Formal Analysis of Smart Home Policies using Compositional Verification.

Conference Paper · January 2009

DOI: 10.3233/978-1-60750-014-8-220 · Source: DBLP

Narges Khakpour
Linnaeus University
14 PUBLICATIONS 89 CITATIONS
SEE PROFILE

Marjan Sirjani
Malardalen University
122 PUBLICATIONS 1,083 CITATIONS
SEE PROFILE

Saeed Jalili
Tarbiat Modares University
87 PUBLICATIONS 375 CITATIONS
SEE PROFILE

Feature Interactions in Software and Communication Systems X
M. Nakamura and S. Reiff-Marganiec (Eds.)
IOS Press, 2009
© 2009 The authors and IOS Press. All rights reserved.
doi:10.3233/978-1-60750-014-8-220

Formal Analysis of Smart Home Policies using Compositional Verification

Narges Khakpour[a,1], Marjan Sirjani[b], Saeed Jalili[a]
[a]*Safety Critical Systems Lab., Department of Electrical and Computer Engineering, Tarbiat Modares University*
[b]*Formal Methods Lab., Department of Electrical and Computer Engineering, University of Tehran*

Abstract Smart spaces contain a large number of computing devices communicating with each other to perform various high-order tasks. They are governed by predefined policies that users can put according to their preferences. In this paper, we investigate the policy interaction problem beyond the smart home domain. We use a formal method for detecting dynamic conflicts between policies. First, we give an abstract model of the system described with an actor-based language. Then, we identify different kinds of conflicts that may exist among policies in smart home domain. To reduce the complexity of model checking, we use compositional verification as well as data abstraction techniques.

Keywords. Smart Spaces, Smart Home, Policy Analysis, Model Checking, Actor-based Models, Compositional Verification

Introduction

Smart Space needs highly adaptive management systems that can adjust their behavior at runtime based on user preferences. Policy is a key mechanism in providing adaptation of the system behavior. Policy based management offers a means for administrators, end-users and application developers to manage and dynamically change the behavior of computing systems[1]. A policy is a rule describing under which condition a specified action must (can or cannot) be performed.

However, policies usually interact with each other that can cause to undesirable effects in the system. Especially, policy conflict has been identified as a potential threat to the realization of effective and usable smart space systems[2]. Understanding and controlling the overall effect of policies is particularly important in smart home domain, where multiple residents may write policies for the same set of resources without coordination. Furthermore, the nature of managing large sets of devices in a distributed area emerges conflict among requirements, which subsequently leads to conflicts into policy specification[3]. Furthermore, since there are a large number of devices at home capable of performing interrelated tasks directed by different policies, thus they need to be configured appropriately to avoid undesirable behavior. Therefore, the inhabitant needs to be conscious of the effects of such interactions.

[1] Corresponding author.

Although the research in the field of policy analysis has gained increasing attention in the past few years, but most researchers investigated policies on its own and abstracted from the systems enforcing these policies. Particularly, the complexity of smart home systems makes capturing the system behavior in verification process inevitable. To tackle this problem, we attempted to employ a model checking approach for detecting policy conflicts in smart home domain. Our solution is based on our research into verification of policy-based systems[4]. Model checking is a well-known technique to verify whether a system satisfies given properties. We model the system using Extended Rebeca[5]. Rebeca[6] is an actor-based language for modeling concurrent and distributed systems which provides a formal foundation. Due to the fact that policy-based systems are usually large-scale and distributed, the model checking complexity of these systems is extremely high. Compositionality is a desirable facility to reduce the complexity of model checking by decomposing a large system into more manageable pieces and proving the correctness of the whole system from that of its immediate components. One of the major advantages of this language in comparison to the other available analogous languages is its capability of compositional verification provided in [5]. Desired properties to discover conflicts are expressed in LTL(Linear Temporal Logic)[7, 8] patterns [4].

This paper has the following major contributions:

1. It studies dynamic policy analysis in smart home domain.
2. It introduces a new approach to verify a policy-based system in smart home domain compositionally, which is a promising step toward making verification of policy-based systems practical.

The remainder of this paper is structured as follows. In Section 1, we have a brief review on Extended Rebeca, Linear Temporal Logic and Smart Homes. Section 2 introduces our modeling approach using a simple example. In Section 3 we deal with verification of our model. We give a summary of related works in Section 4. Finally, Section 5 presents our conclusion.

