Four Exercises in Programming Dynamic Reconfigurable Systems: Methodology and Solution in DR-BIP [*]

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

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

Abstract. DR-BIP is an extension of the BIP component framework intended for programming reconfigurable systems encompassing various aspects of dynamism. A system is built from instances of types of components characterized by their interfaces. The latter consist of sets of ports through which data can be exchanged when interactions take place. DR-BIP allows the description of parametric exogenous interactions and reconfiguration operations. To naturally model self-organization and mobility of components, a system is composed of several architecture motifs, each motif consisting of a set of component instances and coordination rules. The use of motifs allows a disciplined management of dynamically changing coordination rules. The paper illustrates the basic concepts of DR-BIP through a collection of four non-trivial exercises from different application areas: fault-tolerant systems, mobile systems and autonomous systems. The presented solutions show that DR-BIP is both minimal and expressive allowing concise and natural description of non-trivial systems.

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 faults or threats, and to provide the adequate functionality when the external environment changes dynamically. 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 behav-

[***] Institute of Engineering Univ. Grenoble Alpes

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

ior and its coordination. Coordination is *exogenous* in the form of an architecture that describes global coordination rules between the coordinated components. This approach is adopted by numerous Architecture Description Languages (ADL) (see [8] for a survey). The other approach is based on *endogenous* coordination by explicitly using primitives in the code describing the behavior of components. Most programming models use internalized coordination mechanisms. Components usually have interfaces that specify their capabilities to coordinate with other components. Composing components boils down to composing interfaces. This approach is usually adopted with formalisms based on process calculi, such as [1, 10–12].

The obvious advantage of endogenous coordination is that programmers do not have to explicitly build a global coordination model. Consequently, the absence of such a model makes the validation of coordination mechanisms and the study of their underlying properties much harder. Exogenous coordination is advocated for enabling the study of the coordination mechanisms and their properties. It motivated the development of 100+ ADLs [16].

There exists a huge literature on architecture modeling reviewed in detailed surveys classifying the various approaches and outlining new trends and needs [8, 9, 15–17, 19, 22]. However, there is currently no clear understanding about how different aspects of architecture dynamism can be captured. We consider that the degree of dynamism of a system can be characterized as the interplay of dynamic change in three independent aspects.

- The first aspect requires the ability to describe parametric system coordination for arbitrary number of instances of 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.
- The third aspect is currently the most challenging. It meets in particular, the vision of “fluid architectures” or “fluid software” [22] which entails a virtual computing experience allowing 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 migration of components allows a disciplined management of dynamically changing coordination rules. For instance, self-organizing systems may adopt different motifs to adapt their behavior to meet a global property.

The paper proposes the *Dynamic Reconfigurable* BIP (DR-BIP) framework, which encompasses all these three aspects of dynamism. DR-BIP is an extension of BIP[3, 4] —a framework encompassing rigorous design captured as the interplay of behavior, interaction and priorities for static systems —and Dy-BIP [7] —a former extension for handling dynamic interactions. DR-BIP follows an exogenous approach respecting the strict separation between behavior and architecture. It directly embraces multiparty interaction [6]. It characterizes a dynamic architecture as a set of interaction rules implemented by connectors and a set of configuration rules. Although it does not allow ad hoc dynamism, it directly covers all kinds of dynamism at runtime [8]: programmed dynamism, adaptive dynamism, and self-organizing dynamism. It provides support for component/motif creation and removal at runtime. In addition, it directly supports component migration from one motif to another. It supports both programmed and triggered reconfiguration as defined in [9]. The big advantage of using motifs is that when a component joins a motif, its interactions with other components are dictated by both its behavior and the interaction rules in its new motif. So, a motif is a "world" where components live and from which they can migrate to join other "worlds" [22]. DR-BIP shares the same conceptual framework with DReAM [13], which uses an extension of interaction logic with data transfer and reconfiguration. The main difference with DR-BIP is the possibility to express coordination as a conjunction of constraints.

The paper is organized as follows. Section 2 provides a brief overview of the key DR-BIP concepts, namely architectural motifs and motifs-based systems. Section 3 presents DR-BIP models and execution results for use case systems exhibiting different degrees of dynamism. Finally, section 4 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 modeling 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.

