for components x_1, x_2 deployed on adjacent nodes (that is, $\mathcal{D}(x_1) \mapsto \mathcal{D}(x_2)$) connect their ports $x_1.out$ and $x_2.in$.[1] This rule *defines* three interactions between the components, namely $\{b_1.out\ b_3.in\}$, $\{b_3.out\ b_2.in\}$ and $\{b_2.out\ b_1.in\}$. The reconfiguration rule given in Fig. 2 (bottom) allows to extend the ring by adding one more component. The rule is applicable as long as the number of component instances $|\mathcal{B}|$ is less than or equal to 10. When executed, a new component x of type C is created at initial state *idle* (that is, $x := \mathcal{B}.\text{create}(C, idle)$), a new node n is added to the cycle map (that is, $n := \mathcal{H}.\text{extend}()$) and the component x is deployed on the node n (that is, $\mathcal{D}.\text{attach}(x, n)$).

Notice that the distinction between reconfiguration and interaction rules allows separation of concerns in modeling dynamic architectures. On one hand, reconfiguration rules are used to update the motif structure (components, map, deployment) under specific conditions (as depicted by the red arrows in Fig. 1). On the other hand, interaction rules use the motif structure to define how the components of the motif are interconnected (as depicted by the green arrows in Fig. 1). This approach associates interaction rules with motifs, and these rules remain unchanged when components are created or removed.

The reason for choosing maps and deployments as a mean for structuring motifs is 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 they also provide a very convenient way to express various forms of reconfiguration. Both maps and deployments are implemented as dynamic collections of objects, with specific

[1] The dot operator is used interchangeably to access a component's port/data, and to access a motif's components/deployment/map, and to apply primitives over a motif's deployment/map.

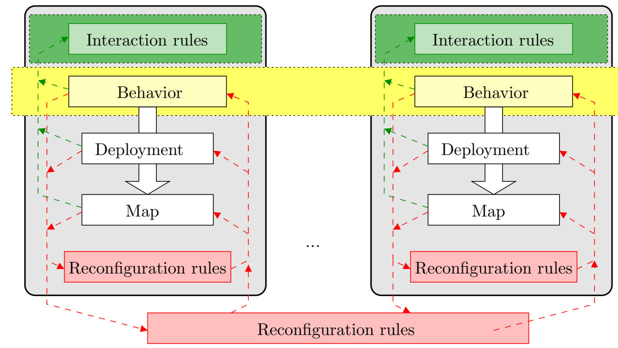

Fig. 3 Motif-based system concept

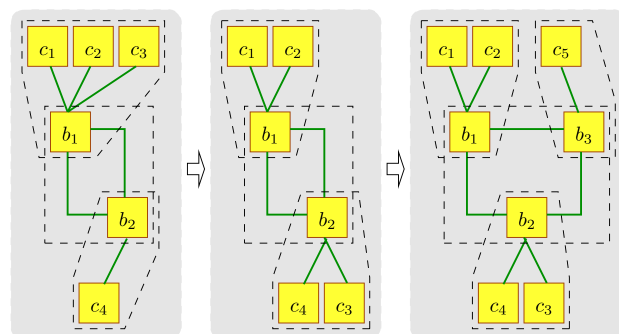

Fig. 4 An example of system reconfigurations

interfaces, in a similar way to standard collection libraries available for standard programming languages.

2.2 Motif-based systems

Several types of motifs may be defined separately by specifying the types of hosted components, parametric interactions and reconfiguration rules. Then, systems are described by superposing a number of motif instances of certain motif types. In this manner, the overall system architecture captures specific architectural/functional properties by design.

Systems are defined as collections of motifs sharing a set of components as depicted in Fig. 3. Each motif can evolve independently of the others, depending only on its internal structure and associated rules. Furthermore, several motifs can synchronize all together to jointly perform a reconfiguration of the system. Coordination between motifs is therefore possible either implicitly by means of shared components or explicitly by means of inter-motif reconfiguration rules. These rules allow joint reconfiguration of several motif instances. They also allow two additional types of actions, respectively, creation and deletion of motif instances, and exchanging component instances between motifs.

Both coordination mechanisms were proven useful and easy to use in practice. On the one hand, global reconfiguration rules provide an imperative way of changing several motifs simultaneously, e.g., to migrate a component between motifs, to merge several motifs into a single one, etc. On the other hand, sharing a component between several motifs allows controlling local reconfiguration in the motifs. For instance, local reconfiguration rules may be enabled in some motif and disabled in another one, depending on the state of the component.

How these two coordination mechanisms are combined depends on the dynamics of the considered system architecture. In some cases, the dynamics can be captured by a *fixed* number of motifs with a very restricted form of global reconfiguration (e.g., only migration of components between motifs). This is the situation for the dynamic multicore task system in Sect. 6.1 and the self-organizing robot colonies in Sect. 6.3. In other cases, the dynamics is captured by an *evolving* number of motifs and complex global reconfiguration (e.g., merging or splitting existing motifs). This is the situation for the dynamic token ring example, as well as for the highway traffic system in Sect. 6.2.

Figure 4 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 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 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. Furthermore, a new motif is created containing b_3 and c_5.

2.3 Execution model

The evolution of motif-based systems in DR- BIP 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 the absence of reconfigurations, the system keeps a fixed static architecture and behaves like an ordinary BIP system. 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 in the architecture. However, these changes have no impact on components, i.e., all running components preserve their state although components may be created/deleted. This independence between execution steps is illustrated in Fig. 5.

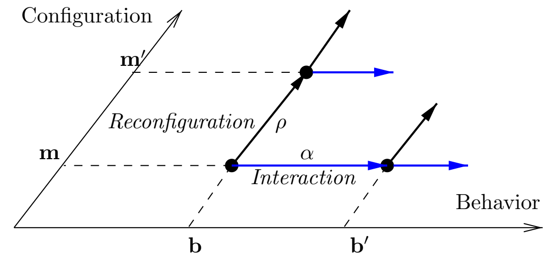

Fig. 5 Reconfiguration versus interaction steps

3 Component-based systems

BIP [4,5] is the underlying component-based framework for 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 $\mathbf{v}$

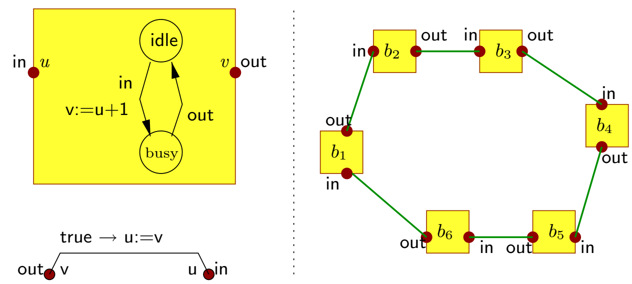

Fig. 6 Component types, interactions and systems in BIP

previously updated according to $\mathbf{v}''_p$. Whenever a p-labeled successor exists from a state, we say that p is *enabled* in that state.

