International Journal on Software Tools for Technology Transfer (2020) 22:437–455
https://doi.org/10.1007/s10009-020-00555-2

FOUNDATION FOR MASTERING CHANGE

Special Section REoCAS

# The DReAM framework for dynamic reconfigurable architecture modelling: theory and applications

Rocco De Nicola[1] · Alessandro Maggi[1] · Joseph Sifakis[2]

Published online: 6 March 2020


**Abstract**
Modern systems evolve in unpredictable environments and have to continuously adapt their behaviour to changing conditions. The "DReAM" (Dynamic Reconfigurable Architecture Modelling) framework has been designed for modelling reconfigurable dynamic systems. It provides a rule-based language, inspired from Interaction Logic, which is expressive and easy to use encompassing all aspects of dynamicity including parametric multi-modal coordination with creation/deletion of components as well as mobility. Additionally, it allows the description of both endogenous/modular and exogenous/centralized coordination styles and sound transformations from one style to the other. The DReAM framework is implemented in the form of a Java API bundled with an execution engine. It allows us to develop runnable systems combining the expressiveness of the rule-based notation together with the flexibility of this widespread programming language.

**Keywords** Architecture description languages · Software architectures · Domain-specific languages · Formal methods · Reconfigurable systems · Dynamic systems

## 1 Introduction

The ever increasing complexity of modern software systems has changed the perspective of software designers who now have to consider new classes of systems, consisting of a large number of interacting components and featuring complex interaction mechanisms. These systems are usually distributed, heterogeneous, decentralized and interdependent, and are operating in an unpredictable environment. They need to continuously adapt to changing internal or external conditions in order to efficiently use their resources and to provide adequate functionality when the external environment changes dynamically. Dynamism, indeed, plays a crucial role in these modern systems and can be guaranteed by exploiting the expressive power offered by the three following features:

1. the parametric description of interactions between instances of components for a given system configuration;
2. the reconfiguration involving creation/deletion of components and management of their interaction according to a given architectural style;
3. the migration of components between predefined architectural styles.

The first feature implies the ability of describing the coordination of systems that are parametric with respect to the number of instances of types of components; examples of such systems are Producer–Consumer systems with $m$ producers and $n$ consumers or Ring systems consisting of $n$ identical interconnected components.

The second feature is related to the ability of reconfiguring systems by creating or deleting components and managing their interactions taking into account the dynamically changing conditions. In the case of a reconfigurable ring, this would require having the possibility of removing a component which self-detects a failure and of adding it back after recovery. Added components are subject to specific interaction rules according to their type and their position in the system. This is especially true for mobile components which are subject to dynamic interaction rules depending on the state of their neighbourhood.

✉ Alessandro Maggi
alessandro.maggi@imtlucca.it

[1] IMT School for Advanced Studies Lucca, Lucca, Italy

[2] Université Grenoble Alpes, Grenoble, France

The third aspect is related to the vision of "fluid architectures" [1] or "fluid software" [2] and builds on the idea that applications and objects live in an environment (we call it a *motif*) which corresponds to an architectural style that enforces specific coordination and reconfiguration rules. Systems' dynamicity is modelled by allowing applications and objects to migrate among motifs; such dynamic migration allows a disciplined and easy-to-implement management of dynamically changing coordination rules. For instance, self-organizing systems may adopt different coordination motifs to adapt their behaviour in order to guarantee global properties.

The different approaches to architectural modelling and the new trends and needs are reviewed in detailed surveys such as [3–7]. Here, we consider two criteria for the classification of existing approaches: *exogenous vs. endogenous* and *conjunctive vs. disjunctive* modelling.

*Exogenous modelling* assumes that components are architecture-agnostic and guarantee a strict separation between a component behaviour and its coordination. Coordination is specified globally by rules applied to sets of components. The rules involve synchronization of events between components and associated data transfer. This approach is adopted by Architecture Description Languages (ADL) [5]. It has the advantage of providing a global view of the coordination mechanisms and their properties.

*Endogenous modelling* requires adding explicit coordination primitives in the code describing components behaviour. Components are composed through their interfaces, which are used to expose their coordination capabilities. An advantage of endogenous coordination is that it does not require programmers to explicitly build a global coordination model. However, validating a coordination mechanism and studying its properties becomes much harder without such a model.

*Conjunctive modelling* uses logics to express coordination constraints between components. In particular, it allows modular description of compound systems as one can associate with each component its coordination constraints. The global coordination of a system can then be obtained as the conjunction of individual constraints of its constituent components.

*Disjunctive modelling* consists in explicitly specifying system coordination as the union of the executable coordination mechanisms such as semaphores, function call and connectors.

Merits and limitations of the two approaches are well understood. Conjunctive modelling allows abstraction and modular description but it is exposed to the risk of inconsistency in case there is no architecture satisfying the specification.

This paper expands upon [8] by giving a more detailed description of the DReAM theoretical framework; moreover it introduces the Java executable implementation of the framework and uses it on a number of illustrative case studies for validation and early assessment.

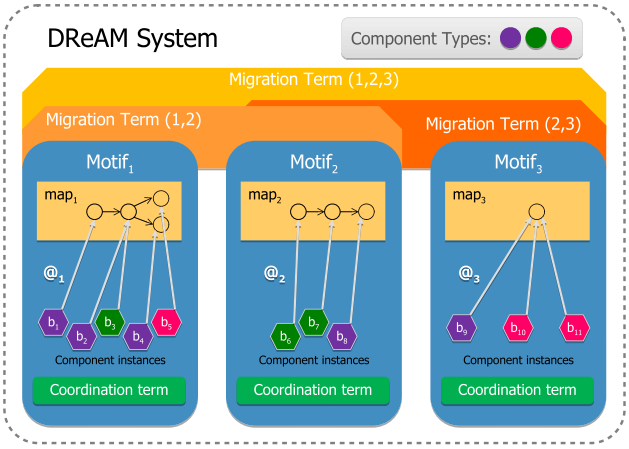



**Fig. 1** Overview of a DReAM system

DReAM uses a logic-based modelling language that encompasses the four styles mentioned above as well as the three mentioned features. A system consists of instances of types of components organized in a collection of motifs. Component instances can migrate between motifs depending on global system conditions. Thus, a given type of component can be subject to different rules when it is in a "ring" motif or in a "pipeline" one. Using motifs allows natural description of self-organizing systems (see Fig. 1).

Coordination terms in a motif involve an interaction part and an associated operation. The former is modelled as a formula of the first-order Interaction Logic [9] used to specify parametric interactions between instances of types of components. The latter enables transfer of data between the components involved in the interaction. In this way, a parametric coordination between classes of components can be specified. The rules built from these terms allow both conjunctive and disjunctive specification styles. In the paper, we will study to what extent a mathematical correspondence can be established between the two styles. In particular, we will see that conjunctive specifications can be translated into equivalent disjunctive global specifications while the converse is not true in general.

To enhance expressiveness of the different kinds of dynamism, each motif is also equipped with a map, which is a graph defining the topology of the interactions in the motif. To parametrize coordination terms for the nodes of the map, an addressing function @ is provided which defines the position @ ($c$) in the map of any component instance $c$ associated with the motif. Additionally, each node is equipped with a local memory that can be accessed by components and used as a shared memory. Maps are also very useful to express mobility, in which case the connectivity relation of the map represents possible moves of components. Finally, our language allows us to modify maps by adding or removing nodes

and edges, as well as to dynamically create and delete component instances.

The rest of the paper is organized as follows.

Section 2 presents the Propositional Interaction Logic (PIL) and shows how it can be used to model static architectures when the involved components are transition systems. It studies the relationship between conjunctive and disjunctive styles and shows that for each conjunctive model there exists an equivalent disjunctive model and vice-versa.

Section 3 lifts the results of the previous section to components and interactions with data. Coordination constraints are expressed in the PILOps language whose terms are guarded commands where guards are PIL formulas and commands are operations on data. PILOps is the core language of the DReAM framework.

Section 4 provides a formal definition of the DReAM framework. Coordination constraints are expressed in a first-order extension of PILOps which allows quantification over component variables involved in rules and guards. The section contains also the definition of the operational semantics of DReAM models and an abstract syntax for a domain-specific language encompassing the basic modelling concepts.

Section 5 describes the prototype Java-based modelling and execution framework and provides a number of illustrative examples and some benchmarks.

Section 6 discusses related work and a brief account of the relationships with the main representatives of existing frameworks.

The concluding section summarizes the main results and discusses directions for their further extension and application to real-life dynamic systems with a focus on autonomous and self-modifying systems.

## 2 Static architectures: the PIL coordination language

We introduce the Propositional Interaction Logic (PIL) [9] used to model interactions between a given set of components.

### 2.1 Components

A system model in PIL is the composition of interacting *components*, which are labelled transition systems where the labels are *port names* and the states are *control locations*. Components are completely coordination-agnostic, as there is no additional characterization to ports and control locations beyond their names (e.g. we do not distinguish between input/output ports or synchronous/asynchronous components).

**Definition 1** (*Component*) Let $\mathcal{P}$ and $\mathcal{S}$ respectively be the domain of ports and control locations. A component is a transition system $B = (S, P, T)$ with

- $S \subseteq \mathcal{S}$: finite set of control locations;
- $P \subseteq \mathcal{P}$: finite set of ports;
- $T \subseteq S \times P \times S$: finite set of transitions; $p \in P$ is the port offered for interaction, and each transition is labelled by a different port.

Each component has one implicit loop transition $\{s \xrightarrow{idle} s\}_{s \in S}$ for each control location $s \in S$ over a dedicated port $idle \in P$. It is assumed that the sets of ports and control locations of different components are disjoint.

Transitions $(s, p, s')$ are also denoted by $s \xrightarrow{p} s'$. Idle ports have been introduced to simplify the theoretical development of the framework. In the rest of the paper, the set of ports of a component without the idle port will be referred as the *set of active ports* $P^* = P \setminus \{idle\}$.

A system specification is characterized by a set of components $B_i = (S_i, P_i, T_i)$ for $i \in [1, n]$. The *configuration* $\gamma$ of a system is the set of the current control locations of each constituent component:

$$\gamma = \{s_i \in S_i\}_{i \in [1 \ldots n]} \tag{1}$$

Given the finite set of ports of a system $P = \bigcup_{i \in [1 \ldots n]} P_i$, an interaction $a$ is any finite subset of $P$ such that:

- every port $p_i \in a$ belongs to a component $B_i$ of the system;
- every component $B_i$ participates in the interaction $a$ with exactly one port, i.e. $P_i \cap a \neq \emptyset$ and if $p_i \in a$ and $p_j \in a$, then $B_i \neq B_j$ for $i, j \in [1 \ldots n]$.

The set of all interactions $I(P)$ is a subset of $2^P$.

Given a set of components $B_1 \ldots B_n$ and the set of interactions $A_\gamma$ allowed for the configuration $\gamma = \{s_1, \ldots, s_n\}$, we can define a system $A_\gamma(B_1, \ldots, B_n)$ using the following operational semantics rule:

$$\frac{a \in A_\gamma \qquad \forall p \in a \cap P_i : s_i \xrightarrow{p} s_i'}{\{s_i\}_{[1 \ldots n]} \xrightarrow{a} \{s_i'\}_{[1 \ldots n]}} \tag{2}$$

where $s_i$ is the current control location of component $B_i$, and $a$ is an interaction containing exactly one port for each component $B_i$. Components $B_j$ not "actively" involved in the interaction will participate with their idle port $idle_j$ and their state will be unchanged, i.e. will have $s_j' = s_j$.

