Formal Analysis of Policy-based self-Adaptive Systems

Narges Khakpour
Tarbiat Modares University
Tehran, Iran
nkhakpour@modares.ac.ir

Ramtin Khosravi
University of Tehran
Tehran, Iran
rkhosravi@ece.ut.ir

Marjan Sirjani
University of Tehran
Tehran, Iran
Reykjavik University,
Reykjavik, Iceland

Saeed Jalili
Tarbiat Modares University
Tehran, Iran
sjalili@modares.ac.ir

ABSTRACT

PobSAM is a flexible actor-based model with formal foundation for model-based development of self-adaptive systems. In PobSAM policies are used to control and adapt the system behavior, and allow us to decouple the adaptation concerns from the application code. In this paper, we use the actor-based language Rebeca to model check PobSAM models. Since policies are used to govern the system behavior, it is required to verify if the governing policies are enforced correctly. To this aim, we present a new generic classification of the policy conflicts and provide temporal patterns expressed in LTL to detect each class of conflicts. Moreover, we propose LTL patterns for checking the correctness of adaptation. An approach based on static analysis of adaptation policies is presented to check the system stability as well.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking, Formal methods; F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Verification, Languages, Design, Theory

1. INTRODUCTION

Increasingly, software systems require adaptations at runtime due to changes in the operational environment, user requirements, upgrades of software modules, and failure or substitution of devices. However, there are several challenges in developing self-adaptive systems which must be addressed appropriately.

Due to the fact that self-adaptive systems are often complex systems with a great degree of autonomy, providing mechanisms to ensure whether the system is operating correctly is a fundamental challenge, where model-based development is considered as an effective approach. *Flexibility* is another main concern to achieve adaptation. Since hard-coded mechanisms make tuning and adapting of long-run systems complicated, we need methods for developing adaptive systems that provide a high degree of flexibility. To this end, [1] proposed 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. PobSAM has a formal foundation that employs an integration of algebraic formalisms and actor-based models [2]. A policy provides a means to manage and dynamically change the system behavior. PobSAM has two types of policies: governing policies which are used to govern the system behavior and adaptation policies which are concerned with adapting the system behavior by changing the governing policies. From the practical point of view, PobSAM is similar to the IBM's PMAC (Policy Management for Autonomic Computing) framework [3] to some extent.

To assure that a self-adaptive system developed by PobSAM behaves properly, we need to provide approaches to verify it during the development process. Particularly, as policies direct the system behavior, it is required to understand and control the overall effect of governing policies on the system behavior. Governing policies often interact with each other that can cause undesirable effects in the system. Hence, providing approaches to detect different kinds of policy conflicts is crucial. Furthermore, the correctness of the adaptation process of the PobSAM models is of great importance, specially, *stability* is an important property of an adaptive system that needs to be checked.

Two main requirements of a model-driven approach are simplicity and ease of use. The encapsulation of state and computation, and the asynchronous communication make actors a natural and simple way to model distributed systems. In this paper, we incorporate an actor-based language named Rebeca [4] to analyze PobSAM models. The Java-like syntax and object-based style of Rebeca make this formalism easy to use for developers. Moreover, Rebeca provides a set of tools for model checking of Rebeca models in addition to transforming Rebeca models into the existing model checkers such as Spin [6] and NuSMV[5]. Due to the fact that PobSAM is an actor-based model, we use Rebeca to verify and analyze PobSAM models.

Contribution. In this paper, we model and analyze PobSAM models using Rebeca. We introduce a new classification of conflicts that may exist among governing policies. Then, we propose LTL patterns to express each type of conflicts. Providing conflict patterns enables us to automate a significant portion of policy analysis process. Moreover, we introduce a number of correctness properties of the adaptation process in the context of PobSAM models. Then, we use static analysis of adaptation policies in addition to model checking technique to verify those properties.

The remainder of this paper is organized as follows. In Section 2, we have a brief review on PobSAM and Rebeca. Section 3 deals with specifying PobSAM models in Rebeca. In Section 4, first we classify and detect the governing policy conflicts. Then, we verify a number of adaptation properties in the context of PobSAM models. We give a summary of related work in Section 5. Finally, Section 6 presents our conclusions and plans for future work.

2. PRELIMINARIES

2.1 PobSAM

PobSAM (Policy-based Self-Adaptive Model) is a flexible actor-based model with formal foundation to develop and prototype self-adaptive systems, introduced in [1]. In this model, policies are used to control and adapt the system behavior. Policies provide a high-level of abstraction and allow us to decouple the adaptation concerns from the application code. Thus, we can change the system behavior as well as adaptation schemas by changing policies.

