A Formal Model to Integrate Behavioral and Structural Adaptations in Self-adaptive Systems

Narges Khakpour, Jetty Kleijn, Marjan Sirjani

A Formal Model to Integrate Behavioral and Structural Adaptations in Self-Adaptive Systems

Narges Khakpour[1], Jetty Kleijn[2], and Marjan Sirjani[3]

[1] Linnaeus University, Sweden
[2] LIACS, Leiden University, The Netherlands
[3] Mälardalens Högskola, Sweden and Reykjavik University, Iceland

Abstract. An approach for modelling adaptive complex systems should be flexible and scalable to allow a system to grow easily, and should have a formal foundation to guarantee the correctness of the system behavior. In this paper, we present the architecture, and formal syntax and semantics of HPobSAM which is a model for specifying behavioral and structural adaptations to model large-scale systems and address reusability concerns. Self-adaptive modules are used as the building blocks to structure a system, and policies are used as the mechanism to perform both behavioral and structural adaptations. While a self-adaptive module is autonomous to achieve its local goals by collaborating with other self-adaptive modules, it is controlled by a higher-level entity to prevent undesirable behavior. HPobSAM is formalized using a combination of algebraic, graph transformation-based and actor-based formalisms.

1 Introduction

The growth and adaptation of a system is realized by behavioral adaptation and/or structural adaptation. While structural adaptation aims to adapt the system behavior by changing its architecture, behavioral adaptation focuses on modifying the functionalities of computational entities. Behavioral adaptation is usually suitable for the cases that minor changes are required to adapt the system. Structural adaptation is more scalable and suitable for large-scale and distributed adaptations. Yet changing the system structure to achieve minor changes is rather expensive. Hence, both behavioral and structural adaptations are often required to design complex adaptive systems.

A system must be able to evolve and grow continually even in unforeseen situations. Since an adaptation requirement might be unknown at design time, adaptive behavior must be built in a way that is *flexible* and *modifiable* at runtime. Furthermore, to guarantee the functionality of a complex software system, we have to provide mechanisms to ensure that the system is operating correctly. Here formal methods can play a key role.

Several frameworks and models have been inspired by natural systems to design large-scale adaptive systems [5, 22, 26, 27]. Although, they support self-organization, self-adaptability and long-lasting evolvability, they are not provided with a formal foundation. Moreover, specification and analysis of dynamic

adaptation have been given lots of attention in the last decade [8, 14, 19, 25, 20, 1, 2] where most of the approaches deal with either behavioral adaptation or structural adaptation [6, 7]. However, dynamic adaptation and self-* properties are restricted to responding to short-term changes, while systems must be additionally able to evolve and grow to cover the long-term evolution of systems [9]. Therefore, we need an approach to design complex software systems which supports behavioral and structural adaptations to tackle the long-term evolution, flexibility, complexity, scalability and assurance problems.

The use of policies has been given attention as a powerful mechanism to achieve flexibility in adaptive and autonomous systems which allows one to *"dynamically"* specify the requirements in terms of high level goals. A policy is a rule describing under which conditions a specified subject must (can or cannot) perform an action on a specific object [15]. PobSAM (Policy-based Self-Adaptive Model) is a policy-based model with formal foundation for developing and modeling self-adaptive systems that supports *behavioral adaptation*. A PobSAM model consists of a set of managers and actors. Managers control the behavior of actors by enforcing policies. This model provides a high degree of flexibility at the behavioral level by allowing one to change policies dynamically. However, it only supports behavioral adaptation.

In this paper, we consider an extension of PobSAM [14, 15], called HPobSAM (Hierarchical PobSAM) to support modeling large-scale adaptive systems. In H-PobSAM, *self-adaptive modules* have been added to PobSAM as a structuring feature. A self-adaptive module consists of managers, actors and possibly other self-adaptive modules. The notion of a *role* is introduced to specify structure-independent adaptations. Roles are dynamically assigned to self-adaptive modules and actors. Structural adaptation occurs by changing the roles of entities which leads to creation, removal or changing the interactions of entities. The managers are responsible to perform structural adaptations using *structural adaptation policies* that are defined in terms of roles.

HPobSAM is used in [13] to model a case study in the area of smart airports. In [13], we refer to an unpublished technical report for a complete description of HPobSAM. Here we present the description, architecture, and formal syntax and semantics of HPobSAM. We use prioritized hierarchical hypergraph (hh-graph) transition systems to specify the operational semantics of HPobSAM. Prioritized hh-graph transition systems are essentially classical prioritized state transition systems augmented with a function mapping states into hh-graphs and transitions into partial morphisms, i.e. a state is provided with a hh-graph indicating the current system structure.

Formal methods have been proposed for the modeling and analysis of adaptive software systems, but they are not always suitable for designing large-scale software systems. We propose a flexible policy-based approach with formal foundation to design large-scale software systems. Compared to existing work, our approach has the following novel features:

1. We present a formal extension of PobSAM to model large-scale systems that is flexible and supports both structural and behavioral adaptations. We

use structural adaptation policies as a mechanism for performing structural adaptation that can be modified at runtime, without the need to change the low-level programs.
2. We present an operational semantics for HPobSAM whose semantics rules allow us to transform a substructure that is specified only partially, i.e. we can add or remove a self-adaptive module whose internal structure is not known completely. This feature is an advantage in open systems where limited knowledge is available about the entities.

2 Case Study

Here, we introduce the running example of the paper shown in Figure 1. Consider a service-based system that dynamically adapts its behavior to provide an appropriate quality of service to clients. The system includes several clusters of application servers that require data provided by the data servers. The *cache handler* is used to determine the best cluster for handling a request considering the quality of service constraints, and the *logger* monitors the incoming requests. The *request receiver* analyzes the requests and transmits them to the *request dispatcher* of the proper cluster. The latter forwards the request to an application server in the cluster. When a request is processed, the result is sent back to the *request receiver* component. This component sends the result back to the requester and/or to the *cache handler*.

