An ECA-P Policy-based Framework for Managing Ubiquitous Computing Environments

Chetan Shiva Shankar, Anand Ranganathan and Roy Campbell
Dept. of Computer Science
University of Illinois at Urbana-Champaign
{chetan, ranganat, roy}@cs.uiuc.edu

Abstract

Ubiquitous Computing Environments feature massively distributed systems containing a large number of devices, services and applications that help end-users perform various kinds of tasks. One way by which administrators and end-users can manage these environments is through the use of policies. In particular, *obligation policies* are used to specify what actions *must* or *must not* be performed by the environment on the occurrence of certain events. Obligation policies are often specified as Event-Condition-Action (ECA) rules. However an important problem in ubiquitous computing systems is that different users and administrators may have conflicting policies for managing the system. Hence, a key challenge in policy-based management is detecting and resolving conflicts between multiple policy rules that get activated by a single event. Existing approaches are limited in power and scope mainly because they do not have semantic information about the effects of policy actions; hence, they cannot infer that two actions may conflict unless they are explicitly stated to be conflicting. In this paper, we propose an extended model of ECA called Event-Condition-Action-Post-Condition (ECA-P), where developers and administrators can annotate actions with their effects. The ECA-P model allows inferring that actions may conflict based on conflicting post-conditions. Detected conflicts are resolved using meta-rules that specify preferred system states. The ECA-P framework also detects failures in policy execution by using post-conditions to verify successful completion of policy actions. We present the details of the framework.

1. Introduction

Ubiquitous computing systems contain a large number of heterogeneous and mobile devices, services and applications. A commonly-used way of managing these complex environments is through policies. Policy-based management has been used in network switches [5], content distribution networks [6], distributed systems [7] and ubiquitous computing systems [8]. Policies are a means of specifying and influencing management behavior within a system, without coding the behavior into the manager agents [3]. These policies may be used for managing different aspects of a system such as Fault, Configuration, Accounting, Performance and Security Management [1]

Our notion of a ubiquitous system is a physically-bounded collection of devices, applications and services managed by a distributed meta-operating system, called Active Space. An active space supports mobile devices, applications and data. Mobile devices can enter an active space and get seamlessly integrated into the system. These devices can then use services, applications and data present in the space. In addition, applications and data can migrate across devices in the space. We use policies to help manage the dynamism and configuration of resources in Active Spaces.

Typically, policy-based management systems use some form of obligation policies to guide system behavior. Obligation policies specify what actions an entity *must* or *must not* perform on the occurrence of an event [3]. Obligation policies are specified as Event-Condition-Action (ECA) rules. These rules specify the actions to be performed when a certain event occurs and the specified condition is satisfied. A typical obligation policy in the management system may look like, “if a mobile device enters the active space and the space is running, request credentials”. A mobile device entering the active space generates an event in the location system. The management system receives this event, checks the state of the active space and if the space is running, requests the device for its credentials.

However, the current state of the art in policy-based frameworks suffers from two main limitations. Firstly, they have limited ways of detecting and resolving conflicts in policies. Secondly, they do not have mechanisms to ensure that policies are enforced or executed correctly. These limitations severely limit the effectiveness of policies as a way of managing ubiquitous computing environments.

Policy conflicts can arise when multiple policies guide system behavior. A conflict occurs when an event triggers multiple policy-actions that cannot occur together as specified by the system administrator [4]. Previous research efforts on conflict detection have focused on detecting modality conflicts [3], conflicts due to temporal events and roles [9] and conflicts due to opposing obligation and prohibition policies [8]. Conflicts that arise from effects of policy actions have not been addressed. For example, rule R1 may state, “if a person enters the active space, start authorization application” and rule R2 may state, “if a person enters the active space, suspend applications”. Suspending applications stops all applications running in the space including the authorization application. This leads to a

DRAFT

conflict between the two rules. We have found that such conflicts occur quite often in active spaces where the administrator who designs policy rules is oblivious of all effects of the policy actions since actions are developed by an active space developer. In order to detect such conflicts, information about the effects of policy actions is required.