A PobSAM model consists of a set of self-managed modules (SMMs). An SMM is a collection of actors which can manage their behavior autonomously according to the predefined policies. An SMM structure can be conceptualized as the composition of three layers: the managed actors layer, the view layer and the managers layer. *The managed actors layer* is dedicated to the functional behavior of an SMM and contains computational actors. The *view layer* is composed of a set of views that provides an abstraction of actors state for the managers. A view is the actual state variable, or a function or a predicate applied to state variables of actors. Autonomous managers of the managers layer are responsible for managing actors' behavior according to the predefined policies.

A manager may has different configurations while dynamic adaptation is performed by switching among those configurations. Each configuration consists of two classes of policies: governing policies and adaptation policies. A manager directs the actor behavior by sending asynchronous messages to the actors according to the governing policies. Adaptation policies are used for dynamic adaptation. The 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 policies of C, respectively.

Governing Policies.

Whenever a manager receives an event, it identifies all the policies that are triggered by that event. For each of these policies, if the policy condition evaluates to true, its action is requested to execute by instructing the relevant actors to perform actions through sending asynchronous messages. Governing policies are expressed using a simple algebra as follows, in which P indicates an arbitrary policy set:

$$P \overset{\text{def}}{=} P \cup P \mid P - P \mid P \cap P \mid \{p\} \mid \emptyset$$

A simple action policy $p = [o,\varepsilon,\psi,\alpha]$ consists of priority o, event ε, optional condition ψ and action α. Actions can be composite or simple. A simple action is of the form $r.\ell(v)$ which denotes that the message $\ell(v)$ is sent to actor r. Composite actions are created by composing simple actions using sequential composition $(\alpha;\beta)$, parallel composition $(\alpha \parallel \beta)$, non-deterministic choice $(+)$ and conditional choice $([\omega?\alpha : \beta])$ operators as follows:

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

Adaptation Policies.

Whenever an event which requires adaptation occurs, relevant managers are informed. However, the adaptation cannot be done immediately and when the system reaches a safe state, the manager switches to the new configuration. Therefore, a new mode of operation called adaptation mode is introduced in which a manager runs before switching to the next configuration. This feature allows us to guide the adaptation process safely. There are two kinds of adaptations called *loose adaptation* and *strict adaptation*. Under loose adaptation, the manager enforces old policies while in the strict adaptation all the events are ignored until the system passes the adaptation mode and reaches a safe state. Adaptation policies are defined using an algebraic language as follows:

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

Adaptation policies of a manager are defined as the composition of simple adaptation policies using $\oplus$ operator. Here, composition of two policies means that those policies are potential to be triggered. The simple adaptation policy $\lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta}$ specifies when the triggering condition δ yields and there is no triggered adaptation policy with the higher priority, the manager evolves to the strict or loose adaptation modes based on λ. When the condition of applying adaptation γ becomes true, the adaptation is performed. Also, ϑ denotes the policy priority. D is an arbitrary configuration defined as follows where $[\omega?D : E]$ and $D \Box E$ represent conditional and non-deterministic choices of configurations respectively, and ω and C denote a logical expression and a simple configuration, respectively:

$$D, E \overset{\text{def}}{=} [\omega?D : E] \mid D \Box E \mid C \mid 0$$

2.2 Rebeca

Rebeca [4] is an actor-based language for modeling concurrent asynchronous systems which allows us to model the system as a set of reactive objects called *rebecs* interacting by message passing. The main building blocks of a Rebeca model are rebecs instantiated from the reactive classes. Reactive classes act as a template for states, behavior and interfaces of active objects. Each rebec provides methods called message servers (*msgsrv*) which can be invoked by other rebecs. 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 to which it can send messages. A rebec serves a message by dequeueing it from the queue and executing the corresponding message server. Execution of the message servers is atomic. The operational semantics of a Rebeca model is defined as a labeled transition system in which a rebec's state is defined as its state variables and queue. Also, transitions are made by execution of the message servers. Rebeca is supported by model checker Modere [8] whereby we are able to model check properties expressed with CTL and LTL. Rebeca supports specifying properties in terms of rebec's state variables as well as the messages in the queue of rebec.

3. SPECIFICATION OF POBSAM MODELS

In this section, we elaborate modeling PobSAM models using Rebeca through a case study.

3.1 Illustrative Case Study: Light Control System

In a Light Control System (LCS)[1], the illumination of a room is controlled automatically to prevent electrical energy wasting. To this aim, several sensors have been embedded to provide the physical properties of the environment. Also, the actuators adjust the intensity of different lights placed in the room according to the predefined policies. They also make lights to switch on/off automatically depending on several factors. We use different policies to control lights when there is a fire, when inhabitants are on vacation and in the normal conditions. The following policies are defined to control the lighting of a room in the normal and fire modes:

Default policies (normal conditions)

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.