1. Preliminaries

1.1. Extended Rebeca

Rebeca is an actor-based language for modeling concurrent systems which allows us to model the system as a set of reactive objects interacting by asynchronous 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[6]. Each rebec provides methods called message servers which can be invoked by other rebecs. Each rebec has an unbounded buffer for coming messages named queue. Furthermore, the rebecs' statevars are in responsible of capturing the rebec state. The Knownrebecs of a rebec denotes the rebecs which it can send messages to. A rebec serves a message by dequeueing it from the queue and executing the corresponding message server. Rebeca is supported by model checker Modere whereby we are able to model check properties expressed with CTL and LTL.

One of the most important problems in model checking is the state-explosion problem. Compositional verification has been introduced to tackle this problem which decomposes a system into a set of components. In compositional verification the goal is to check properties of the components of a system and deduce global properties from those local properties. Extended Rebeca is an extension of Rebeca which supports compositional verification. To verify a model in Extended Rebeca compositionally, the model is decomposed into a set of components and the environment. Each component includes a collection of rebecs while others make its environment. The behavior of environment is modeled by sending messages to the components. Instead of putting the external messages in the queue of rebecs, it is supposed that they are present in all states and they are processed fairly interleaved with the internal messages. This makes the model checking activity more efficient. In a Rebeca component model, we call environment rebecs as *external* and all other rebecs as *internal* rebecs.

1.2. Linear Temporal Logic (LTL)

Due to the lack of space, we only introduce LTL in brief. LTL is a temporal logic that extends propositional logic by introducing four basic temporal operators: the existential operator F, the global operator G, the until operator , and the next operator X. In this paper, we will restrict ourselves to an informal definition of the temporal operators. An LTL formula is evaluated over a sequence of states. Given formulas ψ and φ, $G\psi$ (read as always ψ) states that ψ is true in all states. $F\psi$ (read as finally ψ) means ψ is eventually true. $\psi \mathcal{U} \varphi$ (read as ψ until φ) denotes ψ is true until φ becomes true. $X\psi$ (read as next ψ) asserts ψ is true in the next state.

1.3. Smart Homes

At homes, you will find a wide range of electrical and electronic devices such as lights, thermostats, electric blinds, fire and smoke detection sensors, white goods such as washing machines, as well as entertainment equipment. Smart Home connects those devices and enables inhabitants to monitor and control them from a common UI. The home network also allows the devices to coordinate their behavior in order to fulfill complex tasks without human intervention[9].

In a home automation system, sensors are devices that provide smart home with physical properties of the environment by sensing the environment. In addition, actuators are physical devices that can change the state of the world respondent to sensed data by sensors. Every actuator in the smart house has a certain intentional effect on a domain, which a sensor that senses that particular domain can observe[10]. 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. Policies allow the system to act autonomously in case of certain event. Therefore, residents must be able to control all devices through defining different policies according to their needs and preferences.

2. Modeling Smart Home System

2.1. Running Example

We present a small simplified example to illustrate our approach. Smart homes can have different features; we will focus on only the features that are likely to have most interaction. We take into account the following features:

- **Light Management** This feature allows lights to switch on and off automatically depending on several factors. In addition, the intensity of the different lights placed in a room can be adjusted regarded to the predefined policies.

- **Doors/Windows Management** Inhabitants have to able to manage windows and doors automatically. In addition, if windows have blinds, these should be rolled up and down automatically too. By means of this feature, the state of doors and windows is monitored.

- **Heating Management** Inhabitants must be able to adjust the heating of the house to their preferred value. The heating control will adjust itself automatically in order to save energy.

- **Presence Simulation** In order to avoid burglary, when inhabitants leave the house for a long time, the house occupation must be simulated by either switching the lights on/off or rolling the blinds up/down automatically according to a predefined schedule.

- **Emergency/Security Management** In critical situations, e.g. fire, burglary, water intrusion etc, system should query the resident if it suspects a problem, and if so, it does the revisory actions, prevents from spreading the incident, issues a call for outside help when necessary, etc.