Policy actions may not execute to completion due to various reasons such as changing active space configuration, device and component failure or software errors. For example, the above policy assumes that once the *suspend applications* activity is invoked all applications stop. Some applications may not terminate if there is unsaved data in them and therefore the action cannot be completed. Typically, most management systems assume that actions are successfully executed and so do not monitor policy enforcement. Bettini et. al. [11] motivates the need for obligation monitoring in policy management systems and discusses the measures to be taken if obligation is not fulfilled. Actions in active spaces can fail due to a variety of reasons such as change in configurations, device and application mobility and so on. Hence it is necessary to monitor the execution of actions to ensure that the system is not left in an inconsistent or invalid state as a result of action failure. If an action fails, fault tolerance policies guide the system to recover from failure or to degrade gracefully.

Policies based on ECA rules are unable to address the above problems since they do not contain any semantic information of policy actions. We have developed an extended model of ECA called *Event-Condition-Action-Post-Condition (ECA-P)* for specifying management rules for an active space. The ECA-P model specifies the state of the system, after a policy action has executed, as a post-condition. Policies are still specified using ECA rules but the actions are annotated with post-conditions to enable analysis of policy rules. This allows our management system to detect conflicts due to effects of policy actions and monitor policy enforcement.

Two or more management rules are considered to be conflicting if under the same event-condition they force the system into *conflicting post-conditions*. Previous conflict detection mechanisms [3, 9, 10] defined conflicting rules as ones that have *conflicting actions* for the same event-condition. Defining conflicts using conflicting post-conditions approach enables detecting conflicts due to effects of policy actions like in the above example. We show that the set of conflicts detected using conflicting actions approach is a subset of the set of conflicts detected by the conflicting post-conditions approach. We have developed static and dynamic conflict detection techniques and discuss them in this paper. Conflicts detected statically are resolved by the user while conflicts detected dynamically, at runtime, are resolved by resolution policies. We have developed a policy resolution framework to specify resolution rules that are used to resolve policy conflicts at runtime. Resolution rules specify the state that an active space is preferred to reach from an existing state. We present algorithms for the above techniques.

In section 2, we introduce the ECA-P based management system and the policy structure. In section 3, we present our static and dynamic conflict detection algorithms. Section 4 presents dynamic resolution technique to resolve conflicts at runtime and discusses the resolution algorithm. We discuss the architecture and implementation of the management system in sections 5 and 6 respectively. We analyze various algorithms presented in the paper in section 7. We discuss related work in section 8 and finally conclude the paper.

2. ECA-P based Management System

Policies are written by an administrator in an ECA language, similar to PDL [10] that we have developed. The language uses similar operators and constructs and support parameterized events. Policies are compiled and loaded into the management system by the active space administrator. The management system subscribes to events specified in the policy and initiates corresponding actions when those events are fired. Figure 1 shows the steps involved in loading a policy and initiating management actions.

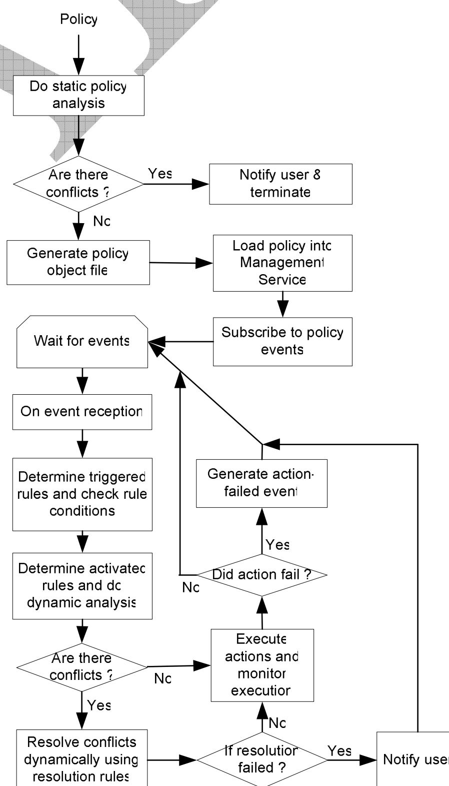

Fig. 1 Flowchart representing sequence of operations involved in loading a policy and initiating management actions

DRAFT

A policy is compiled and conflicts that are detected statically are listed for the user to resolve. If no conflicts are detected, a policy object file is generated and loaded into the management system. The policy object file contains the rules in a format suitable for loading into the management system. The management system subscribes to policy events and waits for the occurrence of an event. When an event is fired, the system determines the set of rules that need to be executed and checks for conflicts among them using a dynamic analysis algorithm. Since all rule conflicts cannot be detected during the static analysis done at the compilation stage, we also need to do dynamic analysis during runtime. Resolution rules are used to resolve conflicts and non-conflicting actions are executed. Successful execution of the action is verified by the policy-monitoring component of the system. If there was a failure, an action-failed event is generated. The system again waits for events.

