A Layered Implementation of DR-BIP Supporting Run-Time Monitoring and Analysis

Antoine El-Hokayem, Saddek Bensalem, Marius Bozga, and Joseph Sifakis

Univ. Grenoble Alpes, CNRS, Grenoble INP*, Verimag

Abstract. Reconfigurable systems are emerging in many application domains as reconfiguration can be used to cope with unpredictable system environments and adapt by delivering new functionality. The Dynamic Reconfigurable BIP (DR-BIP) framework is an extension of the BIP component framework enriched with dynamic exogenous reconfiguration primitives, intended to support rigorous modeling of reconfigurable systems. We present a new two-layered implementation of DR-BIP clearly separating between execution of reconfiguration operations and execution of a fixed system configuration. Such a separation of concerns offers the advantage of using the mature and efficient BIP engine as well as existing associated analysis and verification tools. Another direct benefit of the new implementation is the possibility to monitor a holistic view of a system's behavior captured as a set of traces involving information about both the state of the system components and the dynamically changing architecture. Monitoring and analyzing such traces poses interesting questions regarding the formalization and runtime verification of properties of reconfigurable systems.

1 Introduction

The current trend for adaptive and resilient systems changes the perspective of system designers who have to consider systems that are reconfigurable and self-organizing. This requires conceptual models to better understand and properly take into account the different types of dynamism and corresponding coordination mechanisms for their description. In particular, it is desirable to have a rigorous and disciplined approach that would allow envisioning how a system with static coordination structure can be progressively modified to enhance its adaptivity and resilience.

Consider a platoon system of an automated highway where an arbitrary number of autonomous cars are moving in the same direction, in a single lane, at different cruising speeds. Cars dynamically organize into platoons, i.e., groups of cars cruising at the same speed and closely following a leader car. Platooning confers advantages such as increasing the road capacities, providing a more steady-state traffic flow, and reducing the risk of traffic congestion [4]. Organizing as platoons requires non-trivial dynamic coordination between cars. All the cars belonging to a platoon must agree on a common

* Institute of Engineering Univ. Grenoble Alpes

The research performed by these authors was partially funded by H2020-ECSEL grants CPS4EU 2018-IA call - Grant Agreement number 826276.

cruising speed dictated by the leader car. Platoons may dynamically merge or split. A merge can take 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 reaching some minimal distance K. After the merge, the speed of the new platoon is updated consistently for all cars. A platoon may also split upon the request of a car to leave the platoon. This results in the creation of two platoons. A leading platoon that increases its speed whereas the newly formed tail platoon decreases its speed to achieve some separation distance. However, not all cars can initiate a split, as they may leave other cars stranded. After splitting, the resulting platoons should have at least some minimal size S. For such traffic systems, the global system dynamics depends both on the behavior of individual cars as well as the organization into platoons and their coordination.

Providing insight into the interplay between static coordination and various types of dynamism and reconfiguration has been a driving concern for the design of the DR-BIP component framework [9]. DR-BIP is an extension of the BIP framework which has been used for more than a decade for modeling component-based systems with static architectures. In BIP, a system is built from architecture-agnostic components coordinated using interactions and priorities. We have thoroughly formalized operational semantics for BIP and studied results comparing its expressiveness with respect to existing modeling formalisms [5]. The theoretical framework has been implemented by a toolset integrating an execution engine, code generators for different types of execution platforms, and verification tools. DR-BIP supports an incremental modeling methodology considering that a system consists of a set of motifs, a kind of "worlds" where components "live". Motifs are dynamic architectures integrating components coordinated according to specific rules. To model component mobility, a motif is equipped with a data structure which is a graph representing a map. An addressing function is used to associate with each component a node of the map. *Reconfiguration rules* deal with: 1) component dynamism e.g., creation/deletion of components; 2) map dynamism e.g., updating the map structure; 3) component mobility e.g., changing the addressing function, and 4) modifying connectors that define interactions between components. Furthermore, it is possible to express reconfiguration between motifs e.g., component migration, which confers the ability for system self-organization.

The paper presents the DR-BIP *language* for constructing dynamic reconfigurable systems using BIP components and connectors as well as a *layered implementation* of DR-BIP that re-uses the BIP Engine. Our previous work on DR-BIP introduced the concepts for programming reconfigurable systems [9,8], nonetheless, its implementation was limited to a restricted abstract language of simple components and motifs, and had little support for executing reconfiguration rules. The DR-BIP language has been designed to integrate full-fledged components and connectors described in BIP, arbitrary motif maps and addressing functions described as external C++ structures, and a high-level declarative syntax for reconfiguration rules. The implementation of DR-BIP semantics relies on code generation (for motifs, reconfiguration rules, components, etc) and clearly separates reconfiguration issues from the execution of static configurations. It involves two separate computational phases: the first deals with the execution of reconfiguration rules that determine the overall static coordination structure which we refer to as the *instantiated BIP model*; the second executes for the instantiated BIP model,

the interactions between components using the BIP Engine. The two phases alternate in a global computation cycle synchronized by a well-defined protocol. The present implementation offers the advantage of using the mature and efficient BIP execution engine. Furthermore, the underlying separation of concerns allows better comprehension and management of the intricate semantics of DR-BIP and enhances confidence in the reconfiguration engine implementation. Another direct benefit from the studied implementation is the possibility to monitor a holistic view of the system behavior as a set of traces involving dynamic configurations in addition to component states. The proposed implementation of DR-BIP provides insight into system behavior as the combination of both architectural and component state information, as well as monitoring execution traces at different levels of detail. Analyzing such traces poses interesting questions regarding the formalization and verification of properties of reconfigurable systems. Nonetheless, the development of a specific logic for expressing reconfiguration properties and the synthesis of monitors are beyond the scope of this paper.

