DReAM: Dynamic Reconfigurable Architecture Modeling

Rocco De Nicola[1(✉)], Alessandro Maggi[1(✉)], and Joseph Sifakis[2(✉)]

[1] IMT School for Advanced Studies Lucca, Lucca, Italy
{rocco.denicola,alessandro.maggi}@imtlucca.it
[2] Université Grenoble Alpes, Grenoble, France
Joseph.Sifakis@univ-genoble-alpes.fr

Abstract. Modern systems evolve in unpredictable environments and have to continuously adapt their behavior to changing conditions. The "DReAM" (Dynamic Reconfigurable Architecture Modeling) framework, has been designed for modeling reconfigurable dynamic systems. It provides a rule-based language, inspired from Interaction Logic, expressive and easy to use, and 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 to develop runnable systems combining the expressiveness of the rule-based notation together with the flexibility of this widespread programming language.

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, decentralised and interdependent, and are operating in an unpredictable environments. They need to continuously adapt to changing internal or external conditions in order to efficiently use of resources and to provide adequate functionality when the external environment changes dynamically. Dynamism, indeed, plays a crucial role in these modern systems and it can be captured as the interplay of changes relative to the three features below:

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.

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11246, pp. 13–31, 2018.
https://doi.org/10.1007/978-3-030-03424-5_2

Architecture modeling languages should be equipped with concepts and mechanisms which are expressive and easy to use relatively to each of these features.

The first feature implies the ability of describing the coordination of systems that are parametric with respect to the numbers 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 adding 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.

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

The different approaches to architectural modeling 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 *declarative vs. imperative* modeling.

Exogenous modeling considers that components are architecture-agnostic and respect a strict separation between a component behavior and its coordination. 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 modeling* requires adding explicit coordination primitives in the code describing components' behavior. Components are composed through their interfaces, which 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 modeling uses logics to express coordination constraints between components. It allows in particular modular description as one can associate with each component its coordination constraints. The global system coordination can be obtained in that case as the conjunction of individual constraints of its constituent components. *Disjunctive modeling* 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 modeling allows abstraction and modular description but it involves the risk of inconsistency in case there is no architecture satisfying the specification.

This paper introduces the DReAM framework for modeling Dynamic Reconfigurable Architectures. DReAM uses a logic-based modeling 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).

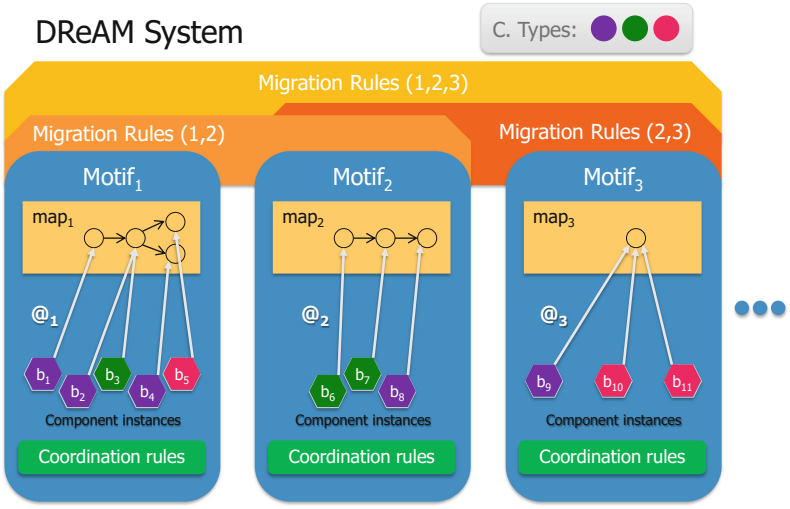


Fig. 1. Overview of a DReAM system

Coordination rules in a motif involve an interaction part and an associated operation. The former is modeled as a formula of the first order Interaction Logic [8] used to specify parametric interactions between instances of types of components. The latter specifies transfer of data between the components involved in the interaction. In this way, we can characterize parametric coordination between classes of components. The rules allow both conjunctive and disjunctive specification styles. We 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 equipped with a map, which is a graph defining the topology of the interactions in this motif. To parametrize coordination rules for the nodes of the map, an

address function @ is provided defining the position $@(c)$ in the map of any component instance c associated with the motif. Maps are also very useful to express mobility of components, in which case the connectivity relation of the map represents possible moves of components. Finally the language allows the modification of maps by adding or removing nodes and edges, as well as the dynamic creation and deletion of component instances.