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

Example 4 (Component type) Figure 6 (left, top) illustrates graphically a component type. The component has two ports (in, out) attached with variables (respectively, u, v. 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+1.

3.2 Systems of components

Systems of components $\Gamma(B)$ are obtained by composing a finite set of component instances $B = \{b_1, \dots, b_n\}$ 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 $i \in I$, $p_i \in ports(b_i)$. 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 $Q = \langle b \mapsto q \mid b \in B,\ q \in spsstates(b)\rangle$, the set of labels $\Sigma \subseteq \mathcal{P}(ports(B) \times \mathcal{P}(\mathbf{V}))$ contains the ports and sets of values exchanged on interactions and transitions $\rightarrow$ are defined by the rule:

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

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 $((\ell_1, \mathbf{v}_1), \dots, (\ell_n, \mathbf{v}_n))$, 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 5 (System of components) Figure 6 (left, bottom) depicts 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 are transferred from the out port to the in port. Figure 6 (right) illustrates a system obtained by composing six b_i instances with six out in interactions in a ring structure.

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 subsections present 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 of type H^t, its set of nodes is denoted by $dom(H)$ and is a subset of $N(H^t)$. For any primitive $op \in \Omega(H^t)$, we use the dotted notation $H.op(\ldots)$ 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 6 (Maps as directed graphs) 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 predicates allowing to express edge constraints $\cdot \mapsto \cdot$, path constraints $\cdot \mapsto^* \cdot$, etc., with the usual meaning.

Example 7 (Maps as cycle graphs) In the dynamic token ring example from Fig. 2, the map type is a *cycle graph* consisting of a single cycle, where (i) vertices compose the domain, (ii) primitives include init, extend, remove to, respectively, initialize to an empty cycle, extend by one vertex (inserted arbitrarily), remove one specified vertex from the cycle and (iii) predicates allow for checking edge constraints $\cdot \mapsto \cdot$, as usual.

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. In particular, we will use the primitive attach to associate a component instance to a node of the map.

Given a deployment $D : B \to dom(H) \cup \{\perp\}$, for a subset of components $B' \subseteq B$ we denote by $D_{|B'}$ the restriction of D to B', that is, the partial function $D_{B'} : B' \to dom(H) \cup \{\perp\}$ where $D_{|B'}(b) = D(b)$ for all $b \in B'$. Similarly, for an arbitrary map H' we denote by $D_{|H'}$ the restriction of D to H', that is, the partial function $D_{|H'} : B \to dom(H') \cup \{\perp\}$ where $D_{|H'}(b) = D(b)$ if $D(b) \in dom(H) \cap dom(H')$ and $\perp$ otherwise.

4.2 Motif types

Henceforth, we consider a given finite collection of component types, map types and deployment types.

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

- the triple $(\mathcal{B}, \mathcal{H}, \mathcal{D})$ consists of motif meta-variables, that is, typed symbols used to denote, respectively, the set of component instances, the map and the deployment of component instances on 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 valuation of meta-variables $\mathcal{B}, \mathcal{H}, \mathcal{D}$ as, respectively, B, H, D where (i) B is a finite set of components instances with types belonging to the predefined set of component types, (ii) H is a map instance of the type of $\mathcal{H}$, (iii) D is a deployment instance of the type of $\mathcal{D}$ which associates component instances from B to nodes of the map, formally $D : B \to dom(H) \cup \{\perp\}$.

The meaning of the rules is explained in the next subsections. Note that motif configuration can dynamically change as the meta-variables are being updated in reconfiguration rules. Furthermore, component instances can interact as dictated by interaction rules. Overall, we tacitly restrict to syntactically consistent motif definitions, that is, where the interaction and reconfiguration rules are restricted to use only the map and deployment primitives defined for the types of $\mathcal{H}$ and $\mathcal{D}$, respectively.

Example 8 (Dynamic token ring motif type) Figure 7 illustrates the structure of a Ring motif type defined for the dynamic token ring system. In any configuration, the behavior B contains several component instances, all of the same type C presented in Example 4. The map H is a cycle graph (or equivalently, a circular linked list) with specific primitives presented in Example 7. The deployment D assigns components to locations of the map in a bijective manner.

Moreover, we consider that our Ring motif type contains one interaction rule denoted as sync-ring-inout for defining

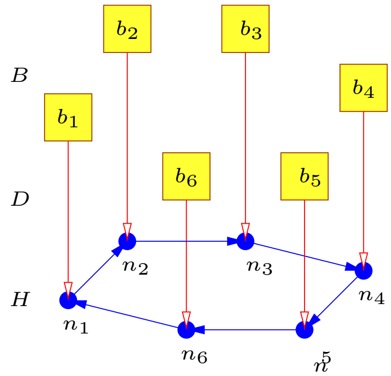

Fig. 7 A configuration of the dynamic token ring motif type

interactions and three reconfiguration rules denoted, respectively, do-ring-init, do-ring-insert and do-ring-remove for dynamic reconfiguration, as follows:

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

For the sake of readability, we use the generic textual syntax of rules proposed in Sect. 2. This textual representation is actually a readable alternative for the abstract representation introduced in Definition 1. The relation between the two representations will be clarified in the following subsections.

4.3 Rule parameters and constraints

The motif evolution is defined by interaction and reconfiguration rules. The set of 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 set constraints on parameters, $\Psi^{\mathcal{H}}$ denotes a constraint on the map (conforming to the map logic $\mathcal{L}(H^t)$, for H^t being the type of $\mathcal{H}$) and $\Psi^{\mathcal{D}}$ denotes a constraint on the deployment (conforming to the deployment logic $\mathcal{L}(D^t)$, for D^t being the type of $\mathcal{D}$). For example, the sync-ring-inout interaction rule in Example 8 has two parameters x_1, x_2 denoting components of type C. The rule constraint $\mathcal{D}(x_1) \mapsto \mathcal{D}(x_2)$ checks if x_1 and x_2 are deployed on adjacent nodes on the map, using the $\mapsto$ predicate defined for cycle graphs.

For fixed motif configuration 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:

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

That means, equalities and/or set 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:

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

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_d (resp. op'_d) are operations on (resp. sets of) 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:

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

In the above, we tacitly lift the interpretation of ζ to port interactions P_I, guards G_I and actions A_I which are all constructed from rule parameters $\mathcal{Z}$. The resulting 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 9 (Interaction rules) The ring motif type presented in Example 8 has a unique interaction rule denoted sync-ring-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. Consider the motif configuration depicted in Fig. 7. The interpretation of rule parameters $\zeta = \{x_1 \mapsto b_3, x_2 \mapsto b_4\}$ satisfies the rule constraint and therefore defines the binary interaction (P_a, G_a, F_a) where $P_a = \{b_3.out, b_4.in\}$, $G_a =$ true, $F_a = (b_4.u := b_3.v)$. The set of all defined interactions, for all interpretations of rule parameters satisfying the rule constraint, is depicted in Fig. 6.

