PobSAM: Policy-based Managing of Actors in Self-Adaptive Systems

Narges Khakpour[1] ,Saeed Jalili[2]

Tarbiat Modares University, Tehran, Iran

Carolyn Talcott[3]

SRI International, Menlo Park, California

Marjan Sirjani[4]

Reykjavk University, Reykjavk, Iceland
University of Tehran and IPM, Tehran, Iran

MohammadReza Mousavi[5]

Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract

In this paper, we present a formal model named PobSAM (Policy-based Self-Adaptive Model) for modeling self-adaptive systems. In this model, policies are used as a mechanism to direct and adapt the behavior of self-adaptive systems. A PobSAM model consists of a set of self-managed modules(SMM). An SMM is a collection of autonomous managers and managed actors. Managed actors are dedicated to functional behavior while autonomous managers govern the behavior of managed actors by enforcing suitable policies. To adapt SMM behavior in response to changes, policies governing an SMM are adjusted, i.e., dynamic policies are used to govern and adapt system behavior. We employ the combination of an algebraic formalism and an actor-based model to specify this model formally. Managers are modeled as meta-actors whose policies are described using an algebra. Managed actors are expressed by an actor model. Furthermore, we provide an operational semantics for PobSAM described using labeled transition systems.

Keywords: Adaptive systems, Policy-based Computing, Component-based Design, Algebra, Actor Models

[1] Email: nkhakpour@modares.ac.ir
[2] Email: sjalili@modares.ac.ir
[3] Email: clt@cs.stanford.edu
[4] Email: msirjani@ut.ac.ir
[5] Email: m.r.mousavi@tue.nl

1 Introduction

Motivation Increasingly, software systems are subjected to adaptation at run-time due to changes in the operational environments and user requirements. Adaptation is classified into two broad categories [1]: structural adaptation and behavioral adaptation. While structural adaptation aims to adapt system behavior by changing system's architecture, the behavioral adaptation focuses on modifying the functionalities of the computational entities.

There are several challenges in developing self-adaptive systems. Due to the fact that self-adaptive systems are often complex systems with greater degree of autonomy, it is more difficult to ensure that a self-adaptive system behaves as intended and avoids undesirable behavior. Hence, one of the main concerns in developing self-adaptive systems is providing mechanisms to trust whether the system is operating correctly where *formal methods* can play a key role.

Zhang et al. [2] proposed a model-driven approach using Petri Nets for developing adaptive systems. They also presented a model-checking approach for verification of adaptive system [3,4] in which an extension of LTL with "adapt" operator was used to specify the adaptation requirements. In this work, system was modeled using a labeled transition system. Furthermore, authors in [5,6] used labeled transition systems in low level of abstraction to model and verify the embedded adaptive systems. Kulkarni et al. [7] proposed a theorem proving approach to verify the structural adaptation of adaptive systems. All the proposed formal models hard-code the adaptation logic which leads to system's inflexibility.

Flexibility is another main concern to achieve adaptation. Since, hard-coded mechanisms make tuning and adapting of long-run systems complicated, so we need methods for developing adaptive systems that provide a high degree of flexibility. Recently, the use of policies has been given attention as a rich mechanism to achieve flexibility in adaptive system. A policy is a rule describing under which condition a specified subject must (can or cannot) do an action on a specific object. In [8,9,10,11,24], policies are used as structural adaptation mechanism. Additionally, [12,13] proposed architectures for engineering autonomic computing systems that use policies for behavioral adaptations.

This paper In this paper we propose a formal model called PobSAM(Policy-based Self-Adaptive Model) for developing and specifying self-adaptive systems that employs policies as the principal paradigm to govern and adapt system behavior. We model a self-adaptive system as a collection of interacting actors directed to achieve particular goals according to the predefined policies. A PobSAM model consists of a set of Self-Managed Modules(SMMs). An SMM is composed of a collection of autonomous managers and managed actors. Autonomous managers are meta-actors responsible for monitoring and handling events by enforcing suitable policies. Each manager adapts its policies dynamically in response to the changing circumstances according to adaptation policies. The behavior of managed actors is governed by managers, and cannot be directly controlled from outside.

PobSAM has a formal foundation that employs an integration of algebraic formalisms and Actor-based models. The computational (functional) model of PobSAM is based on actor-based models while an algebraic approach is proposed to specify policies of managers. Operational semantics of PobSAM is described with labeled transition systems. The proposed model is suitable for cases where a set of predefined policies is known in advance.

In our previous work [14], we proposed a formal model for policy-based self-adaptive systems using an actor-based language Rebeca [15]. In [14], we added policies as rules to the Rebeca code and we focused on policy conflict detection. Combining adaptation concerns with system functionality in [14] increases the complexity of the model as well as the formal verification process. In order to address these drawbacks, we need an approach in which adaptation concerns are separated from system functionality. Here, we extracted the policy rules, specified by algebra, from Rebeca code. Moreover, the policies are presented in two classes, separating the policies governing the actors behavior from the policies which determine the adaptation strategy (when and how the system passes the adaptation phase safely). Additionally, here we added modules as an encapsulation mechanism in which each module can manage itself autonomously.

Contribution Formal methods are proposed for analysis of adaptive systems, mainly in low levels of abstraction, and flexible policy-based approaches are proposed for designing adaptive systems without formal foundation. Here, we propose a flexible policy-based approach with formal foundation to support modeling and verification of self-adaptive systems. Policies allow us to separate the rules that govern the behavioral choices of a system from the system functionality giving us a higher level of abstraction, so, we can change system behavior without changing the code or functionality of the system. We are also concerned about the adaptation strategy, to pass the adaptation phase safely and in the right moment. As an example, we are able to change and reason about the scheduling of jobs using policies independent of the system code. Although our approach can support both structural and behavioral adaptation, in this paper we focus on the behavioral adaptation. The formal foundation, the modular model, and separation of adaptation rules will help us in developing rigorous analysis techniques.