The system should be able to adapt its behavior to provide the requested service properly. The behavioral adaptation is done by dynamically balancing the load of clusters/servers and can be effective to some extent. However, if the load of system becomes high enough such that the current number of servers cannot handle the requests, structural adaptations come into play. We need to adapt the system structure by adding or replacing the clusters to improve the system throughput.

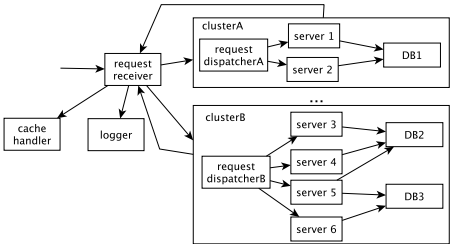

Fig. 1. The architecture of server clusters.

3 Overview of PobSAM

A PobSAM model is composed of three layers [14, 15]:

- The functional behavior of the system is implemented by the *actor layer* and contains computational entities, i.e., the actors.
- The *view layer* consists of view variables that provide an abstraction of the actors' states for the managers. A view variable is an actual state variable, or a function applied to state variables of actors.
- The main layer of PobSAM is the *manager layer* containing the autonomous managers. Managers control the behavior of actors according to predefined policies. A manager may have different configurations of which one is active at a time. Behavioral adaptation is performed by switching among those configurations. A configuration contains two classes of policies: governing policies and behavioral adaptation policies. A manager directs the actors' behavior by sending messages to them according to the governing policies. A governing policy is of the form $\langle o, e, \psi \rangle \bullet a$ where $o \in \mathbb{N}$ is the policy priority, $e \in \mathcal{E}$ is an event, ψ is the policy condition defined over views, and a is the policy action. Whenever a manager receives an event e it identifies all the governing policies that are triggered by that event. For each of the triggered policies, if the policy condition evaluates to true and there is no other triggered governing policy with priority higher than o, action a is requested to be executed by instructing the relevant actors to do so (by sending them asynchronous messages). The behavioral adaptation policies are used to perform behavioral adaptations by switching among different configurations.

Example 1. We model the request dispatcher of a cluster as a manager that is responsible to manage and control the behavior of the cluster. This manager has two configurations `lowConf` and `highConf` to control the cluster behavior respectively, in low-loaded and high-loaded conditions. The servers are modeled as the actors responsible for handling incoming requests. The view layer provides some information about the processing power of each server, their current loads, the whole throughput of the cluster, and the average number of handled requests by each server. The following governing policy of `lowConf` with priority n defined for the request dispatcher of the cluster A states that when a new request x is received and the load of server1 is less than l, ask server1 to handle the request: $g = \langle n, \mathsf{newreq}(x), \mathsf{load1} < l \rangle \bullet (\mathsf{server1.handle}(x))$.

4 The Architecture of HPobSAM

The components of a HPobSAM model are (i) self-adaptive modules, (ii) actors, (iii) the multi-level view layer, (iv) managers, and (v) roles. A system at the highest level is defined as a self-adaptive module. Figure 2 gives a schematic view of the HPobSAM architecture.

The concept of self-adaptive modules is inspired by SMC (Self-Managed Cells) [24] for structuring complex adaptive systems. A Self-Adaptive Module (SAM) is a policy-based building block which is able to automatically adapt its behavior in a complex dynamic environment. A self-adaptive module contains (i) possibly

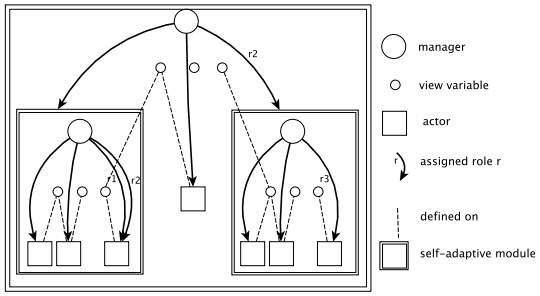

Fig. 2. A typical self-adaptive module.

other lower-level self-adaptive modules, (ii) the actors, (iii) a view layer, and (iv) a manager. To cater for large-scale systems, multiple self-adaptive modules are composed and aggregated hierarchically into a single larger self-adaptive module. A self-adaptive module may provide services to other self-adaptive modules. Note that the services are provided and used by the manager of a self-adaptive module.

A manager is aware of its substructure and is responsible for performing structural and behavioral adaptations of its module. The managers are provided with a new type of policies, so-called structural adaptation policies to perform structural adaptation. When the system requires adaptation, different managers are informed and they plan various adaptations to adapt the system behavior to the current context. Hence, adaptation is performed in a *distributed* manner in the system and not a single entity is responsible for performing an adaptation.

In PobSAM, the view layer provides information about the actor layer to the managers. In HPobSAM, a view layer exists at multiple levels. Each self-adaptive module has a view layer defined based on the view layers of its self-adaptive modules in addition to the actors' state variables of that module. The view layer acts as a tuple space to coordinate interactions of self-adaptive modules and a self-adaptive module can have controlled access to the view layer of other self-adaptive modules.

The structure of a system can change due to adding or removing an actor or a self-adaptive module, and modifying the actors and/or the self-adaptive modules interconnections. If the policies of a manager are described in terms of individual actors or self-adaptive modules, any modification of the manager's underlying substructure (i.e. by joining or deleting actors or self-adaptive modules) influences the specification of its policies and the view layer, and subsequently, policies and view variables have to be redefined to become consistent with the new structure. To tackle this problem of structure-dependent policies, we use the notion of *roles* to refer to the agents with the same functionality. The roles are assigned by a manager to the actors and the self-adaptive modules that it controls, and managers' policies as well as view variables are described in terms of roles. A structural reconfiguration is realized by changing the roles assigned

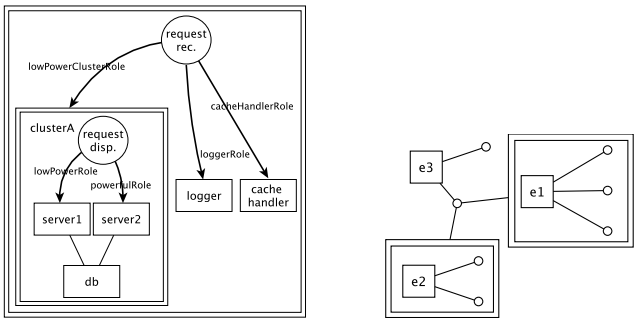

Fig. 3. (a) HPobSAM architecture of running example; (b) a hierarchical hypergraph.

to the entities, and hence, the managers' policies become structure-independent and do not have to be modified after a structural reconfiguration.