2 Static Architectures - The PIL Coordination Language

We introduce the Propositional Interaction Logic (PIL) [8] used to model interactions between a given set of components. A system model 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;*
- *$T \subseteq S \times P \cup \{idle\} \times S$: finite set of transitions. Transitions (s, p, s') are also denoted by $s \xrightarrow{p} s'$; $p \in P$ is the port offered for interaction, and each transition is labelled by a different port.*

A component has a special port $idle \notin P$ that is associated to implicit loop transitions $\{s \xrightarrow{idle} s\}_{s \in S}$. This choice is made to simplify the theoretical development of our framework. Furthermore it is assumed that the sets of ports and control locations of different components are disjoint.

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

$$\Gamma = \{s_i \in S_i\}_{i \in [1..n]} \tag{1}$$

Given the set of ports $\mathcal{P}$, an interaction a is any finite subset of $\mathcal{P}$ such that no two ports belong to the same component. The set of all interactions is isomorphic to $I(\mathcal{P}) = 2^{\mathcal{P}}$.

Given a set of components $B_1 \ldots B_n$ and the set of interactions γ, we can define a system $\gamma\,(B_1, \ldots, B_n)$ using the following operational semantics rule:

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

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[1].

2.1 Propositional Interaction Logic (PIL)

Let $\mathcal{P}$ and $\mathcal{S}$ be respectively the domains of ports and control locations. The formulas of Propositional Interaction Logic $\mathsf{PIL}(\mathcal{P}, \mathcal{S})$ are defined by the syntax:

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

where $\pi : 2^\Gamma \mapsto \{\mathbf{true}, \mathbf{false}\}$ 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 Γ. The semantics is defined by the following satisfaction relation $\models_\Gamma$:

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

A monomial $\bigwedge_{p\in I} p \wedge \bigwedge_{p\in J} \neg p, I \cap J = \emptyset$ denotes a set of interactions a s.t.:

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

Note that *idle* ports of components can appear in PIL formulas. Given a component with ports P and idle port *idle*, the formula $idle \equiv \bigwedge_{p\in P} \neg p$, while $\neg idle \equiv \bigvee_{p\in P} p$.

As we can describe sets of interactions using PIL formulas, we can redefine rule (2) as follows, where Ψ is a PIL formula.

$$\frac{a \models_\Gamma \Psi \qquad \forall p \in a : s_i \xrightarrow{p} s'_i}{\{s_i\}_{[1..n]} \xrightarrow{a} \{s'_i\}_{[1..n]}} \tag{5}$$

2.2 Disjunctive vs. Conjunctive Specification Style

It is shown in [8] how a function β can be defined $\beta : I(P) \rightarrow PIL(P, S)$ associating with an interaction a its characteristic PIL formula $\beta(a)$. For example, if $P = \{p, q, r, s, t\}$ then for the interaction $\{p, q\}$, $\beta(\{p, q\}) = p \wedge q \wedge \neg r \wedge \neg s \wedge \neg t$[2]. For the set of interactions γ caused by the broadcast of p to ports q and r,

[1] Components B_j not "actively" involved in the interaction will participate with their *idle* port s.t. $s'_j = s_j$.

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

$\beta(\gamma) = p\neg s\neg t$. For the set of interactions γ consisting of the singleton interactions p and q, $\beta(\gamma) = (p\neg q \vee \neg pq) \wedge \neg r\neg s\neg t$. Finally $\beta(\{idle\}) = \neg p\neg q\neg r\neg s\neg t$ as $idle$ is the only port not belonging to P.

Note that the definition of the function β requires knowledge of P. This function can be naturally extended to sets of interactions γ: for $\gamma = \{a_1, \ldots, a_n\}$, $\beta(\gamma) = \beta(a_1) \vee \ldots \vee \beta(a_n)$.

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 [8] 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 Ψ_p holds and this defines interaction patterns from other components in which the port p needs to be involved.

For example, the interaction involving 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, tr_1, tr_2 and tr_1r_2.