For the sake of brevity, here we only point to four policies defined for emergency management module:

P 1. On detection of fire, system must run the following scenario:
- Shut all windows and doors without locking
- Trigger the alarm
- Call the fire station

P 2. Whenever the Gas/Smoke sensor is triggered, system should
- Unlocked all the windows
- Close the gas valve

P 3. Once a PIR sensor indicates movement in area X1 while inhabitants are on vacation, system should
- At first, call the police
- Then lock the doors and windows and trigger the alarm

P 4. On opening window/door if the alarm is active, system must carry out the following actions:
- Lock all doors and windows provisionally
- Trigger the alarm

2.2. Policy Definition for Smart Home Systems

The policy language for smart home was built on our work for verification of policy-based systems[4]. We categorize policies as obligation and authorization policies; however, here we restrict ourselves only to the obligation policies. Obligation policies build up the core part of a policy-based system and define which *actions* a *subject* must perform on an *object* when an *event* occurs, given some *conditions* hold. An obligation policy consists of an *event*, an optional *condition*, and an *action*. An *event* is a named event that triggers execution of a policy and can be either an external or a system event. A *condition* is a binary expression evaluated to determine the validity of a policy. *Conditions* can make the validity of policy dependent on the system-dependent conditions like the time or its current state. An *action* describes a task to be executed and may need additional parameters. *Actions* can be composite or simple. *Composite actions* are created by composing simple action using sequential and parallel operators, with sequential operator having higher precedence than parallel. *Composite actions* are useful in situations where triggering an *event* causes the execution of a chain of actions which their order of execution is too important. As an example consider a policy that states "whenever a burglary is detected, the system must call for help and locks all the windows/doors firstly, then it must trigger the alarm". The actions "calling for help" and "closing the doors and windows" must be performed in parallel before execution of "triggering the alarm" action.

2.3. The Model of a Smart Home

In this section, first we present the Rebeca model of smart home, and then we deal with compositional modeling of smart home using Extended Rebeca.

2.3.1. The Rebeca Model of Smart Home

In the proposed approach, at first we provide the core model of the system to be modeled. The core model denotes the internal computation of objects without considering policies that govern the behavior of system. Then, we merge policies with the core model using a simple procedure which can be done automatically[4].

We model a smart home as a collection of interacting objects including smart devices, sensors and actuators. Thus, we identify three kinds of rebecs that may exist in a smart home model. Sensors embedded in different locations at home detect events. While at first sight, it seems to consider each sensor as a rebec, however it leads to the increase in the complexity of the model. Thus for the sake of simplicity, we make an abstraction and consider a rebec named *environment* to abstract the behavior of all sensors receiving external events. This rebec is in charge of notifying *smart devices* of the occurred events. In a smart home application, smart devices are in charge of enforcing policies while actuators are controlled by smart devices through policies. Hence, *smart devices* handle received events from the *environment* and other rebecs by evaluating and enforcing suitable policies. *Actuators* are thereof whose behavior is governed by smart devices and receive messages from smart devices to act properly. Hereafter, we may use smart devices and managers as well as actuators and managed rebecs interchangeably.

```
reactiveclass smart device1( ) {
  knownrebecs { actuator1 _a1; }
  statevars(boolean[5] events;
      //definition of state variables
   }
  msgsrv initial()  {self.monitor(); }
  msgsrv monitor() {
    boolean[10] policy_activity;
    //evaluating and enforcing obligation policies
    policy_activity[0] = events[0] && context0;
    policy_activity[1] = events[2] && context1;
    // setting the activity state of other policies
    if(policy_activity[0] &&
       !(policy_activity[1] || policy_activity[5])){
    // policy1 and policy5 have precedence over policy0
        _a1.msgsrv1(0);
         // enforcing policy
       }
     // enforcing other policies
    }
  //definition of other message servers
  msgsrv trigger_event(byte i)  {
    events[i] = true;
    self.monitor();  }
 }
reactiveclass environment( ) {
  knownrebecs { smart_device1 _sd;}
  statevars(  }
  msgsrv initial() { self.idle();     }
  msgsrv idle()  {
   //generate events
   _sd.trigger_event(i);
   self.idle();}
 }
reactiveclass actuator1( ) {
  knownrebecs{smart_device1 _sd;}
  statevars{
     //definition of state variables
   }
  msgsrv initial(){     }
  msgsrv msgsrv1(int arg1){
    //body of msgsrv1
   }
  //definition of other message servers
 }
main {
  smart_device1 _sd1(_a1):();
  environment _e(_sd1):();
  actuator1 _a1(_sd1):();
  //definition of other rebecs
  }
```

Figure 1. The typical Rebeca model of a Smart Home

An abstract Rebeca model of a smart home application has been illustrated in Figure 1. In this model, we represent events by *event* variables (e.g. *event[i]*). An *event* variable is set when its corresponding event triggers and becomes reset after handling.