Automatic code generation for implementation and/or analysis purposes is one of the most prominent features of model-based design methodologies. There exists a tremendous number of modeling formalisms and associated code generation flows dedicated to static systems. On one hand, general purpose formalisms such as UML/PAPYRUS [13], AADL/OSATE [11], TASTE [19], PTOLEMY [7] are state-of-the-art tools oriented towards design and implementation. On the other hand, domain specific formalisms and/or software libraries such as GENOM [15] and ROS [22] focus on simulation, rapid prototyping and implementation targeting only specific application domains.

Code generation and runtime analysis is becoming more challenging for reconfigurable systems. Approaches have been explored for several general-purpose architecture description languages with reconfiguration capabilities such as π-ADL [6], ARCHJAVA [1], C2SADEL [17] to cite only a few. Nonetheless, these approaches did not reach the same level of acceptance as for static systems. Architecture dynamism and reconfiguration is of much interest for domain specific languages as well. Formalisms such as BUZZ [20] for swarm robotics and PARACOSM [14] for autonomous driving systems are gaining popularity. However, most of the concepts supported are ad-hoc and hardly re-usable beyond their domain. We position DR-BIP as a general-purpose architecture description language for the description of dynamic reconfigurable systems.

The paper is structured as follows. Section 2 recalls the DR-BIP concepts and introduces the key features of the DR-BIP language. Section 3 presents the DR-BIP layering principle and its concrete implementation. Section 4 provides insights on the software architecture and the tooling for generating code used to execute DR-BIP models. Section 5 presents the provided monitoring support and illustrates its application to the analysis of a platoon system as presented earlier. Section 6 discusses challenging future work directions on modeling and analysis of dynamic reconfigurable systems. Finally, Section 6 concludes about the benefits from the presented layered implementation. We provide an online artifact with the tools, the simulation experimental data, and the documentation needed to recreate it [23].

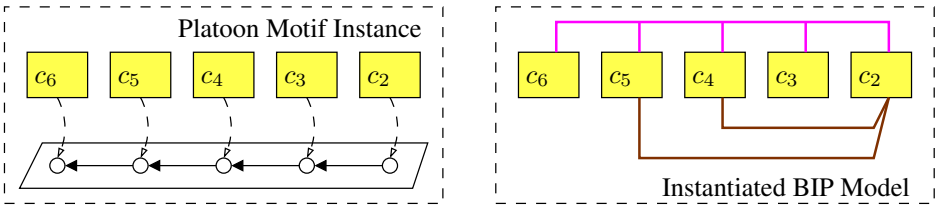


Fig. 1: An example motif and its instantiation in BIP.

2 DR-BIP Overview

We consider DR-BIP models integrating (1) components and connector types described in the BIP language and (2) motif types and reconfiguration rules described in the DR-BIP language. We recall that in BIP, connector types specify interactions between components along their interfaces. This section introduces the concepts underlying DR-BIP and key features of the language.

2.1 Motifs and reconfiguration rules

Motifs are dynamic structures consisting of (i) a set of components, (ii) a map and (iii) an addressing function [9]. Maps are underlying data structures (typically a directed graph) used to organize interactions between components. Addressing functions provide an association between the components and the nodes of maps.

Example 1. Figure 1 showcases on the left an instance of a motif type named Platoon. It consists of the set of five components $\{c_2 \dots c_6\}$, the map consisting of a chain of five nodes, and the addressing function associating each component with one node in the chain. Component c_2 is associated with the head of the chain, thus giving it the role of the leader. Components $c_3, \cdots c_6$ are associated with other nodes and have the role of followers of the platoon.

Motifs are associated with local reconfiguration rules dictating their evolution. These rules define how components are interconnected, and modify the motif i.e., add/remove components, update the map, and update the addressing function. Rules are executed depending on conditions evaluated on the motif and component states. More precisely, local reconfiguration rules have the general form:

`rule` rule-name (arg_1, ... arg_n)
`when` ($cond_1$, ... $cond_n$) { $action_1(args)$, ... $action_m(args)$ }

Their execution consists in selecting a set of components of the motif and assigning them to formal parameters arg_i. Assignments are constrained by a "`when`"-clause, providing a Boolean condition $cond_i$ for each formal parameter. For each valid assignment to formal arguments *args*, a sequence of actions $action_j$ is executed modifying the motif. An assignment of the parameters such that a "`when`"-clause is satisfied, is called a *match*. Further details on the computation of matches are presented in Sect. 3.2.

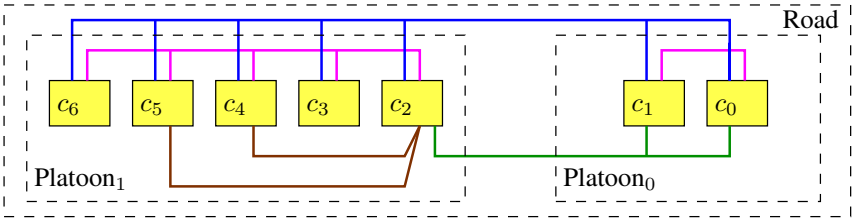


Fig. 2: An example platoon system consisting of 7 cars organized in 2 platoons.

We distinguish between two types of configuration rules: 1) *connecting rules* whose actions deal solely with the creation of connectors, and 2) *updating rules* whose actions deal with the creation or deletion of components and the modification of the map or of the addressing function. The distinction is important because connecting rules do not change the motif; they affect solely the instantiated BIP model.

Example 2. Figure 1 illustrates the Platoon motif (left), and the instantiated BIP model resulting from the application of connecting rules (right). Connector `SpeedUpdate` (pink) is used to synchronize speed with the leader. Furthermore, cars that are allowed to split from the platoon are connected with the leader, as they need to adjust speeds. We call this connector `SplitStep`. For example, c_3 cannot split, since by doing so, it will leave c_2 stranded. Since both c_4 and c_5 can split, two connectors (shown in brown) of `SplitStep` are created.