### 2.2 Propositional Interaction Logic (PIL)

Let $\mathcal{P}$ and $\mathcal{S}$ be, respectively, the domains of ports and control locations. Furthermore, let $\Gamma$ be the set of all system configurations. The formulas of Propositional Interaction Logic PIL($\mathcal{P}$, $\mathcal{S}$) are defined by the following syntax:

$$\text{(PIL formula)} \quad \Psi ::= p \in \mathcal{P} \mid \pi \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \tag{3}$$

where $\pi : \Gamma \mapsto \{\mathbf{true}, \mathbf{false}\}$ is a *state predicate*. We use logical connectives $\vee$ and $\Rightarrow$ with the usual meaning.

The models of the logic are interactions on $\mathcal{P}$ for a configuration $\gamma$. The semantics is defined by the following satisfaction relation $\models_\gamma$ between an interaction $a$ and a PIL formula:

$$\begin{array}{ll} a \models_\gamma \mathbf{true} & \text{for any } a \\ a \models_\gamma p & \text{if } p \in a \\ a \models_\gamma \pi & \text{if } \pi(\gamma) = \mathbf{true} \\ a \models_\gamma \Psi_1 \wedge \Psi_2 & \text{if } a \models_\gamma \Psi_1 \wedge a \models_\gamma \Psi_2 \\ a \models_\gamma \neg\Psi & \text{if } a \not\models_\gamma \Psi \end{array} \tag{4}$$

A monomial $\bigwedge_{p\in I} p \wedge \bigwedge_{p\in J} \neg p$ with $I \cap J = \emptyset$ characterizes a set of interactions $a$ such that:

1. the positive terms correspond to *required ports* for the interaction to occur;
2. the negative terms correspond to *inhibited ports* to which the interaction is "closed";
3. the non-occurring terms are *optional ports*.

When the set of optional ports is empty, then the monomial is a single interaction, and it is characterized by $\bigwedge_{p\in a} p \wedge \bigwedge_{p\notin a} \neg p$.

Note that *idle* ports of components can appear in PIL formulas. Given a component with active ports $P^*$ and idle port *idle*, the following equivalences for PIL formulas hold:

$$idle \equiv \bigwedge_{p\in P^*} \neg p \qquad \neg idle \equiv \bigvee_{p\in P^*} p$$

Since we can describe sets of interactions using PIL formulas, we can redefine the operational semantics rule (2) as follows:

$$\frac{a \models_\gamma \Psi \qquad \forall p \in a : s_i \xrightarrow{p} s_i'}{\{s_i\}_{[1\ldots n]} \xrightarrow{a} \{s_i'\}_{[1\ldots n]}} \tag{5}$$

where $\Psi$ is a PIL formula.

### 2.3 Disjunctive versus conjunctive specification style

In [9] it is shown how to define a function $\beta : I(P) \to PIL(P)$ which associates a characteristic PIL formula $\beta(a)$ to an interaction $a$.

For example, if $P = \{p, q, r, s, t\}$ then for the interaction $\{p, q\}$, we have $\beta(\{p, q\}) = p \wedge q \wedge \neg r \wedge \neg s \wedge \neg t$[1]. For the set of interactions $A$ modelling the broadcast of $p$ to ports $q$ and $r$, we have $\beta(A) = p\neg s\neg t$. For the set of interactions $A$ consisting of the singleton interactions $p$ and $q$, we have $\beta(A) = (p\neg q \vee \neg pq) \wedge \neg r\neg s\neg t$.

Please notice that the definition of function $\beta$ requires knowing $P$ and that this function can be naturally extended to sets of interactions $A = \{a_1, \ldots, a_n\}$; in this case we have $\beta(A) = \beta(a_1) \vee \cdots \vee \beta(a_n)$.

A set of interactions is specified in *disjunctive style* if it is described by a PIL formula which is a disjunction of monomials. A dual style of specification is the *conjunctive style* where the interactions of a system are the conjunction of PIL formulas. A methodology for writing conjunctive specifications proposed in [9] considers that each term of the conjunction is a formula of the form $p \Rightarrow \Psi_p$, where the implication is interpreted as a causality relation: for $p$ to be true, it is necessary that the formula $\Psi_p$ holds, and this defines interaction patterns of other components in which the port $p$ needs to be involved.

For example, the interaction involving a strong synchronization between $p_1$, $p_2$ and $p_3$ is defined by the formula $f_1 = (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_3) \wedge (p_3 \Rightarrow p_1)$. Broadcast from a sending port $t$ towards receiving ports $r_1, r_2$ is defined by the formula $f_2 = (\mathbf{true} \Rightarrow t) \wedge (r_1 \Rightarrow t) \wedge (r_2 \Rightarrow t)$. The non-empty solutions are the interactions $\{t\}$, $\{t, r_1\}$, $\{t, r_2\}$ and $\{t, r_1, r_2\}$.

Notice that, by applying this methodology, we can associate to a component, with $P^*$ as set of active ports, a constraint $\bigwedge_{p\in P^*} (p \Rightarrow \Psi_p)$ that characterizes the set of interactions where some non-idle port of the component may be involved. Thus, if a system consists of components $B_1, \ldots, B_n$ with sets of active ports $P_1^*, \ldots, P_n^*$ respectively, then the PIL formula $\bigwedge_{i\in[1,n]} \bigwedge_{p\in P_i^*} (p \Rightarrow \Psi_p)$ expresses a global interaction constraint. Such a constraint can be put in a disjunctive form where monomials characterize global interactions. Notice that the disjunctive form obtained in this manner contains the monomial $\bigwedge_{p\in P^*} \neg p$, where $P^* = \bigcup_{i\in[1\ldots n]} P_i^*$, which is satisfied by the interaction where every component performs the *idle* action. This trivial remark shows that in the PIL framework it is possible to express interaction constraints of each component separately

[1] For the sake of conciseness, from now on we will omit the conjunction operator on monomials.

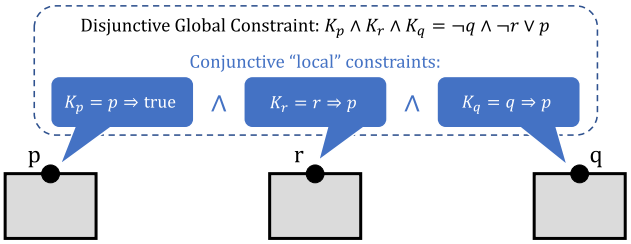
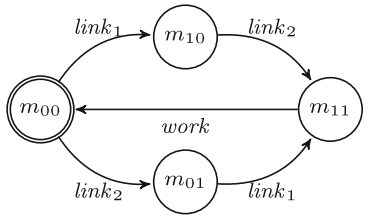
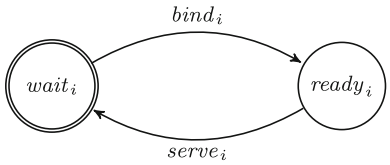



**Fig. 2** Broadcast example: disjunctive versus conjunctive specification




**(a)** The *master* component




**(b)** The $slave_i$ component

**Fig. 3** *master* and $slave_i$ components

and compose them conjunctively to get global disjunctive constraints.

It is also possible to put in conjunctive style a disjunctive formula $\Psi$ specifying the interactions of a system with set of active ports $P^*$. To translate $\Psi$ into a form $\bigwedge_{p \in P^*} (p \Rightarrow \Psi_p)$ we just need to choose $\Psi_p = \Psi\,[p = \textbf{true}]$ obtained from $\Psi$ by substituting **true** to $p$. Given the inherent property of supporting the *idle* interaction, the translated conjunctive formula will be equivalent to $\Psi$ only if the latter allows global idling. Consider broadcasting from port $p$ to ports $q$ and $r$ (Fig. 2). The possible interactions are characterized by the active ports $p$, $pq$, $pr$, $pqr$ and $\emptyset$ (i.e. idling). The disjunctive-style formula is: $\neg p \neg q \neg r \vee p \neg q \neg r \vee pq \neg r \vee p \neg q r \vee pqr = \neg q \neg r \vee p$. The equivalent conjunctive formula is: $(q \Rightarrow p) \wedge (r \Rightarrow p)$ that simply expresses the causal dependency of ports $q$ and $r$ from $p$.

The example below illustrates the application of the two specification styles.

***Example 1*** (Master–Slaves) Let us consider a simple system consisting of three components: *master*, $slave_1$ and $slave_2$. The *master* performs two sequential requests to $slave_1$ and $slave_2$, and then performs some computation with them.

Figure 3 provides a graphical representation of the activities of such components.

The set of allowed interactions $\gamma$ for the set of components $\{master, slave_1, slave_2\}$ can be represented by the following PIL formula using the disjunctive style, where we let $idle_{s_i}$ denote the idle port of component $slave_i$:

$$\Psi_{disj} = \big(link_1 \wedge bind_1 \wedge idle_{s_2}\big) \vee \big(link_2 \wedge bind_2 \wedge idle_{s_1}\big) \vee (work \wedge serve_1 \wedge serve_2)$$

Alternatively, the same interaction patterns can be modelled using the conjunctive style:

$$\begin{aligned}\Psi_{conj} = {} & (link_1 \Rightarrow bind_1) \wedge (link_2 \Rightarrow bind_2) \wedge \\ & (bind_1 \Rightarrow link_1) \wedge (bind_2 \Rightarrow link_2) \wedge \\ & (work \Rightarrow serve_1 \wedge serve_2) \wedge (serve_1 \Rightarrow work) \wedge \\ & (serve_2 \Rightarrow work)\end{aligned}$$

The two formulas differ in the admissibility of the "no-interaction" interaction. That is, the conjunctive formula $\Psi_{conj}$ allows all the components to not interact by performing a transition over their *idle* ports, while $\Psi_{disj}$ does not. To allow it in the disjunctive case, we could instead consider the following formula, where $idle_m$ denotes the idle port of component *master*:

$$\Psi'_{disj} = \Psi_{disj} \vee idle_m \wedge idle_{s_1} \wedge idle_{s_2}$$

## 3 Static architectures with transfer of values: the PILOps coordination language

We expand the PIL framework by introducing data exchange between components. In order to do so, the definition of component will be extended with local variables and the coordination constraints will be expressed with PILOps, which expands PIL with a notation that is inspired by guarded commands. Finally, we extend the definitions for disjunctive and conjunctive styles and study possible connections between the two.

### 3.1 PILOps components

**Definition 2** (PILOps Component) Let $\mathcal{S}$ be the set of all component control locations, $\mathcal{X}$ the set of all local variables, and $\mathcal{P}$ the set of all ports. A component is a transition system $B{:=}(S, X, P, T)$, where:

- $S \subseteq \mathcal{S}$: finite set of control locations;
- $X \subseteq \mathcal{X}$: finite set of local variables;
- $P \subseteq \mathcal{P}$: finite set of ports;
- $T \subseteq S \times P \times S$: finite set of transitions; $p \in P$ is the port offered for interaction, and each transition is labelled by a different port.

Each component has one implicit loop transition $\{s \xrightarrow{idle} s\}_{s \in S}$ for each control location $s \in S$ over a dedicated port *idle*. $P^* = P \setminus \{idle\}$ will be used to denote the set of active ports of a component.

It is assumed that the sets of ports, local variables and control locations of different components are disjoint. Each transition $(s, p, s')$ can also be denoted by $s \xrightarrow{p} s'$.

A system specification includes a set of PILOps components $B_i = (S_i, X_i, P_i, T_i)$ for $i = [1, n]$. The *configuration* $\gamma$ of a system is still described by the set of the current control locations of each constituent component, but now it also includes the *valuation function* $\sigma : \mathcal{X} \mapsto \mathsf{V}$ which maps local variables to values:

$$\gamma = \left(\{s_i \in S_i\}_{[1..n]}, \sigma\right) \tag{6}$$

We will use $\Gamma$ and $\Sigma$ to denote the set of all configurations and the domain of valuation functions, respectively. Interactions are still sets of ports belonging to different components.

## 3.2 Propositional interaction logic with operations (PILOps)

Let $\mathcal{P}$, $\mathcal{X}$ and $\mathcal{S}$, respectively, be the domains of ports, local variables and control locations.