As shown, *environment* generates external events non-deterministically and informs *smart devices* about the changes by invoking *trigger_event* message server. In this model, a message server named *monitor* has been considered per each *smart device* to monitor the system state, which handles events by enforcing the predefined policies. Policies are expressed as a set of rules in the body of monitor where their conditional part is defined as a guarded expression. The policy context is defined based on the rebecs' state variables and the global variables. Whenever an event is received by a *smart device*, it identifies all the obligation policies that are triggered by that event. For each of these policies, the policy condition is evaluated, if one exists. If the condition is evaluated to true, the action part of the triggered policy is appealed to execute by instructing relevant actuator rebecs to perform actions through sending asynchronous messages. In certain situations, multiple non-conflicting policies may need to be enforced and the enforcement order of those policies affects the system state. We assume there is a total order defined over obligations that represents the order of enforcing policies in our model.

The Rebeca model of our running example contains 6 fundamental reactive classes corresponding to actuators includes door, window, light, thermostat, alarm and blind shutter. In cases, identical object types may have different functionality, e.g. interior doors certainly function differently from outer doors. Subsequently, the modeler is supposed to model them as different reactive classes. In addition to actuators, the model contains *smart devices* including *LightCntrlr*, *TempCntrlr*, *EmergencyManager*, *DWCntrlr* and *PSimulator*.

2.3.2. The Compositional Model of Smart Home

The huge state space of the Rebeca model makes it inadequate to verify formally by model checking. Our formal framework allows us to perform compositional verification to address the state space explosion problem. In order to verify our model compositionally, first we should decompose the model into a set of components. We should choose the highly coupled rebecs together as a component. In smart home model, a smart device usually controls multiple actuators. Therefore, we consider a smart device and its governed actuators as a component and the other rebecs as its environment. As mentioned above, an *event* can be either an external event or a system event. Relating to a component, the system events occurring in other components are considered as external to that component. Therefore, we model the environment as an external class sending messages to the components.

Figure 2 shows the simplified model of the *EmergencyManagment* component and its environment in our example. It is worth noting that an actuator rebec may be used in model checking of different components, since it can be controlled by multiple smart devices. As an example, the doors and windows are controlled by several smart devices including *LightCntrlr*, *HeatCntrlr*, *EmergencyManager* and *DWCntrlr*. Therefore, a *door* rebec may receive messages from those smart device rebecs. As a result, when it is included in the component model of a smart device, it must receive the messages from other smart devices. As an example, *alarm* receives messages "trigger_alarm", "activate_alarm" and "deactivate_alarm" messages from environment shown in Figure 2.

```
externalclass XEnvironment of Environment{
envars{
     }
initial(){
        }
sends{
trigger_event(?(0 .. 3));
trigger_alarm(alarm_ID);
activate_alarm(alarm_ID);
deactivate_alarm(alarm_ID);
//sending other messages to EmergencyManagement component
  }
}
main {
  emergencyManager _em( _a, _w1, _d, _v):();
  window _w1(_em):(1);
  door _d1(_em):();
  alarm _a():();
  valve _v1(_em):();

  //definition of other instances
  XEnvironment xEnv(_em,_a);
  Components:
  {_em,_w1,_d1,_a,_v1};
  {xEnv};
 }
```

Figure 2. The simplified model of EmergencyManagment Component

Data Abstraction Another technique to reduce the state space is employing data abstraction methods. To deal with unbounded data domains or large constants, we apply the concept of data domain abstraction[11]. Data values from a large or infinite domain are thereby mapped to a smaller finite domain using a homomorphic abstraction function, provided that the domain abstraction is compatible with the operations of the system [12]. As an example, consider temperature attribute which ranges over integer domain which can be reduced to the abstract domain {freezing, cold, temperate, warm, hot}. Moreover, to make the model checking procedure more efficient, we take advantage of the symmetric behavior of rebecs instantiated from a reactive class and consider one rebec per those reactive classes. The modeler must be careful about using this technique to reduce verification complexity.