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 (component instances, map nodes, maps, deployments, etc.). The local context is updated using standard assignments. As mentioned already, we tacitly restrict to syntactically correct rules, that is, where primitive operations conform to the types of the different symbols used, including meta-variables as well as rule parameters.

The syntax of reconfiguration guards and actions is as follows:

$$\begin{array}{rl} \text{guard:} & G_R ::= G_I \\ \text{action:} & A_R ::= x := \mathcal{B}.\mathsf{create}(B^t, q) \mid \mathcal{B}.\mathsf{delete}(x) \mid \\ & \quad z := \mathcal{H}.update_H(e_R, \cdots e_R) \mid \\ & \quad z := \mathcal{D}.update_D(e_R, \cdots e_R) \mid \\ & \quad z := e_R \mid A_R, A_R \\ \text{expression:} & e_R ::= z \mid \mathcal{B} \mid \mathcal{H} \mid \mathcal{D} \mid op(e_R, \cdots e_R) \end{array}$$

That is, guards are the same as for interaction rules and define constraints on components data. In the definition of reconfiguration actions, the symbol x denotes a rule parameter interpreted as component instance, and z denotes an arbitrary local rule parameter. The intuitive meaning of reconfiguration actions is as follows. The action $x := \mathcal{B}.\mathsf{create}(B^t, q)$ 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 $\mathcal{B}.\mathsf{delete}(x)$ 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 $z := \mathcal{H}.update_H(\ldots)$ denotes an update of the map according to a primitive operation $update_H$ from $\Omega(H^t)$, for H^t being the type of $\mathcal{H}$. Whenever an extra-value is returned by the primitive, it can be (optionally) assigned to the local parameter z. If no extra-value is returned, the assignment to z is omitted. Similarly, the action $z := \mathcal{D}.update_D(\ldots)$ denotes an update of the deployment according to a primitive operation $update_D$ from $\Omega(D^t)$, for D^t being the type of $\mathcal{D}$. Finally, the action $z := e_R$ denotes an update of a rule parameter z according to the expression e_R. Expressions are constructed from rule parameters z and meta-variables $\mathcal{B}, \mathcal{H}, \mathcal{D}$ using a set of predefined operations op, with given interpretation.

Formally, the semantics $[\![A_R]\!]$ of a reconfiguration action A_R is defined as a function[2] updating the motif configuration (B, H, D), the set of component configurations ($\mathbf{b}$) and the parameter interpretation (ζ):

$$\begin{array}{l} [\![x := \mathcal{B}.\mathsf{create}(B^t, q)]\!](B, H, D, \mathbf{b}, \zeta) = \\ \qquad (B \cup \{b\}, H, D', \mathbf{b}', \zeta') \\ \quad \text{where } b = (B^t, k)\ \textit{fresh},\ D' = D[b \mapsto \bot], \\ \qquad \mathbf{b}' = \mathbf{b}[b \mapsto q],\ \zeta' = \zeta[x \mapsto b] \\ [\![\mathcal{B}.\mathsf{delete}(x)]\!](B, H, D, \mathbf{b}, \zeta) = (B \backslash \{b\}, H, D_{|B\backslash\{b\}}, \mathbf{b}, \zeta) \\ \quad \text{where } b = \zeta(x) \in B \\ [\![z := \mathcal{H}.update_H(e_1, \cdots e_n)]\!](B, H, D, \mathbf{b}, \zeta) = \\ \qquad (B, H', D_{|H'}, \mathbf{b}, \zeta') \\ \quad \text{where } H', v' = H.update_H(\zeta(e_1), \cdots \zeta(e_n)), \\ \qquad \zeta' = \zeta[z \mapsto v'] \\ [\![z := \mathcal{D}.update_D(e_1, \cdots e_m)]\!](B, H, D, \mathbf{b}, \zeta) = \\ \qquad (B, H, D', \mathbf{b}, \zeta') \\ \quad \text{where } D', v' = D.update_D(\zeta(e_1), \cdots \zeta(e_m)), \\ \qquad \zeta' = \zeta[z \mapsto v'] \\ [\![z := e]\!](B, H, D, \mathbf{b}, \zeta) = (B, H, D, \mathbf{b}, \zeta[z \mapsto \zeta(e)]) \\ [\![A_{R1}, A_{R2}]\!](B, H, D, \mathbf{b}, \zeta) = \\ \quad ([\![A_{R2}]\!] \circ [\![A_{R1}]\!])(B, H, D, \mathbf{b}, \zeta) \end{array}$$

In the above, for an expression e we denoted by $\zeta(e)$ its valuation given the interpretation ζ of rule parameters and

[2] up to the choice of fresh component instance.

the implicit assignment of meta-variables ($\mathcal{B} \mapsto B, \mathcal{H} \mapsto H, \mathcal{D} \mapsto D$).

Example 10 (Reconfiguration rules) The ring motif type introduced in Example 8 contains three reconfiguration rules. The rule do-ring-init initializes the motif with a ring of two components. The rule do-ring-create creates a new component in the ring. The rule do-ring-remove($x : C$) removes an idle component x from the ring, provided it contains more than three components.

4.6 Operational semantics

The semantics of component composition within a motif involves two categories of atomic interleaved steps, namely interaction steps and reconfiguration steps. An interaction step corresponds to the execution of an interaction (as in BIP) from a set of interactions defined by the interaction rules. Reconfiguration steps correspond to the execution of a reconfiguration rule.

Formally, the operational 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 $\mathbf{b} = \langle b \mapsto q \mid b \in B, q \in \mathit{spsstates}(b)\rangle$,
- the labels of Σ correspond to valid interactions α constructed on components and an additional reconfiguration action label 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

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

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

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 (that is, according to the operational semantics in the static case presented earlier in Sect. 3) 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 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 Sect. 2 and illustrated in Fig. 5.

5 Motif-based systems

We consider systems defined as finite collection of motif instances, with predefined types, and sharing a finite set of component instances. 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. Henceforth, we consider a given finite set of motif types. As for components, a motif instance m is a couple (M^t, k) for some motif type M^t and $k \in \mathbb{N}$.