While local reconfiguration rules apply to a single motif and are given access to that motif only, global reconfiguration rules apply to more than one motif. In this case, the formal parameters can be assigned any motif or component. Moreover, actions can perform migration of components between motifs as well as the creation or deletion of motif instances. The execution of all reconfiguration rules both local and global results in creating an instantiated BIP model containing components interconnected by a set of connectors.

Example 3. Figure 2 illustrates a fully instantiated BIP model of the platoon system. We have one Road motif grouping all 7 cars, and two platoons: $\mathrm{Platoon}_0$ grouping two cars, and $\mathrm{Platoon}_1$ grouping five. We notice two `SpeedUpdate` connectors (pink), one for each platoon, and two `SplitStep` connectors (brown) in $\mathrm{Platoon}_1$, as no car is allowed to split in $\mathrm{Platoon}_0$. The `GlobalStep` connector (blue) allows all cars to move synchronously. Finally, to perform a merge, the two platoons can interact using connector `MergeForward` (green) resulting from the execution of a global connecting reconfiguration rule between the two platoons. This connector involves the leader of the heading platoon, the tail and the leader of the second platoon. It allows checking proximity and updating the speed.

2.2 The DR-BIP Language

The DR-BIP language is a declarative language for the description of: (1) imports that expose all building blocks for the dynamic model (i.e., component types, connector

types, additional BIP predicates, map types and addressing functions); (2) a set of motif types with their associated local and global reconfiguration rules; and (3) a designated "initializer" global reconfiguration rule to initialize the system.

Imports. The DR-BIP language allows component and connector types to be imported from a BIP model. Additionally, external general purpose data structures and functions from the host language (C++) can be used for the implementation of maps and addressing functions. Listing 1.1 provides an example of imports. It includes the reference to the BIP package containing component and connector types (Line 1) which can be used in DR-BIP rules. The listing also includes specifications of signatures of the data structures used to build maps and addressing functions that can be found in a file `platoon.cpp`. We distinguish methods that modify objects from those that do not (using keyword `const`). We see in lines 3-9 the declaration of the data structure `PlatoonMap` used as a map. The methods prefixed with `const` do not modify the map. For example, the method `allowedSplit` checks whether a split of the platoon by the car assigned to that node is allowed. The method `assign` replaces all nodes of the current map with those of another `PlatoonMap`. Finally, we import the signature of additional BIP predicates, described as Boolean functions over elements of the BIP model. Line 11 introduces the predicate `AtSplitLocation` which checks if a given `Car` component (whose type is described in BIP) is in a control location where a split is allowed.

Listing 1.1: DR-BIP external declarations.

```
1  model platoon
2  import from "platoon.cpp" {
3    map PlatoonMap {
4        const bool isLeader(Node)
5        const bool allowedSplit(Node)
6        const PlatoonMap[] splitAt(Node)
7        void  assign(PlatoonMap)
8        ...
9   }
10   addressing PlatoonAddressing { ... }
11   predicate AtSplitLocation(Car)
12 }
```

Motif types and reconfiguration rules The definition of a set of motif types follows the declaration of imports in a DR-BIP specification. For each motif, the types of map and addressing functions are declared, along with their associated reconfiguration rules.

As explained previously, each reconfiguration rule consists of (1) a label specifying its name, (2) a list of formal parameters referring to components or motifs affected by the rule's actions (3) a "when"-clause consisting of a list of Boolean conditions used to filter the relevant components and assign them to formal parameters, and (4) one or more reconfiguration actions.

The concrete specification of reconfiguration rules in the DR-BIP language uses the following predefined symbols for every motif: the set of managed components (C), the map (S), and the addressing function (@). These variables can be used in the "when"-clause by calling the `const` methods of the map and addressing functions[1]. Furthermore, all methods for S and @ can be used as reconfiguration actions.

Listing 1.2: The `Platoon` motif and the `SplitIR` connecting rule.

```
1 motif Platoon<PlatoonMap, PlatoonAddressing> {
2   connecting rule SplitIR(Car leader, Car follower)
3     when ((C.size() > 3) && S.isLeader(@leader),
4       follower != leader && S.allowedSplit(@follower)) {
5         new SplitStep(follower, leader)
6 }}
```

Example 4. Listing 1.2 displays the motif `Platoon` along with the local connecting rule `SplitIR`. Line 1 declares the motif with the identifier `Platoon`, and associates with it the map `PlatoonMap` and addressing function `PlatoonAddressing` (which are imported as explained in Listing 1.1). `Platoon` has a connecting reconfiguration rule `SplitIR`. It creates a connector (of type `SplitStep`) between each follower and the leader to allow the follower to split. Thus, it is parametrized by a *follower* and a *leader*. The "`when`"-clause provides two conditions, one for each parameter. The first condition (line 3) specifies that there must be at least 3 components in the component set and queries the map to ensure that the node associated with *leader* is the head of the platoon in the map. Since function `isLeader` expects a node, the addressing function is used (`@leader`) to get the associated node. In the second condition (line 4), we ensure that the *follower* is any other car in the platoon where a split is allowed. To check if splitting is allowed, we make use of the method `allowedSplit` associated with the map. For every match, we create a new connector (`SplitStep`) with the two associated components. ■

3 Layered Implementation

We adopt a two-layered approach for the execution of DR-BIP models. The first layer is the *interaction layer*. It deals with an instantiated BIP model involving a fixed number of components and their connectors. This model can be simulated and executed using the BIP engine, and analyzed using existing tools of the BIP toolset. The second layer is the *reconfiguration layer*. It deals with dynamic changes of the set of components and the way they are connected. It includes the reconfiguration rules that determine how the instantiated BIP model evolves. We note that this implementation principle is general and can be applied to any kind of dynamic reconfigurable system.