Example 2. Figure 3 (a) partially shows the HPobSAM architecture of our example. The whole system is a self-adaptive module that contains (i) several lower-level self-adaptive modules each corresponding to a cluster, (ii) a manager modeling the request receiver, and (iii) two actors for the cache handler and the logger. The roles `lowPowerClusterRole`, `loggerR`, and `cacheHandlerRole` are assigned by the request receiver to the cluster A (as a low-power cluster), the logger, and the cache handler, respectively.

5 The Syntax of HPobSAM

In this section, we first briefly introduce hierarchical hypergraphs that are used to model the system structure; then we specify the structural modeling of H-PobSAM; and,finally, we give the syntax of HPobSAM.

5.1 Hierarchical Hypergraphs Overview

A hypergraph is a generalization of a graph, where an edge can connect any number of nodes.

Definition 1. ***(Hypergraph)*** *A hypergraph is a tuple $G = (N, E, \theta)$, where N is the set of nodes, E is the set of hyperedges, $\theta : E \to N^*$ is the tentacle function mapping each hyperedge to a unique finite non-empty multiset of nodes.* [4]

[4] Note that we choose to represent multisets as elements of N^*, i.e. strings of occurrences of elements from N. Thus, e.g., the string bab represents the multiset $\{a, b, b\}$

Given two hypergraphs G_1 and G_2 with $G_i = (N_i, E_i, \theta_i)$ for $i = 1, 2$, a hypergraph morphism $m : G_1 \to G_2$ is a pair of mappings $m = (m_N, m_E)$ with $m_N : N_1 \to N_2$ and $m_E : E_1 \to E_2$, such that for all $e \in E_1$, the multiset defined by $\theta_2(m_E(e))$ is the multiset defined by $m_N(\theta_1(e))$. [5]

Such a morphism is injective (surjective, bijective) if both m_N and m_E are injective (respectively surjective, bijective, partial or total). If there is a bijective morphism $m : G_2 \to G_1$, then G_1 and G_2 are isomorphic.

Hierarchical hypergraphs [10] are hypergraphs in which some hyperedges, called *frames*, may refer to hypergraphs that can be hierarchical again, with an arbitrary but finite depth of nesting.

Definition 2. ***(Hierarchical Hypergraph)****. Let $\mathcal{X}$ be a set of symbols called variables. Let $\mathcal{H} = \mathcal{H}_0(\mathcal{X})$ be a set of triples $H = \langle G, F, cts \rangle$ where G is a hypergraph, $F = \emptyset$, and cts the trivial function from F to $\mathcal{X}$.*
For $i > 0$, $\mathcal{H}_i(\mathcal{X})$ consists of all triples $H = \langle G, F, cts \rangle$ where $G = (N, E, \theta)$ is a hypergraph, $F \subseteq E$ is the set of frame hyperedges of G, and $cts : F \to \mathcal{H}_{i-1}(\mathcal{X}) \cup \mathcal{X}$ assigns to each frame its content.
The class $\mathcal{H}(\mathcal{X}) = \bigcup_{i \geq 0} \mathcal{H}_i(\mathcal{X})$ is the set of hierarchical hypergraphs (hh-graphs) derived from $\mathcal{H}$ with variables in $\mathcal{X}$.

Example 3. Figure 3(b) shows a hh-graph which has hyperedges $\{e_1, e_2, e_3\}$, seven nodes depicted by circles, and two frames depicted using double-lined rectangles.

The concept of a graph morphism can be generalized to the hierarchical case [10]. Let $\mathcal{X}$ be a set of variables. For $i = \{1, 2\}$, let $H_i = \langle G_i, F_i, cts_i \rangle$ be two hypergraphs with variables in $\mathcal{X}$, and let X_i denote the set $\{f \in F_i \mid cts_i(f) \in \mathcal{X}\}$ of *variable (or primitive) frames* of H_i.

Definition 3. ***(Hierarchical Morphism).*** *A hierarchical morphism m from H_1 to H_2 is a pair $m = (\bar{m}, m^f)$ where $f \in F_1 \backslash X_1$ and*
(i) $\bar{m} : G_1 \to G_2$ is a graph morphism;
(ii) for all frames $f \in F_1$, $\bar{m}_E(f) \in F_2$, and if $\bar{m}_E(f) \in X_2$ then $f \in X_1$;
(iii) $m^f : cts_1(f) \to cts_2(\bar{m}_E(f))$ is a hierarchical morphism for all $f \in F_1 \backslash X_1$.

A hierarchical morphism is injective (surjective, bijective, partial or total) if both $\bar{m}$ and m^f are injective (respectively surjective, bijective, partial or total).

With graph constraints, certain graph properties can be expressed. In particular, it can be formulated that a graph G must (or must not) contain a certain subgraph G'. An atomic graph constraint ($\mathsf{gcons}(C, C')$) informally states that if a graph G contains the sub-graph C (premise), then it contains the sub-graph C' (conclusion) too [11].

---

as do *abb* and *bba*. Moreover, for every hyperedge e the string $\theta(e)$ is not empty; and for every hyperedge $e' \neq e$, the multisets represented by $\theta(e)$ and $\theta(e')$ are not the same.

[5] Note that the application of m_N to the string $\theta_1(e)$ yields a string in N_2^*.