P5 The system must turn the lights off, when the natural light is sufficient.

P6 If the room is reoccupied after more than T1 minutes since the last person has left the room, the default light scene must be established.

Defined policies in the fire mode

P1 Turn on the emergency light.

P2 Disconnect the power outlets.

P3 When the fire is extinguished, turn off the emergency light.

3.2 Modeling PobSAM Models in Rebeca

In this section, we deal with modeling different layers of the PobSAM models in Rebeca through the LCS case study. Figure 2 gives the Rebeca model of this example.

Managed Actors

The managed layer of a PobSAM model is specified using an extension of Rebeca [7] in which a rebec exposes a number of its state variables publicly. Although all the state variables of the rebecs are defined local, however Rebeca allows us to define public variables. So, we define a public state variable as a global variable of the Rebeca model. At the managed layer, there are two kinds of rebecs. Since self-adaptive systems are often open systems and their behavior depends on the surrounding environment, thus it is required to capture the environment behavior in our modeling of the system. So, we consider some rebecs that model the environment of system, and they are in charge of notifying the autonomous managers of event occurrences through invoking the *triggerevent* message server of manager rebecs. External events are generated non-deterministically. In the LCS example, sensors embedded in different parts of the system sense the environment properties and inform the system about changes. We abstract the behavior of all sensors in the *Environment* rebec. Another type of rebecs are thereof whose behavior is either governed by the manager rebecs or cannot be directed from outside generally. In the LCS case study, the lights are managed actors whose behavior is controlled by policies. We consider the *light* reactive class to model these actors. Since the managers need to know the total intensity of lights, thus we declare the intensity of a light as a global state variable in our model.

Views

As stated above, views provide an abstraction of the managed actors state for the managers. Thus, the views must be observed by the managers. We define a view as a global state variable in Rebeca which is updated after execution of the relevant message servers. As an example, the total intensity of lights is needed to adjust the intensity of different lights. Therefore, we define a global state variable named *totalintensity* to capture the overall intensity of the lights.

Managers

A PobSAM manager is an actor which is modeled as a rebec in Rebeca. Managers are responsible to monitor and handle events by adapting and enforcing suitable policies. Rebeca is a state-based formalism, therefore we represent events by event variables (e.g. event[i]). An event variable is set when its corresponding event is triggered and becomes reset after the event handling. Since execution of a message server is atomic, and dequeuing and serving a message are considered as a single transition, we need to examine the actors' state after execution of a message server and then notify the autonomous manager rebecs of the raised events.

A manager is in charge of enforcing governing policies as well as performing adaptation by applying adaptation policies. As mentioned in Section 2.1, the behavior of a manager depends on the mode in which it is running. A manager can run in different modes including normal, loose adaptation, strict adaptation and enforcement modes. To capture the current mode of a manager, we consider a state variable named *mode* for the manager rebecs. Also, each manager has two state variables to maintain its current configuration and the next configuration to be switched to it, named *config* and *nextconfig* respectively. We consider a message server named *selfadapt* for each autonomous manager rebec which performs dynamic adaptation by changing the current configuration of the manager. When a manager is in the normal mode and an event that requires adaptation is received, it looks for an adaptation policy with the highest priority among the triggered adaptation policies. If the adaptation policy to be enforced requires loose adaptation

$$C_n \overset{\text{def}}{=} \langle \{p_{n1}, p_{n2}, p_{n3}, p_{n4}, p_{n5}, p_{n6}\}, \lfloor C_f \rfloor_{fire,true,S,1} \oplus \lfloor C_v \rfloor_{vacReq,iscnfrmd,L,2} \rangle$$

$$C_f \overset{\text{def}}{=} \langle \{p_{f1}, p_{f2}, p_{f3}\}, \lfloor onVac?\,C_v : C_n \rfloor_{firePutout,true,L,1} \oplus \lfloor C_v \rfloor_{vacReq,isPutout \wedge iscnfrmd,L,2} \rangle$$

Figure 1: Definition of the normal and fire configurations of the *lightcntrlr*

and the conditions of applying adaptation do not hold, the manager switches to loose adaptation mode by setting the *mode* and *nextconfig* state variables. When the conditions of applying the adaptation becomes true, it will switch to the new configuration. If strict adaptation is needed, the manager waits for holding the conditions of applying adaptation using *wait* statement provided by Rebeca. While a manager is in the adaptation mode (both loose and strict adaptations), it does not handle adaptation events anymore.