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 ($\mathcal{Z}^\star$, $\Psi^\star$, $G^\star$, $\mathcal{Z}_L^\star$, $A_R^\star$) 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 and/or set 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:

$$
\begin{array}{r}
M, \mathbf{m}, \zeta \models \Psi^{0*} \text{ iff } \zeta_{\mathbf{m}} \models \Psi^{0*} \\
M, \mathbf{m}, \zeta \models \langle y : \Psi\rangle \text{ iff } B, H, D, \zeta_{\mathbf{m}} \models \Psi \text{ where} \\
m \mapsto (B, H, D) \in \mathbf{m}, \zeta(y) = m \\
M, \mathbf{m}, \zeta \models \Psi_1^* \wedge \Psi_2^* \text{ iff } M, \mathbf{m}, \zeta \models \Psi_1^* \text{ and } M, \mathbf{m}, \zeta \models \Psi_2^* \\
M, \mathbf{m}, \zeta \models \neg\Psi^* \text{ iff } M, \mathbf{m}, \zeta \not\models \Psi^*
\end{array}
$$

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:

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

Inter-motif reconfiguration guards and actions are defined by the following grammar:

$$
\begin{array}{rl}
\text{guard: } G_R^\star & ::= G_I \\
\text{action: } A_R^\star & ::= y := \mathcal{M}.\mathsf{create}(M^t, (e_R^\star, e_R^\star, e_R^\star)) \mid \\
& \quad \mathcal{M}.\mathsf{delete}(y) \mid y.\mathcal{B}.\mathsf{migrate}(x) \mid \\
& \quad \langle y : A_R\rangle \mid z := e_R^\star \mid A_R^\star, A_R^\star \\
\text{expression: } e_R^* & ::= z \mid y.\mathcal{B} \mid y.\mathcal{H} \mid y.\mathcal{D} \mid op(e_R^\star, \cdots, e_R^\star)
\end{array}
$$

That is, guards are the same as for interaction rules. For inter-motif reconfiguration actions, we use the $\mathcal{M}$ symbol to refer the current set of existing motif instances. Also, the y symbol denotes a rule parameter interpreted as motif instance, and z a rule parameter of arbitrary type. The action $y := \mathcal{M}.\mathsf{create}(\, M^t, (e_B, e_H, e_D))$ 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 $\mathcal{M}.\mathsf{delete}(y)$ denotes the deletion of the motif instance y, that is, its removal from the set of motif instances. The action $y.\mathcal{B}.\mathsf{migrate}(x)$ 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 A_R to be executed in the context of the motif instance y and $z := e_R^\star$ an assignment of expression $e_R^\star$ to a local rule parameter z. As for intra-motif reconfiguration rules, expressions $e_R^\star$ are constructed from rule parameters z and meta-variables $\mathcal{B}$, $\mathcal{H}$, $\mathcal{D}$ associated with motif instances y, using a set of available primitives op.

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:

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

In the above, for an expression e we denoted by $\zeta_{\mathbf{m}}(e)$ its valuation in the extended context $\zeta_{\mathbf{m}}$.

Example 11 (Inter-motif reconfiguration rule) Consider an inter-motif reconfiguration rule for two ring motifs:

do-ring-merge(y_1, y_2 : Ring) ≡
  <u>when</u> $y_1.\mathcal{B} \cap y_2.\mathcal{B} = \emptyset$ and $|y_1.\mathcal{B}| + |y_2.\mathcal{B}| \le 10$
  <u>do</u> z_B := union($y_1.\mathcal{B}$, y_2.B),
    z_H := merge-cycle($y_1.\mathcal{H}$, $y_2.\mathcal{H}$),
    z_D := union($y_1.\mathcal{D}$, $y_2.\mathcal{D}$),
    $\mathcal{M}$.create(Ring, (z_B, z_H, z_D)),
    $\mathcal{M}$.delete(y_1), $\mathcal{M}$.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 $M = \{m_1, m_2, \ldots\}$ 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 $\mathbf{b} = \langle b \mapsto q \mid b \in B, q \in \mathit{spsstates}(b) \rangle$ are the component configurations,
- the set of labels Σ correspond to valid interactions α on component instances, a local reconfiguration action label ρ and an inter-motif reconfiguration action label 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

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

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

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

Rules (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.[3] 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.). The DR- BIP prototype can be retrieved from [39].

We have effectively used DR- BIP for programming reconfigurable systems in different application domains [21]. We provide tentative solutions using the DR- BIP formalism and evaluate their performance at executing dynamically changing configurations.

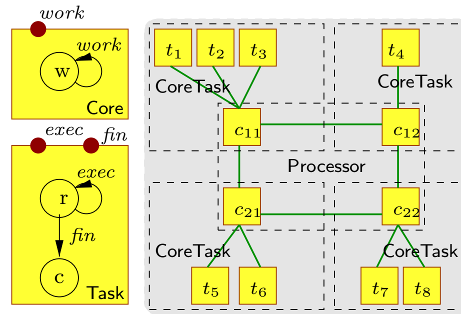

Fig. 8 Multicore task system

6.1 Dynamic multicore task system

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 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 than the load of the departing core minus some constant (K).

Figure 8 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

[3] maps are restricted to simple graphs, e.g., chain, cycle, star.

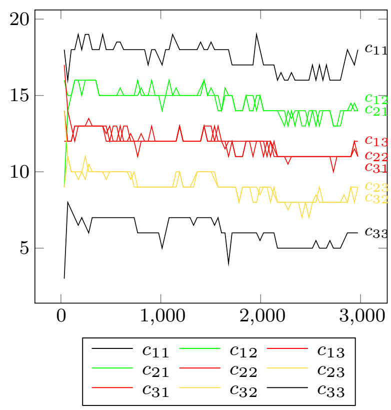

Fig. 9 Task load across 3000 steps

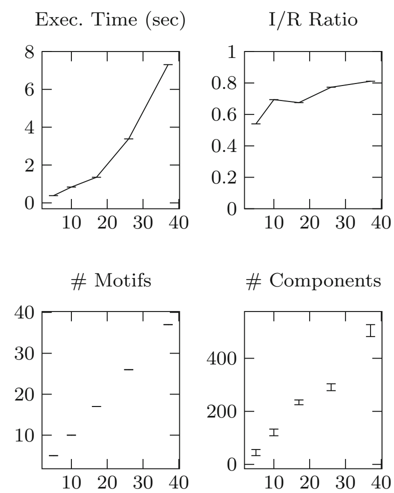

Fig. 10 Dynamic multicore task system measurements—the x-axis indicates the number of motifs in the initial configuration (i.e., $n^2 + 1$ for $n = 2, 3, 4, 5, 6$). The meaning of y-axis is indicated at the top

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 the task completion are represented by the rules:

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

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:

$$\text{do-migrate}(y_1, y_2 : \text{CoreTask}, y_3 : \text{Processor},$$
$$\qquad x_1, x_2 : \text{Core}, x_3 : \text{Task}) \equiv$$
$$\quad \underline{\text{when}}\ \langle\, y_1 : x_1 \in \mathcal{B}\,\rangle \text{ and } \langle\, y_2 : x_2 \in \mathcal{B}\,\rangle \text{ and}$$
$$\qquad \langle\, y_3 : \mathcal{D}(x_1) \mapsto \mathcal{D}(x_2)\,\rangle \text{ and}$$
$$\qquad |y_1.\mathcal{B}| \;>\; |y_2.\mathcal{B}| + \text{K and } x_3 \in y_1.\mathcal{B}$$
$$\quad \underline{\text{do}}\ y_2.\mathcal{B}.\text{migrate}(x_3), \langle y_1 : \mathcal{B}.\text{delete}(x_3)\,\rangle$$

To simplify notations in reconfiguration rules, we rely henceforth on sandwiching constraints/actions with angle brackets to specify the scope. For example, $\langle y_1 : x_1 \in \mathcal{B}\,\rangle$ is a constraint stating that x_1 is a component instance in motif y_1.