Our approach is driven by two main goals: *separation of concerns* and *simplicity*. Separation of concerns means that we reason separately about each layer. We reuse

[1] The set of components (C) is a C++ `std::set`, and relevant methods can be used.

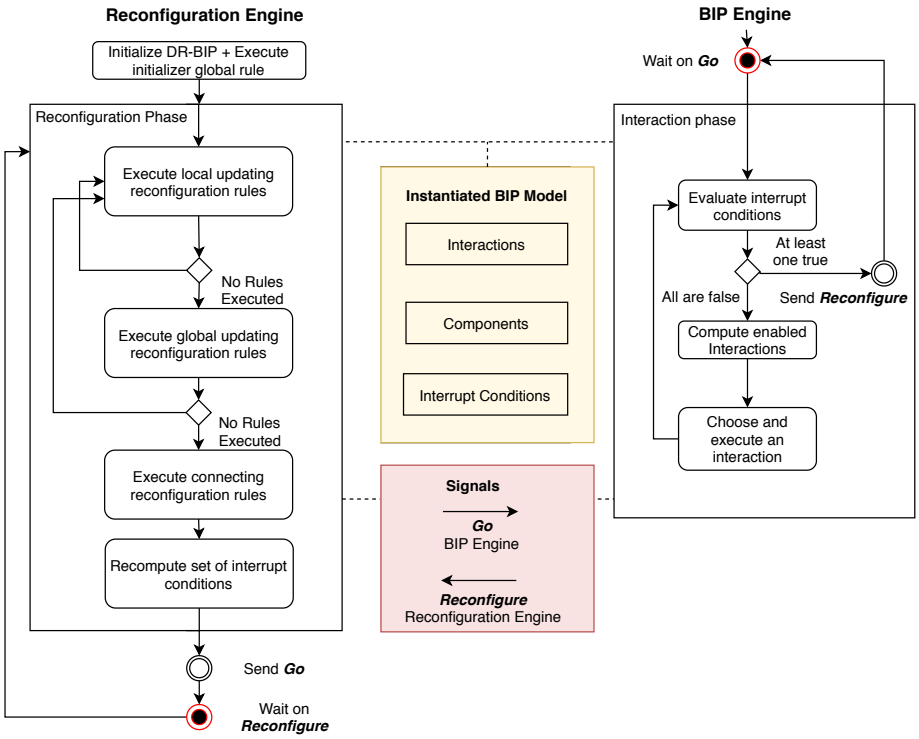


Fig. 3: Interplay between the reconfiguration engine and BIP engine.

predefined elements from the interaction layer such as types of components and connectors as well as existing engines for execution, analysis, and verification (cf. [2]), while adequately extending all the concepts of BIP. Simplicity means that we ease the learning and use of the language by introducing a minimal set of new concepts and constructs. For the reconfiguration layer, we only provide constructs for building the rules modifying the instantiated BIP model that is used by the interaction layer. In the reconfiguration layer, elements of the BIP model and all necessary topological structures are external to the language and are imported.

3.1 Execution Principle

The execution of a DR-BIP model uses a protocol for the collaboration between two engines (shown in Fig. 3): the reconfiguration engine handling the reconfiguration layer, and the BIP engine handling the interaction layer. The two engines share an instantiated BIP model which consists of: the set of components, the set of connectors between these components, and a set of interrupt conditions. An interrupt condition is a predicate on a given state of the instantiated BIP model. When an interrupt condition becomes true, the system needs to be reconfigured. Normal execution is stopped in the BIP engine, and control is transferred to the reconfiguration engine so that it applies reconfiguration.

Two signals determine which layer is to execute: *Go* and *Reconfigure*, respectively, initiate phases of execution of the BIP engine and the reconfiguration engine.

The BIP engine evaluates all interrupt conditions, and if *none* of them holds, it computes *enabled* interactions. If an interrupt condition holds, control is transferred to the reconfiguration engine until a *Go* signal is received. In BIP, interactions are specified by connectors. They represent synchronized state changes of several components. An interaction is *enabled* if all inter-connected components are able to synchronize. The BIP engine finds all enabled interactions and, if any, one is selected for execution. Following the execution, the states of all inter-connected components are updated. If no enabled interaction is found, the BIP engine has reached a deadlock state of the instantiated BIP model and the entire execution of the dynamic reconfigurable system is halted. Otherwise, it continues executing as long as no interrupt condition holds.

The reconfiguration engine executes the reconfiguration rules and modifies the instantiated BIP model. First updating reconfiguration rules are executed iteratively until stabilization. They consist in adding/removing components and updating the structure of the motifs. In our implementation, local updating reconfiguration rules execute first until stabilization, followed by global updating reconfiguration rules. When stabilization is reached, connecting rules are executed to re-create the connectors between components, and then the set of interrupt conditions is updated. Control is then passed to the BIP engine. It is important to note that to avoid interference, reconfiguration rules are executed *sequentially*.

3.2 Reconfiguration Engine Details

We present additional details about the reconfiguration engine and the way it computes matches and modifies the set of interrupt conditions.

Computing matches. The computation of matches constitutes the core operation in the reconfiguration phase. It can be computationally expensive as generally, a condition for a given parameter can depend on other parameters. For efficiency reasons, in our implementation, we restrict conditions to reference only previous parameters in the list (i.e., a condition for parameter at index n can reference only parameters $[0..n]$). This ensures that matches can be computed incrementally without the need to backtrack, stopping immediately if no assignment is found.