Note that by applying this methodology we can associate to a component with set of ports P a constraint $\bigwedge_{p\in P}(p \Rightarrow \Psi_p)$ that characterizes the set of interactions where some port of the component may be involved. So if a system consists of components $C_1, \ldots, C_n$ with sets of 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 disjunctive form whose monomials characterize global interactions. Notice that the disjunctive form obtained in that manner contains the monomial $\bigwedge_{p\in P} \neg p$, where $P = \bigcup_{i\in[1..n]} P_i$, which is satisfied by the interaction where every component performs the $idle$ action. This trivial remark says that in the PIL framework it is possible to express for each component separately its interaction constraints and compose them conjunctively to get global disjunctive constraints.

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

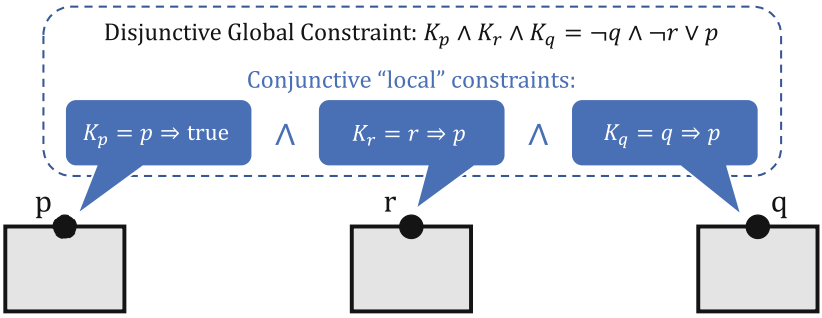
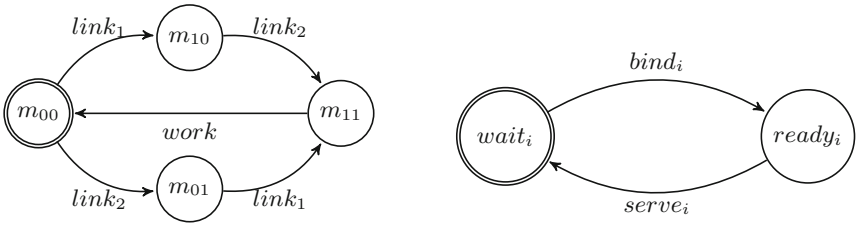


Fig. 2. Broadcast example: disjunctive vs conjunctive specification

Figure 3 shows the representation of such components.



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

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

The set of allowed interactions γ for the set of components $\{master, slave_1, slave_2\}$ can be represented via the following PIL formula using the disjunctive style:

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

where $idle_{s_i} \equiv \neg bind_i \wedge \neg serve_i$ is the *idle* port of $slave_i$. Alternatively, the same interaction patterns can be modeled using the conjunctive style:

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

The two formulas differ in the admissibility of the "no-interaction" interaction; the conjunctive formula Ψ_{conj} allows all components to avoid interaction by performing a transition over their *idle* ports. To allow it, the formula $idle_m \wedge idle_{s_1} \wedge idle_{s_2}$ must be added to the chain of disjunctions in Ψ_{disj}.

3 Static Architectures with Transfer of Values: PILOps

We expand the PIL framework to allow 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 to a notation that is inspired by guarded commands. Finally, we extend the definitions for disjunctive and conjunctive styles and study their connections.

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, P and T are as in Definition 1 and $X \subseteq \mathcal{X}$ is a finite set local variables. As for ports and control locations, it is assumed that sets of local variables for different PILOps components are disjoint.*

A system is a set of coordinated components $B_i = (S_i, X_i, P_i, T_i)$ for $i = [1, n]$. The *configuration* Γ of a system is described by the control locations of its components, and also the *valuation function* $\sigma : \mathcal{X} \mapsto \mathsf{V}$ mapping local variables to values:

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

Interactions are still sets of ports belonging to different components. Using a term of PILOps to compose components, the system configuration Γ evolves to a new configuration Γ' by performing an interaction a, represented by $\Gamma \xrightarrow{a} \Gamma'$.

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

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

- operators & and $\parallel$ are *associative* and *commutative*, and & has higher precedence than $\parallel$;
- $\pi : 2^{\Gamma} \mapsto \{\mathbf{true}, \mathbf{false}\}$ is a state predicate;
- $\delta : 2^{\sigma} \mapsto 2^{\sigma}$ is an operation that transforms the valuation function σ.