Figure 9 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, and 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. This highlights the task migration process cascading from the top left core to the bottom right core.

Figure 10 illustrates the evolution of the dynamic multicore task system for different initial configurations. We vary the number of cores in the processor from 4 to 36 cores. Each core is initialized with a random load as discussed above. The system initial size varies between 46 and 482 component instances as depicted in the figure. Each configuration is simulated for 1000 random steps. As the number of cores increases in size, the execution time increases reaching a maximum of 7.3 s. The motif instance count remains constant across each configuration; however, the component instance count varies as tasks are being created and deleted once completed. Also note that the average ratio of executed interactions versus reconfigurations is 0.7, since the task load converges to a similar value across cores and less task migrations (i.e., reconfigurations) are required.

6.2 Autonomous highway traffic system

This exercise is inspired from autonomous traffic systems for automated highways [6]. The system consists of a single-lane one-way road where an arbitrary number of autonomous homogeneous self-driving cars are moving in the same direction, at different cruising speeds. Cars are organized into platoons, i.e., groups of cars cruising at the same speed and closely following a leader car. Platoons may dynamically merge or split. A merge takes place if two platoons are close enough, i.e., the distance between the tail car of the first pla-

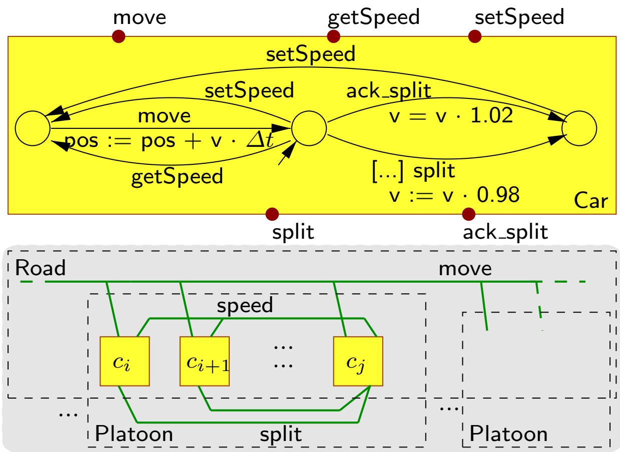

Fig. 11 Automated highway traffic system

toon and the leader car of the second is smaller than some constant K. After the merge, the speed of the new platoon is set to the speed of the first platoon. A platoon may split when an arbitrary car requests to leave the platoon, e.g., in order to perform some specific maneuver. After the split, the leading platoon will increase its speed by 2%, whereas the tail platoon will reduce its speed by 2%.

Figure 11 illustrates the motif-based system in DR- BIP. We use a component type Car to model the behavior of a car. Each car maintains its position pos and speed v. The position pos is updated on the move transition. Transitions setSpeed and ack_split are used by leader cars only to, respectively, define the platoon speed and acknowledge a platoon split. Similarly, transitions getSpeed and split are used by follower cars only to, respectively, synchronize on the leader speed and initiate a platoon split.

The Road motif type contains all cars without additional structuring. The Platoon motif type is structured as a chain of cars. The map of the platoon motif is a (dynamic) linear graph of locations, and the deployment assigns a single car to every position of the map. The Road motif defines a single interaction by the rule sync-road-move, which synchronizes the move ports of *all* cars and therefore performing a joint update of their positions. The Platoon motif defines several interactions by the rules sync-platoon-speed and sync-platoon-split. The first rule synchronizes the speed of the leading car with the speed of all follower cars. The second rule allows any follower car to initiate a split maneuver and become a leader in a newly created platoon.

sync-road-move(X : Car) $\equiv$
  <u>when</u> $X = \mathcal{B}$ <u>sync</u> X.move
sync-platoon-speed(x : Car, X : Car) $\equiv$
  <u>when</u> $X = \mathcal{B} \backslash x$ and $\mathcal{D}(x) = \text{head}(\mathcal{H})$
  <u>sync</u> x.setSpeed X.getSpeed
    true $\rightarrow X.v := x.v$

sync-platoon-split(x_1, x_2 : Car) $\equiv$
  <u>when</u> $\mathcal{D}(x_1) = \text{head}(\mathcal{H})$ and $x_1 \neq x_2$
  <u>sync</u> x_1.ack_split x_2.split

Two reconfiguration rules do-platoon-merge and do-platoon-split handle the merging and the splitting of platoons, respectively:

do-platoon-merge(y_1, y_2 : Platoon, x_1, x_2 : Car) $\equiv$
  <u>when</u> $\langle y_1 : \mathcal{D}(x_1) = \text{tail}(\mathcal{H}) \rangle$
    and $\langle y_2 : \mathcal{D}(x_2) = \text{head}(\mathcal{H}) \rangle$
  <u>do</u> abs(x_1.pos $-$ x_2.pos) $< K \rightarrow$
    $z_B :=$ union($y_1.\mathcal{B}$, $y_2.\mathcal{B}$),
    $z_H :=$ append($y_2.\mathcal{H}$, $y_1.\mathcal{H}$),
    $z_D :=$ union($y_1.\mathcal{D}$, $y_2.\mathcal{D}$),
    $\mathcal{M}$.create(Platoon, (z_B, z_H, z_D)),
    $\mathcal{M}$.delete(y_1), $\mathcal{M}$.delete(y_2)

do-platoon-split(y : Platoon, x : Car) $\equiv$
  <u>do</u> $\langle y : n := \mathcal{D}(x)$,
    $z_{H_1} :=$ sublist$_1$($\mathcal{H}, n$), $z_{D_1} :=$ extract($\mathcal{D}, z_{H_1}$),
    $z_{B_1} :=$ components(z_{D_1}),
    $z_{H_2} :=$ sublist$_2$($\mathcal{H}, n$), $z_{D_2} :=$ extract($\mathcal{D}, z_{H_2}$),
    $z_{B_2} :=$ components(z_{D_2}) $\rangle$,
    $\mathcal{M}$.create(Platoon, ($z_{B_1}, z_{H_1}, z_{D_1}$)),
    $\mathcal{M}$.create(Platoon, ($z_{B_2}, z_{H_2}, z_{D_2}$)) ,
    $\mathcal{M}$.delete(y)

Note that we use specific map primitives head, and tail which point, respectively, to the position of the leader and tail of a platoon, namely the beginning and the end of the list. Furthermore, we use the primitive append which appends and links two maps of type linked list together. Finally, primitives sublist$_{1,2}$ extract sublists from a linked list, respectively, ending before/starting at the node given as argument. The primitive extract computes a restricted deployment for component instances attached to a subset of nodes of the map.