Example 5. We revisit the connecting rule in Listing 1.2 applied on Example 1. The `Platoon` motif shown in Fig. 1 has the set of components $\mathrm{C} = \{c_2, c_3, c_4, c_5, c_6\}$, where c_2 is the leader and c_6 is the tail of the platoon. This rule has two formal parameters of component type `Car` namely, `leader` and `follower`. Since it is a local reconfiguration rule, parameters are assigned components of type `Car` belonging to the motif (in our case C). The "`when`"-clause consists of 2 conditions, one for each formal parameter. The computation begins by evaluating the first condition to calculate the possible assignments for `leader`. The predicate $|\mathrm{C}| > 3$ holds for all possible assignments. However, `S.isLeader(@leader)` is true only for c_2. The final set of assignments for `leader` is therefore $\{c_2\}$.

We now evaluate the possible assignments for `follower`. Note that it is possible to reference also `leader` as the set of possible assignments is computed. Indeed the condition `leader != follower` is evaluated for each component assigned to `leader` (i.e., for all elements in $\{c_2\}$). Furthermore, since no split can result in a platoon with less than 2 cars, `S.allowedSplit(@follower)` holds only for c_4 and c_5. The possible assignments for the second parameter are thus $\{c_4, c_5\}$. The rule executes for each of the following two matches: $\langle c_2, c_4\rangle$ and $\langle c_2, c_5\rangle$, creating two `SplitStep` connectors (shown on the right of Fig. 1 in brown). ■

Executing rules. After finding possible matches for a given rule, it is possible to execute it multiple times. In the case of updating reconfiguration rules (both local and global), the rule is executed with the first match, before stopping and re-evaluating all the reconfiguration rules. This is important, as a rule action can modify the state of (one or more) motif component set, map, or addressing function, requiring all respective clauses to be recomputed. However, in the case of connecting rules, the rule is executed for all possible matches, as connecting rules only add connectors to the model without modifying the motifs.

Modifying the set of interrupt conditions. Interrupt conditions are predicates on the state of the instantiated BIP model that initiate reconfiguration. For each rule, we construct the conjunction of all conditions in the "`when`"-clause, and instantiate it to account for all possible assignments of parameters, adding each instantiation as an interrupt condition. The set of interrupt conditions models the disjunction over all such interrupt conditions, as when one holds at least one rule has a match and can execute. Since the sets of components and motifs change dynamically after reconfiguring, the set of possible assignments changes, and the set of interrupt conditions is updated to account for it. Recall that during the execution of the instantiated BIP model, motifs remain unchanged, when generating interrupt conditions, only BIP predicates are considered to change values. Therefore, optimizations by partial evaluation are applied to minimize the number of instantiated interrupt conditions.

4 Software / Code Architecture

The two-layered execution principle (Fig. 3) is mirrored in the software code, where BIP and the reconfiguration layer are written separately, and compiled separately to generate all necessary C++ files. Then, all C++ files are compiled together to form a single executable process. Figure 4 showcases the compilation scheme for DR-BIP models. Note the clear separation between BIP language, DR-BIP language, and external code.

BIP compiler modifications. We modified the BIP compiler to generate the BIP engine code that accounts for interrupt conditions. As shown in Fig. 4, the BIP compiler also generates a separate file containing all type information on connectors and components. The BIP engine itself has been modified to work on a dynamic number of components and connectors. Related connectors and interrupt conditions are grouped according to the rules that generate them. In this way, a group acts as a "namespace" for the corresponding rule allowing (1) dynamic change in connectors and interrupt conditions to

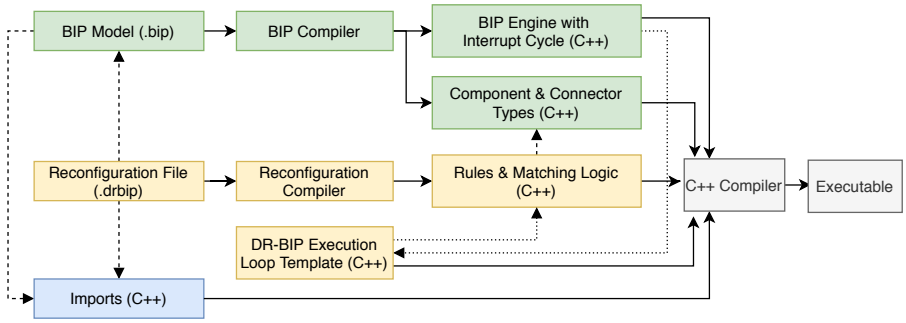


Fig. 4: Compilation scheme for the reconfiguration engine alongside BIP. Solid arrows indicate input/output relation, dashed arrows indicate logical dependencies and dotted arrows highlight the control-flow between the various elements.

be done in bulk for performance gains; and (2) isolation of connectors and interrupt conditions of one rule from those of other rules. In BIP, connectors are originally conceived to operate on a fixed number of components. To overcome this limitation, an extra annotation to the BIP model has been introduced to specify connector types with a variable number of components.

Reconfiguration compiler. The reconfiguration compiler performs the parsing, analysis, and code-generation for the reconfiguration layer. It re-uses the BIP compiler to load information on interaction and component types for the considered BIP model. Furthermore, it parses the elements from the input reconfiguration file to generate the reconfiguration engine code. Type-checking is done at this phase using information from the BIP model and the external signatures of BIP predicates, maps, and addressing functions. To analyze, and eventually generate code, the reconfiguration compiler provides an infrastructure for executing a sequence of passes on the model. The reconfiguration compiler performs two passes: the first annotates rules and motifs with necessary information to minimize the number of interrupt conditions; the second pass generates C++ code. It outputs the necessary code to (1) create data-structures to manage components; (2) compute matches, execute, and generate interrupt conditions for rules; and (3) initialize and destroy motifs dynamically.