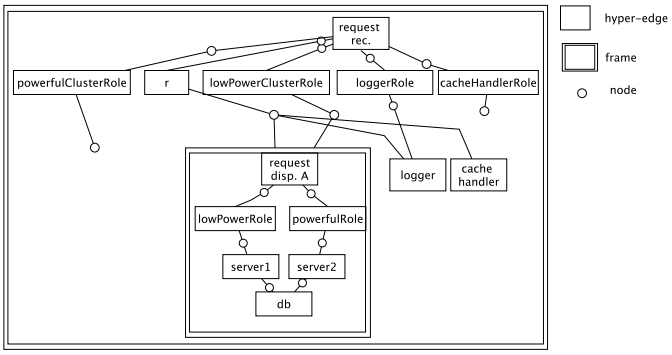

Fig. 4. Part of the hierarchical hypergraph model of our example.

Definition 4. ***(Atomic Graph Constraint).*** *Let C and C' be two graphs. An atomic graph constraint is specified as a graph morphism $k : C \to C'$.*

A graph G satisfies atomic graph constraint $\mathsf{gcons}(C, C')$ *specified by the graph morphism $k : C \to C'$. if, for every injective graph morphism $p : C \to G$, there exists an injective graph morphism $q : C' \to G$ with $q \circ k = p$. A graph constraint is a boolean formula over atomic graph constraints: (i) True and every atomic graph constraint are graph constraints, and (ii) if c and c' are two graph constraints, then $c \vee c'$, $c \wedge c'$ and $\neg c$ are graph constraints.*

5.2 HPobSAM Syntax

Structural Modeling The system structure is modeled as a hh-graph. We model role assignments as nodes, self-adaptive modules as frames, and managers, actors and roles as hyperedges. The hh-graph $H = (G, \mathcal{K}, cts)$ describes how several elements of a self-adaptive module κ are connected together logically. The set of self-adaptive modules of κ is given by $\mathcal{K}$, and cts gives their internal structure. The hypergraph G shows the first-level internal structure of κ defined as follows:

$$G = (N, E, \theta) \ , \ E = \{m\} \cup A \cup R \cup \mathcal{K}$$

where m is the manager of κ, A indicates the set of κ's actors, and R indicates the set of roles assigned by m.

Example 4. Figure 4 partially depicts the hh-graph of our example.

Views A self-adaptive module κ has its own view layer V consisting of view variables defined over the state variables of its (immediate) actors (A) and the view variables of its (immediate) self-adaptive modules ($\mathcal{K}$), i.e., a view variable $v \in V$ is a function over V, $\mathcal{K}$, and the state variables of the actors in A.
Managers A manager m is defined as a tuple $m = \langle C, c_0, \kappa, V, H \rangle$, with C the (finite) set of configurations of m, $c_0 \in C$ its initial configuration, κ the self-adaptive module of which m is the manager, V the (finite) set of view variables

of κ, and the hierarchical hypergraph $H = (G, \mathcal{K}, cts)$ describes how m is logically connected to other agents.

A configuration $c \in C$ is defined as $c = \langle P_G, P_B, P_S \rangle$, where P_G, P_B and P_S indicate the governing policy set, the behavioral adaptation policies set, and the structural adaptation policy set of c, respectively. A primitive action of a governing policy is of the form $\mathsf{r.msg}$ and is intended to send the message msg to some actors/self-adaptive modules with role r. The behavioral adaptation policies are not influenced by this extension (See Section 3).

A structural adaptation policy $sp \in P_S$ is defined as $sp{=}\langle o, e, \psi_H \rangle \bullet a_H$ consists of priority $o \in \mathbb{N}$, event e, condition ψ_H and an action a_H. The condition ψ_H can be defined as a combination of ordinary boolean expressions defined over the view layer and graph constraints defined over H, the internal structure of κ. Let as be an actor or a self-adaptive module. The action a_H is a strategy to apply a dynamic reconfiguration with the primitive actions of the forms

- $\mathsf{r.msg}$ to send the message msg to the agents with role $r \in R$,
- $\mathsf{join(r, as)}$ for assigning role r to as,
- $\mathsf{quit(r, as)}$ for releasing as from role r,
- $\mathsf{add(as)}$ for adding as to the substructure of m, and
- $\mathsf{remove(as)}$ for removing as from the substructure of m.

The condition ψ_H of a structural adaptation policy is defined as follows where $\mathsf{gcons}(Y, Y')$ is an atomic graph constraint:

$$\psi_H = (\exists r \in R).\psi_H \mid (\forall r \in R).\psi_H \mid \psi_H \wedge \psi'_H \mid \neg\psi_H \mid \mathsf{gcons}(Y, Y')$$

Example 5. The policy $\mathsf{PolicyA}$ states that when the request load is high, the cache handler is activated, i.e. the role $\mathsf{cacheHandlerRole}$ is assigned to the cache handler by executing the action $\mathsf{join(cacheHandlerRole, cachehandler)}$. Then, the logger is deactivated ($\mathsf{quit(loggerRole, logger)}$) and a new cluster with powerful servers ($\mathsf{clusterD}$) is added to the system. The operators ; and || are resp. the sequential and parallel composition of the algebra CA^a that is used to specify policy actions

(See [15]):

$$\mathsf{PolicyA} = \langle 1, \mathsf{onhighload}, \top \rangle \bullet (\mathsf{join(cacheHandlerRole, cachehandler)};$$
$$\mathsf{quit(loggerRole, logger)})||(\mathsf{add(clusterD)}\ ;\ \mathsf{join(powerfuleClusterRole, clusterD)})$$

Self-Adaptive Modules A self-adaptive module κ is formally defined as $\kappa = \langle V, H_\kappa \rangle$ where V and H_κ respectively represent the view layer and the hh-graph of κ. Observe that H_κ is a hyperedge with the content H as defined above.

6 Structural Operational Semantics

We present prioritized hh-graph transition systems to define the operational semantics of HPobSAM models. Prioritized hh-graph transition systems are essentially prioritized state transition systems [15] augmented with a function

mapping states into hierarchical hypergraphs and transitions into partial hierarchical morphisms. Thus every state is provided with a graph indicating the current system structure.

Definition 5. ***(Prioritized State Transition System)*** *A prioritized state transition system is a tuple $T = \langle S, \delta, L, s_0 \rangle$ where S is a set of states, $s_0 \in S$ is the initial state, L is a set of labels, and $\delta \subseteq S \times L \times S$ is a set of transitions.*

Labels $l \in L$ are of the form (ϕ, α, n) and a transition $s \xrightarrow{(\phi,\alpha,n)} s'$ means that it is possible to perform action α under condition ϕ in state s when there is no enabled transition with higher priority than n in state s, and then make a transition to s'.