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.

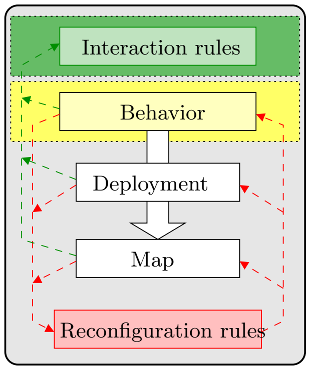

Fig. 1: Motif Concept

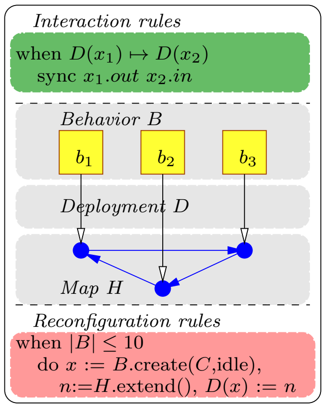

Fig. 2: Motif Example

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. 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-args*) ≡
    [**when** *rule-constraint*]
    sync *interaction-ports*
      [*interaction-guard* →
        interaction-action$^+$]

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

Both sets of rules are interpreted on the current motif configuration. *Formal-args* 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 (*interaction-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 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.

Fig. 2 illustrates the proposed motif concept for describing a dynamic ring architecture. Three components b_1, b_2, b_3 are deployed into a three-position circular map. Given the deployment function D, the interaction rule reads as follows: for components x_1, x_2 deployed on adjacent nodes $D(x_1) \mapsto D(x_2)$

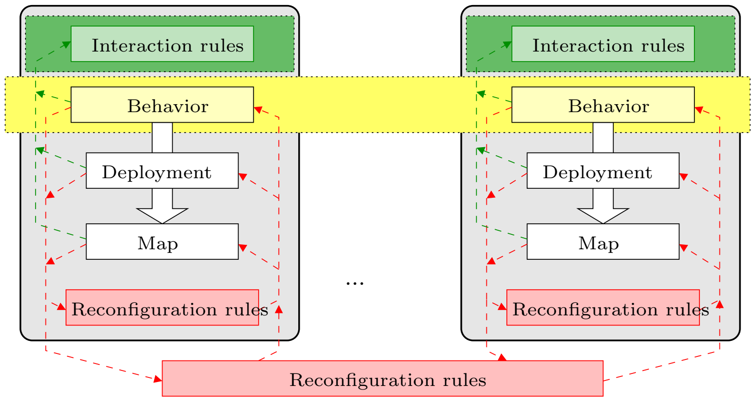

Fig. 3: Motif-based System Concept

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 allows to extend the ring by adding one more component. The rule is applicable as long as the number of component instances $|B|$ is less than 10. When executed, a new component x is created with initial state *idle* ($x := create(C, idle)$), a new node n is added to the circular map H ($n := H.extend()$) and the component x is deployed on the node n ($D(x) := n$).

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 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

[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.

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.

The inter-motif reconfiguration 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.

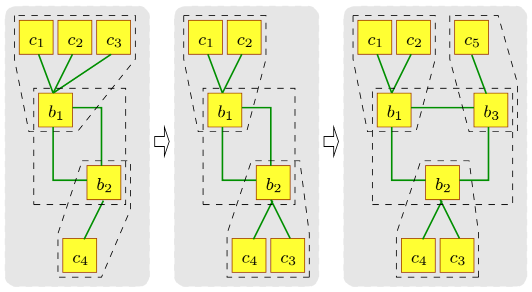

Fig. 4: An example : system reconfigurations

Fig. 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

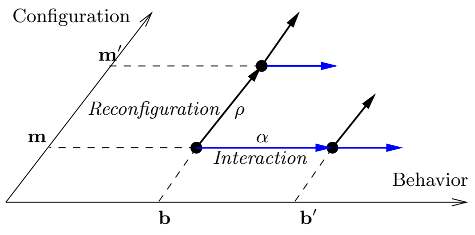

Fig. 5: Reconfiguration vs Interaction Steps

The behavior 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.