Execution control flow. The DR-BIP implementation compiles both engines in one executable. In the resulting executable, the BIP engine begins executing, and yields control to the reconfiguration engine using the two function calls `init` and `reconfigure`. Function `init` is called once to initialize the reconfiguration engine. Function `reconfigure` is called whenever reconfiguration is needed. Alongside the reconfiguration compiler, a template (C++ file) provides an implementation for functions `init` and `reconfigure`. The `init` implementation calls the "initializer" global reconfiguration rule, then performs a reconfiguration phase, while `reconfigure` performs the reconfiguration phase as described in Fig. 3. So, control flow goes from the BIP engine main loop to the template, which then invokes the relevant methods for the rules and

motifs generated by the configuration layer compiler to perform reconfiguration. By providing the general execution infrastructure in a template separately from the rules specific to the dynamic model, it is possible to manage, inspect, and customize the DR-BIP state and execution independently of a given model. Template modification is key for instrumenting DR-BIP and outputting traces for analysis and monitoring, as we will discuss in Sect. 5.

5 Monitoring, Analysis and Profiling Support

In Sect. 4, we introduced the two elements allowing DR-BIP modular support for tracing, inspection, and run-time monitoring and analysis. On the one hand, the reconfiguration compiler provides a modular pass on the model that can be used to generate further code and "hooks" for tracing and generating run-time events. On the other hand, the DR-BIP template provides all the structures and control flow primitives needed to inspect, analyze and log relevant elements of the model: the instantiated BIP model shown in Fig. 3, including all BIP component instances internals, sets of all active motif instances and their reconfiguration rules, as well as global reconfiguration rules.

In addition to execution, our implementation provides the key ingredients that enable profiling, analyzing, and monitoring the execution of DR-BIP models. In this section, we describe our approach for inspecting dynamic reconfigurable architectures and generating a trace for analysis and profiling. We first present the concept of DR-BIP traces. Then, we evaluate quantitative properties on two scenarios of platoon system, we profile the execution time of these scenarios, and report on performance metrics.

5.1 DR-BIP Traces

The *state* of the instantiated BIP model is usually represented as a tuple giving for each component its current control location and a valuation of its variables [3,10]. Nonetheless such a concept of state is not adequate for dynamic reconfigurable systems as it ignores their structure represented by the motifs and the outcome of applying reconfiguration rules. For this reason we use the concept of *configuration* which encompasses the usual notion of component state and additionally accounts for architectural aspects. A *configuration* of a DR-BIP model consists of (i) the set of its motifs, and (ii) for each motif the set of its associated components and the connectors generated by executing its connecting rules. It is represented as a set of hyper-graphs, one for each motif. The set of nodes is the set of the components of the motif. Each hyper-edge corresponds to a connector relating the involved components. We call *stateful configuration* a pair consisting of a configuration and component states at a given execution step. A DR-BIP trace is the sequence of stateful configurations.

Example 6. Fig. 5 shows a stateful configuration of a platoon system with 12 cars. The configuration shows the cars grouped by platoons while the position of the cars is part of their state. Combining information from both makes it possible to calculate distances between platoons, and the space occupied by all the cars. Our example has 4 platoons (shown different colors) occupying a total of 10.41 units. We infer from the respective positions that two platoons are about to merge as they are separated by 0.41 units. ■

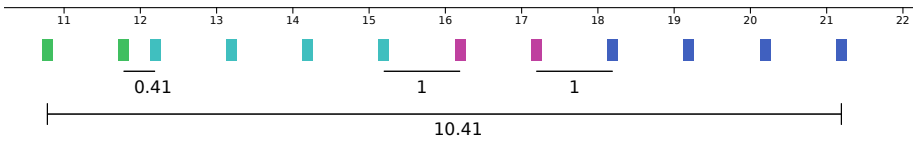


Fig. 5: Stateful configuration of a 12 car platoon system with 4 platoons (each platoon is assigned a unique color).

5.2 Properties for Dynamic Reconfigurable Systems

By analyzing DR-BIP traces we are able to check properties not only about components and their states, but also about the evolution of configurations as a result of reconfiguration actions. Monitoring and checking such qualitative or quantitative properties requires the evaluation of predicates on stateful configurations.

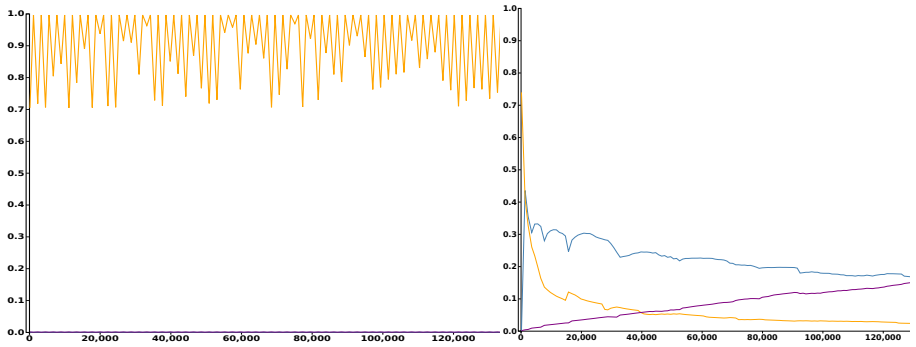
Qualitative properties can be formalized in logics. Clearly, a propositional formula can be checked on a given stateful configuration. If a formula involves modalities, it should be evaluated on sequences of stateful configurations. While formalizing qualitative properties for dynamic reconfigurable systems is not in the scope of this paper, we present an example safety property ensuring that a car can belong only to one platoon, and all cars in the same platoon have the same speed as the leader. The property can be expressed as:

$$\Box\, \forall p, p' \in \texttt{Platoon}\, \forall c \in \texttt{Car}\, \exists c' \in p.\mathrm{C}\, \exists X \subseteq p.\mathrm{C}:$$
$$(p' \neq p \wedge c \in p.\mathrm{C}) \implies (c \notin p'.\mathrm{C} \wedge (\texttt{SpeedUpdate}(c', X)) \wedge c.\mathrm{speed} = c'.\mathrm{speed})).$$

This means that for any sequence of stateful configurations, the argument of $\Box$ holds. For a given stateful configuration, we can verify if a car belongs to a platoon ($c \in p.\mathrm{C}$) by checking if it is a vertex in the hypergraph of the motif. Furthermore, the leader is determined by verifying the presence of the edge associated with the connector ($\texttt{SpeedUpdate}(c', X)$) in the corresponding hypergraph. Using the state of each car, we verify that the cars have the same speed as the leader ($c.\mathrm{speed} = c'.\mathrm{speed}$).