3. Verification

3.1. Conflict Detection

Given the model of system, we are able to do vast kinds of analysis, but in this paper we limit our analysis to only detection of the undesirable behavior of the system rooted in conflicting policies. First, we are supposed to give an informal definition of the term "policy conflict". Two policies ρ_i and ρ_j are in conflict whenever one of the following conditions holds[4]:

- Simultaneous activation of policies ρ_i and ρ_j leads to a state wherein the system cannot decide on the policy to be enforced.
- Applying ρ_i leads to a situation where makes executing the action of ρ_j impossible. As a typical example we can point to two policies that turn off a device and start an application on the same device respectively.
- Enforcing ρ_i and ρ_j (that not necessarily become enabled simultaneously) result in executing two conflicting actions. We say two actions are in conflict if either execution of an action violates the effect of the other or satisfying the post-condition of both actions is unfeasible due to the constraints of the system. As an example in the smart home application, assume two policies: "CD player can play music for three hours starting 7:00 pm" and "Shuts down all audio/video devices after 9:00 pm"[13]. Obviously, the second policy will make the action of the first policy ineffective before it can finish.

Policies provide a means of specifying the desired behavior of the system at a high level. However, a policy may not be enforced properly due to conflict with other policies, i.e. a triggered policy can be enforced correctly provided that it has no conflict with other policies. We established a taxonomy of various undesirable behaviors and formalized definition of those conflicts based on the formal semantics of a policy-based system in[4]. Table 1 gives the taxonomy of conflicts and their informal definition that we use in smart home domain.

Table 1. Informal definition of policy conflicts

Conflict/Anomaly Type	**Informal Definition**
Action Conflict	Two policies with conflicting actions are triggered simultaneously
Effect Inference Conflict	Enforcement of a policy overrides the effect of another policy
Inexecutable Action Conflict	Enforcement of a policy make performing the action of another policy inexecutable by violating its prerequisites
Goal Conflict	User goals cannot be satisfied simultaneously due to the existence of conflicting policies
System Invariant Violation Conflict	Enforcing a policy will violate the system invariants or user goals
Action Redundancy Anomaly	Two policies with non-idempotent action are triggered simultaneously
Policy Redundancy Anomaly	A policy subsumes another one such that the second policy never can enforce in the system while it has been triggered
Unenforceable Policy Anomaly	A policy will never become activated because its triggering conditions never become true

We believe that an adequate way of detecting conflicts is by a collection of temporal formulas that specifies the desired behavior of the system and are defined over the set of predicates that denotes states or raised events of the system. We found LTL as an adequate formalism to describe the desired behavior of the system. Based on the formal definition of conflicts, we provide temporal specification patterns to discover conflicts. In the proposed approach, we investigate the existence of possible conflicts by pairwise comparison of policies. Detection of possible conflicts between each pair policies is performed separately by verifying a temporal property.

Definition 1. The obligation policy ρ_o is defined as the ordered quintuplet $\rho_o =< e, cond, r_s, r_t, \alpha >$ where its entries denote the event, condition, subject, target and the action of policy respectively.

In the formula (1), $\mathcal{T}_{\rho_i}$ denotes the activation condition of policy ρ_i in a state. Informally it asserts that the ρ_i's event and condition become true and none of its prior policies get triggered at that state. Also, its corresponding event will be served in the next state after enforcing policy.

$$\mathcal{T}_{\rho_i} \leftrightarrow (e_i \wedge cond_i \wedge (\neg \bigvee_{\rho_k \in pre(\rho_i)} \mathcal{T}_{\rho_k})) \wedge X(\neg e_i) \qquad (1)$$

Table 2 gives the LTL patterns to detect policy conflicts defined in Table 1 where $\mathcal{T}_{\rho_j}$ denotes the activation condition of policy ρ_j and $r.queue_v$ denotes the message placed in the index v of r's queue. In addition, ψ_{ρ_i} and φ_{ρ_i} represent the post-condition of enforcing ρ_i and the required condition of preserving the effect of ρ_i's action respectively. ω is an invariant that should be hold throughout the system execution applied by high-level policies or system constraints and ϑ_i denotes the prerequisite of executing α_i.