2.1 Policy Structure

Our management framework uses policies that are formulated as sets of event-condition-action rules of the form

on *event* **if** *condition* **do** *action*

A policy rule is read as: When *event* occurs in a situation where *condition* is true, then *action* is executed. The *action* is a call to a method in a library of actions where each action is annotated with a *post-condition* by the action designer (programmer). Therefore, for the purposes of analysis (and in the rest of the paper) we consider our policy rules to be of the form event-condition-action-post-condition (ECA-P), although post-conditions are not specified as part of the rules. Our policy rule framework extends that of Policy Description Language (PDL) [10] by adding a post-condition as an "extension" to the rule.

There are three basic classes of symbols: primitive event symbols, action symbols and constant symbols. Primitive event symbols represent basic events that can be subscribed to in the system. For example, ObjectEnter and ObjectExit are primitive event symbols. Each action symbol denotes the name of a procedure that can be invoked in the system. An action is of the form proc(t_1, …, t_n) where proc is the name of a procedure and t_is are parameters. For example, suspend(light_app) is an action.

A policy is a finite collection of ECA-P rules, having *event*, *condition*, *action* and *post-condition* parts. The event part of a policy rule is a term of the form $e(t_1, \dots t_n)$ where e is a primitive event symbol of n arguments and each t_i is a constant, a typed variable of the form *T v,* where T represents a data type and v is a variable. The action part of a policy rule is an action term of the form $a(t_1, \dots , t_n)$ where a is an action symbol of n arguments and each t_i is a variable that appears in the event or condition parts of the rule or a constant. The condition and post-condition parts of an ECA-P rule is a conjunction of predicates, $p_1, \wedge \dots \wedge p_n$, where each p_i is a predicate of the form $pred(t_1, .., t_n)$ and each t_i is a constant or a variable.

Typical management rules used in our active space are shown in figure 2. The policy language uses terms and methods defined by services in our active space. For example, *ObjectEnter(Person p)*, is an event term that represents an event that is fired by the location system when a person enters the active space. *Person* is a user-defined data type and values of this data type are of the form *"Name"*. These policy rules specify actions to be performed when a person enters or exits an active space. Rule R1 starts an authorization application, to authorize users, when a person other than the owner of the active space enters it. Rule R2 stops all applications running in the space if user named Tom enters the space. When the active space owner exits the space, rule R3 gets triggered and checkpoints all applications in the space. Rule R4 suspends the space if no person is in the space. The post-conditions of actions are shown in braces, below each action, for ease of reading and are not specified as part of the policy rules.

R1: on(ObjectEnter(Person p))
  if (not_equal(role(p), "owner"))
  do (start(authorization_app))
  { status(authorization_app, running)}

R2: on(ObjectEnter("Tom"))
  if (true)
  do (stop_apps())
  { status(log_app, stopped),
  status(authorization_app, stopped)}

R3: on(ObjectExit(Person p))
  if (equal(role(p), "owner"))
  do (start(log_app))
  { status(log_app, running)}

R4: on(ObjectExit(Person p))
  if (equal(person_count(),0)) //space is empty
  do (suspend()) //suspend space
  { status(log_app, stopped),
  status(authorization_app, stopped)}

Fig. 2 Typical active space policy

2.2 Monitoring

Once a rule-action has been executed, the management service monitors the action completion by checking the post-condition of the action. We do not currently assume persistent actions and assume that actions run to completion in a finite time. Therefore, verifying post-conditions suffices in our system. If actions are persistent or have timing constraints then monitoring would require some temporal-reasoning approaches as proposed in [11].

If a post-condition evaluates to false, the action is assumed to have failed and the management service sends an *action-*

DRAFT

failed event to itself. If fault-management rules have been specified the management service takes appropriate actions.