Structure of the paper This paper is organized as follows. In Section 2 we introduce an example to illustrate our approach. Section 3 introduces the PobSAM model in brief. Sections 4 and 5 introduce the syntax and Semantics of PobSAM respectively. Section 6 presents related work and compares our approach with the existing approaches. In Section 7, we present our conclusions and plans for the future work.

2 Smart Home

In a home automation system, sensors are devices that provide smart home with the physical properties of the environment by sensing the environment. In addition, actuators are physical devices that can change the state of the world in response to the sensed data by sensors. The system processes the data gathered by the sensors, then it activates the actuators to alter the user environment according to the predefined set of policies. Smart homes can have different features. Here, we take into account three features including: (1) The lighting control which allows lights to switch on/off automatically depending on several factors. In addition, the intensity of the lights placed in a room can be adjusted according to the predefined policies. (2) Doors/Windows management that enable inhabitants to manage windows and doors automatically. In addition, if windows have blinds, these should be rolled up and down automatically too. (3) Heating control which allows inhabitants to adjust the heating of the house to their preferred value. The heating control will adjust itself automatically in order to save energy.

The smart home system is required to adapt its behavior according to the changes of the environment. To this aim, we suppose system runs in normal, vacation and fire modes and in each context it enforces various sets of policies to adapt to its current conditions. For the reason of space, here we only identify policies defined for lighting control module while the system runs in normal and fire modes as follows:

Defined policies in normal mode

P1 Turn on the lights automatically when night begins.

P2 Whenever someone enters an empty room, the light setting must be set to default.

P3 When the room is reoccupied within T1 minutes after the last person has left the room, the last chosen light setting has to be reestablished.

P4 The system must turn the lights off, when the room is unoccupied.

Defined policies in fire mode

P1 Turn on the emergency light.

P2 Disconnect power outlets.

P3 Upon putting out fire, turn off the emergency light.

3 Modeling Concepts of PobSAM

Self-Managed Module (SMM) is the policy-based building block of PobSAM. A PobSAM is composed of a set of SMMs that an SMM may contain a number of SMMs structured hierarchically. An SMM is a set of actors which can manage their behavior autonomously according to predefined policies. PobSAM supports interactions of an SMM with the other SMMs in the model. To this aim, each SMM provides well-defined interfaces for interaction with other SMMs. In the smart home case study, we consider three SMMs including *LightClModule*, *TempClModule* and *DWClModule* to manage lighting, temperature and door/windows respectively.

An SMM structure can be conceptualized as the composition of three layers illustrated in Figure 1.

- **Managed Layer** This layer is dedicated to the functional behavior of SMM and contains computational actors. Actors are governed by autonomous managers using policies to achieve predefined goals. Henceforth, we use the terms managed actors and actors interchangeably.
- **Autonomous Managers Layer** Autonomous managers are meta-actors that can operate in different configurations. Each configuration consists of two classes of policies: governing policies, and adaptation policies. Using governing poli-

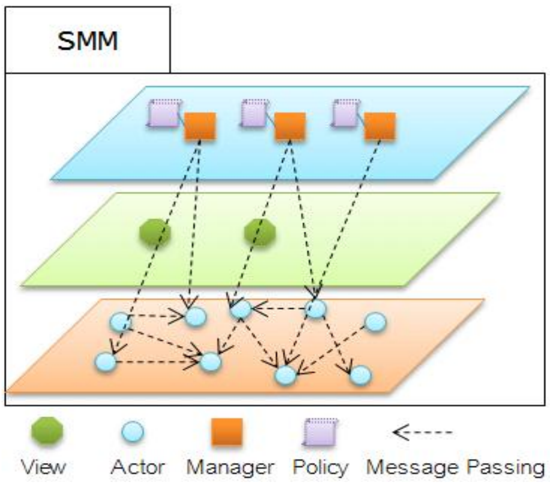

Fig. 1. The PobSAM Model

cies, the manager directs the behavior of actors by sending messages to them. Adaptation policies are used to switch between different configurations to adapt the module behavior properly. Moreover, a manager may contain a set of local variables.

- **View Layer** In PobSAM, each actor provides its required state information to the relevant managers. Not all aspects of the operational environment have direct influence on the behavior of managers, the views provide only the required information for managers. The view layer is composed of views that provides a view or an abstraction of an actor's state that is adequate for the managers' needs. The distinction between the underlying computational environment and the required state information of actors makes analyzing managers much simpler.

Example 3.1 In our example, *LightClModule* module contains an autonomous manager named *LightMngr*, two *light* actors (*light1* and *light2*), an outlet actor and a number of views indicating the overall light intensity of room and the status of the lights.

4 PobSAM Syntax

Figure 2 shows a typical SMM containing manager meta-actors, views and actors which we elaborate in the sequel.