Figure 12 illustrates the evolution of the system involving 200 cars along 2000 sampled steps. Each line describes a configuration of the system. We show 13 sampled non-consecutive configurations. A thin black rectangle represents a platoon. Its length is proportional to the number of cars contained. Its position in the line corresponds to its position on the road. For reference, we show the evolution of a particular car by highlighting it in yellow. Initially, all the cars belong to the same platoon. As the system evolves the initial platoon splits into several platoons, which then keep splitting/merging back, etc.

Figure 13 summarizes the execution of several initial configurations. We evaluate the performance and track the system evolution while varying the number of cars in the initial platoon from 200 to 600 cars. Each configuration is simulated for 3000 random steps. Notice that the component instance count remains constant across each configuration as

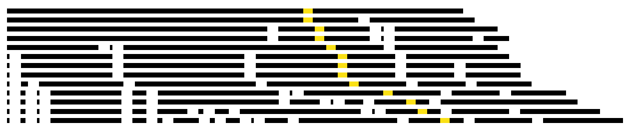

Fig. 12 Automated highway traffic evolution along few steps

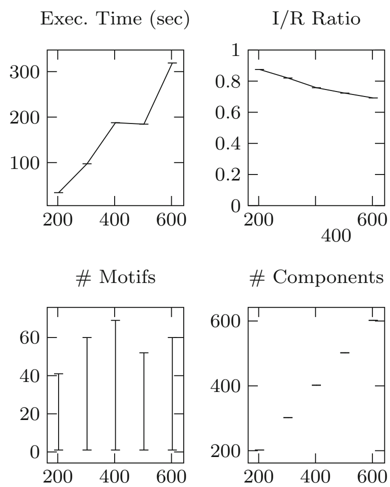

Fig. 13 Measurements on automated highway traffic systems

cars only rearrange within different platoons. However, the motif instance count varies as platoons merge/split. Finally, execution time increases reaching a maximum of 5 min and the average ratio of executed interactions versus reconfigurations is 0.77.

6.3 Self-organizing robot colonies

This exercise is inspired by swarm robotics [34]. A number of identical robots are randomly deployed on a field and have a mission to locate an object (the prey) and to bring it near another object (the nest). The robots know neither the position of the nest nor the position of the prey. They have limited communication and sensing capabilities, i.e., they can display a status (by turning on/off some colored leds) and can observe each other as long as they are physically close in the field. We consider hereafter the swarm algorithm proposed in [34]. In a first phase, the robots self-organize into an exploration path starting at the nest. The first robot detecting the nest initiates the path, i.e., stops moving and displays a specific (on-path) status. Any robot that detects (robots on) the path, begins moving along the path toward its tail, explores a bit further its neighborhood and gets connected as well (i.e., becomes the new tail, stops moving and displays the on-path status). Two cases may occur, either no new robot gets connected to the path within some delay, hence the tail robot disconnects and moves randomly (away from the path), or the tail robot detects the prey and the second phase starts. The path stays in place, while additional robots

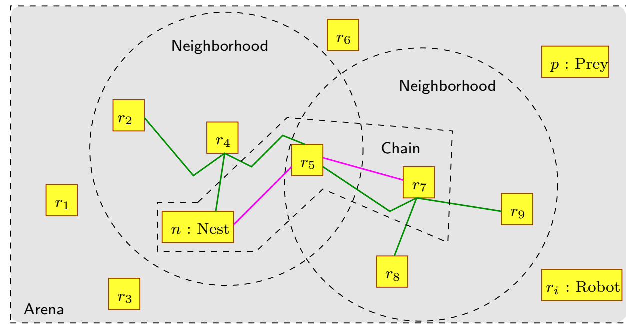

Fig. 14 Self-organizing robot colonies

converge near the prey. When enough robots have converged, they start pushing the prey along the path toward the nest. The path gets consumed, and the system will stop when the prey gets close enough to the nest.

We model the first phase of the algorithm above using three different types of components and three different types of motifs as illustrated in Fig. 14. The Arena motif contains all the robots, the nest and the prey component instances. No map and deployment are used as no specific architecture is enforced by this motif. This motif defines a global tick interaction used to model the synchronous passage of time within the system. Whenever the tick interaction is triggered, the robots update their positions, i.e., they move on the field.

For every robot, its Neighborhood motif is used to represent its visibility range, i.e., the set of robots physically close to it in the field. This motif uses a star-like location map. The inner robot is deployed at the center and the visible neighbors on the leaves. The motif defines a set of binary observe status interactions which are used by the inner robot to collect all the available information from its neighbors. Finally, the Chain motif represents the exploration chain linking robots to the nest. It uses a linear map to deploy the robots belonging to the chain. This motif defines a set of binary next prev interactions which are used to communicate along the chain.

For this example, reconfiguration is used to redefine the content of the Neighborhood and Chain motifs. For the former, as robots are moving in the field, they continuously enter or leave the visibility range of other robots. We use two inter-motif reconfiguration rules to update the neighborhood information:

do-neighborhood-enter(y_1 : Neighborhood, y_2 : Arena,
  x_1, x_2: Robot) $\equiv$
  <u>when</u> $\langle y_1 : \mathcal{D}(x_1) = \text{center}(\mathcal{H})$ and $\neg(x_2 \in \mathcal{B})\rangle$ and
    $\langle y_2 : x_2 \in \mathcal{B}\rangle$
  <u>do</u> dist(x_1.pos, x_2.pos) $\leq R_{min} \rightarrow$
    $y_1.\mathcal{B}$.migrate(x_2),
    $\langle y_1 : n := \mathcal{H}$.extend(), $\mathcal{D}$.attach(x_2,n) $\rangle$

do-neighborhood-leave(y_1 : Neighborhood,

x_1, x_2: Robot) $\equiv$
<u>when</u> $\langle y_1 : \mathcal{D}(x_1) =$ center($\mathcal{H}$) and $x_2 \in \mathcal{B}\rangle$ and
$x_1 \neq x_2$ and
<u>do</u> dist(x_1.pos, x_2.pos) $\geq R_{max} \rightarrow$
$\langle y_1 : n := \mathcal{D}(x_2)$, $\mathcal{B}$.delete(x_2), $\mathcal{H}$.remove(n) $\rangle$

The rules above describe the reconfiguration allowing any robot x_2 to enter (resp. leave) the neighborhood y_1 of any different robot x_1 whenever the distance between x_1 and x_2 is smaller than R_{min} (resp. greater than R_{max}). The evolution of the chain is also described by reconfiguration. At any time, the tail can disconnect or a robot can connect if it is close enough to the tail.