3. Policy Conflict Detection

A policy conflict arises when, on an event-condition, multiple conflicting actions become eligible for execution and the system cannot decide on the action to be executed. For example, in the policy in figure 2 rules R1 and R2 result in conflict if a person named *Tom*, who is not the owner of the active space, enters the space since the rule R1 starts the authorization application while rule R2 stops it. Similarly, if the owner of the active space exits the space and no other person is in the space, the policy leads to a conflict since the checkpoint application is started in rule R3 while it is implicitly stopped in rule R4. Previous research works on conflict detection for obligation policies define conflicts as sets of *action constraints* [4, 7, 8]. Action constraints specify the set of actions that cannot occur together: $\neg(a_1 \wedge \ldots \wedge a_n)$ where a_i is an action term. While action constraints can detect modality conflicts, which are inconsistencies arising from actions being permitted and prohibited [3], conflicts due to temporal events and roles [9] and conflicts due to opposing obligation and authorization policies [10], they cannot detect conflicts arising from effects of actions. This is because these obligation policies do not contain information about the effects of the action. ECA-P rules specify post-conditions of actions and this enables the system to detect conflicts arising from the effects of actions. In the above example, the action *stop_apps()* stops all applications including the authorization application and this results in a conflict between rules R1 and R2. Stopping the authorization application is an effect of the *stop_apps()* action and this has to be explicitly specified if such conflicts have to be detected.

We define policy conflicts as violations of *post-condition constraints*. A *post*-condition constraint is a predicate that expresses a state that the system should not reach and is an expression of the form $\neg(p_1 \wedge \ldots \wedge p_m)$, where each p_i is a predicate representing a part of the state of the system.

We say that a set of post-conditions, K satisfies a post-condition constraint pc in a system if K is a model of pc, in the standard model theoretic sense [4]. This means that the post-conditions in K do not violate constraint pc. We use the standard notation $K \models pc$ to denote this relationship.

Definition 1. K is said to be a *non-conflicting post-condition set* for pc, if $K \models pc$. In other words, the post-condition predicates in K can all be simultaneously true in the system and so the system can reach a state that is expressed by the conjunction of those predicates. If $K \not\models pc$, then K is said to be a conflict.

Similarly, an action set S satisfies an action constraint ac if S is a model of ac and is denoted as $S \models ac$ [4]. If $S \not\models ac$, then S is said to be a conflict in the policy.

Definition 2. Given an action constraint $ac = \neg(a_1 \wedge \ldots \wedge a_n)$, we define an *equivalent post-condition constraint*, epc of ac, as a predicate $\neg(p_1 \wedge \ldots \wedge p_n)$ such that p_i represents the post-condition of action a_i. Note that p_i can be a conjunction of other predicates. It is represented as $epc =_e ac$

Theorem 1. The set of conflicts detected using action constraints is a subset of the set of conflicts detected using equivalent post-condition constraints.
Proof: Consider an action constraint ac and a post-condition constraint pc for a policy such that $pc =_e ac$. To prove the above theorem, we have to show that for every action set S, $S \not\models ac$, there is a post-condition set K such that $K \not\models pc$.

Let $K = \{p_i \mid \forall a_i \in S, p_i$ is the post-condition predicate of $a_i\}$

$S \not\models ac \quad \Rightarrow \exists a_1, \ldots \exists a_s \in S \mid \{a_1, \ldots, a_s\} \not\models ac$
$\quad \Rightarrow \{p_1, \ldots, p_s\} \not\models pc$, where p_i is the post-condition predicate of a_i and $pc =_e ac$
$\quad \Rightarrow K \not\models pc \quad$ (since $p_i \in K$)

Therefore, all conflicts detected by action constraints can be detected using equivalent post-condition constraints.

Our management system uses a combination of static and dynamic conflict detection techniques. Static detection is used by the policy compiler to detect conflicts at compile time. These conflicts include conflicts among rules whose event and condition parts can be statically matched. Dynamic detection is used at run-time to detect conflicts among rules that have been triggered.

We present our algorithms for static and dynamic conflict detection below. The static detection algorithm displays a set of conflicting rules which the user has to resolve. All conflicting rules cannot be detected by static analysis since this requires establishing equivalence among event and condition parts of rules. Determining equivalence between any two event and condition expressions requires model-checking or theorem-proving techniques, which are computationally intensive since they involve generating a large number of states. In fact, the problem of determining if two relational expressions are equal can even be undecidable. We, thus, employ dynamic conflict detection techniques to detect remaining conflicts at runtime (when we can evaluate all the conditions and thus determine conflicts), and use meta-rules to resolve them. Some policy-based systems [4, 8] employ resolution policies to resolve all policy conflicts. This requires specification of resolution rules for all possible conflicts. We felt that detecting and resolving as many conflicts as possible at policy compile time (before the policy is actually loaded into the system) allows catching and resolving some kinds of conflicts at an early stage and hence, reduces the number of dynamic

DRAFT

resolution rules that need to be specified. This is similar to the approach taken by other rule-based systems such as parser-generators where the user has to resolve all conflicting production rules before the parser is generated.