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:

$$\begin{array}{ll} a \models_{\Gamma} \ \Psi \rightarrow \Delta & \text{if } a \models_{\Gamma} \Psi \\ a \models_{\Gamma} \Phi_1 \,\&\, \Phi_2 & \text{if } a \models_{\Gamma} \Phi_1 \text{ and } a \models_{\Gamma} \Phi_2 \\ a \models_{\Gamma} \ \Phi_1 \parallel \Phi_2 & \text{if } a \models_{\Gamma} \Phi_1 \text{ or } a \models_{\Gamma} \Phi_2 \end{array} \tag{8}$$

In other words, the operators & and ∥ for PILOps terms are equivalent to the logical ∧ and ∨ from the interaction semantics perspective.

Operations in Δ are treated differently: operations of rules combined with "&" are either performed all together if the associated PIL formulas hold for a, Γ or not at all if at least one formula does not, while for rules combined with the"∥" operator a maximal union of operations satisfying the PIL formulas will be executed. We indicate the set of operations to be performed for Φ under a, Γ as $[\![\Phi]\!]_{a,\Gamma}$, which is defined according to the following rules:

$$\begin{aligned}
[\![\Psi \to \Delta]\!]_{a,\Gamma} &= \begin{cases} \Delta & \text{if } a \models_\Gamma \Psi \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\Phi_1 \;\&\; \Phi_2]\!]_{a,\Gamma} &= \begin{cases} [\![\Phi_1]\!]_{a,\Gamma} \cup [\![\Phi_2]\!]_{a,\Gamma} & \text{if } a \models_\Gamma \Phi_1 \text{ and } a \models_\Gamma \Phi_2 \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\Phi_1 \parallel \Phi_2]\!]_{a,\Gamma} &= [\![\Phi_1]\!]_{a,\Gamma} \cup [\![\Phi_2]\!]_{a,\Gamma}
\end{aligned} \tag{9}$$

Two PILOps terms Φ_1, Φ_2 are *equivalent* if, for any interaction a and configuration Γ, $[\![\Phi_1]\!]_{a,\Gamma} = [\![\Phi_2]\!]_{a,\Gamma}$.

Axioms for PILOps. The following axioms hold for PILOps terms:

& is associative, commutative and idempotent (10)

$\Psi_1 \to \Delta_1 \;\&\; \Psi_2 \to \Delta_2 = \Psi_1 \wedge \Psi_2 \to \Delta_1 \cup \Delta_2$ (11)

$\Phi \;\&\; \mathbf{true} \to \emptyset = \Phi$ (12)

$\parallel$ is associative, commutative and idempotent (13)

$\Psi_1 \to \Delta \parallel \Psi_2 \to \Delta = \Psi_1 \vee \Psi_2 \to \Delta$ (14)

$\Psi \to \Delta_1 \parallel \Psi \to \Delta_2 = \Psi \to \Delta_1 \cup \Delta_2$ (15)

$\mathbf{false} \to \Delta \parallel \Phi = \Phi$ (16)

Absorption: $\Phi_1 \parallel \Phi_2 = \Phi_1 \parallel \Phi_2 \parallel \Phi_1 \;\&\; \Phi_2$ (17)

Distributivity: $\Phi \;\&\; (\Phi_1 \parallel \Phi_2) = \Phi \;\&\; \Phi_1 \parallel \Phi \;\&\; \Phi_2$ (18)

Normal disjunctive form (DNF): (19)

$\Psi_1 \to \Delta_1 \parallel \Psi_2 \to \Delta_2 = \Psi_1 \wedge \neg\Psi_2 \to \Delta_1 \parallel \Psi_2 \wedge \neg\Psi_1 \to \Delta_2 \parallel \Psi_1 \wedge \Psi_2 \to \Delta_1 \cup \Delta_2$

Note that PILOps strictly contains PIL as a formula Ψ can be represented by $\Phi \to \emptyset$. The operator & is the extension of conjunction with neutral element $\mathbf{true} \to \emptyset$ and ∥ is the extension of the disjunction with an absorption (17) and distributivity axiom (18). 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 (17) and there is no conjunctive normal form.