4.1 Actors

The encapsulation of state and computation, and the asynchronous communication make actors a natural way to model distributed systems. Therefore, we use an actor-based model to specify the computational environment of a self-adaptive system. To this aim, an extension of Rebeca is used. Rebeca is an actor-based language [15] for modeling concurrent asynchronous systems which allows us to model the system as a set of reactive objects called rebecs interacting by message passing. Each rebec provides methods called message servers (msgsrv) which can be invoked

```
SMM SMM1
 Managers
   Manager ManagerName1(InitialConfiguration11)
     // definition of local variables
     <ViewName11,..., ViewName1n>;
     Configurations
         ConfigurationName1={gp11,,gp1p}<ap11|  |ap1q>;
         .
         .
         .
     EndC
     Policies
         //Definition of governing policies(gps)
         GoverningPolicyName1: on eventi if condi do actionsi;
         .
         .
         .
         //Definition of adaptation policies(aps)
         AdaptationPolicyName1:
              on eventj if condj switchto Configuration1 when condk;
         .
         .
         .
     EndP
   EndM
  // definition of other managers
 EndMS
 Views //definition of views
   Datatype1 ViewName1 as expr1;
   Datatype2 ViewName2 as expr2;
     .
     .
     .
 EndV
 Actors//definition of actors
   reactiveclass Classname1() {
      Knownrebecs{}
      Statevars{  Public datatype v1;
                  Private datatype v2;}
      msgsrv initial() {}
      msgsrv msgsrv1(){}
   }
   main {
      ...
      Classname1 rebec1(...):(...);
      ...
   }
 EndA
 EndS
```

Fig. 2. The Typical Syntax of a PobSAM Model

by others. Each rebec has an unbounded buffer for coming messages named queue. Furthermore, the rebecs' state variables (statevars) are responsible of capturing the rebec state. The known rebecs of a rebec (Knownrebecs) denotes the rebecs which it can send messages. In our extension, an actor can expose a number of its state variables to the managers (Figure 2). The exposed state variables are used in the definition of views.

Example 4.1 In the *LightClModule* SMM, the managed layer comprises a set of *light* rebecs controlled by *LightMngr* based on their current policies. We consider a reactive class named *Light* to model the lights which contains *setIntensity*, *switchOn* and *switchOff* message servers as well as *intensity* and *status* state variables.

4.2 Views

In PobSAM, the views are defined in terms of the public state variables of actors. a view variable could be the actual state variable, or a function or a predicate applied to state variables. Views enable managers not to be concerned about internal behavior of actors and they provide an abstraction of actor's state to managers.

Example 4.2 In the *LightClModule* SMM, the *LightMngr* does not require the exact values of the lights intensities and providing overall intensity as low, medium or high values is sufficient to decide. The overall intensity is defined based on the *intensity* statevar of the *light* rebecs as a view.

4.3 Managers

In our model, policies direct the system behavior, and adaptation is achieved by changing policies. A manager can be in various configurations enforcing different policy sets. As shown in Figure 2, a manager is defined in terms of its possible configurations, its view of the actor layer and its local variables.

Governing Policies

Whenever a manager receives an event, it identifies all the policies that are triggered by that event. For each of these policies, the policy condition is evaluated if one exists. If the condition evaluates to true, the action part of the triggered policy is requested to execute by instructing the relevant rebecs to perform actions through sending asynchronous messages. We express governing policies using a simple algebra as follows, in which P and Q indicate the policy sets:

$$P, Q \overset{\text{def}}{=} P \cup Q | P - Q | P \cap Q | \{p\} | \emptyset$$

$P \cap Q$ means that intersection of policy sets P and Q is used to direct actors. P-Q reduces policy set P by eliminating all the policies in the second set Q. $P \cup Q$ represents the union of P and Q governing the actors simultaneously. {p} denotes a policy set with the simple policy p as its member. $P \cap Q$ and $P \cup Q$ are commutative and associative.

A simple action policy p=$[o,\varepsilon,\psi,\alpha]$ consists of priority o, event ε , optional condition ψ and action α. In PobSAM, events are defined as the execution of a message server, sending a message to a rebec, creating new actor or a specific condition holds in the system. Actions can be composite or simple. A simple action is in form of $r.\ell(v)$ which denotes message $\ell(v)$ is sent to actor r. Composite actions are created by composing simple actions as follows:

$$\alpha, \beta \overset{\text{def}}{=} \alpha \to \beta | \alpha \parallel \beta | \alpha + \beta | [\omega?\alpha : \beta] | r.\ell(v)$$

Thus a composite action can be the sequential the sequential $(\alpha \to \beta)$ or parallel $(\alpha \parallel \beta)$ execution of actions α and β. Also, an action can be chosen to execute non-deterministically(+). Term $([\omega?\alpha : \beta])$ represents that action α is chosen to be executed if ω holds, else β will be chosen. + and $\|$ are commutative and associative.

Example 4.3 The governing policies of the *lightMngr* in the normal configuration are as follows where *night*, *occpd*, *rccpd* and *unoccpd* denote events and *c1*, *c2*, *d1* and *d2* are arguments indicating the light intensity.

$$p_{n1} \overset{\text{def}}{=} [1, night, true, light1.switchon() \parallel light2.switchon()]$$
$$p_{n2} \overset{\text{def}}{=} [2, occpd, true, light1.setIntensity(d1) \parallel light2.setIntensity(d2)]$$
$$p_{n3} \overset{\text{def}}{=} [3, rccpd, true, light1.setIntensity(c1) \parallel light2.setIntensity(c2)]$$
$$p_{n4} \overset{\text{def}}{=} [4, unoccpd, true, light1.switchoff() \parallel light2.switchoff()]$$

Adaptation Policies
One of the main characteristics of a formal model to specify a self-adaptive system is considering adaptation semantics. To this end, we should deal with a number of issues such as when should an adaptation be performed in the system? Relating to PobSAM, when must the manager's policies be modified? Following modifying policies, when do the enforcement of new policies begin? Is the system behavior restricted during adaptation?

Whenever an event requiring adaptation occurs, relevant managers in different SMMs are informed. However, the adaptation cannot be done immediately and when the system reaches a safe state, the managers switch to the new configuration. Therefore, we introduce a new mode of operation named adaptation mode in which a manager runs before switching to the next configuration. While the manager is in the adaptation mode, it is likely that events occur which need to be handled by managers. To handle these cases, we introduce two kinds of adaptations named loose adaptation and strict adaptation. Under loose adaptation a manager enforces old policies, while under strict adaptation all events will be ignored until the manager exits the adaptation mode and the system reaches the safe state. For example in our smart home example, when *LightMngr* is in the fire configuration and there is a request for the vacation mode, while fire has not been put out it keeps enforcing policies of the fire configuration by switching to the loose adaptation mode. Also, when *LightMngr* is in the normal configuration, once fire is detected, it stops enforcing its current policies by switching to the strict adaptation mode.