Table 2. LTL patterns to detect policy conflicts

Conflict/Anomaly Type	**Formal Definition**
Action Conflict	$\neg F(\mathcal{T}_{\rho_i} \wedge \mathcal{T}_{\rho_j})$
Effect Inference Conflict	$G(\mathcal{T}_{\rho_i} \rightarrow F((\varphi_{\rho_i} \rightarrow \psi_{\rho_i}) U \neg\varphi_{\rho_i})$
Inexecutable Action Conflict	$G(\mathcal{T}_{\rho_i} \rightarrow G((r_{t,i}.queue[0] = \alpha_i \rightarrow \vartheta_i) \wedge X(r_{t,i}.queue[0] \neq \alpha_i)))$
System Invariant Violation Conflict	$G\omega$
Action Redundancy Anomaly	$\neg F(\mathcal{T}_{\rho_i} \wedge \mathcal{T}_{\rho_j})$
Policy Redundancy Anomaly	$G(\mathcal{T}_{\rho_i} \rightarrow \mathcal{T}_{\rho_j})$
Unenforceable Policy Anomaly	$\neg F\mathcal{T}_{\rho_i}$

The main difficulty with compositional verification is that local properties are often not preserved at the global level[14]. However, as we detect the conflicts which may exist among the policies of a smart device, therefore the local properties of each component will make the global properties that we must check on the system model.

3.2. Verification Results

As the number of properties to be checked is high, we do not try to produce a complete list of conflicts which may exist between policies. We restrict ourselves to give a number of conflicts detected in verification of *EmergencyManagment* component. For verification, we used a PC equipped with an Intel Core 2 2.6 GHz and 3 GByte of memory with Windows XP.

Conflict I. Action Conflict
To detect action conflicts, first we should identify the conflicting actions manually. Table 3 shows conflicting actions in *EmergencyManagment* component partially.

As shown in Table 3, the lock and unlock door/window actions are conflicting. Therefore, P1 and P3 are potentially in action conflict. Checking Property 1 confirmed that those policies are in action conflict, i.e. they are triggered simultaneously.

Property 1. $\neg F(\mathcal{T}_{\rho_1} \wedge \mathcal{T}_{\rho_3})$

Conflict II. Effect Inference Conflict
In order to detect Effect Inference conflicts, we should state that the consequence of applying a policy must hold explicitly while the preserving conditions being satisfied. As an example, while fire has not been put out, the windows/doors must be kept closed and unlocked. Also, the alarm should not become deactivated. Checking Property 1 shows that the effect of this policy will be overridden. Investigating counterexamples shows that this policy has effect inference conflict with P3. If meanwhile a burglary happens, system will lock all the doors and windows while fire has not been put out yet.

Property 2. $G\left(\mathcal{T}_{p1} \rightarrow F\left((\neg firePutOut \rightarrow (door1.closed \wedge \neg door1.locked) \wedge (window1.closed \wedge \neg window1.lock) \wedge alarm.gone_{off})\right) U\ firePutOut\right)$

Table 3. Conflicting actions

Action	Conflicting Actions
Open window/door	Close window/door, Lock window/door
Close window/door	Open window/door
Lock window/door	Open window/door, Unlock window/door
Unlock window/door	Lock window/door
Activate alarm	Deactivate alarm
Deactivate alarm	Activate alarm, Trigger alarm
Trigger alarm	Deactivate alarm
Open valve	Close valve
Close valve	Open valve

Conflict III. Inexecutable Action Conflict
The required condition for alarm to become triggered is being active. However, checking Property 3 reveals that in some cases triggering alarm is inexecutable. By investigating the counterexamples, we found that this happened when the inhabitant makes the alarm inactive manually which is modeled by sending messages from *XEnvironment*.

Property 3. $G(\mathcal{T}_{p3} \rightarrow G((alarm.queue[0] = trigger \rightarrow alarm.isactive) \wedge X(alarm.queue[0] \neq trigger)))$

Anomaly I. Action Redundancy Anomaly
Since there is no non-idempotent action in our example, therefore policies do not have action redundancy anomaly.