Operations. Operations δ in PILOps are assignments on local variables of components involved in an interaction of the form $x := f$, 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 σ as:

$$(x := f)(\sigma) = \sigma\left[x \mapsto f\left(\sigma\left(y_1\right), \ldots, \sigma\left(y_k\right)\right)\right] \tag{20}$$

A set of assignment operations Δ is performed using a *snapshot semantics*. When Δ contains multiple assignments on the same local variable, the results are *non-deterministic*.

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

$$\frac{a \models_\Gamma \Phi \qquad \forall p \in a : s_i \xrightarrow{p} s_i' \qquad \sigma' \in [\![\Phi]\!]_{a,\Gamma}(\sigma)}{\left(\{s_i\}_{[1..n]}, \sigma\right) \xrightarrow{a} \left(\{s_i'\}_{[1..n]}, \sigma'\right)} \tag{21}$$

where $[\![\Phi]\!]_{a,\Gamma}(\sigma)$ is the set of valuation functions obtained by applying the operations $\delta \in [\![\Phi]\!]_{a,\Gamma}$ to σ in every possible order (using a *snapshot semantics*).

3.3 Disjunctive vs. Conjunctive Specification Style in PILOps

We define disjunctive and conjunctive style specification in PILOps. We associate with $p \Rightarrow \Psi_p$ an operation Δ_p to be performed when an interaction involving p is executed according to this rule. We call the PILOps term describing this behavior the *conjunctive term* $[p, \Psi_p, \Delta_p] = (\neg p \to \emptyset \parallel p \wedge \Psi_p \to \Delta_p)$. Δ_p may be executed when p is involved in some interaction; otherwise, no operation is executed. The conjunction of terms of this form gives a disjunctive style formula. Consider for instance, the conjunction of two terms:

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

The disjunctive form obtained by application of the distributivity axiom (18) is a union of four terms corresponding to the canonical monomials on p and q and leading to the execution of no operation, either operation Δ_p, Δ_q or both. It is easy to see that the conjunctive and disjunctive forms below are equivalent:

$$\begin{gathered}\&_{p \in P} \, (\neg p \to \emptyset \parallel p \wedge \Psi_p \to \Delta_p)\\ \Big\|_{I \cup J = P} \Big(\bigwedge_{i \in I} p_i \wedge \Psi_{p_i} \bigwedge_{j \in J} \neg p_j \to \bigcup_{i \in I} \Delta_{p_i} \Big) \qquad \text{where } \bigcup_{p_i \in \emptyset} \Delta_{p_i} = \emptyset.\end{gathered}$$

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 canonical form and will be 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 Δ_{p_i} such that $\Delta_{IJ} = \bigcup_{i\in I} \Delta_{p_i}$. That is, the operation associated with a port participating to an interaction is the same. This condition also determines the limits of the conjunctive and compositional approach.

Example 2 (Master-Slaves). Let us expand Example 1 by attaching data transfer between the *master* component and the two $slave_1$ and $slave_2$ components. We assume that the *master* has a $buffer$ local variable taking the value obtained by adding the values stored 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 γ does not change, but using PILOps we can characterize the desired behaviour using the disjunctive style as follows:

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

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

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

4 The DReAM Framework

In this Section we present the DReAM framework, allowing dynamism and reconfiguration which extends the static framework in the following manner. Components are instances of types of components and their number can dynamically change. 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 topology of the underlying architecture as well as component mobility.

4.1 Component Types and Component Instances

DReAM systems are constituted by *instances of component types*. Component types in DReAM correspond to PILOps components (see Definition 2), while 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 $\mathcal{C}$ 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 genericity, 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 $c.X$*[3]*. We use the same notation to denote ports, states and variables belonging to a given component instance (e.g. $c.p \in c.P$) and assume that ports of different component instances are still disjoint sets, 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:

$$\frac{(s, p, s') \in T}{c.s \xrightarrow{c.p} c.s'} \tag{22}$$

4.2 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 $\mathcal{P}$, the DReAM coordination language is defined by the syntax:

$$\begin{array}{rl}
\text{(DReAM term)} & \rho ::= \Phi \mid D\{\Phi\} \mid \rho_1 \;\&\; \rho_2 \mid \rho_1 \parallel \rho_2 \\
\text{(declaration)} & D ::= \forall c : m.b \mid \exists c : m.b \mid D_1, D_2 \\
\text{(PILOps term)} & \Phi ::= \Psi \rightarrow \Delta \mid \Phi_1 \;\&\; \Phi_2 \mid\; \Phi_1 \parallel \Phi_2 \\
\text{(PIL formula)} & \Psi ::= c.p \in \mathcal{P} \mid \pi \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \\
\text{(set of ops.)} & \Delta ::= \emptyset \mid\; \{\delta\} \mid \Delta_1 \cup \Delta_2
\end{array} \tag{23}$$