Example 1. In the LCS case study, *_lightcntrlr* is a manager rebec that controls the lights. A *lightcntrlr* has three configurations C_f , C_v and C_n corresponding to the fire, vacation and normal configurations, respectively. Figure 1 shows the normal and fire configurations of the *lightcntrlr* in which p_{mj} denotes the policy j defined for the configuration C_m. *vacReq*, *fire*, *firePutout* are public state variables while *isPutout* , *iscnfrmd* and *onVac* are *lightcntrlr*'s local variables.

To enforce governing policies, a message server named *enforce* is considered for each manager rebec, which receives and handles events by interpreting the governing policies of the current configuration of manager. While a manager is in the normal or loose adaptation modes, it handles events by enforcing the triggered governing policies based on the priority of policies. A triggered governing policy is enforced if there is no triggered governing policy with a higher priority. Governing policies are expressed as a set of rules in the body of *enforce*. The conditional part of a governing policy is defined as a guarded expression. The policy context is defined in terms of the global state variables associated with the view layer. As mentioned before, a policy action can be composite. To enforce a sequential composite action, we use the *wait* statement provided by Rebeca to handle synchronization between those sequential actions. In case of a sequential action with identical rebecs, there is no need for synchronization since messages of the rebec's queue are processed in FIFO order. Rebeca provides a non-deterministic choice operator which allows us to specify the non-deterministic choice of actions. Rebeca supports concurrent running of actions, thus describing parallel execution of actions is straightforward.

4. VERIFICATION

We can perform different kinds of analysis on PobSAM models. In general, properties to be checked about an adaptive system can be categorized as adaptation properties, functional properties or both together. Correctness properties of the functional layer (managed actors) of PobSAM models are application-specific which can be checked using standard model checking techniques. In this paper, we are only concerned with verification of different properties re-

```
//DEFINITION OF VIEWS AND PUBLIC STATE VARIABLES
byte totalintensity; byte mode;
byte l1_intensity; byte l2_intensity;
reactiveclass lightcntrlr(3) {
  knownrebecs {light _l1; // other known rebecs
                     }
  statevars{ boolean[5] events; byte wap;
             byte config; byte nextconfig; }
  msgsrv initial() {}
  msgsrv enforce() {
   boolean triggeredP1; //definition of local variables
   triggeredP1 = events[0];
   if(config==0){ // the normal configuration
       //enforcing policies of the normal configuration
       if(triggeredP1){
          _l1.switchon(); _l2.switchon(); } }
      if(config==1) // the fire configuration
       { //enforcing governing policies of the fire mode
         }
     if(config==3) // the vacation configuration
       { //enforcing governing policies of the vacation mode
         } }
  msgsrv selfadapt(){
   //lightcntrlr is in the loose adaptation mode
   if(mode == 1 && canswicth)
     { config = nextconfig;
       mode = 0;}
   else
   if(mode == 0) //lightcntrlr is in normal mode
    {if(config == 0) // in the normal configuration
     { if(event[3] == true) // fire event
         { mode = 2; config = 1;
           wait(cond); mode = 0; }
       //definition of other adaptation policies
      } else
     if(config == 1){// in the fire configuration
       if(event[4] == true) // vacation request
        { if(isconfrmd == true)
             config = 3;
           else { nextconfig = 3; mode = 1;
            // set the adaptation policy that causes
            // going to the loose adaptation mode
            wap = 2;
               }}
       //definition of other adaptation policies
      } else
    if(config==3)// the vacation configuration
     { //applying the adaptation policies }
     }
     self.enforce();
   }
  msgsrv triggerevent(byte i)  {
   events[i] = true;
   self.selfadapt(); } }
//DEFINITION OF ACTORS
reactiveclass environment(2) {
 knownrebecs {lightcntrlr _lc1;     }
 statevars{  }
 msgsrv initial() { self.idle();    }
 msgsrv idle()  {
   if(mode != 2){
   //generate events non-deterministically
   _lc1.triggerevent(i);
   self.idle();}
  }
}
reactiveclass light1(2) {
 knownrebecs{ lightcntrlr _lc1;}
 statevars{ byte status; // on or off
  }
 msgsrv initial(){    }
 msgsrv switchon(){
  status = true; }
//definition of other massage servers
 }
main {
 lightcntrlr _lightcntrlr(_l1, _l2):();
 environment _e(_lightcntrlr):();
 light _l1(_lightcntrlr):();
 light _l2(_lightcntrlr):();
}
```

Figure 2: The typical Rebeca model of a PobSAM

lated to the managers layer. Correctness properties of the managers layer are related to the adaptation concerns (i.e., adaptation policies) or behavioral concerns (i.e., governing policies) which we discuss in this section.