The terms of PILOps$(\mathcal{P}, \mathcal{X}, \mathcal{S})$ are defined by the following syntax:

$$\begin{array}{rl} \text{(PILOps term)} & \Phi ::= \Psi \to \Delta \mid \Phi_1 \,\&\, \Phi_2 \mid \Phi_1 \parallel \Phi_2 \\ \text{(PIL formula)} & \Psi ::= p \in \mathcal{P} \mid \pi \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \\ \text{(set of ops.)} & \Delta ::= \emptyset \mid \{\delta\} \mid \Delta_1 \cup \Delta_2 \end{array} \tag{7}$$

where:

- operators & and $\parallel$ are *associative* and *commutative*, with & having higher precedence than $\parallel$;
- $\pi : \Gamma \mapsto \{\mathbf{true}, \mathbf{false}\}$ is a state predicate;
- $\delta : \Sigma \mapsto \Sigma$ is an operation that transforms a valuation functions $\sigma \in \Sigma$.

The models of the logic are still interactions $a$ on $\mathcal{P}$, where the satisfaction relation is defined by the set of rules (4) for PIL with the following extension:

$$\begin{array}{ll} a \models_\gamma \Psi \to \Delta & \text{if } a \models_\gamma \Psi \\ a \models_\gamma \Phi_1 \,\&\, \Phi_2 & \text{if } a \models_\gamma \Phi_1 \wedge a \models_\gamma \Phi_2 \\ a \models_\gamma \Phi_1 \parallel \Phi_2 & \text{if } a \models_\gamma \Phi_1 \vee a \models_\gamma \Phi_2 \end{array} \tag{8}$$

In other words, the conjunction and disjunction operators & and $\parallel$ for PILOps terms are equivalent to the logical $\wedge$ and $\vee$ from the interaction semantics perspective.

Operations in $\Delta$ are treated in a different way: operations associated with rules combined with "&" will be either performed all together if the associated PIL formulas hold for $a, \gamma$ or not at all if at least one formula does not, while for rules combined with the "$\parallel$" operator a largest union of operations satisfying the PIL formulas will be executed.

We indicate the set of operations to be performed for $\Phi$ under $a, \gamma$ as $[\![\Phi]\!]_{a,\gamma}$, which is defined according to the following rules:

$$[\![\Psi \to \Delta]\!]_{a,\gamma} = \begin{cases} \Delta & \text{if } a \models_\gamma \Psi \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\Phi_1 \,\&\, \Phi_2]\!]_{a,\gamma} = \begin{cases} [\![\Phi_1]\!]_{a,\gamma} \cup [\![\Phi_2]\!]_{a,\gamma} & \text{if } a \models_\gamma \Phi_1 \wedge a \models_\gamma \Phi_2 \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\Phi_1 \parallel \Phi_2]\!]_{a,\gamma} = [\![\Phi_1]\!]_{a,\gamma} \cup [\![\Phi_2]\!]_{a,\gamma} \tag{9}$$

Rules (9) allow to define a notion of *semantic equivalence* between PILOps terms:

$$\Phi_1 = \Phi_2 \text{ iff } [\![\Phi_1]\!]_{a,\gamma} = [\![\Phi_2]\!]_{a,\gamma} \text{ for any } a, \gamma \tag{10}$$

### 3.2.1 Axioms for PILOps

Given the equivalence relation (10), the following axioms hold for PILOps terms:

$$\& \text{ is associative, commutative and idempotent} \tag{11}$$

$$\Psi_1 \to \Delta_1 \,\&\, \Psi_2 \to \Delta_2 = \Psi_1 \wedge \Psi_2 \to \Delta_1 \cup \Delta_2 \tag{12}$$

$$\Phi \,\&\, \mathbf{true} \to \emptyset = \Phi \tag{13}$$

$$\parallel \text{ is associative, commutative and idempotent} \tag{14}$$

$$\Psi_1 \to \Delta \parallel \Psi_2 \to \Delta = \Psi_1 \vee \Psi_2 \to \Delta \tag{15}$$

$$\Psi \to \Delta_1 \parallel \Psi \to \Delta_2 = \Psi \to \Delta_1 \cup \Delta_2 \tag{16}$$

$$\mathbf{false} \to \Delta \parallel \Phi = \Phi \tag{17}$$

$$\text{Absorption: } \Phi_1 \parallel \Phi_2 = \Phi_1 \parallel \Phi_2 \parallel (\Phi_1 \,\&\, \Phi_2) \tag{18}$$

$$\text{Distributivity:} \tag{19}$$

$$\Phi \,\&\, (\Phi_1 \parallel \Phi_2) = (\Phi \,\&\, \Phi_1) \parallel (\Phi \,\&\, \Phi_2)$$

$$\text{Normal disjunctive form (DNF):}$$

$$\Psi_1 \to \Delta_1 \parallel \Psi_2 \to \Delta_2 = \Psi_1 \wedge \neg\Psi_2 \to \Delta_1 \parallel \Psi_2 \wedge \neg\Psi_1 \to \Delta_2 \parallel \Psi_1 \wedge \Psi_2 \to \Delta_1 \cup \Delta_2 \tag{20}$$

Note that PILOps strictly contains PIL as a formula $\Psi$ can be represented by $\Psi \to \emptyset$. The operation & is the extension of conjunction with neutral element $\mathbf{true} \to \emptyset$ and $\parallel$ is the extension of the disjunction with an absorption (18) and distributivity axiom (19). The DNF is obtained by application of the axioms. Note some important differences with PIL: the usual absorption axioms for disjunction and conjunction are

replaced by a single absorption axiom (18) and there is no conjunctive normal form.

#### 3.2.2 Operations

Operations $\delta$ in PILOps are assignments on local variables of components involved in an interaction of the form $x{:=}f\,(y_1,\ldots,y_k)$, where $x \in \mathcal{X}$ is the local variable subject to the assignment and $f : \mathsf{V}^k \mapsto \mathsf{V}$, is a function on local variables $y_1,\ldots,y_k$ ($y_i \in \mathcal{X}$) on which the assigned value depends.

We can define the semantics of the application of the assignment $x{:=}f$ to the valuation function $\sigma$ as:

$$\delta(\sigma) = (x{:=}f)\,(\sigma) = \sigma\left[x \mapsto f\left(\sigma\left(y_1\right),\ldots,\sigma\left(y_k\right)\right)\right] \tag{21}$$

The application of a set of operations $\Delta$ to a valuation function $\sigma$ produces a set of functions $\left\{\sigma'_j\right\}_{[1\ldots m]} \subseteq \Sigma$. Each function $\sigma'_j$ is the result of the application of an operation $\delta'_j$ to $\sigma$, where $\delta'_j$ is obtained by the *composition* of all $\delta_i \in \Delta$ in a given order.

Formally, given $\Delta = \{\delta_i\}_{[1\ldots n]}$, let the symmetric group $\mathfrak{S}_\Delta$ of all the $n!$ permutations on the set of operations $\Delta$ be:

$$\mathfrak{S}_\Delta = \left\{\left(\alpha_j\left(\delta_1\right)\cdots\alpha_j\left(\delta_n\right)\right)\right\}_{[1\ldots n!]}$$

where $\alpha_j\left(\delta_i\right)$ is the $j$th permutation of operation $\delta_i$. Then, the application $\Delta\left(\sigma\right)$ produces the set $\left\{\sigma'_j\right\}_{[1\ldots n!]}$ as follows:

$$\Delta\left(\sigma\right) = \left\{\left(\alpha_j\left(\delta_1\right)\circ\cdots\circ\alpha_j\left(\delta_n\right)\right)\left(\sigma\right)\right\}_{j\in[1\ldots n!]} \tag{22}$$

where:

- $\left(\alpha_j\left(\delta_1\right)\cdots\alpha_j\left(\delta_n\right)\right) \in \mathfrak{S}_\Delta$ for $j \in [1\ldots n!]$;
- the operator $\circ$ is the function composition (e.g. $f \circ g\left(x\right) = f\left(g\left(x\right)\right)$).

Having introduced the extended syntax of PILOps and formally characterized the semantics of the operations, we can define the operational semantics of the language by appropriately expanding rule (5).

A PILOps term $\Phi$ is a coordination mechanism that, applied to a set of components $B_1\ldots B_n$, gives a system defined by the following rule:

$$\frac{a \models_\gamma \Phi \qquad \forall p \in a : s_i \xrightarrow{p} s'_i \qquad \sigma' \in [\![\Phi]\!]_{a,\gamma}\left(\sigma\right)}{\left(\{s_i\}_{[1\ldots n]},\sigma\right) \xrightarrow{a} \left(\left\{s'_i\right\}_{[1\ldots n]},\sigma'\right)} \tag{23}$$

where $[\![\Phi]\!]_{a,\gamma}\left(\sigma\right)$ is the set of valuation functions obtained according to (22) by applying the operations $\Delta = [\![\Phi]\!]_{a,\gamma}$ to the valuation function $\sigma$.

Notice that the valuation function $\sigma'$ is selected with no specific criterion, therefore the effects of multiple operations on system configurations are generally *non-deterministic*. This behaviour can be mitigated by the adoption of a *snapshot semantics* for the valuation function update (i.e. every local variable is evaluated using the same initial valuation function), which prevents the inversion of access/update operations from producing different outcomes. This limits non-determinism in PILOps to the application of multiple assignments on the same local variable.

### 3.3 Disjunctive versus conjunctive specification style in PILOps

To define how the conjunctive specification style introduced in Sect. 2.3 can be extended to PILOps, we associate with $p \Rightarrow \Psi_p$ an operation $\Delta_p$ to be performed when an interaction involving $p$ is executed according to this rule. We call the PILOps term describing this behaviour the *conjunctive term*:

$$p \rhd \Psi_p \to \Delta_p = \left(\neg p \to \emptyset \parallel p \wedge \Psi_p \to \Delta_p\right) \tag{24}$$

We will refer to $p$ in a conjunctive term as the *dependent* port and to $\Psi_p$ as the *dependency*.

The conjunction of terms of this form through the operator & gives a disjunctive-style formula. Consider for instance, the conjunction of two terms:

$$\begin{aligned}
&\left(p \rhd \Psi_p \to \Delta_p\right) \,\&\, \left(q \rhd \Psi_q \to \Delta_q\right)\\
&= \left(\neg p \to \emptyset \parallel p \wedge \Psi_p \to \Delta_p\right) \& \left(\neg q \to \emptyset \parallel q \wedge \Psi_q \to \Delta_q\right)\\
&= \neg p \wedge \neg q \to \emptyset \parallel p \wedge \neg q \wedge \Psi_p \to \Delta_p \parallel\\
&\qquad q \wedge \neg p \wedge \Psi_q \to \Delta_q \parallel p \wedge q \wedge \Psi_p \wedge \Psi_q \to \Delta_p \cup \Delta_q
\end{aligned}$$

The disjunctive form obtained from the application of the distributivity axiom (19) is the union of four terms corresponding to the canonical monomials on $p$ and $q$ and leading to the execution of operation $\Delta_p$, $\Delta_q$, both or none. It is easy to see that for a set of ports $P$ the conjunctive form

$$\mathop{\&}_{p\in P}\left(\neg p \to \emptyset \parallel p \wedge \Psi_p \to \Delta_p\right)$$

is equivalent to the disjunctive form

$$\mathop{\Big\|}_{I\cup J=P}\left(\bigwedge_{i\in I} p_i \wedge \Psi_{p_i} \bigwedge_{j\in J} \neg p_j \to \bigcup_{i\in I} \Delta_{p_i}\right)$$

where $\bigcup_{i\in\emptyset}\Delta_{p_i} = \emptyset$.