– *Declarations* define the context of the term by declaring quantified ($\forall|\exists$) component variables (c) associated to 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;
– $\pi : 2^{\Gamma} \mapsto \{\mathbf{true}, \mathbf{false}\}$ is a state predicate on the system configuration Γ;
– $\delta : 2^{\Gamma} \mapsto 2^{\Gamma}$ is an *operation* that transforms the system configuration Γ.

[3] 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. $b.c.s$.

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.

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

Declaration Expansion for Coordination Terms. Given that DReAM systems host finite numbers of component instances, first-order logic quantifiers can be eliminated by enumerating every component instance of the type specified in the declaration. We thus define the *declaration expansion* $\langle\rho\rangle_\Gamma$ of ρ under configuration Γ via the following rules:

$$\begin{aligned} \langle\Phi\rangle_\Gamma &= \Phi & \big\langle\forall c : m.b\{\Phi\}\big\rangle_\Gamma &= \mathop{\&}_{c^*\in m.b.C} \Phi\,[c^*/c] \\ \langle\rho_1 \;\&\; \rho_2\rangle_\Gamma &= \langle\rho_1\rangle_\Gamma \;\&\; \langle\rho_2\rangle_\Gamma & \big\langle\exists c : m.b\{\Phi\}\big\rangle_\Gamma &= \mathop{\|}_{c^*\in m.b.C} \Phi\,[c^*/c] \\ \langle\rho_1 \parallel \rho_2\rangle_\Gamma &= \langle\rho_1\rangle_\Gamma \parallel \langle\rho_2\rangle_\Gamma & \big\langle D_1, D_2\{\Phi\}\big\rangle_\Gamma &= \big\langle D_1\{\,\langle D_2\{\Phi\}\rangle_\Gamma\,\}\big\rangle_\Gamma \end{aligned} \tag{24}$$

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

By applying (24), any term can be transformed into a PILOps term, whose semantics is defined in Sect. 3.2:

4.3 Motif Modeling

A motif characterizes an independent dynamic architecture involving a set of component instances C subject to specific *coordination terms* parameterized 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 $m := \langle C, \rho, Map_0, @_0\rangle$, where $C \subseteq \mathcal{C}$ is the set of component instances assigned to the motif, ρ is the coordination term regulating interactions and reconfigurations among them, and $Map_0, @_0$ are the initial configurations of the map associated to the motif and of the addressing 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 e.g. geographic map or some conceptual structure, e.g., cellular structure of a memory. In DReAM a map is specified as a graph $Map = (N, E)$, where:

- N is a set of nodes or locations (possibly infinite);
- E is a set of edges subset of $N \times N$ that defines the connectivity relation between nodes.

The relation E defines a concept of neighborhood for components.

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

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* Γ_m of motif m is represented by the tuple

$$\Gamma_m = \langle m.C.s, m.C.\sigma, m.Map, m.@\rangle \tag{25}$$

$$\equiv \left\langle \{c.s\}_{c \in m.C}, \{c.\sigma\}_{c \in m.C}, m.Map, m.@ \right\rangle \tag{26}$$

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.

Reconfiguration Operations. Reconfiguration operations realize 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.

Operational Semantics of Motifs. Terms ρ of the coordination language are used to compose component instances in a motif. The latter can evolve from a configuration Γ_m to another Γ''_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:

$$\frac{a \models_{\Gamma_m} \langle\rho\rangle_{\Gamma_m} \qquad \Gamma_m \xrightarrow{a} \Gamma'_m \qquad \Gamma''_m \in [\![\langle\rho\rangle_{\Gamma_m}]\!]_{a,\Gamma_m}(\Gamma'_m)}{\Gamma_m \overset{a}{\rightsquigarrow} \Gamma''_m} \tag{27}$$

– $\Gamma_m \xrightarrow{a} \Gamma'_m$ 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:

$$\frac{\forall c.p \in a : c.s \xrightarrow{c.p} c.s' \quad \text{with } c \in m.C}{\langle m.C.s, m.C.\sigma, m.Map, m.@\rangle \xrightarrow{a} \langle m.C.s', m.C.\sigma, m.Map, m.@\rangle} \tag{28}$$

– $[\![\langle\rho\rangle_{\Gamma_m}]\!]_{a,\Gamma_m}(\Gamma'_m)$ is the set of motif configurations obtained by applying the operations $\delta \in [\![\langle\rho\rangle_{\Gamma_m}]\!]_{a,\Gamma_m}$ in every possible order (evaluated using a snapshot semantics).