Quantitative properties can be evaluated in a similar manner. We specify the degree of satisfaction of a quantitative property by using score functions normalized in the interval $[0..1]$. We assume that the degree increases as the scores get close to zero.

Example 7. To analyze quantitative properties of the platoon system in Example 6, we define three score functions: (1) uniform separation (US), (2) target size (TS), and (3) road occupancy (RO). Uniform separation is used to assess how uniform the distance between platoons is. It is computed from the set of distances between platoons, normalized in an interval $[0..1]$ to obtain a set of distances d_i, for which we compute their mean $\overline{d}$, and the standard deviation as follows: $\mathrm{US} = \sqrt{\frac{\Sigma((d_i - \overline{d})^2)}{|d|}}$. Thus, for uniform distances the score converges to 0. TS is used to estimate how close the size of platoons is to an ideal target size. It is computed similarly to the score US as the deviation from the ideal size. For this example, we fix the ideal size to be 2. Lastly, RO measures the total space occupied by all cars on the road. It is computed as follows:



(a) S_1 (initial distance = $0.3 < K$) (b) S_2 (initial distance $1.0 = 2K$)

Fig. 6: The evolution of scores in two scenarios of platoon systems. The Y-axis shows the score (in the range $[0..1]$), while the X-axis shows the number of steps of the system (number of rules and interactions executed). The scores are: uniform separation (blue), target size (orange), and road occupation (purple).

$\mathrm{RO} = 1 - ((L - \Sigma d_i)/L)$ where L is the distance between the leading car of the first platoon and the tail car of the last platoon. By combining the three scores we aim to measure deviation with respect to configurations where the platoons have a specific size, while maintaining similar distances between them that are not too large, so as to not waste road space. The scores computed for the stateful configuration in Fig. 5 are as follows: $\mathrm{US} = 0.34, \mathrm{TS} = 0.14$ and $\mathrm{RO} = 0.23$. Indeed, we observe that the distance between the left-most two platoons (0.41) significantly differs from the distance between the two others (1). Additionally, we see two platoons of size 4, and a total separation distance of 2.41 units between platoons (23% of the total distance occupied by the cars).

We consider two different large platoon systems: S_1 and S_2 differing only in the initial distance between cars. Both S_1 and S_2 start with a single platoon of 1,000 cars. In S_1 cars are initially separated by $K = 0.3$ which is less than the minimal distance needed to merge platoons, while in S_2 cars are initially separated by $2K$. By taking the initial distance between cars smaller than the merge distance, platoons have very little possibility to diverge, as they will be merged very soon after a split. On the contrary, with an initial distance much larger than K, platoons are not likely to be merged soon after a split. We illustrate the evolution of scores in Fig. 6. In the case of S_1, cars form at most 2 platoons during the simulation, and we are unable to reach platoon size 2, as splits are rapidly followed by merges. In addition, distances between platoons remain equal in S_1 as splits are rapidly followed by merges allowing no possibility for the cars to separate. For S_2, we see a convergence towards the target platoon size. However, we observe larger and diverging separation between platoons, as well as differentiation of the distances between platoons. ■

Table 1: Fixed-length simulation of 5,000 interactions of platoon system capturing maximum number of motifs (**M**) and execution times (**BIP**, **DR-BIP**) when varying initial distance, number of cars (**N**), and number of reconfiguration phases (**RP**).

	S_1				S_2			
N	**RP**	**M**	**BIP** (s)	**DR-BIP** (s)	**RP**	**M**	**BIP** (s)	**DR-BIP** (s)
100	48	4	0.16	0.05	178	77	0.18	0.52
	13	4	0.13	0.01	30	40	0.14	0.05
500	48	4	0.68	0.59	569	306	0.83	30.40
	14	4	0.72	0.18	47	65	0.75	0.77
1000	48	4	1.40	2.08	729	504	1.52	118.00
	14	4	1.44	0.62	51	73	1.44	1.94

5.3 Profiling DR-BIP Performance for Platoon System

We use performance metrics to profile the behavior of both DR-BIP and the system being executed. Measurements of reconfiguration frequency and time allow fine-tuning rules and models so as to enhance performance.

Recall from Example 7, we have two platoon systems that differ only in the initial distance between cars: S_1 and S_2. Both S_1 and S_2 start with a single platoon of N cars. Cars are initially separated by $0.3 < K$ for S_1 and $2K$ for S_2, where K is the minimal distance needed to merge platoons. We performed a fixed-length simulation up to 5,000 BIP interactions for S_1 (resp. S_2), and report the result in Table 1 averaging values over 10 simulations. To control the number of reconfiguration phases, we modify the model so that a car cannot split before moving for a certain amount of time.

Consider the case of 1000 cars and high reconfiguration rate (row 5). The average total execution time for S_1 (resp. S_2) is 3.48s (resp. 119.52s). A reconfiguration phase occurs after executing 104 (resp. 7) interactions. The percentages of the time spent reconfiguring are respectively 60% and 99%. For S_1, we can see that the maximum number of motifs present after each reconfiguration phase (**M**) is 4, indicating the presence of at most 2 platoons (with 1 road and 1 motif responsible to merge platoons). For S_2, we see the creation of multiple motifs, increasing to 729 motifs, which results in an increase of execution time for DR-BIP. Growing number of car and motif instances increases the time required for matching motif rules and global reconfiguration rules. When reconfiguration is less frequent, and with fewer motif instances, we observe reasonable execution times. For S_2, when a reconfiguration phase occurs every 98 interactions (row 6), the runtime is reduced from 119.52s to 3.38s.