4.1 Policy Analysis

Policy conflict detection is a significant analysis that we are concerned with it in this research. To define conflicts formally, we have to begin with some informal understanding of the term "policy conflict". Two policies ρ_i and ρ_j are in conflict if one of the following conditions holds:

- Simultaneous activation of policies ρ_i and ρ_j leads to a state wherein the system cannot choose a policy to enforce.
- Applying ρ_i leads to a situation where makes executing the action of ρ_j impossible. As a typical example, we can consider two policies that turns off a device and starts an application on the same device, respectively.
- Enforcing ρ_i and ρ_j (that not necessarily become enabled simultaneously) results in executing two conflicting actions. We say two actions are in conflict if either execution of an action violates the effect of the other action or satisfying the post-conditions of both actions is infeasible due to the constraints of the system.

4.1.1 Conflict Classification

According to the informal definition of conflicts, we have developed a classification of various conflicts that may exist among policies.

Conflict I. **Action Conflict**
Two arbitrary actions have conflict if they cannot be executed simultaneously. Action conflict arises when two policies with conflicting actions are triggered simultaneously. The conflicting actions shall be defined by the modeler. As an example in the LCS example, increasing and decreasing the light intensity are two conflicting actions.

Conflict II. **Effect Inference Conflict**
In some cases, enforcing a policy overrides the effect of other policies. In this situation, the system no longer works effectively since a policy cancels the effect of the other policy. To illustrate this conflict, consider a scenario that P5 requires a light to be off due to the sufficient natural light while P2 states that whenever someone enters the room, the system must turn on the lights, and as long as the room is occupied, the chosen light scene has to be maintained. Obviously, P2 makes the action of P5 ineffective, while the natural light is adequate.

Conflict III. **Inexecutable Action Conflict**
To execute an action, some prerequisites are required to be preserved by the system. However, the enforcement of a policy may make the action of another policy inexecutable by violating its prerequisites. In the LCS example, suppose that the prerequisite of increasing the light intensity action is that the light must be "on". Thus, in order to enforce a policy that increases the light intensity, no policy that turns off the light must be applied before this policy.

Anomaly I. **Action Redundancy**
Some actions are idempotent, i.e., the number of times that they execute does not influence on the final system state. However, multiple execution of an action can cause to different subsequent states, e.g. turning off a device is an idempotent action while decreasing the light intensity is not. Several execution of a non-idempotent action may lead to the undesired behavior. As an example, assume the *inc* message server that increases the light intensity by variant units. If two policies become triggered simultaneously and both invoke this message server independently, it leads to the growth of the light intensity more than the intended value.

Anomaly II. **Policy Redundancy**
A policy is redundant when the system behavior will not change if that policy is removed. Redundancy is considered as a critical anomaly since redundant policies increase the search time and space requirements of the policy evaluation process.

Anomaly III. **Unenforceable Policy**
Another kind of policy anomaly is unenforceable policy that occurs when the system contains a policy that never becomes activated.

4.1.2 Conflict Detection

We take a model checking approach to detect conflicts. We found LTL as an adequate formalism to describe the desired behavior of the system. Rebeca supports specifying properties in terms of rebec's state variables as well as the messages in the queue of the rebec. Based on the definitions of conflicts, we have provided temporal specification patterns to discover conflicts. Each conflict is detected separately via checking a temporal property. It is worth noting that when a property is violated, the model checker generates counterexamples that are helpful to identify the conflicting policies and to determine how to resolve the conflicts. Inspection of the counterexamples often reveals the policies that lead to conflict.

Henceforth, we suppose that the governing policies ρ_i and ρ_j are defined as $\rho_i = [o_i, \varepsilon_i, \psi_i, \alpha_i]$ and $\rho_j = [o_j, \varepsilon_j, \psi_j, \alpha_j]$ where $\alpha_i = r_i.\ell_i(\upsilon_i)$ and $\alpha_j = r_j.\ell_j(\upsilon_j)$. ρ_i and ρ_j are defined for the configuration C_1 of the manager M. Without loss of generality, we assume policies to have simple actions.

A triggered policy can be enforced if its prior policies are not activated. In Formula 1, $\mathcal{T}_{\rho_i}$ denotes the activation condition of policy ρ_i. Informally, it asserts that ρ_i is triggered if its event and conditions are true and none of its prior policies are triggered at that state. Also, the disjunction with $\bigcirc(\neg\varepsilon_i)$ is because of reseting events after applying the governing policies.

$$\mathcal{T}_{\rho_i} \equiv (M.config = C_1 \wedge \varepsilon_i \wedge \psi_i \wedge \neg \bigvee_{\rho_k \in pre(\rho_i)} \mathcal{T}_{\rho_k}) \wedge \bigcirc(\neg\varepsilon_i) \tag{1}$$