Our prototype implementation of DR-BIP includes a concrete language to describe motif-based systems and an interpreter (implemented in JAVA) for the operational semantics. The language provides syntactic constructs for describing component and motif types, with some restrictions on the maps and deployments allowed[2]. The interpreter allows the computation of enabled interactions and (inter-motif) reconfiguration rules on system configurations, and their execution according to predefined scheduling policies (interactive, random, etc).

3 Four Exercises

We present hereafter four exercises for programming dynamic reconfigurable systems. We provide tentative solutions using the DR-BIP formalism and evaluate their performance at executing dynamically changing configurations.

3.1 Dynamic Token Ring System

A *token ring* consists of two or more identical components interconnected using uni-directional 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 leaving at least two components in the ring. and 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.

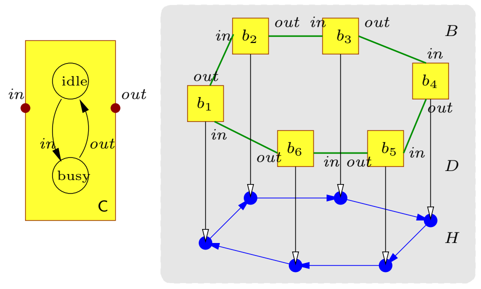

Fig. 6: Dynamic Token Ring

The behavior of component instances and the structure of the ring motif are graphically illustrated in Fig. 6. The map H is a ring of locations, i.e. an instance of a circular linked list type. The deployment D assigns components to locations in a bijective manner.

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

Interactions are defined by the rule *sync-ring-inout*$(x_1, x_2 : \mathrm{C})$, which connects the *out* port of a component x_1 to the *in* port of the component x_2 deployed next to it on the map. The motif reconfiguration is defined by two rules. The rule *do-ring-insert* creates a new component in the ring. The rule *do-ring-remove*$(x : \mathrm{C})$ removes an idle component x from the ring, provided it contains more than 2 components. Finally, the inter-motif reconfiguration rule *do-ring-merge* merges two ring instances y_1, y_2 into a single ring, whenever their sets of component instances are disjoint and together do not exceed 10.

```
sync-ring-inout($x_1, x_2$ : C) ≡ when D($x_1$) ↦ D($x_2$)
    sync $x_1$.out $x_2$.in
do-ring-insert() ≡ do $x$ := B.create(C, idle), $n$ := H.extend(), D($x$) := $n$
do-ring-remove($x$ : C) ≡ when |B| ≥ 3 ∧ $x$.idle
    do $n$ := D($x$), B.delete($x$), H.remove($n$)
do-ring-merge($y_1$, $y_2$ : Ring) ≡ when $y_1$.B ∩ $y_2$.B = ∅  and  |$y_1$.B| + |$y_2$.B| ≤ 10
    do B = $y_1$.B ∪ $y_2$.B, D = $y_1$.D ∪ $y_2$.D, H = merge-cycle($y_1$.H, $y_2$.H),
        create(Ring, (B, H, D)), delete($y_1$), delete($y_2$)
```

Note that we use specific map primitives init, extend, remove, merge-cycle to respectively initialize, extend by one new location, remove one location and merge two cyclic maps. The map predicate $\cdot \mapsto \cdot$ denotes the connection relation between locations.

Fig. 7 illustrates the execution of a dynamic ring system initialized with 10 ring motifs, each having 2 component instances. At each step, either an interaction or a reconfiguration (either within a motif or an inter-motif reconfiguration) is randomly executed. We remark that the number of ring motif instances decreases along the execution as idle components are removed and rings are enabled to merge into a single ring. The number of component instances varies across the execution between 6 and 20 as the *do-ring-insert* and *do-ring-remove* reconfiguration rules are executed.

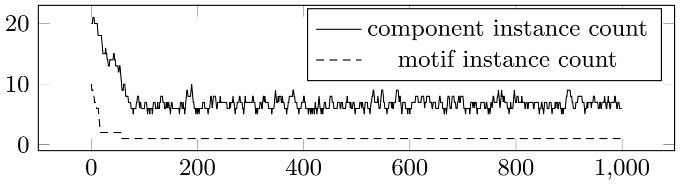

Fig. 7: Dynamic ring system evolution across 1,000 steps

Fig. 8 summarizes the execution of the dynamic ring system for different initial configurations. We evaluate the performance and track the system evolution while varying the number of initial rings from 10 to 100. Each configuration is simulated for 1000 random steps. As the system grows in size and the computation of enabled interactions and reconfigurations gets more complex, the execution time increases reaching a maximum of 14 seconds (first plot). The average ratio of the number of executed interactions vs reconfigurations along the run is around 0.45 (second plot). Finally, the minimum and maximum number of motif and component instances are depicted in the third and fourth plots.