Anomaly II. Policy Redundancy Anomaly
As mentioned above, policy ρ_j is redundant respect to policy ρ_i if the property $G(\mathcal{T}_{\rho_j} \rightarrow \mathcal{T}_{\rho_i})$ holds. In our example, there is no redundant policy.

Anomaly III. Unenforceable Policy Anomaly
In our example, all the policies are enforceable. We checked $\neg \Diamond \mathcal{T}_{\rho_i}$ for each policy which is violated for all cases.

4. Related Works

There is a significant amount of work done in the area of policy analysis. However, there has been very little research done to tackle the problem of policy analysis by capturing the system behavior, as we have done in [4]. [15] proposed use of event calculus (EC) to express both policy and system behavior formally. Then, they use abductive reasoning provided by EC to detect situations in which an inconsistency occurs. However, in this approach all domain-specific inference rules to detect inconsistencies needed to be identified beforehand. In our approach, we propose general domain-independent patterns to discover conflicts. Moreover, works done by Kuninobu et al [16] and [17] have taken a model checking approach to verify policy-based systems, but they did not deal with policy conflicts. Layouni et al [18] proposed an approach to detect action conflicts among policies through analyzing the pre/post-conditions of policy actions. They identified the situations that an action conflict may arise as concurrency conflicts, disabling conflicts and results conflicts. The conflicts that they can detect are corresponded to our action conflict and inexecutable action conflict.

In the area of policy analysis in smart spaces, authors in [13] have proposed a semi-formal approach, called IRIS (Identifying Requirements Interactions using Semi-formal methods) to detect interactions between policies in smart home domain. They categorized policies as System Axioms and Dynamic Behavior policies. A Dynamic Behavior policy specifies the reaction of the system when a certain event occurs. Different types of conflicts can be detected which are correspondent to action conflict, effect inference conflict and goal conflict introduced by our approach. The most drawback of this approach is that it needs an expert to express policies which is an error-prone process. Moreover, the behavior of system is modeled partially by just expressing policies.

Wang et al[19] investigate policy conflict detection and resolution in home care systems. They classify different types of conflicts in policy-based home care systems as: conflicts that result from apparently separate triggers, conflicts among policies of multiple stakeholders, and conflicts resulting from apparently unrelated actions. However, they did not deal with detection of those conflicts specially by capturing system behavior. Authors in [20] proposed a framework to describe and verify integrated services of a home network system(HNS). They described the HNS and the integrated services using an object-oriented language which is transformed into SMV

(SymbolicModel Verifier) language. SMV model checker is used for formal verification. However, they have provided no classification of the feature interactions.

Run-time conflict detection [3, 21-25] is another area of research closed to us which detects policy conflicts during policy execution, i.e. it discovers conflicts online. In this approach, a detection module monitors system at run time to identify unpredicted conflicts. Actually, this class of techniques considers the behavior of system. Whether Run-time conflict detection is useful in policy analysis of large-scale systems but the detection process is usually time consuming and it may has undesirable influence on the system performance.

5. Conclusion and Future Works

In this paper, we dealt with dynamic policy conflict detection in smart home domain using a model checking approach. We used an actor-based language named Extended Rebeca to model the system. Then, we identified different kinds of conflicts which may exist between policies in smart home. To detect each conflict, we checked the satisfiability of the temporal patterns expressed in LTL. To tackle the problem of state explosion, we used compositional verification as well as data abstraction techniques. Our experience confirmed the ease and simplicity of Extended Rebeca as a modeling language for modeling smart homes. Moreover, we believe this research is a promising step toward verification of policy-based systems using compositional verification.

References

[1] M. Beigi, S. Calo, and D. Verma, "Policy Transformation Techniques in Policy-based Systems Management," in *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*: IEEE Computer Society, 2004.

[2] M. R. McGee-Lennon and P. D. Gray, "Addressing Challenges of Stakeholder Conflict in the Development of Homecare Systems," in *Workshop on Software Engineering Challenges for Ubiquitous Computing*, Lancaster, 2006.

[3] D. Nicole, I. Jadwiga, and R. Kerry, "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.

[4] N. Khakpour, M. Sirjani, and S. Jalili, "Model Checking Policy-based Systems," 2008.