Conflict I. **Action Conflict**
To detect an action conflict, first we should define conflicting actions explicitly. Defined policies ρ_i and ρ_j for a rebec r_k with conflicting actions have action conflict potentially. Therefore, the simultaneous triggering of those policies should be investigated. The LTL formula 2 requires

policies ρ_i and ρ_j not to be triggered simultaneously.

$$\Box\neg(\mathcal{T}_{\rho_i} \wedge \mathcal{T}_{\rho_j}) \tag{2}$$

Conflict II. **Effect Inference Conflict**
Let χ_{ρ_i} represent the post-condition of enforcing ρ_i and ϕ_{ρ_i} stands for the condition of preserving the effect of ρ_i's action. We say ρ_i is in effect inference conflict with ρ_j if and only if executing the action of ρ_j leads to the violation of χ_{ρ_i} while ϕ_{ρ_i} is satisfied. To detect effect inference conflicts, we should state that the consequence of applying a policy holds as long as the preserving condition of that policy is satisfied. This can be expressed by the LTL formula 3.

$$\Box(\mathcal{T}_{\rho_i} \rightarrow \Diamond((\phi_{\rho_i} \rightarrow \chi_{\rho_i})U\neg\phi_{\rho_i})) \tag{3}$$

Example 2. In order to detect whether the post-condition of P5 is preserved while there is sufficient natural light, we impose the following property which is violated. Interpretation of counterexamples reveals that P2 overrides the effect of P5.

$$\Box(e_{sl} \rightarrow \Diamond(\mathit{sufflight} \rightarrow (\neg status_{l1} \wedge \neg status_{l2}))\,U\neg \mathit{sufflight})$$

Conflict III. **Inexecutable Action Conflict**
Assume ϑ_i be the prerequisite of executing α_i. The policy ρ_i has inexecutable action conflict in respect to ρ_j provided that the consequence of enforcing ρ_j contradicts ϑ_i and ρ_j has been enforced prior to ρ_i. Thus, for the action α_i to run successfully, ϑ_i must hold in the state in which it is dequeued (formula 4). The policy that causes conflict can be identified by analyzing the counterexamples.

$$\begin{aligned}\Box(\mathcal{T}_{\rho_i} \rightarrow \Box((r_i.queue[0] = \alpha_i \rightarrow \vartheta_i)\wedge \\ \bigcirc(r_i.queue[0] \neq \alpha_i)))\end{aligned} \tag{4}$$

Anomaly I. **Action Redundancy**
Two policies ρ_i and ρ_j defined for the rebec r_k with a common non-idempotent action have action redundancy anomaly provided that they are triggered simultaneously. This can be expressed by the LTL formula $\Box\neg(\mathcal{T}_{\rho_i} \wedge \mathcal{T}_{\rho_j})$.

Anomaly II. **Policy Redundancy** Policy ρ_j is redundant respect to the policy ρ_i if the LTL formula 2 holds. We detect this anomaly by the pairwise comparison of the policies with same actions.

$$\Box(\mathcal{T}_{\rho_i} \rightarrow \mathcal{T}_{\rho_j}) \tag{5}$$

Example 3. In the LCS example, policy P6 is redundant respect to P2 since the property $\Box(\mathcal{T}_{\rho_6} \rightarrow \mathcal{T}_{\rho_2})$ is satisfied by the model.

Anomaly III. **Unenforceable Policy** If policy ρ_i never becomes activated, the formula $\neg\Diamond\mathcal{T}_{\rho_i}$ is held. In our case study, all the policies are enforceable.

4.2 Adaptation Analysis

We define the triggering condition of the C_i's simple adaptation policy $a_p = \lfloor D_p \rfloor_{\delta_p,\gamma_p,\lambda_p,\vartheta_p}$ as formula 6 where C_i is a simple configuration of the manager M. Furthermore, let λ denote the current mode of a manager that can take the values of am (adaptation mode), lm (loose adaptation mode) or nm (normal mode).

$$\begin{aligned}\Gamma_{a_p} \equiv\ (M.\mathit{config} = C_i)\ \wedge \delta_p \wedge \neg \bigvee_{a_k \in pre(a_p)} \Gamma_{a_k} \wedge \\ ((M.\lambda = nm\ \wedge \bigcirc M.\lambda = lm\ ) \vee \bigcirc\ M.\mathit{config} \neq C_i)\end{aligned} \tag{6}$$