A simple configuration C is defined as $C \overset{\text{def}}{=} \langle P, A\rangle$ where P and A indicate the governing policy set and the adaptation policy of C respectively. Adaptation policies are defined using an algebraic language as follows:

$$A \overset{\text{def}}{=} \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta} | A \oplus A$$

in which D, δ , γ , λ and ϑ, respectively denote an arbitrary configuration, the conditions of triggering adaptation, the conditions of applying adaptation, adaptation type (loose or strict) and the priority of adaptation policy. The simple adaptation policy, $\lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta}$, specifies when the triggering condition δ holds and there is no other adaptation policy with the higher priority, manager evolves to the strict or loose adaptation mode based on the value of λ. When the condition of applying adaptation γ becomes true, it will perform adaptation. Adaptation policies of a manager is defined as the composition, $\oplus$, of the simple adaptation policies. Here, composition of two policies means that those policies are potentially to be triggered. $\oplus$ is associative and commutative. D is defined as follows where ω is an arbitrary condition:

$$D, D' \stackrel{\text{def}}{=} [\omega?D : D'] | D \Box D' | C$$

Terms $[\omega?D : D']$ and $D \Box D'$ represent conditional choice and non-deterministic choice respectively. In conditional choice, configuration D is chosen if ω holds, else D' will be chosen. Non-deterministic choice means that the choice between configuration D or D' is made non-deterministically. This operator is associative and commutative.

Example 4.4 In our smart home example, there are three configurations including C_n, C_v and C_f. Formal specification of *lightMngr*'s configurations are as follows in which p_{mj} denotes policy j defined in mode C_m. *vacReq*, *fire*, *firePutout* and *comebackHome* are events while *isPutout* , *isCnfrmd* and *onVac* are *lightMngr*'s local variables. As an example, when the *lightMngr* is in the *fire* mode and there is a request for going on vacation, while fire has net been put out, it can not switch to the C_v configuration.

$C_n \stackrel{\text{def}}{=} \langle \{p_{n1}, p_{n2}, p_{n3}, p_{n4}\}, \lfloor C_f \rfloor_{fire,true,S,1} \oplus \lfloor C_v \rfloor_{vacReq,isCnfrmd,L,2} \rangle$
$C_f \stackrel{\text{def}}{=} \langle \{p_{f1}, p_{f2}, p_{f3}\}, \lfloor onVac?C_v : C_n \rfloor_{firePutout,true,L,1} \oplus \lfloor C_v \rfloor_{vacRequest,isPutout,L,2} \rangle$
$C_v \stackrel{\text{def}}{=} \langle \{p_{v1}, p_{v2}, p_{v3}, p_{v4}, p_5\}, \lfloor C_f \rfloor_{fire,true,S,1} \oplus \lfloor C_n \rfloor_{comebackHome,true,L,2} \rangle$

5 Operational Semantics of PobSAM

5.1 Operational Semantics of Actors and Views

The operational semantics of our extension of Rebeca does not differ from Rebeca. However, any changes in state of the rebecs used in the definition of views must be reflected to the views. Let $I_1, I_2, \ldots, I_n$ denote the defined views of SMM S and η denote the set of defined events that S is concerned with. The state of a view is determined by its current value that is modified by the related events occurring at the actor level. After execution of a message server, the changes of public state variables must be reflected in the views state, too. We specify the operational semantics of the view layer as a labeled transition system.

Let S_B, A_B and $T_B \subseteq S_B \times A_B \times S_B$ be the set of states, the set of actions and the state transition relation of the transition system of the actor layer respectively. The state transition relation of the view layer $T_I \subseteq S_I \times A_I \times S_I$ is defined based on T_B, where S_I and A_I are the set of states and the set of actions of the view layer transition system respectively and $S_I = \langle I_1, I_2, ..., I_n \rangle$. Suppose $I_j(x_1, x_2, .., x_m)$ denotes an arbitrary view defined on public state variables $x_1, x_2, .., x_m$ and $I_j|_{\sigma_s}$ denotes the state of I_j where its state variables are substituted with their corresponding values in state σ_s. For each triple $\langle \sigma_s, a, \sigma_t \rangle \in T_B$, we consider an associated transition $\langle \sigma'_s, a, \sigma'_t \rangle \in T_I$ where ${\sigma'}_s = \langle I_1|_{\sigma_s}, I_2|\sigma_s, ..., I_n|\sigma_s \rangle$ and $\sigma'_t = \langle I_1|_{\sigma_t}, I_2|_{\sigma_t}, ..., I_n|_{\sigma_t} \rangle$ if and only if $a \in \eta$ or $\exists I_k | 1 \leq k \leq n \wedge I_k|_{\sigma_s} \neq I_k|_{\sigma_t}$, i.e.

$$\frac{\sigma_s \xrightarrow{a} \sigma_t \in T_B, (a \in \eta) \vee (\exists I_k | 1 \leq k \leq n \wedge I_k|_{\sigma_s} \neq I_k|_{\sigma_t})}{\sigma'_s \xrightarrow{a} \sigma'_t}$$

5.2 *Operational Semantics of Managers*

We use a labeled transition system to define the operational semantics of managers in which labels have two components. The first component indicates the activation condition of the transition while the second component denotes the action of the transition. Assume that M is a logical expression defined on state variables, the transitions are of the form $P\xrightarrow{M,a}Q$ meaning "if M holds then P has an action a leading to Q". Henceforth, we denote a transition by $P\xrightarrow{\mu}Q$ where $\mu = (M, a)$.