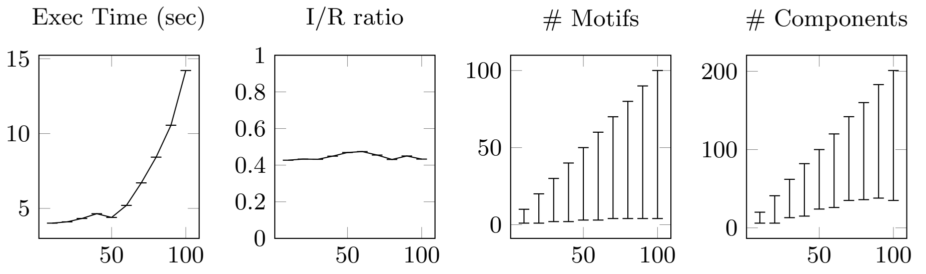

Fig. 8: Dynamic token ring system measurements - the x-axis indicates the number of rings in the initial configuration. The meaning of y-axis is indicated at the top

3.2 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).

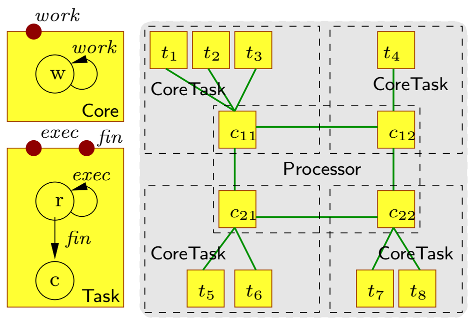

Fig. 9: Multicore Task System

Fig. 9 presents the overall structure of the motif-based system for four cores. We distinguish two types of atomic components, namely Task and Core. Multiple cores are interconnected together in a motif of type Processor. The interconnecting topology reflects the platform architecture (e.g., a 2×2 grid in the figure) and is enforced using a similar grid-like map and deployment. An additional CoreTask motif type is used to represent every core with its assigned tasks.

The interactions in the system are defined within the CoreTask motif. The execution of a task by the core and the task completion are represented by the rules:

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

```
do-migrate($y_1, y_2$ : CoreTask, $y_3$ : Processor, $x_1, x_2$ : Core, $x_3$ : Task) $\equiv$
    when $\langle\ y_1 : x_1 \in \mathsf{B}\ \rangle \wedge \langle\ y_2 : x_2 \in \mathsf{B}\ \rangle \wedge \langle\ y_3 : \mathsf{D}(x_1) \mapsto \mathsf{D}(x_2)\ \rangle \wedge$
            $|y_1.\mathsf{B}|\ >\ |y_2.\mathsf{B}| + \mathsf{K} \wedge x_3 \in y_1.\mathsf{B}$
    do $y_2$.migrate($x_3$), $y_1$.delete($x_3$)
```

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

Fig. 10 illustrates the execution of the dynamic multicore task system with 3×3 cores for 3000 steps. Each core is initialized with a random load between 1 and 20. The constant K is set to 3, hence tasks are allowed to migrate to neighboring cores (left, right, top or bottom) that differ in task load by at least 3 tasks. The cores c_{11}, and c_{33} are used to respectively create new tasks and withdraw completed tasks. These two cores retain the maximum and minimum load. As tasks migrate, the task load of cores converges and balances along the execution having at most a difference of 3 tasks between neighboring cores. For example, in core c_{21} the task load increased from 6 to 14. As expected the cores (c_{21}, and c_{12}) closest to c_{11} maintain a high load and as we move away from c_{11} the core's load gradually decreases. This highlights the task migration process cascading from the top left core to the bottom right core.