6 Conclusion

We presented a two-layered approach for the specification, execution and monitoring of DR-BIP models integrating a reconfiguration and an interaction layer. DR-BIP extends

BIP with the concepts and primitives needed for modeling dynamic reconfigurable systems. It supports modeling reconfiguration involving motifs along with their maps and addressing functions as well as reconfiguration rules for the dynamic change of local and global coordination patterns. We elaborated on the interplay between the reconfiguration engine which manages motifs and executes reconfiguration rules to modify an instantiated BIP model, and the BIP engine which executes the interactions of the latter. We also presented the necessary extensions to the BIP compiler and engine in order to reuse them in the context of dynamic and reconfigurable systems.

Another key contribution is the definition of traces for DR-BIP models allowing a natural high level interpretation of system behavior and run-time monitoring and analysis. Using information from stateful configurations, we can analyze properties characterizing the complex collective behavior of the system components. We demonstrated all these concepts for both qualitative and quantitative properties on a platoon system example inspired from autonomous traffic systems. We focus on two complementary directions for future work. The first is to improve the current implementation. The second is to provide enhanced support for analysis and controlled experimentation through model simulation. We plan to increase the efficiency of the execution of the reconfiguration phase, by allowing the incremental evaluation of reconfiguration rules. For instance, static dependencies between rules can be exploited to avoid the costly global re-evaluation of all rule constraints and computation of matches at every step. Moreover, symbolic representations and/or searching techniques inspired from SAT/SMT solving can be used to isolate the impact of a rule on the system and therefore to restrict the focus of application of rules to that part.

Regarding the support for analysis at runtime, our simulation infrastructure provides all the needed introspection capabilities to extract both architecture (i.e, hypergraph of interconnected components and connectors) and state information (i.e, component state vectors). We plan to integrate the monitoring of behavior and/or configuration properties by introducing temporal modalities [21] into configuration logics [16]. Beyond monitoring, statistical model checking can be used to evaluate properties over multiple traces of the system and to compute levels of confidence for their satisfaction. However, the application of statistical model checking would require the definition of a stochastic semantics for the DR-BIP model. As a first alternative, this can be achieved by keeping non-stochastic the execution of the reconfiguration engine, while leaving stochastic aspects at the BIP level to be handled only by the (statistical) BIP engine [18]. A more challenging alternative would consist in proposing and implementing a stochastic semantics for the reconfiguration engine to be combined with the BIP engine.

Finally, in addition to monitoring and analysis, we plan to develop support for controlled experimentation allowing to bring reconfigurable dynamic systems into specific configurations and check their behavior. This is particularly important for the definition of coverage criteria and validation through the exploration of corner cases and critical situations. It will additionally require the refinement of the BIP model into a model distinguishing between observable and controllable features and their integration.

References

1. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: connecting software architecture to implementation. In: Tracz, W., Young, M., Magee, J. (eds.) Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA. pp. 187–197. ACM (2002)
2. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software **28**(3), 41–48 (2011)
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006). pp. 3–12. IEEE Computer Society (2006)
4. 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)
5. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008 - Concurrency Theory, 19th International Conference Proceedings. Lecture Notes in Computer Science, vol. 5201, pp. 508–522. Springer (2008)
6. Cavalcante, E., Oquendo, F., Batista, T.V.: Architecture-based code generation: From π-ADL architecture descriptions to implementations in the go language. In: Avgeriou, P., Zdun, U. (eds.) Software Architecture - 8th European Conference, ECSA 2014, Vienna, Austria, August 25-29, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8627, pp. 130–145. Springer (2014)
7. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S.R., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. Proceedings of the IEEE **91**(1), 127–144 (2003)
8. 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 (2018)
9. 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 (2018)
10. Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Software and Systems Modeling **14**(1), 173–199 (2015)
11. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language. SEI series in software engineering, Addison-Wesley (2012)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA (1995)
13. Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., Terrier, F.: Papyrus UML: an open source toolset for MDA. In: Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009). pp. 1–4 (2009)
14. Majumdar, R., Mathur, A.S., Pirron, M., Stegner, L., Zufferey, D.: Paracosm: A language and tool for testing autonomous driving systems. CoRR **abs/1902.01084** (2019)

15. Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., Ingrand, F.F.: GenoM3: Building middleware-independent robotic components. In: IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010. pp. 4627–4632. IEEE (2010)
16. Mavridou, A., Baranov, E., Bliudze, S., Sifakis, J.: Configuration logics: Modeling architecture styles. J. Log. Algebraic Methods Program. **86**(1), 2–29 (2017)
17. 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, Los Angeles, CA, USA, May 16-22, 1999. pp. 44–53. ACM (1999)
18. 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)
19. Perrotin, M., Conquet, E., Delange, J., Schiele, A., Tsiodras, T.: TASTE: A real-time software engineering tool-chain overview, status, and future. In: Ober, I., Ober, I. (eds.) SDL 2011: Integrating System and Software Modeling - 15th International SDL Forum Toulouse, France, July 5-7, 2011. Revised Papers. Lecture Notes in Computer Science, vol. 7083, pp. 26–37. Springer (2011)
20. Pinciroli, C., Beltrame, G.: Buzz: An extensible programming language for heterogeneous swarm robotics. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2016, Daejeon, South Korea, October 9-14, 2016. pp. 3794–3800. IEEE (2016)
21. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE Computer Society (1977)
22. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. In: ICRA workshop on open source software. vol. 3, p. 5. Kobe, Japan (2009)
23. Verimag: DR-BIP Artifact. https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/bip/artifacts/drbip-feb2020 (February 2020)