Stability checking.
We must be ensured about the correctness of the system behavior before, during, and after adaptation. Adaptation can cause instability of system state, i.e. the adaptation by a manager may lead to another adaptation, which in turn leads to another, and so on. If this cycle continues without reaching a stable state, we say the system is in an unstable state. Thus, stability of adaptation is an important property that needs to be checked. In PobSAM, stability is defined as follows:
Definition 1. **(Stability)**Manager M is unstable if it switches between the adaptation and normal modes infinitely without evolving to the enforcement mode, i.e., there is a path such as $\sigma = \sigma_i\,\sigma_{i+1}...$ where $(\lambda_j = am \vee \lambda_j = nm)$, $\forall j \geq i$ and λ_j denotes the M's mode in state j.
Checking stability of a manager such as M can be expressed by the LTL formula $\neg\Diamond\Box(M.\lambda = am \vee M.\lambda = nm)$. Due to the state explosion problem of model checking, we propose an approach to verify the stability of a manager by reasoning on the adaptation policies of the manager. This approach allows us to only detect a specific class of instabilities. However, it is helpful to be used beside the model checking approach.
Definition 2. Let $\Psi(D)$ denote the potential successor configurations of D under the condition ω computed as follows:

$$\Psi(\langle\omega\rangle D) = \begin{cases} \Psi(\langle\omega\rangle D_1) \cup \Psi(\langle\omega\rangle D_2) & \text{if } D = D_1 \Box D_2 \\ \Psi(\langle\omega \wedge \omega'\rangle D_1) \cup \Psi(\langle\omega \wedge \neg\omega'\rangle D_2) & \\ & \text{if } D = \langle\omega' ? D_1 : D_2\rangle \\ D & \text{if D is a simple configuration} \\ \emptyset & \text{if } D = 0 \text{ or } D = \langle false\rangle D_1 \end{cases}$$

Definition 3. Let $\mathcal{C} = \{C_1, ..., C_m\}$ denote the set of M's configurations where $C_k = \langle P_k, A_k\rangle, 1 \leq k \leq m$. We say C_j is an immediate successor of C_i denoted by $C_i \leadsto^{pq}_{w} C_j$, if and only if C_i and C_j have respectively the adaptation policies such as $a_p = \lfloor D_p \rfloor_{\delta_p,\gamma_p,\lambda_p,\vartheta_p}$ and $a_q = \lfloor D_q \rfloor_{\delta_q,\gamma_q,\lambda_q,\vartheta_q}$ where $\gamma_p \rightarrow \Gamma_q$, $\langle w\rangle C_j \in \Psi(\langle\delta_p\rangle D_p)$, $a_p \in A_i$ and $a_q \in A_j$.

Definition 4. (Immediate adaptation graph) The *immediate adaptation graph* of a manager is a labeled directed graph $G = \langle V, E\rangle$ whose vertices indicate the manager's simple configurations. The G's edges indicate immediate successor relation between the configurations and $(C_i, C_j, \{pq, w\}) \in E$ if and only if $C_i \leadsto^{pq}_{w} C_j$.
Based on the above definitions, it can be easily seen that if the M's immediate adaptation graph has a cycle such as $C_1 \rightarrow ... \rightarrow C_n$ where $(C_i, C_{i+1}, \{p_i q_i, w_i\}) \in E,\ q_i = p_{i+1}$, $1 \leq i < n,\ q_n = p_1$ and $w_i = w_n = true$, then M is unstable.

Deadlock-freedom.

Deadlock-freedom is a generic property of a system that must be checked. In the context of the adaptation mechanism of PobSAM, we want to verify that "when a manager is waiting for the condition of applying adaptation in the adaptation mode, it would eventually switch to the next configuration". This can be expressed by the LTL formula $\Box(\Gamma_{a_p} \rightarrow \Diamond M.\lambda = nm)$ for the adaptation policy a_p. If this specification is satisfied for all the M's adaptation policies, we conclude that M is deadlock-free during the adaptation process.

Unreachable Configuration.

Moreover, we can check whether all the configurations of a manager can be reached finally or not, i.e. a manager will run all its configurations. This can be expressed by the LTL formula $\Diamond(M.config = C_i)$ for all the configurations of M.

Adaptation policy analysis.

Among the policy conflicts and anomalies introduced for governing policies, only the unenforceable policy anomaly makes sense in the context of adaptation policies. The LTL formula $\Diamond\Gamma_{a_p}$ is satisfied if the simple adaptation policy a_p is not unenforceable.

Checking safety properties during adaptation.