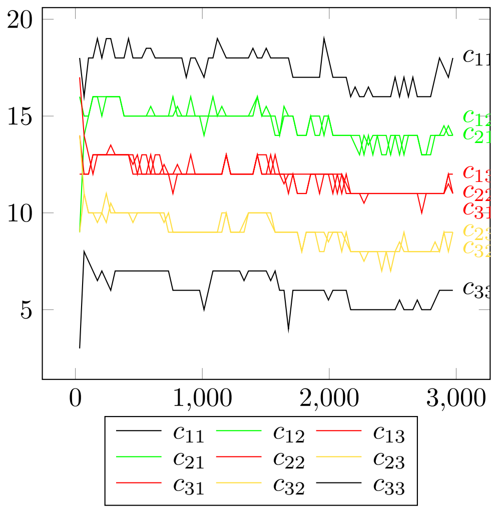

Fig. 10: Task load across 3000 steps

Fig. 11 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 seconds. 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 vs reconfigurations is 0.7, since the task load converges to a similar value across cores and less task migrations (i.e. reconfigurations) are required.

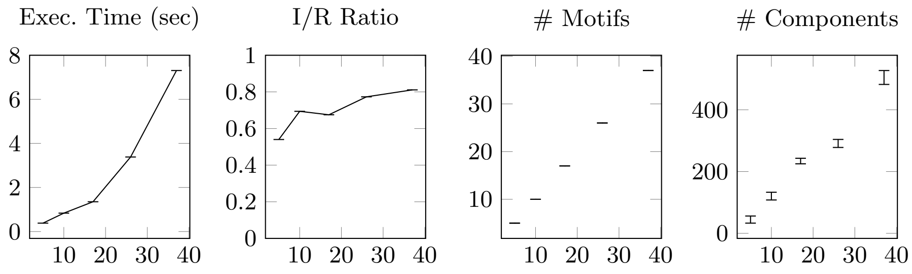

Fig. 11: Dynamic multicore task system measurements - the x-axis indicates the number of motifs in the initial configuration (i.e. $n^2 + 1$ for $n = 2, 3, 4, 5, 6$). The meaning of y-axis is indicated at the top

3.3 Autonomous Highway Traffic System

This exercise is inspired from autonomous traffic systems for automated highways [5]. 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 platoon 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%.

Fig. 12 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.

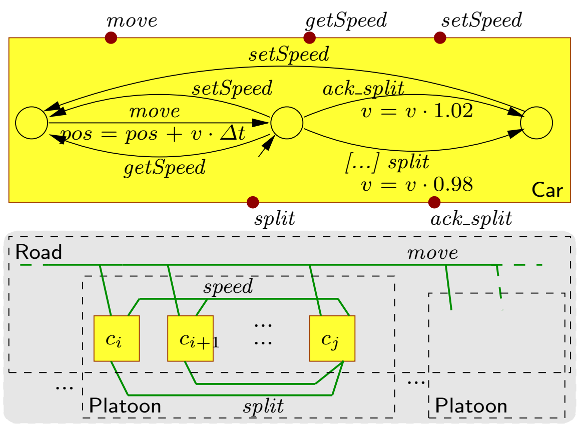

Fig. 12: Automated Highway Traffic System

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.

$\mathsf{sync\text{-}road\text{-}move}(X : \mathsf{Car}) \equiv \underline{\mathsf{when}}\ X{=}\mathsf{B}\ \underline{\mathsf{sync}}\ X.\mathsf{move}$
$\mathsf{sync\text{-}platoon\text{-}speed}(x : \mathsf{Car},\ X : \mathsf{Car}) \equiv \underline{\mathsf{when}}\ X{=}\mathsf{B} \setminus x \wedge \mathsf{D}(x) = \mathsf{H.head}$
  $\underline{\mathsf{sync}}\ x.\mathsf{setSpeed}\ X.\mathsf{getSpeed}\ \underline{\mathsf{do}}\ X.v = x.v$
$\mathsf{sync\text{-}platoon\text{-}split}(x_1,\ x_2 : \mathsf{Car}) \equiv \underline{\mathsf{when}}\ \mathsf{D}(x_1) = \mathsf{H.head} \wedge x_1 \neq x_2$
  $\underline{\mathsf{sync}}\ x_1.\mathsf{ack\_split}\ x_2.\mathsf{split}$