do-chain-connect(y_1 : Chain, y_2 : Neighborhood,
x_1, x_2 : Robot) $\equiv$
<u>when</u> $\langle y_1 : \mathcal{D}(x_1) =$ tail($\mathcal{H}$) and $x_2 \notin \mathcal{B}\rangle$ and
$\langle y_2 : \mathcal{D}(x_1) =$ center($\mathcal{H}$) and $x_2 \in \mathcal{B}\rangle$
<u>do</u> $y_1.\mathcal{B}$.migrate(x_2),
$\langle y_1 : n := \mathcal{H}$.extend(), $\mathcal{D}$.attach(x_2,n) $\rangle$

do-chain-disconnect(y_1 : Chain, x_1 : Robot) $\equiv$
<u>when</u> $\langle y_1 : \mathcal{D}(x_1) =$ tail($\mathcal{H}$) $\rangle$
<u>do</u> x_1.timeout $\rightarrow$
$\langle y_1 : n := \mathcal{D}(x_1)$, $\mathcal{B}$.delete(x_1), $\mathcal{H}$.remove(n) $\rangle$

6.4 Lessons learned

Although very preliminary, these experiments allowed us to draw some conclusions and identify potential lines for improvement:

- Arbitrarily complex interaction and/or architectural reconfiguration patterns are usually decomposable as a superposition of motifs, which, moreover, use relatively restricted forms of maps and addressing functions. No example required a map topology different than the ones mentioned so far (chain, cycle, star).
- While DR- BIP semantics leaves unspecified the choice of next rule to be executed between multiple interaction and reconfiguration rules, some control mechanism is needed to restrict non-determinism and enforce a desirable scenario. For instance, giving high priority to specific reconfiguration rules may enforce atomicity on a long reconfiguration sequence by avoiding interference with execution of interactions (e.g., for migrating a task to some final executing core). In contrast, giving higher priority to interaction rules may be useful when reconfiguration is triggered by external events and will take place only when the system reaches some stable state (e.g., for constraining the insertion of new tasks in the task system).
- The handling of time is very rudimentary. Actually, synchronous time progress is modeled using a multiparty interaction rule involving all timed components in a global motif. For example, all cars are synchronized for making their move action in the Road motif; similarly, all robots are synchronized in the Arena motif, etc. Whereas semantically correct, this representation is cumbersome and shall be improved by using clock variables like in real-time BIP [1] and an implicit semantics of time allowing to separate time-dependent system evolution from functional (interaction, reconfiguration) behavior.
- Going beyond toy examples would require a new implementation of DR- BIP concepts integrating a full-fledged representation of component types (e.g., as in BIP or real-time BIP) as well as richer types of maps and of addressing functions (e.g., defined as abstract data types in some implementation language). This is needed for building detailed models that could be used both for analysis with simulation-based techniques or for concrete implementation and deployment as part of real systems.

7 Related work

There exists a significant number of frameworks dealing with dynamic software and system architectures. We recommend [12,13] for exhaustive surveys and classification of existing approaches and [24,29] for an overview of current and foreseen design challenges. In this section, we restrict ourselves to formal frameworks dealing with an explicit notion of architecture in terms of components and connectors and providing primitives to express architectural reconfigurations. In particular, we do not consider general-purpose programming languages or domain-specific languages.

We distinguish between frameworks for *specification* or for *programming* architectural reconfiguration. In the first category, we include frameworks based on temporal logics such as [2,19], hybrid logics such as [36] or extended configuration logics such as DREAM [18]. These frameworks allow characterizing reconfiguration from the perspective of an external observer. Nonetheless, they do not provide support for implementation of reconfiguration within the system.

The DR- BIP framework is part of the second category dealing with explicit programming of reconfiguration within the system. Most of the reconfigurable ADL frameworks belong to this category. Usually, they can be classified according to the underlying formalism for programming reconfiguration and/or defining their operational semantics, e.g., based on process algebra such as π-ADL [15], MONTIARC [25], PILAR [35] and DARWIN [28]; using graph rewriting rules such as [26] and [37]; using chemical reaction rules such as CHAM [40]; and using specific rules such as GEREL [23], C2SADEL [32] and RAPIDE [27] to cite only a few. According to this classification, the reconfiguration rules of DR- BIP are a specific class of graph rewriting rules allowing to change the architecture seen as a hyper-graph of interconnected BIP components. Our originality lies in the use of maps and addressing functions to express reconfiguration constraints and to organize the different types of rewriting rules as reconfiguration and interaction rules.

The distinction between exogenous and endogenous reconfiguration is another criteria for the classification of existing approaches. Frameworks such as LEDA [14] are endogenous as they allow to freely use reconfiguration primitives, e.g., to create and remove components and connectors, as regular actions of components. π-ADL [15] allows for both endogenous (within components) and exogenous (within subsystems) reconfiguration. Nevertheless, most frameworks are exogenous and try to isolate as best as possible reconfiguration from applicative component behavior. For example, *mode-based* reconfiguration in MONTIARC [25], AADL- SLIM [16] or graph-rewriting rules in [26] are both examples of exogenous reconfiguration. It is also the case of DR- BIP where reconfiguration and interaction rules are kept fully separated from the behavior of components.

Finally, let us briefly discuss the positioning of DR- BIP in the BIP landscape. The BIP framework introduced in [4] and the big majority of its descendants including real-time BIP [1], distributed send/receive BIP [8], stochastic real-time BIP [33], etc., are restricted to static architectures, that is, with a fixed number of components and fixed connectors. The different variants consider specific models for components, semantics of time, specific forms of interaction, etc. The first extension toward dynamic reconfigurable systems has been DyBIP [9] allowing for changes on the connecting topology, whereas the set of components remain fixed. Later on, fully reconfigurable extensions have been studied in relation to specific implementations on some host languages. For example, the functional-BIP [20] allows for implementation in functional languages, whereas Java-BIP [31] has been developed to support concrete use of BIP concepts in relation to industrial Java-based software platforms. With DR- BIP, we try to re-unify these different dynamic variants behind a unique high-level framework, independent of target host languages and/or application domains to provide a common platform for analysis and implementation of dynamically reconfigurable systems.

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 parameterization 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 can 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 [10].