The behavior of a manager depends on the mode in which it is running. A manager can run in different modes such as normal execution, adaptation and policy enforcement. To distinguish managers in different modes, we use different notations. Let $[\mathcal{M}]^s_{\mathcal{R},p}C[\alpha]$ indicate manager $\mathcal{M}$ in the enforcement mode in which,

- C is the configuration in which $\mathcal{M}$ is running and $C \overset{\text{def}}{=} \langle P, A\rangle$.
- $\mathcal{R}$ is the set of triggered policies to be enforced.
- p is the current policy being enforced by $\mathcal{M}$.
- α is the action of a recent policy being executed by $\mathcal{M}$.
- s denotes $\mathcal{M}$'s view of current context in addition to its local variables.

$\mathcal{M}^s_{\emptyset,\emptyset}C[0]$, $|\mathcal{M}|^s_{\mathcal{R},p}C[\alpha]$ and $\|\mathcal{M}\|^s_{\emptyset,\emptyset}C[0]$ indicate $\mathcal{M}$ in normal execution, loose adaptation and strict adaptation modes respectively. We ignore s in the definition of operational semantics of managers.

Policy enforcement semantics

Whenever an event is received by a manager, it identifies all the triggered policies with the policy condition evaluated to true. Then it enforces the identified policies based on their priorities. Once a manager enforces all the triggered policies, it evolves to normal mode. Figure 5.1 gives rules of policy enforcement.

Using NPE1 manager switches to the enforcement mode by identifying the triggered policies to be enforced. NPE2 places the action of a policy with the highest priority in the action part of the manager to be run. NPE3, NPE4 and NPE5 define the semantics of non-deterministic and conditional choice. NPE7 is considered to apply two sequential actions in which r' is an arbitrary actor. To this aim, we used synchronous message passing provided by Extended Rebeca [16]. The corresponding actors of two sequential actions are synchronized after execution of the first action. We considered two message servers for each reactive class named $sendAck(r)$ and $waitAck(r)$ which sends and receives a synchronization message to and from rebec r respectively. NPE8 expresses sending a message to an actor and removing it from the list of actions to be executed. After applying policy p, NPE9 will remove p from the list of activated policies. When there is no policy to be enforced, manager will switch to normal execution mode using NPE10.

As mentioned above, managers in loose adaptation mode are able to enforce policies. Therefore, all the rules introduced for the enforcement mode, except NPE10, are applicable in loose adaptation mode too.

$$(\text{NPE1})\frac{\mathcal{R} = \{\text{p}|(p.\varepsilon \wedge p.\psi) = true\} \wedge \mathcal{R} \neq \emptyset}{\mathcal{M}_{\emptyset,\emptyset}C[0] \xrightarrow{\varepsilon,\tau} [\mathcal{M}]_{\mathcal{R},\emptyset}C[0]} \quad (\text{NPE2})\frac{p \in \mathcal{R} \wedge (\nexists q \in \mathcal{R}|q.o > p.o)}{[\mathcal{M}]_{\mathcal{R},\emptyset}C[0] \xrightarrow{p.\varepsilon\wedge p.\psi,enf(p)} [\mathcal{M}]_{\mathcal{R},p}C[p.\alpha]}$$

$$(\text{NPE3})\frac{\alpha \longrightarrow \alpha'}{[\mathcal{M}]_{\mathcal{R},p}C[\alpha+\beta] \xrightarrow{true,\tau} [\mathcal{M}]_{\mathcal{R},p}C[\alpha']}$$

$$(\text{NPE4})\frac{}{[\mathcal{M}]_{\mathcal{R},p}C[[\omega?\alpha:\beta]] \xrightarrow{\omega,\tau} [\mathcal{M}]_{\mathcal{R},p}C[\alpha]} \quad (\text{NPE5})\frac{}{[\mathcal{M}]_{\mathcal{R},p}C[[\omega?\alpha:\beta]] \xrightarrow{\neg\omega,\tau} [\mathcal{M}]_{\mathcal{R},p}C[\beta]}$$

$$(\text{NPE6})\frac{\alpha \to \alpha'}{[\mathcal{M}]_{\mathcal{R},p}C[\alpha \parallel \beta] \xrightarrow{true,\tau} [\mathcal{M}]_{\mathcal{R},p}C[\alpha' \parallel \beta]}$$

$$(\text{NPE7})\frac{}{[\mathcal{M}]_{\mathcal{R},p}C[r.\ell(v) \to r'.\ell'(v')] \xrightarrow{true,\tau} [\mathcal{M}]_{\mathcal{R},p}C[r.\ell(v) \to r.sendAck(r') \to r'.waitAck(sendAck(r')) \to r'.\ell'(v')]}$$

$$(\text{NPE8})\frac{}{[\mathcal{M}]_{\mathcal{R},p}C[r.\ell(v) \to \beta] \xrightarrow{true,send(r,\ell(v))} [\mathcal{M}]_{\mathcal{R},p}C[\beta]}$$

$$(\text{NPE9})\frac{}{[\mathcal{M}]_{\mathcal{R},p}C[0] \xrightarrow{true,\tau} [\mathcal{M}]_{\mathcal{R}-p,\emptyset}C[0]} \quad (\text{NPE10})\frac{}{[\mathcal{M}]_{\emptyset,\emptyset}C[0] \xrightarrow{true,\tau} \mathcal{M}_{\emptyset,\emptyset}C[0]}$$

Fig. 5.1. Rules of policy enforcement

Policy adaptation semantics

For the sake of readability, we omit p, $[\alpha]$ and $\mathcal{R}$ symbols of managers in normal execution and strict adaptation modes. Figure 5.2 shows rules for adaptation in strict mode in which B, A′ and B′ denote arbitrary adaptation policies. Furthermore, F and D'' denote a simple adaptation policy and an arbitrary configuration, respectively.