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

$\mathsf{do\text{-}platoon\text{-}merge}(y_1,\ y_2 : \mathsf{Platoon},\ x_1,\ x_2 : \mathsf{Car}) \equiv$
  $\underline{\mathsf{when}}\ \langle y_1 : \mathsf{D}(x_1) = \mathsf{H.tail}\rangle \wedge \langle y_2 : \mathsf{D}(x_2) = \mathsf{H.head}\rangle \wedge |x_1.\mathsf{pos} - x_2.\mathsf{pos}| < K$
  $\underline{\mathsf{do}}\ \mathsf{B} := y_1.\mathsf{B} \cup y_2.\mathsf{B},\ \mathsf{H} := \mathsf{append}(y_2.\mathsf{H},\ y_1.\mathsf{H}),\ \mathsf{D} := y_1.\mathsf{D} \cup y_2.\mathsf{D},$
    $\mathsf{create}(\mathsf{P},\ (\mathsf{B},\ \mathsf{H},\ \mathsf{D})),\ \mathsf{delete}(y_1),\ \mathsf{delete}(y_2)$
$\mathsf{do\text{-}platoon\text{-}split}(y : \mathsf{Platoon},\ x : \mathsf{Car}) \equiv$
  $\underline{\mathsf{do}}\ \langle y : \mathsf{H}_1 := \mathsf{H.sublist}(0,\ \mathsf{D}(x)),\ \mathsf{B}_1 := \mathsf{D}^{-1}(\mathsf{H}_1),\ \mathsf{D}_1 := \mathsf{D.restrict}(\mathsf{H}_1),$
    $\mathsf{H}_2 := \mathsf{H.sublist}(\mathsf{D}(x),\ \mathsf{H.length}),\ \mathsf{B}_2 := \mathsf{D}^{-1}(\mathsf{H}_2),\ \mathsf{D}_2 := \mathsf{D.restrict}(\mathsf{H}_2)\ \rangle,$
    $\mathsf{create}(P,\ (\mathsf{B}_1,\ \mathsf{H}_1,\ \mathsf{D}_1)),\ \mathsf{create}(P,\ (\mathsf{B}_2,\ \mathsf{H}_2,\ \mathsf{D}_2)),\ \mathsf{delete}(y)$

Note that we use specific map primitives head, and tail which point to the position of the leader and tail of a platoon, namely the beginning and the end of the list. Furthermore, we use the primitive append which appends and links two maps of type linked list together. Finally, the primitive sublist and length creates a sublist from a linked list and returns the length of the list respectively. The primitive restrict restricts a deployment keeping only the deployment mappings of components in a given map and removes the rest.

Fig. 13 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 nonconsecutive 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.

Fig. 14 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 cars only rearrange within different platoons.

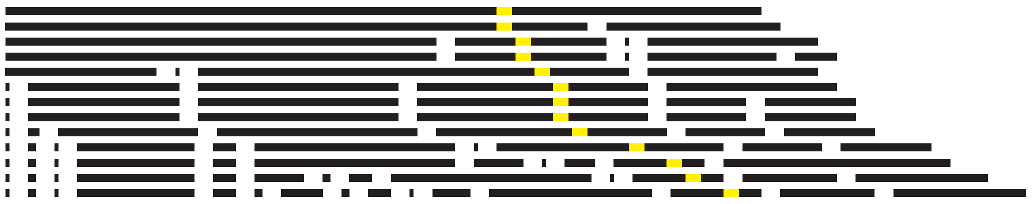

Fig. 13: Automated highway traffic evolution along few steps

However the motif instance count varies as platoons merge/split. Finally, execution time increases reaching a maximum of 5 minutes and the average ratio of executed interactions vs reconfigurations is 0.77.

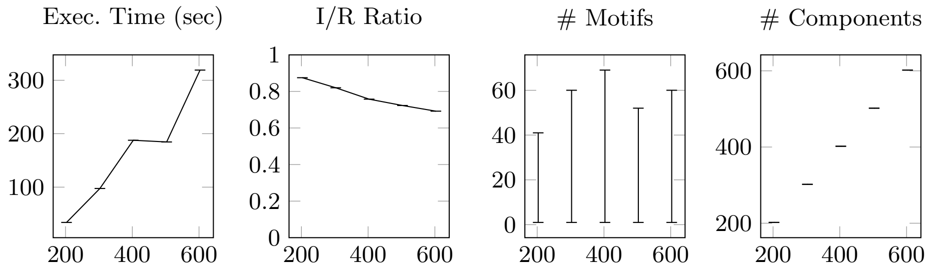

Fig. 14: Measurements on automated highway traffic systems

3.4 Self-Organizing Robot Colonies

This exercise is inspired by swarm robotics [18]. 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 [18]. 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 towards 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 converge near the prey. When enough robots have converged, they start pushing the prey along the path towards 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. 15. 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.