The converse does not hold. Given a disjunctive specification it is not always possible to get an equivalent conjunctive one. If we have a term of the form $\|_{k\in K}\ \Psi \rightarrow \Delta_k$ over a set of ports $P$, it can be put in the canonical form, i.e. the union of canonical terms of the form $\bigwedge_{i\in I} p_i \bigwedge_{j\in J} \neg p_j \rightarrow \Delta_{IJ}$. It is easy to see that for this form to be obtained as a conjunction of causal terms, a sufficient condition is that for each port $p_i$ there exists an operation $\Delta_{p_i}$ such that $\Delta_{IJ} = \bigcup_{i\in I} \Delta_{p_i}$. This condition also determines the limits of the conjunctive/compositional approach. That is, in order to express a disjunctive term with a conjunctive equivalent, it must be possible to deconstruct the set of operations of the former term in subsets of operations associated with each dependent port of the latter.

***Example 2*** (Master–Slaves) Let us expand the example scenario introduced in Example 1 by including data transfer between the *master* component and the two $slave_1$ and $slave_2$ components. More specifically, we provide the *master* component with a *buffer* local variable that will store the sum of values in local variables $mem_1$ and $mem_2$ of the two respective slaves when they all synchronize through the ports $work, serve_1, serve_2$.

The set of allowed interactions $\gamma$ is not going to change, but adopting the PILOps coordination language, we can characterize the desired behaviour using the disjunctive style as follows:

$$\begin{aligned}\Phi_{disj} = {} & link_1 \wedge bind_1 \wedge idle_2 \rightarrow \emptyset \ \| \\ & link_2 \wedge bind_2 \wedge idle_1 \rightarrow \emptyset \ \| \\ & work \wedge serve_1 \wedge serve_2 \\ & \rightarrow buffer{:=}mem_1 + mem_2\end{aligned}$$

The conjunctive style version equivalent to $\Phi_{disj}$ (except for its allowance of the idling of all components) is the following:

$$\begin{aligned}\Phi_{conj} = {} & link_1 \rhd bind_1 \rightarrow \emptyset \,\&\, link_2 \rhd bind_2 \rightarrow \emptyset \,\& \\ & bind_1 \rhd link_1 \rightarrow \emptyset \,\&\, bind_2 \rhd link_2 \rightarrow \emptyset \,\& \\ & work \rhd serve_1 \wedge serve_2 \\ & \rightarrow buffer{:=}mem_1 + mem_2 \,\& \\ & serve_1 \rhd work \rightarrow \emptyset \,\&\, serve_2 \rhd work \rightarrow \emptyset\end{aligned}$$

## 4 The DReAM framework

In this section, we present the DReAM framework, which extends the static framework by introducing support for parametric system specifications and dynamic reconfiguration. In this context, components become instances of types of components and their number can dynamically change. Component instances are assigned to motifs, which describe independent dynamic architectures. Coordination between components in a motif, but also between the motifs constituting a system, is expressed by the DReAM coordination language, a first-order extension of PILOps. In motifs, coordination is parametrized by the notion of "map", which is an abstract relation used as a reference to model the topology of the underlying architecture and component mobility.

### 4.1 Component types and component instances

The basic elements of DReAM systems are *instances of component types*. Component types in DReAM correspond to PILOps components (see Definition 2) without the requirement that their sets of ports, control location and local variables be disjoint. Component instances are obtained from a component type by renaming its control locations, ports and local variables with a unique *identifier*.

To highlight the relationships between component types and their defining sets we use a "dot notation":

- $b.S$ refers to the set of control locations $S$ of component type $b$ (same for ports and variables);
- $b.s$ refers to the control location $s \in b.S$ (same for ports and variables).

**Definition 3** (*Component instance*) Let $\mathcal{C}$ be the domain of instance identifiers $c$ and $B = \langle b_1, \ldots, b_n \rangle$ be a tuple of component types where each element is $b_i = (S_i, X_i, P_i, T_i)$.

A set of *component instances* of type $b_i$ is represented by $b_i.C = \{b_i.c : c \in C\}$, for $1 \leq i \leq n$ and $C \subseteq \mathcal{C}$, and is obtained by renaming the set of control locations, ports and local variables of the component type $b_i$ with $c$, that is $b_i.c = (c.S_i, c.X_i, c.P_i, c.T_i)$. Without loss of generality, we assume that instance identifiers uniquely represent a component instance regardless of its type.

The state of a component instance $b.c$ is therefore defined as the pair $\langle c.s, c.\sigma \rangle$, where $c.\sigma$ is the *valuation function* of the variables $c.X$[2]. $\Sigma$ denotes the domain of all valuation functions, while we can refer to the set of all possible valuation functions for component instance $c$ with $c.\Sigma$. We use the same notation to denote ports, control locations, states and variables belonging to a given component instance (e.g. $c.p \in c.P$). Given that, by construction, each instance identifier is uniquely associated with one component instance, we have that sets of ports, control location and local variables of different component instances are still disjoint, i.e. $c.P \cap c'.P = \emptyset$ for $c \neq c'$.

Transitions for component instances $c.T$ are obtained from the respective component type transitions $T$ via port name substitution, i.e. via the rule:

[2] Notice that when writing, e.g. $c.s$, we are omitting the explicit reference to the component type $b$ and using a shorter notation compared to the complete one, e.g. $b.c.s$.

$$\frac{(s,\, p,\, s') \in T}{c.s \xrightarrow{c.p} c.s'} \tag{25}$$

## 4.2 Motif modelling

A motif characterizes an independent dynamic architecture involving a set of component instances $C$ subject to specific *coordination terms* parametrized by a specific data structure called *map*.

**Definition 4** (*Motif*) Let $\mathcal{C}$ be the domain of component instance identifiers. A *motif* is a tuple $m{:=}\langle C, \rho, Map, @\rangle$, where $C \subseteq \mathcal{C}$ is the set of component instances assigned to the motif, $\rho$ is the coordination term regulating interactions and reconfigurations among them, and $Map$, $@$ are the configurations of the map associated with the motif and of the address function.

We assume that each component instance is associated with exactly one motif, i.e. $m_1.C \cap m_2.C = \emptyset$.

A *Map* is a set of locations and a connectivity relation between them. It is the structure over which computation is distributed and defines a system of coordinates for components. It can represent a physical structure, such as a geographic map, or some conceptual structure, e.g. the cellular structure of a memory. Additionally, each location has a local memory that components can write to or read from. Formally, a map in DReAM is specified as a graph $Map = (N, E, \omega)$, where:

- $N$ is a set of nodes or locations (possibly infinite);
- $E$ is a set of (possibly directed) edges subset of $N \times N$ that defines the connectivity relation between nodes;
- $\omega : N \mapsto \mathsf{V}$ is a valuation function that associates nodes to values, realizing the map memory.

If the map memory is empty, then the only available information for a location is its name. Otherwise, the memory can be shared by different components and used for their coordination.

The relation $E$ defines a concept of neighbourhood for components, which is used in many applications to express coordination constraints or directions for moving components. When these additional topological relations are not needed and $E = \emptyset$, the map can be still used as a simple indexing structure.

Component instances $C$ in a motif and its map are related through the (partial) *address function* $@ : \mathcal{C} \to N$ binding each component in $C$ to a node $n \in N$ of the map.

As we mentioned, maps can be used to model a physical environment where components are moving. If the map is an array $N = \{(i, j) | i, j \in \mathsf{Integers}\} \times \{f, o\}$, the pairs $(i, j)$ represent coordinates and the symbols $f$ and $o$ stand, respectively, for free and obstacle. We can model the movement of $b$, such that $@\,(b) = ((i, j), f)$, to a position $(i + a, j + b)$ provided that there is a path from $(i, j)$ to $(i + a, j + b)$ consisting of free cells.

The *configuration* $\gamma_m$ of motif $m$ is represented by the tuple

$$\begin{aligned}\gamma_m &= \langle C_m.s, C_m.\sigma, Map_m, @_m\rangle \\ &\equiv \left\langle \{c.s\}_{c\in m.C}, \{c.\sigma\}_{c\in m.C}, m.Map, m.@\right\rangle\end{aligned} \tag{26}$$

By modifying the configuration of a motif we can model:

- *Component dynamism*: The set of component instances $C$ may change by creating/deleting or migrating components;
- *Map dynamism*: The set of nodes or/and the connectivity relation of a map may change. This is the case in particular when an autonomous component, e.g. a robot, explores an unknown environment and builds a model of it;
- *Mobility dynamism*: The address function $@$ changes to express mobility of components.

Different types of dynamism can be obtained as the combination of these three basic types. More details on how they can be implemented in DReAM will be discussed in Sect. 4.3.2 where *reconfiguration operations* are introduced.

## 4.3 The DReAM coordination language

The DReAM coordination language is essentially a first-order extension of PILOps where quantification over sets of components is introduced.

Given the domain of ports $\mathcal{P}$ and the set of all possible system configurations $\Gamma$, the DReAM coordination language is defined by the syntax:

$$\begin{aligned}
&\text{(DReAMterm)} && \rho ::= \Phi \mid D\{\Phi\} \mid \rho_1 \,\&\, \rho_2 \mid \rho_1 \parallel \rho_2 \\
&\text{(declaration)} && D ::= \forall c : m.b \mid \exists c : m.b \mid D_1, D_2 \\
&\text{(PILOps term)} && \Phi ::= \Psi \to \Delta \mid \Phi_1 \,\&\, \Phi_2 \mid \Phi_1 \parallel \Phi_2 \\
&\text{(PIL formula)} && \Psi ::= c.p \in \mathcal{P} \mid \pi \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \\
&\text{(set of ops.)} && \Delta ::= \emptyset \mid \{\delta\} \mid \Delta_1 \cup \Delta_2
\end{aligned} \tag{27}$$

where:

- *Declarations* define the context of the term by declaring quantified ($\forall|\exists$) component variables ($c$) associated with instances of a given type ($b$) belonging to a motif $m$;
- Operators & and $\parallel$ are the same as the ones introduced in (7) for PILOps;
- $\pi : \Gamma \mapsto \{\mathbf{true}, \mathbf{false}\}$ is a state predicate on a system configuration $\gamma \in \Gamma$;
- $\delta : \Gamma \mapsto \Gamma$ is an *operation* that transforms a system configuration $\gamma$ in another $\gamma'$, with $\gamma, \gamma' \in \Gamma$.

A DReAM coordination term is *well formed* if its PIL formulas and associated operations contain only component variables that are defined in its declarations. From now on, we will only consider well-formed terms.

Notice that operations have now a more general definition compared to (7), as they can alter the whole system configuration and not just valuation functions for local variables. This allows DReAM to handle *reconfiguration operations*, which will be discussed in more details in Sect. 4.3.2.

Given a system configuration, a coordination term can be translated to an equivalent PILOps term by performing a *declaration expansion* step, which expands the quantifiers and replaces component variables with actual components.

#### 4.3.1 Declaration expansion for coordination terms

Given that DReAM systems host a finite number of component instances, first-order logic quantifiers can be eliminated by enumerating every component instance of the type specified in each declaration. We thus define the *declaration expansion* $\langle\rho\rangle_\gamma$ of $\rho$ under configuration $\gamma$ via the following rules:

$$\begin{aligned}
&\langle\Phi\rangle_\gamma = \Phi \\
&\langle\rho_1 \,\&\, \rho_2\rangle_\gamma = \langle\rho_1\rangle_\gamma \,\&\, \langle\rho_2\rangle_\gamma \\
&\langle\rho_1 \parallel \rho_2\rangle_\gamma = \langle\rho_1\rangle_\gamma \parallel \langle\rho_2\rangle_\gamma \\
&\langle\forall c : m.b\{\Phi\}\rangle_\gamma = \mathop{\&}_{c^* \in m.b.C} \Phi\left[c^*/c\right] \\
&\langle\exists c : m.b\{\Phi\}\rangle_\gamma = \mathop{\Big\|}_{c^* \in m.b.C} \Phi\left[c^*/c\right] \\
&\langle D_1, D_2\{\Phi\}\rangle_\gamma = \left\langle D_1\{\langle D_2\{\Phi\}\rangle_\gamma\}\right\rangle_\gamma
\end{aligned} \tag{28}$$

where $m.b.C$ is the set of component instances of type $b$ in motif $m$, and $\left[c^*/c\right]$ is the substitution of the symbol $c$ with the actual identifier $c^*$ in the associated term.