To verify the correctness of reconfiguration operations, a different approach can be 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 [30]. 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. Abdellatif, T., Combaz, J., Sifakis, J.: Model-based implementation of real-time applications. In: Carloni, L.P., Tripakis, S. (eds.) Proceedings of the 10th International conference on Embedded software, EMSOFT 2010, pp 229–238. ACM, New York (2010)
2. Aguirre, N., Maibaum, T.: A temporal logic approach to the specification of reconfigurable component-based systems. In: 17th IEEE International Conference on Automated Software Engineering, ASE 2002, pp. 271–274. IEEE, New York (2002)
3. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: 1st International Conference on Fundamental Approaches to Software Engineering, FASE'98. LNCS, vol. 1382, pp. 21–37. Springer, Berlin (1998)
4. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), pp. 3–12. IEEE Computer Society (2006)
5. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)
6. Bergenhem, C.: Approaches for facilities layer protocols for platooning. In: IEEE 18th International Conference on Intelligent Transportation Systems, ITSC 2015, pp. 1989–1994. IEEE, New York (2015)
7. Bliudze, S., Sifakis, J.: The algebra of connectors structuring interaction in BIP. IEEE Trans. Comput. **57**(10), 1315–1330 (2008)
8. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. Distrib. Comput. **25**(5), 383–409 (2012)
9. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling dynamic architectures using Dy-BIP. In: Software Composition—11th International Conference, SC 2012, LNCS, vol. 7306, pp. 1–16. Springer, Berlin (2012)
10. Bozga, M., Iosif, R., Sifakis, J.: Checking deadlock-freedom of parametric component-based systems. In: Tools and Algorithms for the Construction and Analysis of Systems—25th International Conference, TACAS 2019, LNCS, vol. 11428, pp. 3–20. Springer, Berlin (2019)
11. Bradbury, J.: Organizing definitions and formalisms for dynamic software architectures. Tech. Rep. 2004-477, Software Technology Laboratory, School of Computing, Queen's University (2004)
12. Bradbury, J., Cordy, J., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, pp. 28–33. ACM, New York (2004)
13. Butting, A., Heim, R., Kautz, O., Ringert, J.O., Rumpe, B., Wortmann, A.: A classification of dynamic reconfiguration in component and connector architecture description. In: Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp), CEUR-WS.org, CEUR Workshop Proceedings, vol. 2019, pp. 10–16 (2017)
14. Canal, C., Pimentel, E., Troya, J.M.: Specification and refinement of dynamic software architectures. In: Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), IFIP Conference Proceedings, vol. 140, pp. 107–126. Kluwer, Dordrecht (1999)
15. Cavalcante, E., Batista, T.V., Oquendo, F.: Supporting dynamic software architectures: from architectural description to implementation. In: Bass, L., Lago, P., Kruchten, P. (eds.) 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015, pp. 31–40. IEEE Computer Society (2015)
16. Cimatti, A., DeLong, R., Stojic, I., Tonetta, S.: Model-based run-time synthesis of architectural configurations for adaptive MILS systems. In: Romanovsky, A.B., Troubitsyna, E., Bitsch, F. (eds.) Computer Safety, Reliability, and Security—38th International Conference, SAFECOMP 2019, Lecture Notes in Computer Science, vol. 11698, pp. 200–215. Springer, Berlin (2019)
17. 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)
18. De Nicola, R., Maggi, A., Sifakis, J.: DReAM: dynamic reconfigurable architecture modeling. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems—8th International Symposium, ISoLA 2018, Lecture Notes in Computer Science, vol. 11246, pp. 13–31. Springer, Berlin (2018)
19. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Using temporal logic for dynamic reconfigurations of components. In: Barbosa, L.S., Lumpe, M. (eds.) Formal Aspects of Component Software—7th International Workshop, FACS 2010, Lecture Notes in Computer Science, vol. 6921, pp. 200–217. Springer, Berlin (2010)
20. Edelmann, R., Bliudze, S., Sifakis, J.: Functional BIP: embedding connectors in functional programming languages. J. Log. Algebr. Methods Program. **92**, 19–44 (2017)
21. 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. Distributed Systems—8th International Symposium, ISoLA 2018, LNCS, vol. 11246, pp. 304–320. Springer, Berlin (2018)
22. El Ballouli, R., Bensalem, S., Bozga, M., Sifakis, J.: Programming dynamic reconfigurable systems. In: Formal Aspects of Component Software—15th International Conference, FACS 2018, Proceedings, Lecture Notes in Computer Science, vol. 11222, pp. 118–136. Springer, Berlin (2018)
23. Endler, M., Wei, J.: Programming generic dynamic reconfigurations for distributed applications. In: International Workshop on Configurable Distributed Systems, 1992, pp. 68–79. IET (1992)
24. Garlan, D.: Software architecture: a travelogue. In: Proceedings of the on Future of Software Engineering, FOSE 2014, pp. 29–39. ACM, New York (2014)
25. Heim, R., Kautz, O., Ringert, J.O., Rumpe, B., Wortmann, A.: Retrofitting controlled dynamic reconfiguration into the architecture description language MontiArcAutomaton. In: Software Architecture—10th European Conference (ECSA'16) (2016)
26. Le Métayer, D.: Software architecture styles as graph grammars. In: In ACM SIGSOFT Software Engineering Notes, vol. 21, issue 6, pp. 15–23. ACM, New York (1996)
27. Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using Rapide. IEEE Trans. Softw. Eng. **21**(4), 336–354 (1995)
28. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: ACM SIGSOFT Software Engineering Notes, vol. 21, issue 6, pp. 3–14. ACM, New York (1996)
29. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**(6), 869–891 (2013)
30. Mavridou, A., Baranov, E., Bliudze, S., Sifakis, J.: Configuration logics: modeling architecture styles. J. Log. Algebr. Methods Program. **86**(1), 2–29 (2017)
31. Mavridou, A., Rutz, V., Bliudze, S.: Coordination of dynamic software components with JavaBIP. In: Formal Aspects of Component Software—14th International Conference, FACS 2017, LNCS, vol. 10487, pp. 39–57. Springer, Berlin (2017)
32. Medvidovic, N., Rosenblum, D.S., Taylor, R.N.: A language and environment for architecture-based software development and evolution. In: Boehm, B.W., Garlan, D., Kramer, J. (eds.) Proceedings of the 1999 International Conference on Software Engineering, ICSE'99, pp. 44–53. ACM, New York (1999)

33. Nouri, A., Mediouni, B.L., Bozga, M., Combaz, J., Bensalem, S., Legay, A.: Performance evaluation of stochastic real-time systems with the SBIP framework. IJCCBS **8**(3/4), 340–370 (2018)
34. Nouyan, S., Gross, R., Bonani, M., Mondada, F., Dorigo, M.: Teamwork in self-organized robot colonies. IEEE Trans. Evol. Comput. **13**(4), 695–711 (2009)
35. Quintero, C.E.C., de la Fuente, P., Barrio-Solórzano, M.: Dynamic coordination architecture through the use of reflection. In: Proceedings of the 2001 ACM Symposium on Applied Computing (SAC), pp. 134–140. ACM, New York (2001)
36. Sanchez, A., Madeira, A., Barbosa, L.S.: On the verification of architectural reconfigurations. Comput. Lang. Syst. Struct. **44**, 218–237 (2015)
37. Taentzer, G., Goedicke, M., Meyer, T.: Dynamic change management by distributed graph transformation: towards configurable distributed systems. In: International Workshop on Theory and Application of Graph Transformations, pp. 179–193. Springer, Berlin (1998)
38. Taivalsaari, A., Mikkonen, T., Systä, 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 2014, pp. 338–343. IEEE Computer Society (2014)
39. Verimag, DR-BIP Prototype (2018). https://www-verimag.imag.fr/~bozga/download/drbip.tgz
40. Wermelinger, M.: Towards a chemical model for software architecture reconfiguration. IEE Proc. Softw. **145**(5), 130–136 (1998)