Static detection technique detects conflicts among rules whose events and conditions can be statically matched. In order to determine if two event terms match we compare their event symbols and types and values of their parameters. Therefore, in the policy in figure 2, event term of rule R1, *on(ObjectEnter(Person p))* matches that of R2, *on(ObjectEnter("Tom"))*, since the event symbol is the same and "Tom" is an instance of data type Person. This is an example of a process called unification used in type-checking in compilers. A condition expression C_1 matches another expression C_2 if the number of predicates of C_1 equals that of C_2 and for every predicate in C_1 there is a corresponding lexically-equivalent predicate in C_2.

Algorithm 1: Static Conflict Detection

```
// initialize
PC – post-condition constraint set
P - the Policy
event(r) – event of rule r
condition(r) – condition of rule r
id(r) – rule identification number of rule r
post(r) – post-condition of rule r

// statically detect conflicts for each rule
for each rule r in P
        e := event(r)
        c := condition(r)
        K = {}
        R = {}   // rule id set
        for each rule s in P and s ≠ r
                if (match(event(s),e) & match(condition(s), c))
                        K = K U {post(s)}
                        R = R U {id(s)}
                endif
        end for
        K = K U {post(r)}
        if (K U PC) is not consistent
                print(R)
        endif
end for
```

Algorithm 1 lists the set of rules that are conflicting. The algorithm is initialized with the post-condition constraint set, PC and the policy specified as ECA-P rules. Each rule, r, in the policy is evaluated against other rules in the policy. If the event and conditions of rule r *match* those of another rule s, the post-condition of rule s is added to set K and a rule identification number that uniquely identifies the rule in the policy is added to set. Once all rules have been evaluated, the post-condition of rule r is added to set K. The union of K and PC is checked for consistency. We use a Prolog reasoner to check for consistency among predicates. The predicates of post-conditions of rules in K are asserted into the Prolog knowledge base. For every post-condition constraint in PC, the truth value of each predicate of the constraint is checked. If more than one predicate evaluates to true for a constraint, it implies that constraint has been violated and the union of K and PC is considered inconsistent.

If the union set is consistent then there are no conflicts among the set of rules that match the events and conditions of rule r. If the union set is inconsistent, the rule identification numbers of the conflicting rules is displayed.

Applying this algorithm to the policy in figure 2 with the post-condition constraint

P1: ¬(status(App, running) ∧ status(App, stopped))

detects that rules R1 and R2 are conflicting, since the authorization application cannot be simultaneously in the *running* and *stopped* states according to the above constraints. However, the conflict between rules R3 and R4 is not detected since their condition expressions are not found to match statically.

Once the user resolves the above conflict, the policy compiler generates a policy object file. This is loaded into the management service. Note that the conflict between rules R3 and R4 still exists in the policy. The management service subscribes to events specified in the policy rules. When an *ObjectExit(Person)* event is received, the management service determines the set of rules triggered by the event, which in this case are rules R3 and R4. The condition expressions of the triggered rules are evaluated. If an expression is satisfied the rule is said to be *activated* and is added to an *activation set*. If the person who exits the space is an "owner" and the space is empty, rules R3 and R4 are both added to the activation set. The activation set is analyzed to determine any conflicts. The algorithm for detecting the conflicts is presented below:

Algorithm 2: Dynamic Conflict Detection

```
PC – post-condition constraint set
AS – activation set
K = {}

for each rule r in AS
        K = K U {post(r)}
end for
if (K U PC) is not consistent
        resolve conflict
```

The algorithm collects the post-conditions of all activated rules and checks if its union with the post-condition

constraint set is consistent. If it is not consistent, a conflict is detected and the rules are sent to a conflict-resolver for resolution. Therefore, for the above example, the algorithm detects a conflict between rules R3 and R4 since *log_app* application cannot be in the *running* and *stopped* states simultaneously, according to the post-condition constraint. The conflicts are sent to a conflict resolver for resolution.