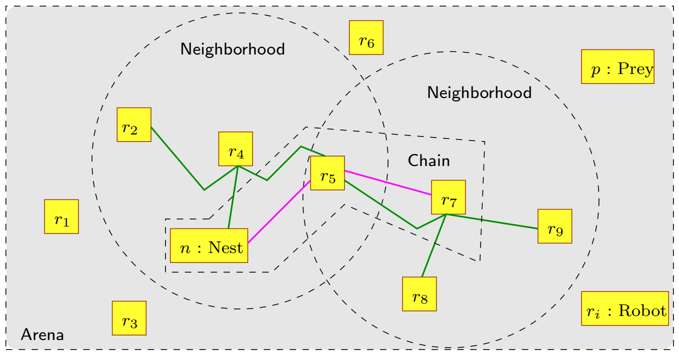

Fig. 15: Self-organizing robot colonies

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 Neigborhood 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) $\equiv$
    when $\langle y_1$ : D($x_1$) = H.center $\wedge$ $x_2 \notin$ B$\rangle$ $\wedge$ $\langle y_2 : x_2 \in$ B $\rangle$ $\wedge$ dist($x_1$, $x_2$) $\leq R_{min}$
        do $y_1$.migrate($x_2$), $\langle y_1 : n$ := H.extend(), D($x_2$) := $n$ $\rangle$
do-neighborhood-leave($y_1$ : Neighborhood, $x_1, x_2$: Robot) $\equiv$
    when $\langle y_1$ : D($x_1$) = H.center $\wedge$ $x_2 \in$ B$\rangle$ $\wedge$ $x_1 \neq x_2$ $\wedge$ dist($x_1$, $x_2$) $\geq R_{max}$
        do $\langle y_1 : n$ := D($x_2$), B.delete($x_2$), 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 its close enough to the tail.

```
do-chain-connect($y_1$ : Chain, $y_2$ : Neighborhood, $x_1, x_2$ : Robot) $\equiv$
    when $\langle y_1$ : D($x_1$) = H.tail $\wedge$ $x_2 \notin$ B $\rangle$ $\wedge$ $\langle y_2$ : D($x_1$) = H.center $\wedge$ $x_2 \in$ B$\rangle$
        do $y_1$.migrate($x_2$), $\langle y_1 : n$ = H.extend(), D($x_2$) := $n$ $\rangle$
do-chain-disconnect($y_1$ : Chain, $x_1$ : Robot) $\equiv$
    when $\langle y_1$ : D($x_1$) = H.tail $\rangle$ $\wedge$ $\langle y_1 : x_1.timeout$ = true $\rangle$
        do $\langle y_1 : n$ := D($x_1$), B.delete($x_1$), H.remove($n$) $\rangle$
```

4 Discussion

The paper presents the DR-BIP framework as well as its basic structuring constructs and their application to programming real-life systems. We show that the proposed framework is minimal and expressive allowing concise modeling. This is achieved by a methodology supporting incremental description through strict separation of concerns. Describing a system as a superposition of motifs allows enhanced flexibility and abstraction. Each motif is a specific dynamic architecture with its own coordination rules. So membership in a motif determines the way a component interacts with other components and the reconfiguration rules it is subject to. This is achieved in particular through maps which are reference structures used to naturally express mobility and dynamically changing environments.

DR-BIP has been designed with autonomy in mind. The examples on Autonomous highway traffic system and Self-organizing robot colonies demonstrate the power of its structuring concepts. Designing systems as a superposition of motifs (architectures) with their own coordination rules tremendously simplifies the description of autonomous behavior. At the conceptual level, motifs correspond to “modes whose behavioral content may change through component migration and can also be transformed by using higher level coordination rules.