$$(\text{SA1})\frac{\nexists F \in A|(\vartheta < F.\vartheta \wedge F.\delta = true)}{\mathcal{M}\langle P, \lfloor D\rfloor_{\delta,\gamma,\lambda,\vartheta} \oplus A\rangle \xrightarrow{\delta\wedge\lambda,\tau} \|\mathcal{M}\|\langle P, \lfloor D\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$$

$$(\text{SA2})\frac{}{\|\mathcal{M}\|\langle P, \lfloor D\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\gamma,adapt} \mathcal{M}_{\emptyset,\emptyset}D}$$

$$(\text{SA3})\frac{\mathcal{M}\langle P, B\rangle \xrightarrow{\mu} \mathcal{M}\langle P, B'\rangle}{\mathcal{M}\langle P, A\oplus B\rangle \xrightarrow{\mu} \mathcal{M}\langle P, A\oplus B'\rangle} \quad (\text{SA4})\frac{\mathcal{M}\langle P, A\rangle \xrightarrow{\mu} \mathcal{M}\langle P, A'\rangle}{\mathcal{M}\langle P, A\oplus B\rangle \xrightarrow{\mu} \mathcal{M}\langle P, A'\oplus B\rangle}$$

$$(\text{SA5})\frac{\|\mathcal{M}\|\langle P, \lfloor D\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\mu} \|\mathcal{M}\|\langle P, \lfloor D'\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}{\|\mathcal{M}\|\langle P, \lfloor D\Box D''\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\mu} \|\mathcal{M}\|\langle P, \lfloor D'\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$$

$$(\text{SA6})\frac{\|\mathcal{M}\|\langle P, \lfloor D\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\mu} \|\mathcal{M}\|\langle P, \lfloor D'\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}{\|\mathcal{M}\|\langle P, \lfloor D''\Box D\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\mu} \|\mathcal{M}\|\langle P, \lfloor D'\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$$

$$(\text{SA7})\frac{}{\|\mathcal{M}\|\langle P, \lfloor \omega?D{:}D'\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\omega,\tau} \|\mathcal{M}\|\langle P, \lfloor D\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$$

$$(\text{SA8})\frac{}{\|\mathcal{M}\|\langle P, \lfloor \omega?D{:}D'\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\neg\omega,\tau} \|\mathcal{M}\|\langle P, \lfloor D'\rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$$

Fig. 5.2. Rules of strict adaptation

SA1 states that in case of strict adaptation when adaptation policy conditions hold, manager $\mathcal{M}$ switch to the strict adaptation mode. SA2 asserts that when the condition for applying the adaptation holds, $\mathcal{M}$ will evolve to normal mode and run configuration D. SA5, SA6, SA7 and SA8 define the semantics of non-deterministic and conditional choice of configurations. SA5, SA6, SA7 and SA8 rules have the higher priority than SA2. To this aim, we use the ordered SOS(Structural Operational Semantics) framework [17] and place SA5, SA6, SA7 and SA8 above SA2. Rules of loose adaptation are identical to strict adaptations rules except for SA1 which is as $\dfrac{\nexists F \in A|(\vartheta < F.\vartheta \wedge F.\delta = true)}{\mathcal{M}\langle P, \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta} \oplus A\rangle \xrightarrow{\delta\wedge\neg\lambda,\tau} |\mathcal{M}|\langle P, \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$(LA1).

5.3 Interaction of Managers and Views

The other kind of transitions is related to the interaction of managers and view layer. Figure 7 demonstrates rules for interaction of managers and the view layer containing both internal and external views of S where $\sigma_s \xrightarrow{a} \sigma_t \in \mathrm{T_I}$ and t indicates the new state of $\mathcal{M}$. s and t are defined as the projection of σ_s and σ_t on M's view respectively, i.e. $s = \sigma_s \uparrow M.v$ and $t = \sigma_t \uparrow M.v$. Construct $\sigma_s \uparrow M.v$ denotes only the state variables of σ_s that are in $M.v$ too. IR1, IR2, IR3 and IR4 express changing $\mathcal{M}$'s view of view layer by state changing at the view layer in the normal, enforcement, loose adaptation and strict adaptation modes respectively.

$$\text{(IR1)}\frac{\sigma_s \xrightarrow{a} \sigma_t, \exists I_j \in s|I_j|_{\sigma_s} \neq I_j|_{\sigma_t}}{\mathcal{M}^s_{\emptyset,\emptyset}\langle P, A\rangle[0] \xrightarrow{true,\tau} \mathcal{M}^t_{\emptyset,\emptyset}\langle P, A\rangle[0]} \qquad \text{(IR2)}\frac{\sigma_s \xrightarrow{a} \sigma_t, \exists I_j \in s|I_j|_{\sigma_s} \neq I_j|_{\sigma_t}}{[\mathcal{M}]^s_{\emptyset,\emptyset}\langle P, A\rangle[0] \xrightarrow{true,\tau} [\mathcal{M}]^t_{\emptyset,\emptyset}\langle P, A\rangle[0]}$$

$$\text{(IR3)}\frac{\sigma_s \xrightarrow{a} \sigma_t, \exists I_j \in s|I_j|_{\sigma_s} \neq I_j|_{\sigma_t}}{|\mathcal{M}|^s_{\mathcal{R},p}\langle P, A\rangle[0] \xrightarrow{true,\tau} |\mathcal{M}|^t_{\mathcal{R},p}\langle P, A\rangle[0]} \qquad \text{(IR4)}\frac{\sigma_s \xrightarrow{a} \sigma_t, \exists I_j \in s|I_j|_{\sigma_s} \neq I_j|_{\sigma_t}}{\|\mathcal{M}\|^s_{\emptyset,\emptyset}\langle P, A\rangle[0] \xrightarrow{true,\tau} \|\mathcal{M}\|^t_{\emptyset,\emptyset}\langle P, A\rangle[0]}$$

Fig. 7. Rules for interaction of managers and the view layer

6 Discussion and Related Work

Flexibility of self-adaptive systems is realized by three different features including separation of concerns, computational reflection and component-based design [18]. We explain PobSAM address these requirements in the sequel.