4. Conflict Resolution using Resolution Policies

Policy conflicts are resolved using rules that specify the post-condition that the system can reach from a given condition. These rules are called *resolution rules* since they determine the rule to be executed from a set of conflicting rules. A set of resolution rules is called a resolution policy.

A resolution rule, m is specified as a simple if-then statement
if {condition} **then** {post-condition}

This is read as: "if the system is in state represented by *condition*, then the system is preferred to reach the state represented by *post-condition*". The resolution technique prioritizes one rule over another by stating that if two conflicting post-conditions can occur, then one of the post-condition is preferred over the other. The resolution algorithm, presented below, chooses the action corresponding to the preferred post-condition. This approach is similar to priority-based conflict resolution schemes [3,4,8] except that post-conditions are prioritized instead of actions. The condition of m is represented as *cond(m)* and post-condition as *post(m)*. The condition and post-condition expressions of resolution rules are represented similar to that of policy rules.

The resolution policy for the example in figure 2 is
M1: *if (event(ObjectExit) ∧ can_reach(status(log_app,running)) ∧ can_reach (status(log_app, stopped))*
then can_reach(status(log_app, running))

event(e) represents a predicate that determines if event *e* is the triggering event for any of the conflicting rules. *can_reach(p)* represents a predicate that determines if predicate *p* is in any of the post-conditions of the conflicting rules. So the resolution rule, M1 implies that if *ObjectExit* event occurs and application *log_app* can reach states *running* and *stopped* by the execution of the conflicting rules, then choose the rule corresponding to the post-condition *can_reach(status(log_app, running))*.

Definition 3. A post-condition predicate p is *preferred* by a resolution policy M = {m_1, … m_n} if there exists a resolution rule m in M, such that post(m) = p and cond(m) is *true*. p is *unpreferred* otherwise.

When a conflict occurs, the conflict-resolver is invoked with the conflicting rules and the resolution policy. The conflict resolution algorithm is presented below:

Algorithm 3: Dynamic conflict resolution

```
C – Conflict rule set                //set of conflicting rules
N – output rule set
M – resolution policy
post(s) – post-condition of rule s

condition(m) –condition of resolution rule m
post-condition(m) – post-condition of resolution rule m
evaluate(p) – returns evaluated value of expression p
preferred(p,M) - returns true if p is preferred by M and false
           otherwise
N = { }
for each rule r in C
        k = true
        for each predicate p in post(r)
                k  = k & preferred(p, M)
        end for
        if k = = true        //rule should be executed
                N = N U {r}
        end if
end for
if cardinality(N) > 1
        notify user
else
        return N

//function satisfy
Boolean preferred(p, M)
        for each resolution-rule m in M
                if (match(post-condition(m), p)
                        return evaluate(condition(m))
                end if
        end for
return false
```

The conflict-resolution algorithm takes in a *conflict-set* C that contains the set of conflicting rules and a resolution policy M and determines the rule to be executed. The post-condition of each rule in the conflict-set is evaluated against M to determine if the rule-action can be executed. This is done by checking to see if each predicate of the post-condition is *satisfied* by M. If all predicates of a post-condition are satisfied by M the corresponding rule is considered for execution by adding it to the *output set* N. If more than one rule is considered for execution, these rules may possibly conflict among since they come from the conflict-set and so we conclude that the conflict resolution process was not able to resolve policy conflicts with the given resolution rules. Detecting the maximal subset of a conflict-set such that no rules of the subset conflict requires advanced analysis techniques such as confluence analysis [14] and is currently not considered. Therefore, the resolution algorithm chooses rule R3. The management system executes the rule action of R3 and ignores R4.

DRAFT

5. Architecture

The management system is implemented as an active space service. Figure 3 illustrates the architecture of the management service. The management service contains a coordinator component that coordinates the interactions among various components of the service. The policy compiler compiles the management policy and generates a policy object file. The action library contains a library of actions that can be invoked from the action part of the policy rule. In addition, the action library contains post-conditions of the actions that are used for conflict detection. The policy loader loads the generated object file into the management service. The service stores the policy rules in a policy store, which is a simple database. The event receiver is responsible for subscribing to events and receiving them when they occur. The event receiver verifies the types of the parameters in the events and notifies the management coordinator of the event occurrence along with the parameters. The management coordinator determines the triggered rules, and uses the condition checker to test the rule condition expressions. If a condition evaluates to true the rule is added to the activation set. The dynamic conflict resolver determines and resolves any conflicts among rules in the activation set. The actions of non-conflicting rules are initiated by the action initiator component. The action initiator is multi-threaded and actions are executed concurrently. Once an action completes execution, the enforcement verifier uses the post-condition to test successful action completion. It sends an action-failed event in case of a fault.