Definition 6. ***(Prioritized hh-Graph Transition System)*** *A prioritized hh-graph transition system is given by a pair $\langle T, g \rangle$, where T is a prioritized state transition system and g is a pair $g = \langle g_1, g_2 \rangle$ of mappings such that $g_1(s)$ is a hh-graph for each state $s \in S$, and $g_2(t) : g_1(s) \to g_1(s')$ is an injective partial hierarchical morphism for each transition $t : s \xrightarrow{l} s' \in \delta$.*

The conditions of a transition $t : s \xrightarrow{l} s' \in \delta$ can contain graph constraints that are to be evaluated over $g_1(s)$. The semantics of the actor layer remains unchanged by this extension. The semantics of the view layer is similarly defined as that of PobSAM [15, 14]. In this paper, we restrict ourselves to introduce the semantics of managers as the core part of HPobSAM.

Overview of a Manager's Semantics We use the notation $[M]^c\langle b, p, a, q, H \rangle$ to describe a manager M where $c = \langle P_G, P_B, P_S \rangle$ is its current configuration, $b \in P_B$ is its triggered behavioral adaptation policy, $p \subseteq P_G \cup P_S$ is its set of triggered governing/structural adaptation policies, a is its current executing action (that can belong to a governing policy or a structural adaptation), q is its input message queue, and H is a hh-graph denoting the substructure of M. The semantics of triggering structural policies is identical to that of governing policies presented in [15]. Hence, we focus on their enforcement and use the notation $M\langle p, a, q, H \rangle$ for the sake of simplicity. The notation $\surd$ is used to show an empty action. The operational semantics of managers in HPobSAM is described by the transition rules for PobSAM proposed in [15] and the transition rules given in Fig. 5 and Fig. 7 which we explain later. The conditions of transitions specifying managers' semantics (e.g. ϕ in Figure 5) are evaluated on M's view and its substructure.

The Semantics of a Manager's Interactions is presented in Figure 5 that contains graph constraints presented in Figure 6. The rules description and the definition of symbols are described in the following. A primitive action of a PobSAM manager is sending an asynchronous message msg to an actor a that results in putting the message msg in a's queue. In HPobSAM, there are three types of interactions that a manager may initiate: (i) sending a message to an actor with the role r, (ii) sending a message to a lower-level self-adaptive module with the

$$\text{MSR1}\ \frac{M\langle p, \mathsf{r.msg}, q, H\rangle \xrightarrow{(\top, \mathsf{r.msg}, n)} M\langle p, \surd, q, H\rangle \qquad H \vDash \mathsf{gcons}(\emptyset, G_1)}{s_a \xrightarrow{(\top, \mathsf{r.msg}, n)} s'_a}$$

$$\text{MSR2}\ \frac{M_1\langle p, \mathsf{r.msg}, q, H\rangle \xrightarrow{(\top, \mathsf{r.msg}, n)} M_1\langle p, \surd, q, H\rangle \qquad H \vDash \mathsf{gcons}(\emptyset, G_2)}{M_2\langle p', a', q', H'\rangle \xrightarrow{(\top,\ \mathsf{r.msg}\ ,\ n)} M_2\langle p', a', q' : \mathsf{msg}, H'\rangle}$$

$$\text{MSR3}\ \frac{M_1\langle p, \mathsf{r.msg}, q, H\rangle \xrightarrow{(\top, \mathsf{r.msg}, n)} M_1\langle p, \surd, q, H\rangle \qquad H_{\kappa'} \cup H_{\kappa} \vDash \mathsf{gcons}(\emptyset, G_3)}{M_2\langle p', a', q', H'\rangle \xrightarrow{(\top,\ \mathsf{r.msg}\ ,\ n)} M_2\langle p', a', q' : \mathsf{msg}, H'\rangle}$$

Fig. 5. The rules for managers' interactions.

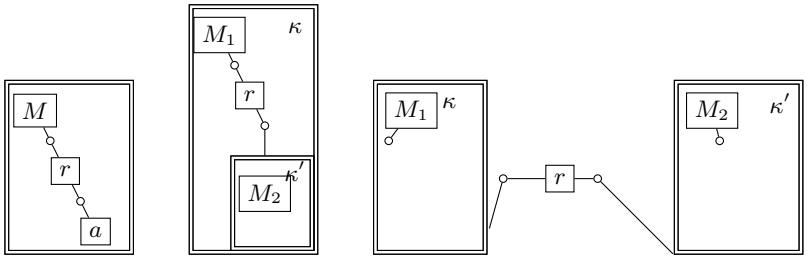

Fig. 6. The graph constraints of interactions semantics.

role r, and (iii) sending a message to the sibling self-adaptive modules with the role r. The operational semantics of case (i) is expressed using the rule MSR1 where G_1 is a graph depicted in Figure 6(a), $\mathsf{gcons}(\emptyset, G_1)$ is a graph constraint that holds if the actor a has the role r, and s_a and s'_a indicate the local states of a before and after receiving the message msg. The rule MSR2 expresses the semantics of case (ii). In this rule, a message is sent to a lower-level self-adaptive module κ' with the role r that contains a manager M_2. The graph G_2 is defined in Figure 6 (b). The manager M_1 in the self-adaptive module κ has assigned the role r to its sibling self-adaptive module κ' that contains the manager M_2. The manager M_1 uses the rule MSR3 to send a message to M_2 (case (iii)) where $\mathsf{gcons}(\emptyset, G_3)$ is a graph constraint with graph G_3 as defined in Figure 6(c) and $H_{\kappa'} \cup H_{\kappa}$ is the union of $H_{\kappa'}$ and H_{κ}.