In many applications, particularly in fault-tolerant systems, some safety properties must be preserved during adaptation. Let ϕ denote a property that must be held during the adaptation of M from the configurations C_i to C_j. This property is expressed by the following LTL formula:

$$\Box(((M.\lambda = am \wedge M.config = C_i \wedge M.nextconfig = C_j) \rightarrow \phi) \\ U(M.\lambda = nm))$$

5. RELATED WORK

Adaptation techniques are classified into two broad categories: structural and behavioral. Given a system, while structural adaptation aims to adapt the system behavior by changing system's architecture at run-time, behavioral adaptation focuses on modifying the functionalities of computational entities. Structural adaptation has been given strong attention in the research community (see [9]), but fewer approaches tackle the behavioral adaptation as in PobSAM is considered. We restrict ourselves to present related work in the area of behavioral adaptation. Regarding verification of adaptive systems, a model-driven approach was proposed for developing adaptive systems in [10]. 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 [11], 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 [12]. Then, they use a model checking approach to verify the system. Kulkarni et al. [13] proposed an approach based on the concept of proof lattice for verifying correctness of dynamic adaptation of components. [14] proposed a framework for model-based development of adaptive embedded systems using labeled transition systems. In this work, model analysis is performed using theorem proving, model checking and specialized verification methods. In [1] PobSAM is compared with the existing approaches.

Another related area of research is policy analysis. While the policy research community has reported a significant amount of work in the area of policy analysis, researchers have given strong attention on analyzing policies alone without modeling the system behavior (e.g. see [15, 16, 17, 18]). This is because these approaches use policies to manage network systems, while modeling network systems is too complicated. There has been very little research done to tackle the problem of policy analysis capturing the system model, as we have done in our research. An outstanding point of our work is introducing a new classification of conflicts. Most work done on policy analysis are restricted to detection of modality conflicts(e.g. [18, 19, 20]). Some researchers dealt with detection of action conflicts, particularly online conflict detection approaches which is covered by our classification as well.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we dealt with formal analysis of PobSAM models using model checking and static analysis techniques. To this aim, we used an actor-based language called Rebeca to model the system. Then, we presented a new generic classification of governing policy conflicts. To detect the conflicts, we provided temporal patterns expressed in LTL which enable us to automate detection of conflicts. Moreover, we identified generic properties of a PobSAM model in term of the adaptation concerns and proposed LTL patterns to verify those properties. Also, we proposed a technique to detect a specific kind of system instability which works based on the formal specification of adaptation policies.

Compositional verification of the managers and actors layers is an ongoing work. Since a manager can run in different configurations with various goals, we attempt to verify a property by dividing it into a set of properties that must be preserved by each configuration.

Acknowledgments. The authors would like to gratefully thank Ehsan Khamespanah for his helpful suggestions and discussions.

7. REFERENCES

[1] N. Khakpour, S. Jalili, C. Talcott, M. Sirjani, and M. Mousavi, "PobSAM: Policy-based Managing of Actors in Self-Adaptive Systems," In Proceedings of the 6th International Workshop on Formal Aspects of Component Software, Eindhoven, The Netherland, 2009, To appear.

[2] G.Agha, "Actors: A Model of Concurrent Computation in Distributed Systems," MIT Press, Cambridge, MA, USA, 1990.

[3] "Autonomic computing", IBM Systems Journal, vol. 42, 2003.

[4] 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.

[5] NuSMV User Manual, Availabe through http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html.

[6] Spin User Manual, Available through http://netlib.bell-labs.com/netlib/spin/

whatisspin.html.

[7] 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.

[8] M. M. Jaghoori, A. Movaghar, and M. Sirjani, "Modere: The Model-Checking Engine of Rebeca," in ACM Symposium on Applied Computing - Software Verificatin Track, 2006, pp. 1810-1815.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] D. Agrawal, J. Giles, K.-w. Lee, and J. Lobo, "Policy Ratification," in Proceedings of 6th IEEE International Workshop on Policies for Distributed Systems and Networks, Stockholm, Sweden, 2005, pp. 223-232.

[16] J. Baliosian and J. Serrat, "Finite State Transducers for Policy Evaluation and Conflict Resolution," in Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks, New York, USA, 2004, pp. 250- 259.

[17] J. Chomicki, J. Lobo, and S. Naqvi, "Conflict Resolution Using Logic Programming," IEEE Transactions on Knowledge and Data Engineering, vol. 15, pp. 244-249, 2003.

[18] E. Lupu and M. Sloman, "Conflict analysis for management policies," in Proceedings of the fifth IFIP/IEEE international symposium on Integrated network management San Diego, California, 1997.

[19] N. Dunlop, J. Indulska, and K. Raymond, "Dynamic Conflict Detection in Policy-Based Management Systems," in Proceedings of the Sixth International Enterprise Distributed Object Computing Conference (EDOC'02): IEEE Computer Society, 2002.

[20] A. Uszok, J. M. Bradshaw, R. Jeffers, N. Suri, P. J. Hayes, M. R. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott, "KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement," in Proceedings of 4th IEEE International Workshop on Policies for Distributed Systems and Networks, Lake Como, Italy, 2003, pp. 93- 96.