For the sake of conciseness, (27) and (28) do not mention additional notations that can be used to ease declaration writing. For instance, explicit reference to the motif hosting the instance variable can be omitted when the term is associated with the motif itself. Similarly, if the associated PILOps term has to apply to groups of instances regardless of their type, the component type can be omitted in the declaration.

By applying (28), any term can be transformed into a PILOps term, whose semantics is defined in Sect. 3.2.

#### 4.3.2 Reconfiguration operations

In addition to the assignment operation introduced for PILOps in Sect. 3.2.2, DReAM can now support more complex reconfiguration operations which enable component, map, and mobility dynamism by allowing transformations of a motif configuration at runtime.
Component dynamism can be realized using the following statements:

- $create(b, n)$: creates an instance of type $b$ at node $n$ of the relevant map;
- $delete(c)$: deletes instance $c$.

Map dynamism can be realized using the following statements:

- $add(n)$: adds node $n$ to the relevant map;
- $remove(n)$: removes node $n$ from the relevant map, along with incident edges and components mapped to it;
- $add(n_1, n_2)$: adds edge $(n_1, n_2)$ to the relevant map;
- $remove(n_1, n_2)$: removes edge $(n_1, n_2)$ from the relevant map.

Mobility dynamism can be realized using the following statement:

- $move(c, n)$: changes the position of $c$ to node $n$ in the relevant map.

### 4.4 Operational semantics of motifs

As described in 4.2, motifs are equipped with terms $\rho$ of the coordination language which are used to compose the component instances assigned to them. Motifs can evolve from a configuration $\gamma_m$ to another $\gamma_m''$ by performing a transition labelled with the interaction $a$ and characterized by the application of the set of operations $[\![\langle\rho\rangle_{\gamma_m}]\!]_{a,\gamma_m}$ iff $a \models \langle\rho\rangle_{\gamma_m}$. Formally this is encoded by the following inference rule:

$$\frac{a \models_{\gamma_m} \langle\rho\rangle_{\gamma_m} \quad \gamma_m \xrightarrow{a} \gamma_m' \quad \gamma_m'' \in [\![\langle\rho\rangle_{\gamma_m}]\!]_{a,\gamma_m}\left(\gamma_m'\right)}{\gamma_m \xrightarrow{a} \gamma_m''} \tag{29}$$

where:

- $\gamma_m \xrightarrow{a} \gamma_m'$ expresses the capability of the motif to evolve to a new configuration through interaction $a$ according to the simple PIL semantics of (5). By expanding the motif configuration, we have indeed:

$$\frac{\forall c.p \in a : c.s \xrightarrow{c.p} c.s' \quad \text{with } c \in m.C}{\langle C_m.s, C_m.\sigma, Map_m, @_m\rangle \xrightarrow{a} \langle C_m.s', C_m.\sigma, Map_m, @_m\rangle} \tag{30}$$

- $[\![\langle\rho\rangle_{\gamma_m}]\!]_{a,\gamma_m}\left(\gamma_m'\right)$ is the set of motif configurations obtained according to (22) by applying the operations $\Delta = [\![\langle\rho\rangle_{\gamma_m}]\!]_{a,\gamma_m}$ to the motif configuration $\gamma_m'$.

The considerations in Sect. 3.2.2 about the non-deterministic effects of multiple operations to the valuation function of PILOps components also apply to motif configurations as the declaration expansion $\langle\rho\rangle_{\gamma_m}$ reduces DReAM terms to PILOps terms.

### 4.5 System-level operational semantics

**Definition 5** (DReAM *system*) Let $B$ be a tuple of component types and $M$ a set of motifs. A *DReAM system* is a tuple $\langle B, M, \mu, \gamma_0\rangle$ where $\mu$ is a *migration term* and $\gamma_0$ is the initial configuration of the system.

The migration term $\mu$ is a coordination term where the operations $\delta$ are of the form $migrate(c, m, n)$, which move a component instance $c$ to node $n$ in the map of motif $m$.

The global configuration of a DReAM system is simply the union of the configurations of the set of motifs $M$ that constitute it:

$$\gamma = \bigsqcup_{m\in M} \gamma_m = \left\langle \bigcup_m m.C.s, \bigcup_m m.C.\sigma, \bigcup_m m.Map, \bigcup_m m.@ \right\rangle \tag{31}$$

where we overloaded the semantics of the union operator to combine different maps in a bigger one characterized by the union of the sets of nodes, edges, and memory locations.

The system-level semantics is described by the following inference rule:

$$\frac{\gamma_m \overset{a_m}{\rightsquigarrow} \gamma'_m \ \text{ for } m\in M \qquad a \models_{\gamma'} \langle\mu\rangle_{\gamma'} \qquad \gamma'' \in [\![\langle\mu\rangle_{\gamma'}]\!]_{a,\gamma'}(\gamma')}{\gamma \xrightarrow{a} \gamma''} \tag{32}$$

where:

- $\gamma' = \bigsqcup_{m\in M} \gamma'_m$;
- $a_m \subseteq a$ is a subset of the global interaction $a$ containing only ports of component instances belonging to motif $m$.

By performing interaction $a$, each motif first evolves on its own according to its coordination term, and then the whole system changes configuration according to the migration term $\mu$.

## 5 An executable implementation of DReAM

The ongoing implementation of the DReAM framework involves two parts: a Java execution engine with an associated API and a domain-specific language (DSL) with an IDE for modelling in DReAM.

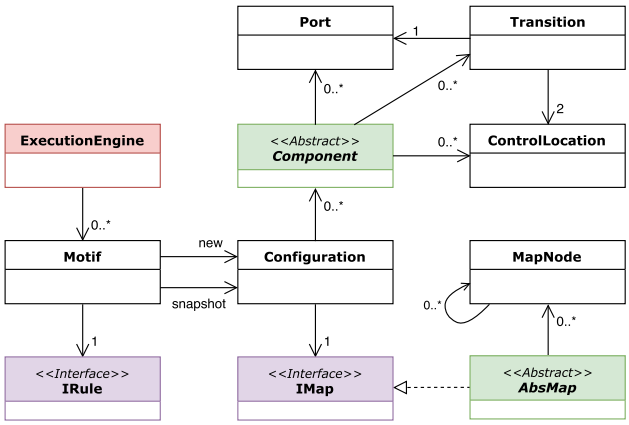



**Fig. 4** Simplified class diagram of the main software elements of the DReAM Java API

The centralized execution engine directly implements the DReAM operational semantics. Components and maps are defined as abstract classes that the programmer can extend with custom methods for which a library of predefined implementations is provided. Furthermore, by using directly the API, the programmer can enrich coordination terms and associated operations with any Java code. Figure 4 illustrates the main Java classes that realize the execution framework.

The `IRule` and `IMap` interfaces are implemented with classes representing DReAM terms and actual map implementations, respectively.

The DSL implements the abstract syntax of DReAM using XText, which also provides an integrated development environment as an Eclipse plug-in with convenient features like syntax highlighting and static checks.

Given the dynamic nature of the modelled systems and the importance of the study of collective behaviours, we are also realizing a pluggable component for the execution engine to visualize the evolution of DReAM system configurations.

We provide an abstract syntax of DReAM, for a system with a set of motifs $M$ (with their respective component instances) and migration term describing how components can leave a motif and join another.

**System** {
  $B = \{b_1, \ldots, b_k\}$ (the set of component types)
  $M = \{m_1, \ldots, m_n\}$ (the set of motifs)
  $\mu$ (migration term)
}
**Motif** $m_i$ {
  $Map_i$ (definition and associated functions/predicates)
  $\rho_i$ (coordination term)
}

Both migration and motif terms are expressions built using operators &, ‖ and the following "basic" terms:

$$\text{conjuctive term: } \forall c : b \in m,\ D\{c.p \rhd \phi_p \rightarrow \Delta_p\} \tag{33}$$

$$\text{disjunctive term: } D\{\phi \rightarrow \Delta_\phi\} \tag{34}$$

restriction term: $AtMost(n, b.p)$ (35)

where:

- $b \in B$ is a component type in the system;
- $D$ is a declaration as defined in (27);
- $\phi_p, \phi$ are PIL formulas;
- $\Delta_p, \Delta_\phi$ are sets of operations;
- $n \in \mathbb{N}$ is a integer;
- $p \in b.P^*$ is an active port of component type $b$.

The conjunctive term (33) matches the one defined for PILOps in Sect. 3.3. Its meaning is that any component instance $c$ of type $b$ belonging to motif $m$ interacts through port $p$ if $\phi_p$ holds, and the corresponding operation is $\Delta_p$.

The disjunctive term (34) is, in fact, a general DReAM coordination term. It characterizes all the interactions satisfying the formula $\phi$, and the corresponding operation is $\Delta_\phi$.

The restriction term (35) can be understood as a useful macronotation for a more complex coordination term forbidding all interactions that involve more than $n$ component instances of type $b$ interacting through port $p$. If no port $p$ is provided, then the restriction applies to every port of the component type $b$.

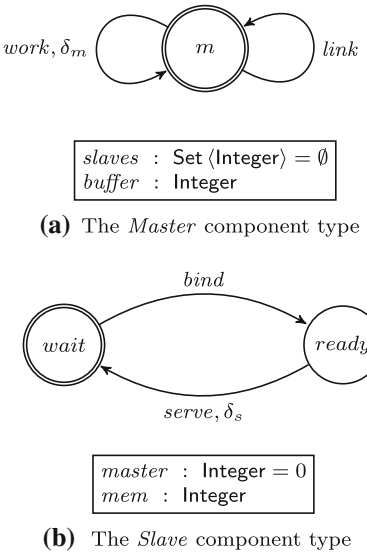
Migration terms are built from the given basic rules where operations $\Delta_p, \Delta_\phi$ involve only migration operations.

Coordination terms are built from the given basic rules where operations $\Delta_p, \Delta_\phi$ involve only assignment and reconfiguration operations. Since coordination terms are defined within the scope of a single motif, the reference to the motif $m$ itself can be omitted.

## 5.1 Applications and benchmarks

We will now present how some simple application scenarios can be modelled using the DReAM coordination language. For validation purposes and to show possible venues of analysis, the following examples have also been implemented and tested using the DReAM Java API.

***Example 3*** (Master–Slaves) Let us revisit the scenario of Example 2 using the DReAM coordination language. The first step is to generalize the components introduced in Example 1 to DReAM component types *Master* and *Slave* (Figure 5). In this case, component types must also provide appropriate local variables that will be used to store the instances to which they get connected to perform the task (i.e. the set of integers *slaves* for the *Master* type and the integer *master* for the *Slave* type). To restore these local variables to their initial value, we can associate operations $\delta_m = slaves{:=}\emptyset$ and $\delta_s = master{:=}0$ respectively with ports *work* and *serve*.