PobSAM decouples the adaptation logic of an SMM from its business logic described at an abstract level using policies. Among the proposed formal approaches to model adaptive systems, [2,20,21] combine the adaptation logic into the business logics. While in [3,6,19] the adaptation concerns have been separated, however, all the proposed formal models hard-code the adaptation logic which leads to system's inflexibility. The proposed model permits us to direct/adapt system behavior by enforcing/modifying policies at an abstract level without re-coding actors and managers; thereby it leads to increasing system flexibility and scalability.

Computational reflection is the ability of a system to monitor and change its behavior subsequently. In PobSAM, managers monitor actor's behavior through views and direct/adapt SMMs behavior. Policies provide us a high-level description of what we want without dealing with how to achieve it. Thus, it can be a suitable mechanism to determine if the goals were achieved using existing policy refinement techniques.

Furthermore, PobSAM uses SMM as a policy-based building block for a modular model where each component is able to adapt its behavior autonomously. This notion makes PobSAM a suitable model to specify self-organizing and cooperating systems too. Although, in this paper we focused on behavioral adaptation, however, PobSAM can support structural adaptation by joining/leaving an actor to/from an SMM dynamically, which is an advantage over the most existing approaches that concentrate on one adaptation type. SMM notion is similar to Self-Managed Cell (SMC) notion proposed in [13] as a paradigm for engineering ubiquitous systems. In this work, an SMC consists of a set of components that constructs an autonomous management domain.

One of the main aspects of modeling a self-adaptive system is specifying adaptation requirements. To this aim, we introduced a two phases adaptation strategy to pass the adaptation phase safely. Upon receiving an adaptation event by a manager, it switches to the adaptation mode. Adaptation mode models transient states during adaptation. When the system reaches a safe state, the adaptation is completed by evolving the manager to the new configuration. We believe that the modular nature of adaptation policies enables us to express adaptation requirements easily and at the high-level of abstraction.

As stated above, PobSAM has decoupled the adaptation layer from the functional layer. Thus, we can verify the adaptation layer independently from the actor layer provided that we have a labeled transition system modeling view behavior. This feature can decrease the complexity of verification procedure.

Dynamic adaptation is a very diverse area of research. While structural adaptation has been given strong attention in the research community(see [22]), fewer approaches tackle behavioral adaptation as we considered. Due to the lack of space, we restrict ourselves to present related work done on formal modeling of self-adaptive systems in addition to applying policy-based approaches in engineering of self-adaptive systems.

Formal verification of adaptive systems is a young research area [23] and only a few research groups already focused on this topic. A model-driven approach was proposed for developing adaptive systems in [2]. In this approach, there are different behavioral variants of a process modeled as Petri nets. At each time, one Petri net runs and reconfiguration is carried out by switching between various Petri nets. In another work [3], they modeled a system as a set of steady-state programs among which the system switches. An extension of LTL with “adapt“ operator was used to specify adaptation requirements before, during and after adaptation [4]. Then, they use a model checking approach to verify the system. Kulkarni et al. [7] proposed an approach based on the concept of proof lattice to verify if a system is in a correct state during and after adaptation in terms of satisfying the transitional-invariants. Furthermore, Schneider et al. [5] presented a method to describe adaptation behavior at an abstract level. After deriving transition systems from a system description, they verify the system using model checking. In their later work [6], they proposed a framework for model-based development of adaptive embedded systems using labeled transition systems. In this work, they verify different properties using theorem proving, model checking and specialized verification methods. [19] proposed a coordination protocol for distributed adaptation of the component-based systems and used Colored Petri Nets for formal verification. In our model, adaptation is performed by applying suitable policies in different contexts, which in nature differs from the proposed approaches. We have proposed a formal model of policy-based self-adaptive systems using Rebeca concentrated on policy conflict detection [14]. Combining adaptation concerns with system functionality in this approach causes an increase in the complexity of model as well as formal verification process. To the best of our knowledge, there is no other work on the formal specification of policy-based self-adaptive systems to compare our approach with.

Employing policies as a paradigm to adapt self-adaptive systems has been given considerable attention during recent years. Works in [8,9,10,11,24,26] used policies as the adaptation logic for structural adaptation, while we use policies as a mechanism for governing as well as adapting system behavior. Furthermore, [24] used policies for a simple type of behavioral adaptation named parameterization, too. [25] proposed an adaptive architecture for management of differentiated networks which performs adaptation by enabling/disabling a policy from a set of predefined QoS policies, but this architecture does not have formal foundation. Anthony [26] presents a policy definition language for autonomic computing systems in which the policies themselves can be modified dynamically to match environmental conditions. However, this work does not deal with modeling system and it is limited to proposing an informal policy language.

7 Conclusion and future works

We proposed PobSAM as a formal model to specify self-adaptive systems which uses policies as the main mechanism to govern and adapt the system behavior. To this aim, we model a system as the composition of a set of autonomous components named SMMs. Each SMM contains two types of actors: managed actors that are dedicated to the functional layer of system and autonomous managers that coordinate actors to achieve predefined goals using policies. This model integrates two formal methods including algebra and actor-based model to specify system. Then, we presented the operational semantics of PobSAM by means of the labeled transition systems.