The Semantics of Structural Adaptation is presented in Figure 7. In this figure, a function $f' = f|\{(e_1, v_1), \ldots, (e_2, v_n)\}$ is defined as $f'(x) = \begin{cases} v_k & x = e_k \\ f(x) & - \end{cases}$. The predicate $\mathsf{conn}(e, n, e') = n \in \theta(e) \cap \theta(e')$ informally states that the hyperedges e and e' are connected through the node n in a hypergraph G. The underlying substructure of M before and after a reconfiguration is respectively H and H' where $H = \langle G, F, cts\rangle$, $G = (N, E, \theta)$, and $H' = \langle G', F', cts'\rangle$, $G' = (N', E', \theta')$.

$$\text{(AAR)}\quad \frac{\mathsf{conn}(M, n_1, \iota), n_2 \in \theta(\iota)\backslash\{n_1\}, \theta' = \theta|\{(as, \{n_2\})\}, as \notin E, E' = E \cup \{as\} \quad \mathsf{hhyper}(as, G_{as}) \wedge \mathsf{wellFormed}(G_{as}) \implies (F' = F \cup \{as\} \wedge cts' = cts|\{(as, G_{as})\})}{M\langle p, \mathsf{add}(as), q, H\rangle \xrightarrow{(\top, \mathsf{add}(as), 1)} M\langle p, \surd, q, H'\rangle}$$

$$\text{(RAR)}\quad \frac{\mathsf{conn}(M, n_1, r), \mathsf{conn}(r, n_2, as), E' = E\backslash\{as\}, E' \neq \emptyset \quad as \in F \implies (F' = F\backslash\{as\}, cts' = cts|\{(as, \mathbf{undef})\})}{M\langle p, \mathsf{remove}(as), q, H\rangle \xrightarrow{(\top, \mathsf{remove}(as), 1)} M\langle p, \surd, q, H'\rangle}$$

$$\text{(JAR)}\quad \frac{\mathsf{conn}(M, n_1, r), as \in E, n_2 \in \theta(r)\backslash n_1, \theta' = \theta|\{(as, \theta(as) \cup \{n_2\})\}}{M\langle p, \mathsf{join}(r, as), q, H\rangle \xrightarrow{(\top, \mathsf{join}(r, as), 1)} M\langle p, \surd, q, H'\rangle}$$

$$\text{(QAR)}\quad \frac{\mathsf{conn}(M, n_1, r), \mathsf{conn}(r, n_2, as), r \neq \iota, \theta' = \theta|\{(as, \theta(as)\backslash\{n_2\})\}}{M\langle p, \mathsf{quit}(r, as), q, H\rangle \xrightarrow{(\top, \mathsf{quit}(r, as), 1)} M\langle p, \surd, q, H'\rangle}$$

Fig. 7. The rules for structural adaptation.

Note that for the sake of readability, only updated components of H are given in the rules.

When the action $\mathsf{add}(as)$ is executed by the manager M, the actor or the self-adaptive module as is added to its underlying structure (Rule AAR). The hyperedge as is added to the hyperedge set ($E' = E \cup \{as\}$), and it becomes connected to the predefined role ι through the node n_2. If as is associated to a hh-graph with the content G_{as} ($\mathsf{hhyper}(as, G_{as})$), it is added to the frame set ($F' = F \cup \{as\}$) and cts is updated to reflect the content of as. The rule RAR is used to remove an actor or a self-adaptive module as ($E' = E\backslash\{as\}$). If as is a self-adaptive module, it is removed from the frame set ($F' = F\backslash\{as\}$) and cts is updated correspondingly.

The rule JAR is used to assign the role r to as. This rule adds the node n_2 to the set of nodes connecting by the hyperedge as ($\theta' = \theta|\{(as, \theta(as) \cup \{n_2\})\}$). Similarly, execution of the primitive action $\mathsf{quit}(r, as)$ results in quitting as from the role r using the rule QAR. In this rule, as is connected to r through the node n_2 and this connection is removed by eliminating n_2 from the nodes connected by as, i.e., ($\theta'(as) = \theta(as)\backslash\{n_2\}$). If an actor or a self-adaptive module quits from all of its roles, since it has the predefined role ι, will remain as an underlying actor of the manager m.

Example 6. Let Figure 4 show the current structure of our example. Figure 8(a) illustrates the structure after the execution of $\mathsf{add}(\mathsf{clusterD})$ in Example 5 that assigns the default role ι to the self-adaptive module $\mathsf{clusterD}$. Then, execution of the action $\mathsf{join}(\mathsf{powerfuleClusterRole}, \mathsf{clusterD})$ leads to the system structure shown in Figure 8(b). To remove or add a cluster, the request receiver does not need to know the internal structure of the cluster which is an advantage of our model.

The set of nodes connected by a set X is defined as $\theta(X) = \bigcup_{e \in X} \theta(e)$. Let a self-adaptive module κ contain a manager M, the set of actors A, the set of

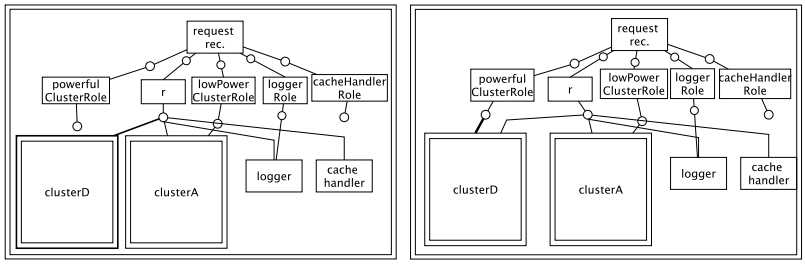

Fig. 8. The graph transformations of Example 6.

self-adaptive modules $\mathcal{K}$ and the set of roles R assigned by M. We define the well-formedness of κ's structure as follows:

Definition 7. ***Well-formed structure*** *The hh-graph $H = (G, \mathcal{K}, cts)$ describing κ's internal architecture, is well-structured if (1) H has at least a managed element, i.e. $A \cup \mathcal{K} \neq \emptyset$, (2) the manager M is only connected to the roles, i.e. $\theta(M) \subseteq \theta(R)$, (3) every role $r \in R$ is connected to M (i.e. $\exists n.\theta(M) \cap \theta(r) = \{n\}$) in addition to the actors and the self-adaptive modules (i.e. $\theta(r) \subseteq \theta(\mathcal{K}) \cup \theta(A)$), (4) every actor $a \in A$ is only connected to other actors or roles, i.e. $\theta(a) \subseteq \theta(R) \cup \theta(A \backslash \{a\})$, (5) every self-adaptive module $\kappa \in \mathcal{K}$ is connected to the role hyperedges, i.e. $\theta(\kappa) \subseteq \theta(R)$, and (6) every self-adaptive module $\kappa \in \mathcal{K}$ is well-formed.*

The following lemma states that the transformation rules used to specify the reconfiguration semantics are sound:

Lemma 1. *If H is a well-formed hh-graph showing the underlying structure of M, then H' obtained after some structural adaptations by M is also a well-formed hh-graph.*

Proof. We prove this by induction on the number of performed structural adaptations. We show the structure after n structural adaptations by H_n.

Base Case If there is no structural adaptation, $H' = H = H_0$ and the conclusion is obvious.

Inductive Step Assume it holds for n adaptations, i.e. H_n is well-formed. We should prove that H_{n+1} is well-formed. To prove this, we should prove that all six conditions are preserved by each of the rules in Figure 7.

None of the rules changes the manager M, the roles and the nodes connected by M, i.e. $\theta'(M) = \theta(M)$ and $\theta'(R) = \theta(R)$, therefore, (1), (2) and the first part of (3) are preserved by all the rules. In the first rule, two cases can happen:

- if as isn't aframe, this rule adds a new edge as that connects only the node n_2 where n_2 belongs to $\theta(\iota)$, i.e. $n_2 \in \theta(\iota)$. The updates performed by this rule include adding the new hyperedge as and setting $\theta'(as)$ to $\{n_2\}$, i.e. $\theta'(as) = \{n_2\}$. From, $n_2 \in \theta(\iota)$ and $\iota \in R$, we can conclude $\theta'(as) \subseteq \theta'(R)$ and

subsequently the item (4) holds. The self-adaptive modules do not change, i.e. $\theta'(\mathcal{K}) = \theta(\mathcal{K})$, hence (5) and (6) are followed from the inductive step hypothesis and the fact that $\theta'(\kappa) = \theta(\kappa)$ for all $\kappa \in \mathcal{K}$.
- if as is a frame, the proof of (5) will be similar to that of (4) in the previous case. This rule also adds as to the frames and (6) is trivially followed from the side-conditions of this rule (i.e. $\mathsf{wellFormed}(\mathsf{G}_{\mathsf{as}})$) and the inductive step hypothesis.

The proof for the rule RAR is similar to that of AAR. The rule JAR only updates the graph by adding n_2 to the nodes connected by as, i.e. $\theta'(as) = \theta(as) \cup \{n_2\}$. If as is a self-adaptive module, from $n_2 \in \theta'(r)$ and the assumption that $\theta(as) \subseteq \theta(R)$, it follows $\theta'(as) \subseteq \theta'(R)$ (i.e. (5) holds). The conditions (4) and (6) are respectively followed from the facts that this rule does not change nodes connected by the actors (i.e. $\theta'(A) = \theta(A)$) and no frame is added or modified by this rule (i.e. $\theta'(R) = \theta(R)$). Similarly, we can prove QAR.

7 Discussion and Related Work

In [13], the suitability of HPobSAM for modeling large-scale self-adaptive systems has been discussed, particularly, it was discussed how the hierarchical structure of this model to support centralized and decentralized adaptations, improves scalability. In [17], the authors refer to [13] and mention that how the hierarchical structure offers a form of controlled autonomy and balances agent autonomy and system controllability, for example to prevent unsafe situations caused by a selfish acting ATV. Since we use hierarchical hypergraphs and a type of graph transformation rules which allows us to add or remove components with no need to be aware of their internal structure, this feature enables us to model open evolving systems where components enter or leave at any time, while their internal structure is unknown. Moreover, we use roles to specify structure-independent adaptation logic which allows us to adapt the system without changing the adaptation logic.

Three different features - separation of concerns, computational reflection and component-based design - guarantee th flexibility of the apporach to develop self-adaptive systems.. Policies are used to adapt the system behavior and the system structure which can be changed and loaded dynamically. This feature provides a high-degree of flexibility and makes HPobSAM a suitable model to model evolving software systems. We believe this work is original in using both structural and behavioral adaptations which are directed by an identical flexible mechanism. The applicability of this model has been shown by applying it on two case studies in the areas of server clusters and an autonomous transportation system in a smart airport [13].

In [14, 15], we have compared PobSAM with existing approaches for modeling behavioral adaptation in terms of flexibility, separation of concerns and formal foundation. The main aim of the research presented here is to extend our formal approach for architectural modeling and structural adaptation of software inten-

sive systems. Hence we focus here on related work concerned with the design of software-intensive systems and formal modeling of structural adaptation.

Another related area of research is structural adaptation which has been given strong attention. Formal techniques have been extensively used to model and analyze dynamic structural adaptation (see [7]). Structural adaptation (or dynamic reconfiguration) is usually modeled using graph-based approaches (e.g. [25, 8]) or ADL-based approaches (e.g. [18, 21]). Compared to the proposed approaches based on graph transformation, we use hierarchical hypergraphs and a type of graph transformation rule which allows us to add or remove components without need to be aware of their internal structure. Moreover, most existing work concentrates on modeling structural changes [7, 6], while we have integrated both behavior and architecture in our model. The authors in [6] model the system as graphs and use graph transformation to model the system behavior. In this work, both behavior and structure are modeled with the same formalism, however handling large and complex graphs would be difficult for large-scale systems. We take the benefit of both an ordinary state-based formalism for specifying behavioral information in addition to graphs as a natural model to express the system structure.