**(a)** The *Master* component type

**(b)** The *Slave* component type

**Fig. 5** *Master* and *Slave* component types

The system only requires the definition of a single motif with a trivial map characterized by a single node. The coordination term characterizing the desired interaction pattern can be expressed, for instance, using the conjunctive style as follows:

$$
\begin{aligned}
\rho = {} & AtMost(1, Master)\ \&\\
& AtMost(1, Slave.bind)\ \&\\
& AtMost(2, Slave.serve)\ \&\\
& \forall m : Master, \exists s : Slave\{\\
& \quad m.link \rhd \|m.slaves\| < 2 \wedge s.bind\\
& \quad \rightarrow m.slaves{:=}m.slaves \cup \{s\}\ \}\ \&\\
& \forall s : Slave, \exists m : Master\{\\
& \quad s.bind \rhd m.link \rightarrow s.master{:=}m\}\ \&\\
& \forall m : Master, \exists s_1, s_2 : Slave\{\\
& \quad m.work \rhd s_1 \neq s_2 \wedge \|m.slaves\| = 2 \wedge\\
& \quad s_1 \in m.slaves \wedge s_2 \in m.slaves \wedge s_1.serve \wedge\\
& \quad s_2.serve \rightarrow m.buffer{:=}s_1.mem + s_2.mem\}\ \&\\
& \forall s : Slave, \exists m : Master\{\\
& \quad s.serve \rhd s.master = m \wedge m.work \rightarrow \emptyset\}
\end{aligned}
$$

The system model composed by the motif $m$ characterized by $\rho$ and the component types $\{Master, Slave\}$ can then be initiated with an arbitrary number of component instances of the available types assigned to $m$. The resulting system will evolve through interactions that conform to the original description of Example 2, meaning that each component instance of type *Master* will connect with two different component instances of type *Slaves* (uniquely bound to that same instance of *Master*) and then they will synchronize to carry

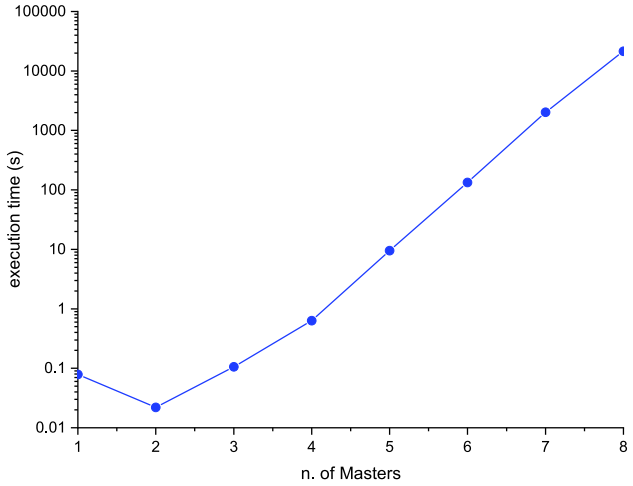



**Fig. 6** Runtime of 20 execution cycles of the implementation of Example 3

out the computation. Notice that the restriction terms guarantee that only one *Slave* instance at a time can connect to a single *Master* instance, and that no more than two *Slave* instances can participate in an interaction with port *serve*.

We used the DReAM Java API to implement the system described in Example 3 to study the performance of the execution engine when varying the number of component instances in the system. For this test we limited the number of execution cycles performed to 20, and we measured the runtime for systems characterized by 1 to 8 *Masters* and, respectively, 2 to 16 *Slaves*.

The results are illustrated in Fig. 6. The exponential growth in the runtime with the number of components is caused by the fact that the current implementation of the execution engine searches exhaustively over the set of all possible interactions collecting all the maximal ones, and then selects one at random.

***Example 4*** (Coordinating flocks of interacting robots) Consider a system with $N$ robots moving in a square grid, each one with given initial location and initial movement direction. Robots are equipped with a sensor that can detect other peers within a specific range $r$ and assess their direction: when this happens, the robot changes its own direction accordingly.

We require that robots maintain a timestamp of their last interaction with another peer: when two robots are within the range of their sensors, their direction is updated with the one having the highest timestamp. For the sake of simplicity we also assume that the grid is, in fact, a torus with no borders.

To model these robots in DReAM we will define a *Robot* component type as the one represented in Fig. 7.

Each *Robot* maintains a local *clock* that is incremented by 1 through an assignment statement in $\delta_{tick}$ every time an instance interacts with port *tick*.

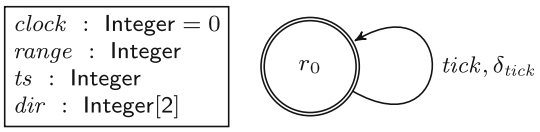



**Fig. 7** The *Robot* component type

A motif that realizes the described scenario can be defined with the conjunction of two coordination terms: one that enforces synchronization between every *Robot* instance through port *tick* allowing information exchange when possible, and another that enables all *Robot* instances performing a *tick* to move:

$$
\begin{aligned}
\rho = & \forall r : Robot\{r.tick \triangleright \textbf{true} \rightarrow move(r, @(r) + r.dir)\} \\
& \& \\
& \forall r_1, r_2 : Robot\{r_1.tick \triangleright r_2.tick \rightarrow \\
& \quad \text{IF } (r_1 \neq r_2) \text{ THEN } ( \\
& \quad\quad \text{IF } (distance(@(r_1), @(r_2)) < r_1.range \wedge \\
& \quad\quad\quad (r_1.ts < r_2.ts \vee (r_1.ts = r_2.ts \wedge r_1 < r_2)) \\
& \quad\quad ) \text{ THEN } ( \\
& \quad\quad\quad r_1.dir{:=}r_2.dir;\ r_1.ts{:=}r_1.clock; \\
& \quad\quad\quad r_2.ts{:=}r_2.clock))\}
\end{aligned}
$$

where:

- we use a map whose nodes are addressed via size-two integer arrays $[x, y]$;
- we are using a $distance(n_1, n_2)$ function that returns the euclidean distance between two points in an n-dimensional space;
- in the inequality $r_1 < r_2$ we use instance variables $r_1, r_2$ in place of their respective integer instance identifiers.

The coordination term $\rho$, which adopts the conjunctive style, can be intuitively understood breaking it into two parts:

1. every robot $r$ can interact with its port $r.tick$, and if it does it also moves according to its stored direction $r.dir$[3];
2. for every robot $r_1$ to interact with its port $r_1.tick$, every robot $r_2$ must also participate in interaction with its port $r_2.tick$ (i.e. interactions through port *tick* are strictly synchronous). Furthermore, for every pair of distinct $(r_1 \neq r_2)$ robots $r_1, r_2$ interacting through their respective *tick* ports: if they are closer than a given range $(r_1.range)$ and either $r_1$ has updated its direction less recently $(r_1.ts < r_2.ts)$ or they have updated their directions at the same time but $r_2$ has a higher instance

[3] Notice that if the direction of a robot is updated at a given time, the robot will move according to this new direction only during the next clock cycle because of the adopted snapshot semantics.

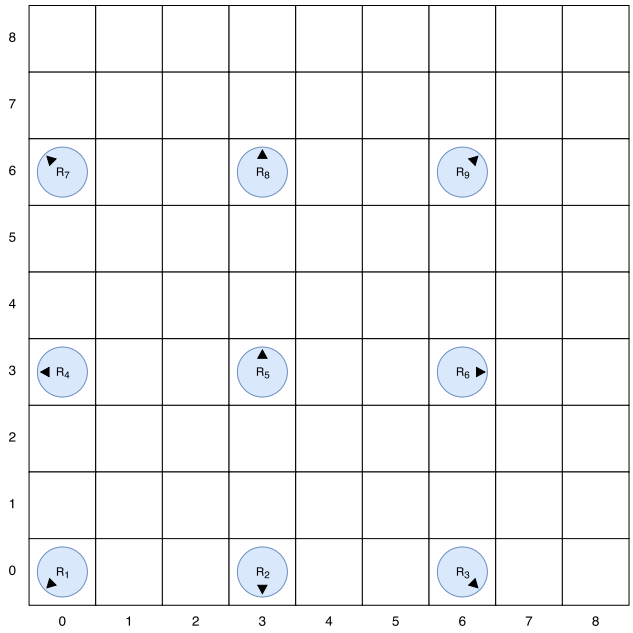



**Fig. 8** Initial system configuration for grid size $s = 9$

identifier ($r_1.ts = r_2.ts \wedge r_1 < r_2$)[4], then $r_1$ will update its direction and timestamp using $r_2$'s.

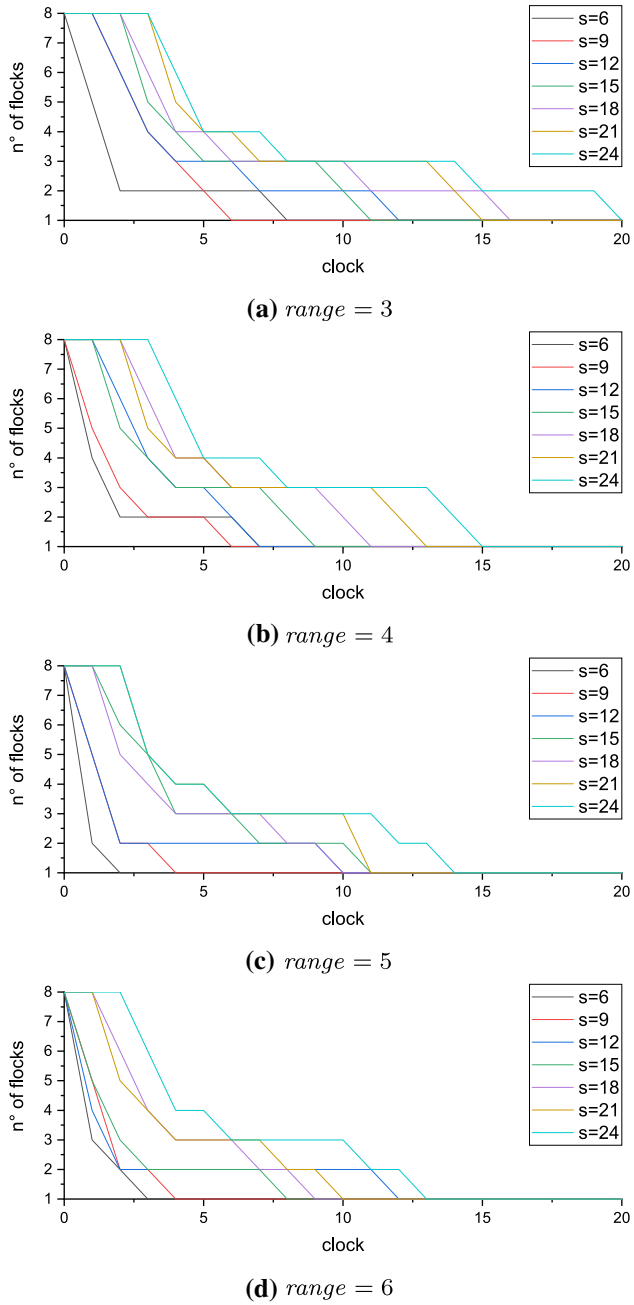
We used the DReAM Java API to implement the system described in Example 4 and study its behaviour while varying the size of the grid and the communication range for a fixed number of robots. Intuitively, we expect to observe a faster convergence in the movement directions as the size of the grid shrinks and/or as the communication range increases.

We fixed the number of robots in the system to 9, and we chose a specific initial direction for each one of them. We also chose the same *range* value for all robots. The mapping of the robots to a grid of size $s \times s$ is realized in such a way that they are uniformly spaced both horizontally and vertically. We chose grid sizes proportional to 3 for uniformity.

An example of the initial configuration for a grid of size $s = 9$ is shown in Fig. 8.

The graphs in Fig. 9 show the trend in the number of flocks (i.e. the number of groups formed by robots moving in the same direction) over time for different values of the given *range* of communication.

Indeed, the results confirm our expectations: the adopted initial setup procedure of the robot's positions and directions allows them to converge to an homogeneous flock within 20 clock ticks, a number which decreases as we increase the communication range. There is also an opposite trend




**(a)** $range = 3$

**(b)** $range = 4$

**(c)** $range = 5$

**(d)** $range = 6$

**Fig. 9** Trends in the number of flocks over time at different communication ranges

when increasing the size of the grid, although it is interesting to see that there are several exceptions to this rule (e.g. for $range = 3$ convergence on the grid $s = 6$ takes more time than on the grid $s = 9$; the same applies for $s = 12$ vs $s = 15$ and $s = 18$ vs $s = 21$).

***Example 5*** (Coordinating flocks of robots with stigmergy) We consider a variant of the previous problem by using stigmergy [10].