4.4 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 μ is a* migration term *and Γ_0 is the initial configuration of the system.*

The migration term μ is a coordination term where the operations δ 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:

$$\Gamma = \bigsqcup_{m \in M} \Gamma_m = \left\langle \bigcup_m m.C.s, \bigcup_m m.C.\sigma, \bigcup_m m.Map, \bigcup_m m.@ \right\rangle \tag{29}$$

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:

$$\frac{\Gamma_m \overset{a_m}{\rightsquigarrow} \Gamma'_m \;\text{ for } m \in M \qquad a \models_{\Gamma'} \langle\mu\rangle_{\Gamma'} \qquad \Gamma'' \in [\![\langle\mu\rangle_{\Gamma'}]\!]_{a,\Gamma'}(\Gamma')}{\Gamma \xrightarrow{a} \Gamma''} \tag{30}$$

- $\Gamma' = \bigsqcup_{m \in M} \Gamma'_m$;
- $a_m \subseteq a$ 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 μ.

The DReAM coordination language and its semantics have been implemented in Java. The implementation involves two parts: a Java execution engine with an associated API and a domain-specific language (DSL) with an IDE for system modeling in DReAM. Details about the implementation as well as examples of systems modeled in DReAM are provided in the long version of this paper [9].

5 Related Work

DReAM allows both conjunctive and disjunctive style modeling of dynamic reconfigurable systems. It inherits the expressiveness of the coordination mechanisms of BIP [8] as it directly encompasses multiparty interaction and extends previous work on modeling parametric architectures [10] 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 [11] the causal rules are of the form $p \Rightarrow \bar{p}$, where p and $\bar{p}$ are input and output port names corresponding to port symbol p. For CSP [12], the causal rules implementing the interface parallel operator parameterized by the channel a are of the form $a_i \Rightarrow \bigwedge_{a_j \in A} a_j$, where A is the set of ports communicating through a.

Also other richer calculi, such as π-calculus [13], that offer the possibility of modeling dynamic infrastructure via channel passing can be modeled in DReAM with its reconfiguration operations. Formalisms with richer communication models, such as AbC [14], offering multicasting 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 [15,16]. Nonetheless, our framework is more general as it does not adopt assumptions about timed synchronous cyclic behavior of components.

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.

6 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 modeled 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.

Flexible Semantics: The language relies on an operational semantics that admits a variety of implementations between two extreme cases. One consists in precomputing 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 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 specification. All results are too recent and many open 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 the 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. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in BIP. IEEE Trans. Comput. **57**(10), 1315–1330 (2008)
9. De Nicola, R., Maggi, A., Sifakis, J.: Dream: Dynamic reconfigurable architecture modeling, arXiv preprint: http://arxiv.org/abs/1805.03724 (2018)
10. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling dynamic architectures using Dy-BIP. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (eds.) SC 2012. LNCS, vol. 7306, pp. 1–16. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30564-1_1
11. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
12. Brookes, S.D., Hoare, C.A., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31**(3), 560–599 (1984)
13. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Inf. Comput. **100**(1), 1–40 (1992)
14. Alrahman, Y.A., De Nicola, R., 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 Inernational Conference, pp. 1–18 (2016)
15. Pinciroli, C., Lee-Brown, A., Beltrame, G.: Buzz: An extensible programming language for self-organizing heterogeneous robot swarms, arXiv preprint arXiv:1507.05946 (2015)
16. 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), pp. 3794–3800. IEEE (2016)
17. Bruni, R., Lafuente, A.L., Montanari, U., Tuosto, E.: Style based reconfigurations of software architectures. Universita di Pisa, Technical report 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: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11246, pp. 304–320. Springer, Cham (2018)