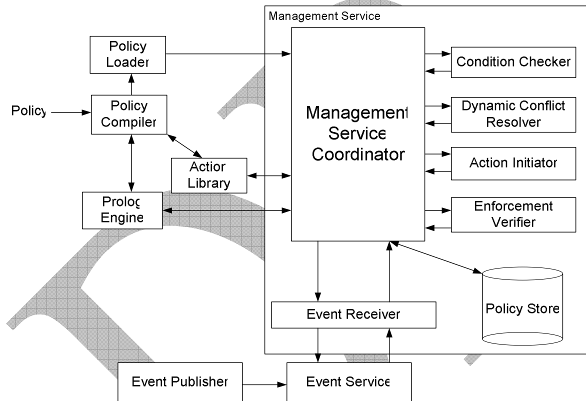

Fig. 3 Management Service Architecture

6. Implementation

Our active space consists of various devices such as plasma displays, tablet PCs, desktops, laptops, cameras and sensors and Gaia operating system provides essential services for discovering resources, sharing data and running applications in the space. The Gaia OS kernel services run on Windows 2000/XP. The services are implemented in C++ and Java using CORBA as the communication middleware. The CORBA implementation for Windows 2000/XP is Orbacus. The management system is implemented in C++ and runs as a service of the Gaia operating system. It uses a Prolog reasoner called XSB and uses C interfaces to communicate with XSB. The policy compiler and loader are implemented in Java. The policy compiler has a parser that is generated using the ANTLR parser generator tool. We use CORBA event channels for event communication and we have extended them to support parameterized events.

7. Evaluation

The management system has been used to manage various active space maintenance operations such as starting services and applications, requesting authorizations when people enter the space, managing configuration of the space when devices are added or removed from the space and so on.

For the static conflict detection algorithm (algorithm 1), the complexity of event matching is $O(b)$ where b is the average number of parameters of event terms. The complexity of matching two conditions is $O(q^2 * t)$ where q is average number of predicates in the conditions and t is the average number of arguments in the predicates. Since the inner loop iterates over each rule, the complexity of the inner loop is $O(n*(b+q^2*t))$ where n is the number of rules in the policy. The consistency checking has $O(n*c)$ Prolog assertions where c is the average number of predicates per post-condition. This is assuming the worst-case situation when all rules in the policy conflict and therefore all predicates in the post-conditions should be asserted. Checking consistency requires $O(d*e)$ queries to the Prolog reasoner, where d is the number of post-condition constraints and e is the average number of predicates per post-condition. Therefore, the complexity of consistency verification is $O(n*c + d*e)$. We do not consider the complexity of Prolog queries in this paper and assume it to be $O(1)$. Since the outer loop iterates n times, the final complexity is $O(n * (n*(b+q^2*t) + n * c + d * e) = O(n^2*(b+q^2*t) + n^2*c + n*d*e)$

By similar analysis, the complexity of the dynamic conflict detection algorithm (algorithm 2) is found to be $O(n + n^2*c + n*d*e)$. For the resolution algorithm (algorithm 3), the preferred method has a complexity of $O(m * (q^2*t)) + O(k)$, where m is the number of resolution rules, q is the average number of predicates in the post-condition of a resolution rule, t is the average number of terms per predicate and k is the number of predicates in the condition part of a resolution rule. Assuming c to be the number of predicates per post-condition of a policy rule and noting that the outer loop iterates over each rule in the conflict rule set, the final complexity is $O(p*c*(m*(q^2*t) + k))$, where p is the size of the conflict rule set.