[4] Since all robots synchronize on the same "clock", many of them might update their respective directions differently at the same time: adding the "tiebreaker" on the instance identifier when timestamps are equal allows data exchange even in these cases.

Instead of letting robots sense each other, we will allow them to "mark" their locations with their direction and an associated timestamp. In this way, each time a robot moves to a node in the map it will either update its direction with the one stored in the node or update the one associated with the node with the direction of the robot (depending on whether the timestamp is higher than the last time the robot changed its direction or not).

The *Robot* component type represented in Fig. 7 can still be used without modifications (the *range* local variable will be ignored).

The coordination term associated with the motif becomes:

$$
\begin{aligned}
\rho' = \forall r : Robot \{ \\
& r.tick \rightarrow \\
& \text{IF } \big( @\,(r)\,.ts > r.ts \big) \text{ THEN } \big( \\
& \quad r.dir{:=}@\,(r)\,.dir;\ r.ts{:=}r.clock; \\
& \quad @\,(r)\,.ts{:=}r.clock \\
& \big) \text{ ELSE } \big( \\
& \quad @\,(r)\,.ts{:=}r.clock;\ @\,(r)\,.dir{:=}r.dir \big); \\
& move(r, @\,(r) + r.dir) \}
\end{aligned}
$$

Notice that we are now adopting a disjunctive-style specification for $\rho'$. We can interpret the term $\rho'$ as follows:

1. every robot $r$ must participate in all interactions with its port $r.tick$, and will move in the map according to its stored direction $r.dir$;
2. every robot $r$ either updates its direction with the one stored in the node $@\,(r)$ if the latter is more recent (i.e. if $@\,(r)\,.ts > r.ts$) or overwrites the direction stored in the node with its own otherwise.

Example 5 has also been implemented using the DReAM Java API. For a comparison with Example 4, we fixed the same parameters regarding number of robots, initial directions, set of tested grid sizes and mapping criterion.

We can reasonably expect a similar correlation between convergence time and grid size as in the case for communicating robots. Indeed, this is confirmed by the graph in Fig. 10, which shows the trends in the number of flocks for different grid sizes.

It is worth observing that convergence time and grid size are, again, not always directly proportional: here it is noticeable how the robots converge to a single flock for grid sizes equal to 15 and 21 in roughly half the time it takes for them to converge on the smaller grid with $s = 12$.

The graphs in Fig. 11 compare directly the convergence trends using the two approaches on grids of different sizes. From these we can appreciate how the stigmergy-based solution performs roughly on-par with the interaction-based one

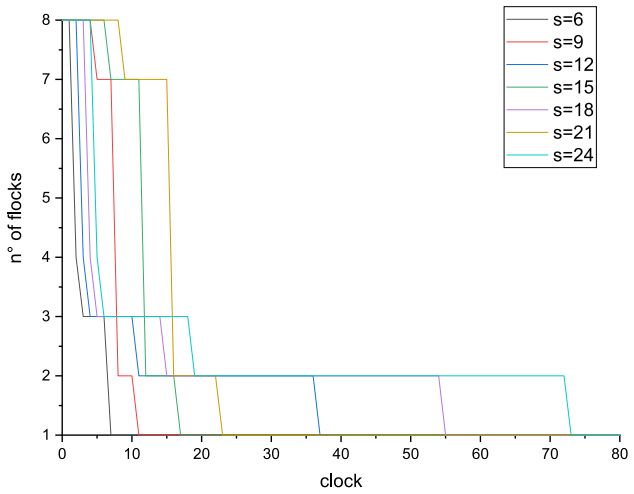



**Fig. 10** Evolution of the number of flocks over time at different communication ranges

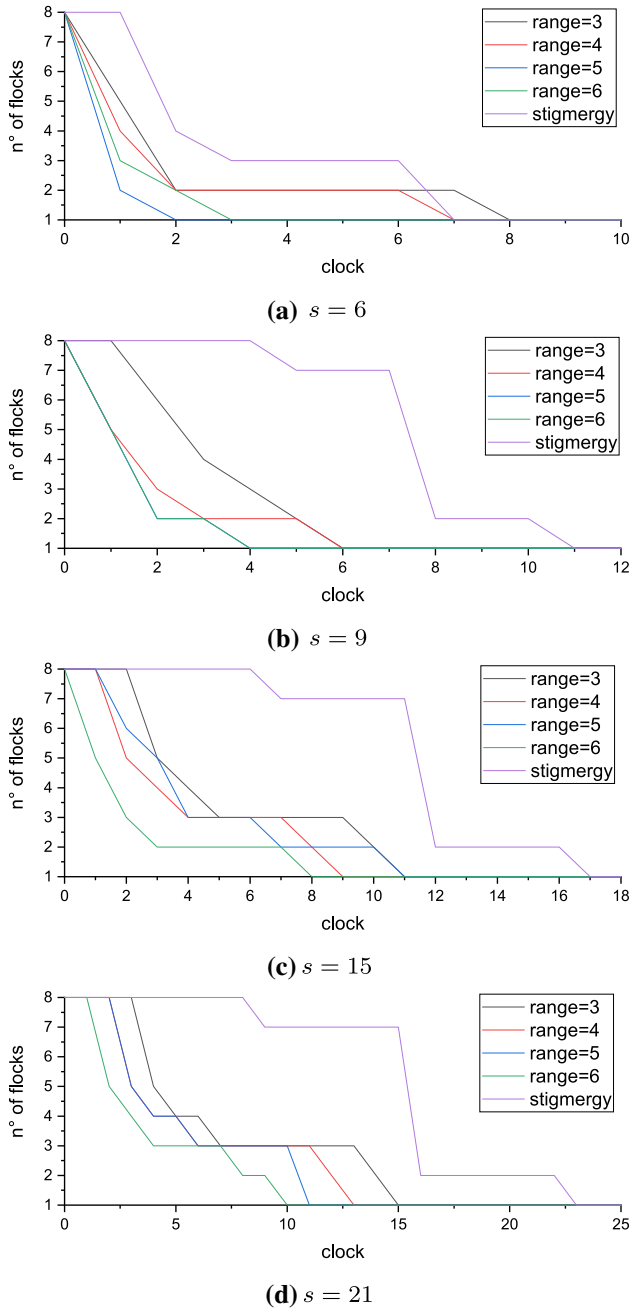
for small maps, progressively losing ground to the latter as the map becomes larger. This comparison also helps to better visualize how the implementation not resorting on sensors initially requires some time to populate the map with information which is proportional with the size of the map itself.

***Example 6*** (Reconfigurable ring) Consider a system of *Nodes* arranged in a *ring* topology where a passing token allows each *Node*, in turns, to communicate with the next. This rather simple coordination scheme is enriched with two dynamic elements characterizing the system:

1. new *Nodes* are created and added to the *ring* constantly, until it reaches a given size limit $N$;
2. *Nodes* can fail and get removed from the *ring* at a rate that increases with the *ring* size.

The *Node* component type is represented in Fig. 12. Each *Node* has two local variables: *payload*, that holds the next value to send, and *buffer*, that stores the last value received. The *payload* variable is initialized via the operation $\delta_{pl}$ associated with the transitions that change the control location of a *Node* from *empty* to *full*. The *out* and *in* ports are used intuitively to model the send and receive actions that a *Node* can perform with its neighbours in the *ring*, while the *init* port is used to handle more conveniently the bootstrapping of the system when all *Nodes* are in the initial *empty* control location.

The motif that will model the system needs to be equipped with an appropriate *Map* to represent the *ring* topology of interest, which is essentially a cyclic directed graph with just one cycle. The functions associated with this kind of *Map* will have to allow adding and removing *Map* nodes by also updating the set of edges in order to preserve the initial properties of the graph. Furthermore, we assume that initially




**(a)** $s = 6$

**(b)** $s = 9$

**(c)** $s = 15$

**(d)** $s = 21$

**Fig. 11** Comparison between the two approaches at different grid sizes

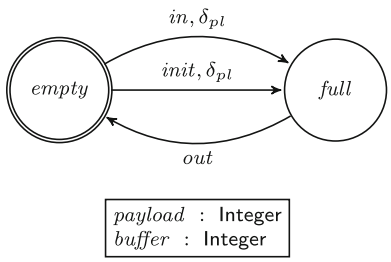



**Fig. 12** The *Node* component type

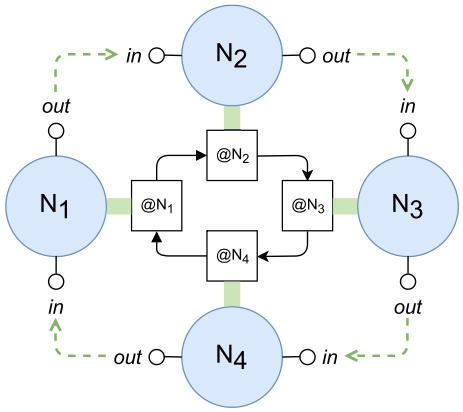



**Fig. 13** A representation of a *ring* with four *Node* instances and the corresponding *Map*

each *Node* instance has a one-to-one correspondence with a *Map* node. Figure 13 illustrates a graphical representation of the *ring* with four *Node* instances, where the data flow and relationship between component instances and locations of the *Map* are highlighted in green.

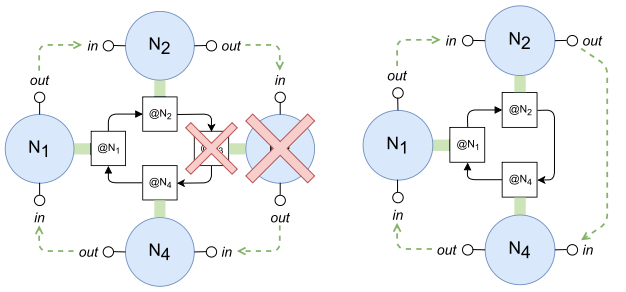
Having these ingredients, we can define a coordination term $\rho$ that realizes the described behaviour adopting the conjunctive style. This allows us to be compositional in our design process, so we will make use of this advantage and divide the problem of defining $\rho$ into three simpler sub-problems.

First, we will model how the *ring* grows and shrinks as new *Node* instances are created and removed. As we previously mentioned, we will design the system in such a way that one new *Node* is added to the *ring* as long as its size is less than $N$. At the same time, any *Node* instance in the system can terminate and get removed from the *ring* with probability $D\left(\frac{ring.size}{N}\right)^2$, where $D \in [0, 1]$. When this happens, the *ring* will automatically reconfigure to preserve the overall structure as illustrated in Fig. 14. To keep the example simple, we encode the information required to model this behaviour directly within the motif: the constants $N$ and $D$ will be statically defined within the coordination term $\rho$, and the current *ring* size will be obtained from a property of the *Map* (i.e. the number of *Map* nodes).

We can encode this behaviour with the following term:

$$
\begin{aligned}
\rho_1 = \; & (ring.size < N) \triangleright \mathbf{true} \\
& \quad \rightarrow create(Node, ring.newAddress) \\
& \& \\
& \forall n : Node \{ \\
& \quad rand < D\,(ring.size/N)^2 \triangleright \mathbf{true} \\
& \quad \rightarrow remove(@\,(n)) \;;\; delete(n) \}
\end{aligned}
$$

where:




**(a)** $N_3$ and the location @ $(N_3)$ are deleted

**(b)** and the *Map* reconfigures to preserve the *ring*

**Fig. 14** How the system handles *Node* deletion

– *ring.newAddress* is a function of the *Map* that creates a new location and returns its address;
– $rand \in [0, 1]$ is a random number.