[5] 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.

[6] 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.

[7] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977, pp. 46-57.

[8] E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic," in *Logic of Programs, Workshop*: Springer-Verlag, 1982.

[9] K. Pohl, G. Böckle, and F. v. d. Linden, *Software Product Line Engineering Foundations, Principles and Techniques*: Springer, 2005.

[10]H. Sumi, M. William, E.-Z. Hicham, K. Jeffrey, K. Youssef, and J. Erwin, "The Gator Tech Smart House: A Programmable Pervasive Space." vol. 38: IEEE Computer Society Press, 2005, pp. 50-60.

[11]M. C. Edmund, G. Orna, and E. L. David, "Model checking and abstraction." vol. 16: ACM, 1994, pp. 1512-1542.

[12]R. Adler, I. Schaefer, T. Schuele, and E. Vecchié, "From Model-Based Design to Formal Verification of Adaptive Embedded Systems," in *Formal Methods and Software Engineering*, 2007, pp. 76-95.

[13]M. Shehata, A. Eberlein, and A. Fapojuwo, "Using semi-formal methods for detecting interactions among smart homes policies," in *Science of Computer Programming*. vol. 67: Elsevier North-Holland, Inc., 2007, pp. 125-161.

[14]E. Clarke, D. Long, and K. McMillan, "Compositional model checking," in *Proceedings of the Fourth Annual Symposium on Logic in computer science* Pacific Grove, California, United States: IEEE Press, 1989.

[15]A. K. Bandara, E. C. Lupu, and A. Russo, "Using Event Calculus to Formalise Policy Specification and Analysis," in *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, Lake Como, Italy, 2003, pp. 26-39.

[16]S. Kuninobu, Y. Takata, N. Nitta, and H. Seki, "Policy Controlled System and Its Model Checking," *IEICE Transactions on Information and Systems*, vol. E88-D, pp. 1685-1696, 2005.

[17]S. Kikuchi, S. Tsuchiya, M. Adachi, and T. Katsuyama, "Policy Verification and Validation Framework Based on Model Checking Approach," in *Proceedings of the Fourth International Conference on Autonomic Computing*: IEEE Computer Society, 2007.

[18]A. F. Layouni, L. Logrippo, and K. J. Turner., "Conflict Detection in Call Control Using First-Order Logic Model Checking," in *Proceedings of 9th International Conference on Feature Interactions in Software and Communications Systems*, Amsterdam, May 2008, pp. 66-82.

[19]F. Wang and K. J. Turner, "Policy Conflicts in Home Care Systems," in *Proceedings of International Conference on Feature Interactions in Software and Communication Systems*, Grenoble, France, 2007, pp. 54-65.

[20]L. Pattara, T. Tatsuhiro, K. Tohru, N. Masahide, and M. Ken-ichi, "Describing and Verifying Integrated Services of Home Network Systems," in *Proceedings of the 12th Asia-Pacific Software Engineering Conference*: IEEE Computer Society, 2005.

[21]A. C. Gemma and J. T. Kenneth, "Ontologies to Support Call Control Policies," in *Proceedings of the The Third Advanced International Conference on Telecommunications*: IEEE Computer Society, 2007.

[22]J. T. Kenneth and B. Lynne, "Policies and conflicts in call control." vol. 51: Elsevier North-Holland, Inc., 2007, pp. 496-514.

[23]H. Lee, J. Park, P. Park, M. Jung, and D. Shin, "Dynamic Conflict Detection and Resolution in a Human-Centered Ubiquitous Environment," in *Universal Access in Human-Computer Interaction. Ambient Interaction*, 2007, pp. 132-140.

[24]E. Syukur, S. W. Loke, and P. Stanski, "Methods for Policy Conflict Detection and Resolution in Pervasive Computing Environments," in *Proceedings of Policy Management for Web workshop in conjunction with WWW 2005 Conference*, Chiba, Japan., 2005, pp. 13-20.

[25]I. Armac, M. Kirchhof, and L. Manolescu, "Modeling and Analysis of Functionality in eHome Systems: Dynamic Rule-based Conflict Detection," in *Proceedings of 13th IEEE International Conference on the Engineering of Computer Based Systems*, Potsdam, Germany, 2006.