There is much more research to pursue in the area of verification of self-adaptive systems. In this paper, we focused on formal modeling of self-adaptive systems. Verification of different properties of adaptation and functional layers of PobSAM models is an ongoing work. We are going to implement a tool to support our approach too. As our model can support both behavioral and structural adaptations, our future researches will be concentrated on specifying structural adaptations. Extending this model for modeling self-organizing systems in which managers need to coordinate together is considered as a future work, too.

References

[1] C. Hofmeister, “Dynamic Reconfiguration, Ph.D. Thesis” in Computer Science Department, University of Maryland, College Park 1993.

[2] J. Zhang and B. Cheng, “Model-Based Development of Dynamically Adaptive Software”, in Proceedings of International Conference on Software Engineering, 2006, pp. 371-380.

[3] J. Zhang, H. J. Goldsby, and B. H. C. Cheng, “Modular verification of dynamically adaptive systems”, in the 8th ACM international conference on Aspect-oriented software development, Charlottesville, Virginia, 2009, pp. 161-172.

[4] J. Zhang and B. H. C. Cheng, “Using temporal logic to specify adaptive program semantics”, Journal of Systems and Software, Architecting Dependable Systems, vol. 79, pp. 1361-1369, 2006.

[5] K. Schneider, T. Schuele, and M. Trapp, “Verifying the adaptation behavior of embedded systems”, in Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, Shanghai, China, 2006, pp. 16 - 22.

[6] R. Adler, I. Schaefer, T. Schuele, and E. Vecchie, “From Model-Based Design to Formal Verification of Adaptive Embedded Systems”, in Proceedings of International Conference on Formal Engineering Methods, 2007, pp. 76-95.

[7] S. S. Kulkarni and K. N.Biyani, “Correctness of Component-Based Adaptation”, in Component-Based Software Engineering, 2004, pp. 48-58.

[8] R. J. Anthony and C. Ekelin, “Policy-driven self-management for an automotive middleware”, in Proceedings of 1st International Workshop on Policy-Based Autonomic Computing Florida, USA, 2007.

[9] C. Efstratiou, K. Cheverst, N. Davies, and A. Friday, “An Architecture for the Effective Support of Adaptive Context-Aware Applications”, in Proceedings of the 2nd International Conference in Mobile Data Management (MDM‘01), Hong Kong, 2001, pp. 15-26.

[10] C. Efstratiou, K. Cheverst, N. Davies, and A. Friday, “Utilising the Event Calculus for Policy Driven Adaptation in Mobile Systems”, in Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks, Monterey, California., 2002, pp. 13-24.

[11] G. Phil and B. Lynne, “A Framework for Policy Driven Auto-adaptive Systems Using Dynamic Framed Aspects”, in Transactions on Aspect-Oriented Software Development II. vol. LNCS 4242, 2006, pp. 30-65.

[12] “Autonomic computing”, IBM Systems Journal, vol. 42, 2003.

[13] M. Sloman and E. Lupu, “Engineering Policy-Based Ubiquitous Systems”, The Computer Journal, to appear, 2009.

[14] N. Khakpour, R. Khosravi, M. Sirjani, and S. Jalili, “A Formal Model for Policy-based Self-Adaptive Systems”, Submitted, 2009.

[15] M. Sirjani, A. Movaghar, A. Shali, and F. S. d. Boer, “Modeling and Verification of Reactive Systems using Rebeca”, Fundamenta Informaticae, vol. 63, pp. 385-410, 2004.

[16] M. Sirjani, F. d. Boer, A. Movaghar, and A. Shali, “Extended Rebeca: A Component-Based Actor Language with Synchronous Message Passing”, in Proceedings of the Fifth International Conference on Application of Concurrency to System Design: IEEE Computer Society, 2005.

[17] M. Mousavi, I. Phillips, M. A. Reniers, and I. Ulidowski, ”Semantics and expressiveness of ordered SOS,” Information and Computation, vol. 207, pp. 85-119, 2009.

[18] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, “A Taxonomy of Compositional Adaptation”, Michigan State University Technical Report MSU-CSE-04-17, 2004.

[19] N. H. Kacem, A. H. Kacem, and K. Drira, “A Formal Model of a Multi-step Coordination Protocol for Self-adaptive Software Using Coloured Petri Nets”, International Journal of Computing and Information Sciences , 2009. To appear.

[20] K. N. Biyani and S. S. Kulkarni, “Concurrency and Complexity in Verifying Dynamic Adaptation: A Case Study”, Michigan State University Technical Report MSU-CSE-05-21, August 2005.

[21] M. Gudemann, F. Ortmeier, and W. Reif, “Safety and Dependability Analysis of Self-Adaptive Systems”, in the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, 2007, pp. 177-184.

[22] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, “A survey of self-management in dynamic software architecture specifications”, in Proceedings of the International Workshop on Self-Manages Systems, Newport Beach, USA, 2004.

[23] B. H. C. Cheng, H. Giese, P. Inverardi, J. Magee, and R. d. Lemos, “Software Engineering for Self-Adaptive Systems: A Research Road Map”, in Software Engineering for Self-Adaptive Systems 2008.

[24] D. Pierre-Charles and L. Thomas, “Towards a Framework for Self-adaptive Component-Based Applications”, in Proceedings of Distributed Applications and Interoperable Systems. LNCS2893, 2003, pp. 1-14.

[25] L. Lymberopoulos, E. Lupu, and M. Sloman, “An Adaptive Policy Based Management Framework for Differentiated Services Networks”, in Proceedings of IEEE Workshop on Policies for Distributed Systems and Networks, Monterey, California, Los Alamitos, California, 2002, pp. 147-158.

[26] R. J. Anthony, “Generic Support for Policy-Based Self-Adaptive Systems”, in Proceedings of the 17th International Conference on Database and Expert Systems Applications, 2006, pp. 108-113.