DRAFT

8. Related Work

Policy-based management has been an active area of research for the past few years and many projects have focused on designing policy languages [8, 10, 12], detecting and resolving policy conflicts [1,3,4,8,9] and monitoring obligation enforcement [11]. Policies in these projects are designed using ECA rules and conflicts are detected by specifying the actions that cannot occur together. Sloman et.al. have developed the Ponder language to specify management policies [7, 12]. They provide a classification of various conflicts and suggest techniques to resolve modality conflicts and conflicts due to opposing obligation and authorization policies. Their policy rule framework does not provide any information about the rule action and so they do not detect conflicts due to effects of policy actions. Our ECA-P model provides information about the rule action as a post-condition and so we can detect conflicts due to effects of the rule actions. Chomicki et.al. [4] define a framework for detecting and resolving conflicts using a concept called *monitors*. A monitor of a set of conflicting actions determines a subset of the actions without conflicts. They define a conflict as a violation of action constraints that specifies the set of actions that cannot occur together. While action constraints can detect conflicts due to conflicting actions, they cannot detect conflicts arising from the effects of actions. In our work, we define conflicts as violation of condition constraints and this enables us to detect conflicts due to action effects. Further, we propose a monitoring system to verify successful completion of actions which they do not. Kagal et.al [8] propose a policy language called Rei for a pervasive computing environment. Rei is based on deontic concepts and includes constructs for rights, prohibitions, obligations and dispensations. They detect modality conflicts and conflicts between obligation and prohibition policies. They do not seem to resolve conflicts and their techniques cannot detect conflicts to action effects. Further, they do not support a monitoring system for verification like we have proposed. Bettini et.al. [11] have designed a obligation monitoring system that monitors successful execution of obligation actions using events. They do not detect or resolve obligation conflicts like we have proposed. Patwardhan et. al. [13] propose an architecture for enforcing policies in pervasive environments. The focus of their work is on enforcing security policies. The focus of our work is on enforcing obligation policies and therefore the techniques are significantly different from theirs.

9. Conclusion

In this paper, we propose a framework based on Event-Condition-Action-Post-Condition (ECA-P) rules for policy-based management of a ubiquitous computing system. ECA-P rules provide information about the effects of an action as a post-condition. This facilitates more powerful reasoning about policies and enables detecting conflicts between rules due to effects of actions. We show how post-condition based conflict analysis detects conflicts detected by action-based analysis techniques in addition to detecting conflicts due to effects of actions. We have designed a management system based on the ECA-P framework that enables policy enforcement and dynamic conflict resolution. We have implemented the system and evaluated the complexities of various algorithms in this paper. In the future, we plan to develop a graphical policy development environment for administrators to specify policies easily.

Reference

1. Lupu E.C., "A Role-Based Framework for Distributed Systems Management, Ph.D. Thesis, Imperial College, London.
2. Ranganathan, A., et. al., "Mobile Polymorphic Applications in Ubiquitous Computing Environments", Mobiquitous 2004: First International Conference on Mobile and Ubiquitous Computing, Boston, 2004.
3. Lupu E. C. et al. "Conflicts in Policy-Based Distributed Systems Management, IEEE Transactions on Software Engineering, Vol. 25, Nov 99, pp. 852-869.
4. Jan Chomicki, et.al, "Conflict Resolution Using Logic Programming", IEEE Transactions on Knowledge and Data Engineering, Vol. 15, 2003.
5. Bhatia R., et. al. "Policy Evaluation for Network Management", INFOCOM 2000, pp.1107-1116.
6. Amiri. K., et. al., "Policy based management of content distribution networks," IEEE Network Magazine, March 2002.
7. M. Sloman, "Policy Driven Management For Distributed Systems", Plenum Press Journal of Network and Systems Management, vol 2, no. 4, Dec. 1994, pp. 333-360
8. Lalana Kagal, et. al. "A Policy Language for a Pervasive Computing Environment", IEEE 4th International Workshop on Policies for Distributed Systems and Networks, June 04 - 06, 2003 Lake Como, Italy
9. Nicole Dunlop, et. al. Dynamic Conflict Detection in Policy-Based Management Systems, EDOC '02, 2002.
10. Lobo J., et. al., "A policy description language", in Proc. of AAAI, Orlando, FL, July 1999.
11. Bettini et.al. "Obligation Monitoring in Policy Management", Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02), 2002.
12. Damianou N., et. al., "The Ponder Specification Language", Workshop on Policies for Distributed Systems and Networks (Policy2001), HP Labs Bristol, 29-31 Jan 2001.
13. Patwardhan A., et. al., "Enforcing Policies in Pervasive Environments", Mobiquitous 04, Boston, 2004.
14. Baralis E. et. al. "Better Static Rule Analysis for Active Database Systems", ACM Transactions on Database Systems, 2000.

DRAFT