To the best of our knowledge, there is no exogenous coordination language such as an ADL addressing all these modeling issues in such a methodologically rigorous manner. DR-BIP has some similarities with simulation and programming frameworks for autonomous mobile systems which nonetheless adopt significant domain-specific restrictions such as Buzz [20, 21].

Future work aims at showing that DR-BIP is expressive enough to directly encompass various coordination mechanisms, in particular unifying the modeling of distributed actor-based systems and thread-based shared memory systems. This can be achieved by considering threads as a special type of mobile components using maps as a shared memory structure. In addition, we aim to study parametric verification techniques for specific types of architectures (motifs) and combine them with correct-by-construction techniques based on the composition of architectures [2]. A formal definition of the DR-BIP is provided in report [14].

References

1. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: International Conference on Fundamental Approaches to Software Engineering. pp. 21–37. Springer (1998)
2. Attie, P., Baranov, E., Bliudze, S., Jaber, M., Sifakis, J.: A general framework for architecture composability. Formal Aspects of Computing 28(2), 207–231 (2016)
3. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)
4. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time systems in BIP. In: SEFM’06 Proceedings. pp. 3–12. IEEE Computer Society Press (2006)

5. Bergenhem, C.: Approaches for facilities layer protocols for platooning. In: Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on. pp. 1989–1994. IEEE (2015)
6. Bliudze, S., Sifakis, J.: The algebra of connectors structuring interaction in bip. IEEE Transactions on Computers 57(10), 1315–1330 (2008)
7. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling dynamic architectures using Dy-BIP. In: International Conference on Software Composition. pp. 1–16. Springer (2012)
8. Bradbury, J.: Organizing definitions and formalisms for dynamic software architectures. Tech. Rep. 2004-477, Software Technology Laboratory, School of Computing, Queen's University (2004)
9. Butting, A., Heim, R., Kautz, O., Ringert, J.O., Rumpe, B., Wortmann, A.: A classification of dynamic reconfiguration in component and connector architecture description languages. In: 4th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp'17) (2017)
10. Canal, C., Pimentel, E., Troya, J.: Specification and refinement of dynamic software architectures. In: Software Architecture, pp. 107–125. Springer (1999)
11. Cuesta, C., de la Fuente, P., Barrio-Solárzano, M.: Dynamic coordination architecture through the use of reflection. In: Proceedings of the 2001 ACM symposium on Applied computing. pp. 134–140. ACM (2001)
12. 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)
13. De Nicola, R., Maggi, A., Sifakis, J.: Dream: Dynamic reconfigurable architecture modeling (full paper). arXiv preprint arXiv:1805.03724 (2018)
14. El Ballouli, R., Bensalem, S., Bozga, M., Sifakis, J.: Dr-bip - programming dynamic reconfigurable systems. Tech. Rep. TR-2018-3, Verimag Research Report
15. Garlan, D.: Software architecture: A travelogue. In: Future of Software Engineering (FOSE'14). pp. 29–39. ACM (2014)
16. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: A survey. IEEE Transactions on Software Engineering 39(6) (June 2006)
17. Medvidovic, N., Dashofy, E., Taylor, R.: Moving architectural description from under the technology lamppost. Information and Software Technology 49(1), 12–31 (2007)
18. Nouyan, S., Gross, R., Bonani, M., Mondada, F., Dorigo, M.: Teamwork in self-organized robot colonies. IEEE Transactions on Evolutionary Computation 13(4) (August 2009)
19. Oreizy, P.: Issues in modeling and analyzing dynamic software architectures. In: International Workshop on the Role of Software Architecture in Testing and Analysis. pp. 54–57 (1998)
20. Pinciroli, C., Beltrame, G.: Buzz: An extensible programming language for heterogeneous swarm robotics. In: Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on. pp. 3794–3800. IEEE (2016)
21. Pinciroli, C., Lee-Brown, A., Beltrame, G.: Buzz: An extensible programming language for self-organizing heterogeneous robot swarms. arXiv preprint arXiv:1507.05946 (2015)
22. Taivalsaari, A., Mikkonen, T., Syst, K.: Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In: IEEE 38th Annual Computer Software and Applications Conference (COMPSAC'14) (2014)