In [3, 4], a coordinated actor model for self-adaptive track-based traffic control systems is introduced which is inspired from PobSAM and Rebeca language [23]. In coordinated actor model, unlike HPobSAM we have a centralised coordinator. Creol is a formal object-oriented language to develop open distributed systems that supports dynamic upgrading of classes [28]. While this language supports some limited levels of dynamism that can be used for behavioural adaptations (e.g. by upgrading a method) or structural adaptations (e.g. by defining new interfaces), however, (i) it is not flexible as HPobSAM is, and (ii) its supported adaptations are limited and fine-grained, e.g. one cannot remove a whole subsystem. DR-BIP [12] is a component framework for programming reconfigurable systems that supports structural adaptations. In contrast to HPobSAM, this framework does not support behavioural adaptation and is not flexible.

8 Conclusion

We provided a formal semantics for HPobSAM which is a formal model to specify structural and behavioral adaptations in large-scale systems. In this model, self-adaptive modules are used as autonomous building blocks to structure a system. We used hierarchical hypergraphs to model the system structure. The proposed semantics rules enable us to add or remove a component of which the internal structure is not given. To support reasoning about systems designed using HPobSAM, we plan to extend a tool developed in [13] to generate Maude specifications from HPobSAM models which will allow us to use the reasoning techniques provided by Maude (e.g. model checking). Furthermore, the behavioural equivalence theory proposed for PobSAM [15, 16] can be slightly extended to support graph morphisms and reason about behavioural/structural equivalence.

Acknowledgment We thank the anonymous reviewers for their helpful comments that improved the paper.

References

1. Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. Modeling and analyzing MAPE-K feedback loops for self-adaptation. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, pages 13–23, 2015.
2. Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. Formal design and verification of self-adaptive systems with decentralized control. *TAAS*, 11(4):25:1–25:35, 2017.
3. Maryam Bagheri, Ilge Akkaya, Ehsan Khamespanah, Narges Khakpour, Marjan Sirjani, Ali Movaghar, and Edward A. Lee. Coordinated actors for reliable self-adaptive systems. In *Formal Aspects of Component Software - 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers*, pages 241–259, 2016.
4. Maryam Bagheri, Marjan Sirjani, Ehsan Khamespanah, Narges Khakpour, Ilge Akkaya, Ali Movaghar, and Edward A. Lee. Coordinated actor model of self-adaptive track-based traffic control systems. *Journal of Systems and Software*, 143:116–139, 2018.
5. Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.
6. Basil Becker and Holger Giese. Modeling of correct self-adaptive systems: a graph transformation system based approach. In *Proceedings of the 5th international conference on Soft computing as trans disciplinary science and technology*, pages 508–516, 2008.
7. Jeremy S. Bradbury, James R. Cordy, Jürgen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings of 1st ACM SIGSOFT Workshop on Self-managed Systems*, pages 28–33. ACM, 2004.
8. Antonio Cansado, Carlos Canal, Gwen Salaün, and Javier Cubo. A formal framework for structural reconfiguration of components under behavioural adaptation. *Electr. Notes Theor. Comput. Sci.*, 263:95–110, 2010.
9. Constanze Deiters, Michael K"oster, Sandra Lange, Sascha Lützel, Bassam Mokbel, Christopher Mumme, and Dirk Niebuhr. Demsy- a scenario for an integrated demonstrator in a smart city. Technical report, NTH - Niedersachsische Technische Hochschule, 2010.
10. Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Comp. Syst. Sci.*, 64(2):249–283, 2002.
11. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. 2006.
12. Rim El Ballouli, Saddek Bensalem, Marius Bozga, and Joseph Sifakis. Programming dynamic reconfigurable systems. In *International Conference on Formal Aspects of Component Software*, pages 118–136. Springer, 2018.
13. Narges Khakpour, Saeed Jalili, Marjan Sirjani, Ursula Goltz, and Bahareh Abolhasanzadeh. Hpobsam for modeling and analyzing it ecosystems - through a case study. *Journal of Systems and Software*, 85(12):2770–2784, 2012.

14. Narges Khakpour, Saeed Jalili, Carolyn L. Talcott, Marjan Sirjani, and Mohammad Reza Mousavi. Pobsam: Policy-based managing of actors in self-adaptive systems. *Electr. Notes Theor. Comput. Sci.*, 263:129–143, 2010.
15. Narges Khakpour, Saeed Jalili, Carolyn L. Talcott, Marjan Sirjani, and Mohammad Reza Mousavi. Formal modeling of evolving adaptive systems. *Science of Computer Programming*, 78:3–26, 2012.
16. Narges Khakpour, Marjan Sirjani, and Ursula Goltz. Context-based behavioral equivalence of components in self-adaptive systems. In *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, pages 16–32, 2011.
17. Edward A. Lee and Marjan Sirjani. What good are models? In *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Pohang, South Korea, October 10-12, 2018, Proceedings*, pages 3–31, 2018.
18. Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.
19. Emanuela Merelli, Nicola Paoletti, and Luca Tesei. Adaptability checking in complex systems. *Science of Computer Programming*, 115-116:23–46, 2016.
20. Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley R. Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 1–12, 2015.
21. Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.
22. Wei-Min Shen, Peter M. Will, Aram Galstyan, and Cheng-Ming Chuong. Hormone-inspired self-organization and distributed control of robotic swarms. *Auton. Robots*, 17(1):93–105, 2004.
23. Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using rebeca. *Fundam. Inform.*, 63(4):385–410, 2004.
24. Morris Sloman and Emil C. Lupu. Engineering policy-based ubiquitous systems. *Computer Journal*, 53(7):1113–1127, 2010.
25. Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 179–193. Springer Berlin / Heidelberg, 2000.
26. Cynthia Villalbaa and Franco Zambonelli. Towards nature-inspired pervasive service ecosystems: Concepts and simulation experiences. *Journal of Network and Computer Applications*, 34(2):589 – 602, 2011.
27. Mirko Viroli, Matteo Casadei, Elena Nardini, and Andrea Omicini. Towards a pervasive infrastructure for chemical-inspired self-organising services. *Lecture Notes in Computer Science*, 6090 LNCS:152 – 176, 2010.
28. Ingrid Chieh Yu, Einar Broch Johnsen, and Olaf Owe. Type-safe runtime class upgrades in creol. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 202–217. Springer, 2006.