Next, we will define how components interact and transfer data. Communication is binary between neighbouring *Nodes* in the *ring*: a *Node* $n_i$ can send data to a *Node* $n_j$ only if the edge $\left(@\left(n_i\right), @\left(n_j\right)\right)$ belongs to the *ring Map* (which we express with the predicate $@\left(n_i\right) \rightarrow @\left(n_j\right)$). Additionally, the $Node$ sending data and the one receiving it have to participate in the interaction with ports *out* and $in$, respectively. We can encode these constraints in the following way:

$$\begin{aligned}\rho_2 = \forall n_1, n_2 : Node\{ & \\ & n_1.out \triangleright @\left(n_1\right) \rightarrow @\left(n_2\right) \wedge n_2.in \rightarrow \emptyset\} \,\&\, \\ \forall n_1, n_2 : Node\{ & \\ & n_1.in \triangleright @\left(n_2\right) \rightarrow @\left(n_1\right) \wedge n_2.out \\ & \rightarrow n_1.buffer{:=}n_2.payload\}\end{aligned}$$

Lastly, we need to handle the transient situations when all *Nodes* are in the *empty* (initial) control location. We can model a system with a single passing token by defining a coordination term that allows just one *Node* instance to perform an *init* provided that every other *Node* instance is *empty* and stays *idle*:

$$\begin{aligned}\rho_3 = & \forall n_1, n_2 : Node\{ \\ & n_1.init \triangleright n_1 = n_2 \vee (n_2.empty \wedge n_2.idle) \rightarrow \emptyset\}\end{aligned}$$

This term will come into play after the creation of the very first *Node* instance in the *ring* and in the event that the *Node* instance holding the token gets removed from the *ring*.

The overall coordination term $\rho$ of the motif modelling the system will be the conjunction of the given terms:

$$\rho = \rho_1 \,\&\, \rho_2 \,\&\, \rho_3$$

**Table 1** Growth probability at varying ring sizes for $N = 20$ and $D = 0.5$

| Ring.size | Growth prob. (%) |
|---|---|
| 1 | 99.8750 |
| 2 | 99.0025 |
| 3 | 96.6628 |
| 4 | 92.2368 |
| 5 | 85.3215 |
| 6 | 75.8613 |
| 7 | 64.2465 |
| 8 | 51.3219 |
| 9 | 38.2605 |
| 10 | 26.3076 |
| 11 | 16.4656 |
| 12 | 9.2420 |
| 13 | 4.5731 |
| 14 | 1.9555 |
| 15 | 0.7058 |
| 16 | 0.2090 |
| 17 | 0.0490 |
| 18 | 0.0087 |
| 19 | 0.0011 |
| 20 | 0.0001 |

We executed Example 6 using the DReAM Java API. Like with Examples 4 and 5, we monitored a simple metric of the system to see how it behaves overall, but in this case we chose the size of the *ring* (i.e. the number of active *Node* instances). Choosing the system parameters $N = 20$ and $D = 0.5$, the probability that the *ring* will grow in size at each iteration can be easily computed as $p_g(n) = (1 - p_d\,(n))^n$, where $n$ is the current size of the *ring* and $p_d(n) = 0.5 \cdot (n/20)^2$ is the probability that at least one *Node* will get deleted. Table 1 displays the values of $p_g$ for $n$ ranging from 1 to $N$.

Figure 15 visualizes the results produced by the first 200 iterations of the execution engine. The graph shows a behaviour that is in line with what we could expect from the system with the given $N$ and $D$: the probability of the *ring* to steadily grow towards the size cap falls below 25% once the population of *Nodes* surpasses 10, and the likelihood that the growth is going to be negated by at least one *Node* instance getting deleted exceeds 95% when the size of the *ring* reaches 13.

## 6 Related work

DReAM allows both conjunctive and disjunctive-style modelling of dynamic reconfigurable systems. It inherits the expressiveness of the coordination mechanisms of BIP [9] as it directly encompasses multiparty interaction and extends previous work on modelling parametric architectures [11]

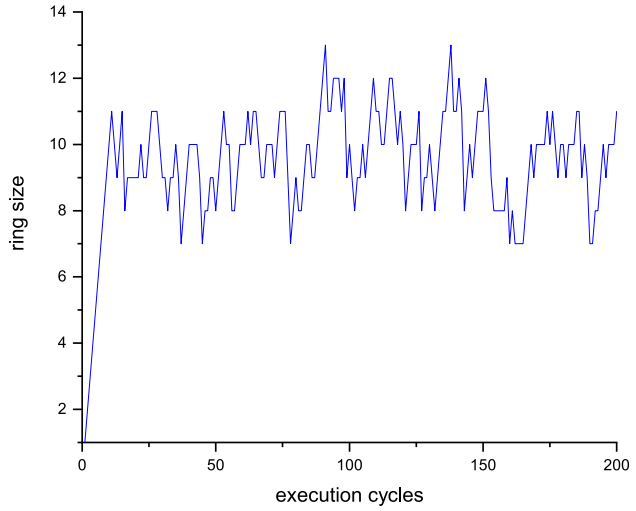



**Fig. 15** Evolution of the number of *Nodes* in the *ring*

in many respects. In DReAM interactions involve not only transfer of values but also encompass reconfiguration and self-organization by relying on the notions of maps and motifs.

When the disjunctive style is adopted, DReAM can be considered as an exogenous coordination language, e.g. an ADL. A comparison with the many ADL's is beyond the scope of the paper. Nonetheless, to the best of our knowledge DReAM surpasses existing exogenous coordination frameworks in that it offers a well-thought and methodologically complete set of primitives and concepts.

When conjunctive style is adopted, DReAM can be used as an endogenous coordination language comparable to process calculi to the extent they rely on a single associative parallel composition operator. In DReAM this operator is logical conjunction. It is easy to show that for existing process calculi parallel composition is a specialization of conjunction in Interaction Logic. For CCS [12] the causal rules are of the form $\mathbf{true} \Rightarrow p \wedge \mathbf{true} \Rightarrow \bar{p}$, where $p$ and $\bar{p}$ are input and output port names corresponding to port symbol $p$. In this context, strong synchronization can also be modelled without resorting to restriction by using causal rules like $p \Rightarrow \bar{p} \wedge \bar{p} \Rightarrow p$. For CSP [13], the interface parallel operator parametrized by the shared channel $a$ can be modelled in PIL by defining a set of ports $A$ implementing $a$ and using causal rules of the form $a_i \Rightarrow \bigwedge_{a_j \in A} a_j$ for all $a_i \in A$.

Also other richer calculi, such as $\pi$-calculus [14], that offer the possibility of modelling dynamic infrastructure via channel passing can be modelled in DReAM with its reconfiguration operations. Formalisms with richer communication models, such as AbC [15], offering multicast communications by selecting groups of partners according to predicates over their attributes, can also be rendered in DReAM. Attribute based interaction can be simulated by our interaction mechanism involving guards on the exchanged values and atomic transfer of values.

DReAM was designed with autonomy in mind. As such it has some similarities with languages for autonomous systems in particular robotic systems such as Buzz [10,16]. Our framework is more general as it does not rely on assumptions about the timed synchronous cyclic behaviour of components. Nonetheless, we are investigating the possibility of introducing the notion of time explicitly as this will be useful when specifying some types of dynamic systems.

The relationships between our approach and graph based architectural description languages such as ADR [17] and HDR [18] will be the subject of future work.

Finally, DReAM shares the same conceptual framework with DR-BIP [19]. The latter is an extension of BIP with component dynamism and reconfiguration. As such it adopts an exogenous and imperative approach based on the use of connectors. A detailed comparison between DReAM and DR-BIP will be the object of a forthcoming publication.

## 7 Discussion

We have proposed a framework for the description of dynamic reconfigurable systems supporting their incremental construction according to a hierarchy of structuring concepts going from components to sets of motifs forming a system. Such a hierarchy guarantees enhanced expressiveness and incremental modifiability thanks to the following features:

**Incremental modifiability of models at all levels**: The interaction rules associated with a component in a motif can be modified and composed independently. Components can be defined independently of the maps and their context of use in a motif. Self-organization can be modelled by combining motifs, i.e. system modes for which particular interaction rules hold.

**Expressiveness**: This is inherited from BIP as the possibility to directly specify any kind of static coordination without modifying the involved components or adding extra coordinating components. Regarding dynamic coordination, the proposed language directly encompasses the identified levels of dynamicity by supporting component types and the expressive power of first-order logic. Nonetheless, explicit handling of quantifiers is limited to declarations that link component names to coordinates.

**Abstract Semantics**: The language relies on an operational semantics that admits a variety of implementations between two extreme cases. One consists in pre-computing a global interaction constraint applied to an unstructured set of component instances and choosing the enabled interactions and the corresponding operations for a given configuration.

The other consists in computing separately interactions for motifs or groups of components and combining them.

The results about the relationship between conjunctive and disjunctive styles show that while they are both equally expressive for interactions without data transfer, the disjunctive style is more expressive when interactions involve data transfer. We plan to further investigate this relationship to characterize more precisely this limitation that seems to be inherent to modular specifications.

All results are still recent and many open research avenues need to be explored. The language and its tools should be evaluated against real-life mobile applications such as autonomous transport systems, swarm robotics or telecommunication systems.

## References


1. Garlan, D.: Software architecture: a travelogue. In: Proceedings of the on Future of Software Engineering, pp. 29–39. ACM (2014)
2. Taivalsaari, A., Mikkonen, T., Systä, K.: Liquid software manifesto: the era of multiple device ownership and its implications for software architecture. In: Proceedings of the 38th Computer Software and Applications Conference, pp. 338–343. IEEE (2014)
3. Bradbury, J.S.: Organizing definitions and formalisms for dynamic software architectures, Technical Report, vol. 477 (2004)
4. Oreizy, P., et al.: Issues in modeling and analyzing dynamic software architectures. In: Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis, pp. 54–57 (1998)
5. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**(6), 869–891 (2013)
6. 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: Pre-proceedings of 4th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering, p. 13 (2017)
7. Medvidovic, N., Dashofy, E.M., Taylor, R.N.: Moving architectural description from under the technology lamppost. Inf. Softw. Technol. **49**(1), 12–31 (2007)
8. De Nicola, R., Maggi, A., Sifakis, J.: Dream: dynamic reconfigurable architecture modeling. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems, pp. 13–31. Springer, Cham (2018)
9. Bliudze, S., Sifakis, J.: The algebra of connectors: structuring interaction in BIP. IEEE Trans. Comput. **57**(10), 1315–1330 (2008)
10. Pinciroli, C., Lee-Brown, A., Beltrame, G.: Buzz: An extensible programming language for self-organizing heterogeneous robot swarms. arXiv preprint arXiv:1507.05946 (2015)
11. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling dynamic architectures using Dy-BIP. In: Software Composition, pp. 1–16. Springer, Berlin (2012)
12. Milner, R.: A calculus of communicating systems (1980)
13. Brookes, S.D., Hoare, C.A., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31**(3), 560–599 (1984)
14. Milner, R., Parrow, J., Walker, D.: A Calculus Of Mobile Processes, I. Inf. Comput. **100**(1), 1–40 (1992)
15. Alrahman, Y. Abd, Nicola, R. De, Loreti, M.: On the power of attribute-based communication. In: Proceedings of the Formal Techniques for Distributed Objects, Components, and Systems—FORTE 2016—36th IFIP WG 6.1 International Conference
16. Pinciroli, C., Beltrame, G.: Buzz: an extensible programming language for heterogeneous swarm robotics. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2016, pp. 3794–3800, IEEE (2016)
17. Bruni, R., Lafuente, A. L., Montanari, U., Tuosto, E.: Style based reconfigurations of software architectures, Universita di Pisa, Tech. Rep. TR-07-17, (2007)
18. Bruni, R., Lluch-Lafuente, A., Montanari, U.: Hierarchical design rewriting with maude. Electron. Notes Theor. Comput. Sci. **238**(3), 45–62 (2009)
19. El Ballouli, R., Bensalem, S., Bozga, M., Sifakis, J.: Four exercises in programming dynamic reconfigurable systems: methodology and solution in DR-BIP. In: ISoLA 2018, vol. 11246. Springer (2018)