Ordering Management Actions in Pervasive Systems using Specification-enhanced Policies

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

Abstract

A pervasive system features a plethora of devices, services and applications organized as a large distributed system. One approach to managing such systems is by policies where administrators specify the management action to be taken in different situations using Event-Condition-Action (ECA) rules. An important problem with policy-based management of a pervasive system is that multiple rules can get triggered on a single event and the behavior of the system depends on the order of rule enforcement. Systems managed using ECA policies do not provide guarantees about system behavior when multiple rules are concurrently triggered.

In this paper, we present a novel rule framework called Event-Condition-Precondition-Action-Postcondition (ECPAP) that combines axiomatic specifications with ECA rules for specifying management rules. ECPAP rules contain action specifications in first-order predicate logic that enables us to reason about the enforcement order. We define a notion called enforcement semantics for policy-based management and show how this can be used to provide guarantees about system behavior. We present the details of the framework.

1. Introduction

Pervasive systems constitute large collections of heterogeneous and mobile devices, services and applications. Commercial deployment of these systems in smart offices, aware homes and other establishments requires an infrastructure that enforces organizational guidelines for usage of these systems. Policy-based management is a popular approach for enforcing such organizational requirements in network switches [3], content distribution networks [4], distributed systems [5] and pervasive systems [6, 1]. Policies are a means of specifying and influencing management behavior within a system, without coding the behavior into the manager agents [8]. These policies may be used for managing different aspects of a system such as Fault, Configuration, Accounting, Performance and Security [7].

Our notion of a pervasive system is a physically-bounded collection of devices, applications and services, called *Active Space* [9]. Gaia distributed meta-operating system [17] provides services for discovering new devices and services, integrating services of mobile devices with that of the space, migrating applications and data across devices and for various other functionalities. We use policies to manage the dynamism and configuration of resources of an active space. Policies guide the behavior when mobile devices are brought into the active space; applications are started; device file systems are mounted and so on.

Typically, policy-based management systems use policies designed using Event-Condition-Action (ECA) rules. These rules specify the action to be performed when a certain event occurs and the specified condition is satisfied. A typical rule would look like, “if a device physically enters the active space and the device is owned by the space owner, mount device file system”. A mobile device entering the active space generates an event in our location system [10]. The management system receives this event, checks the device ownership and mounts the device file system onto the active space file system.

Management policies are designed by system administrators who modify them periodically to conform to organizational and user requirements. Policies get altered by addition and deletion of rules as devices and applications are added or removed from the system, organization and user needs change and due to various other system dynamisms. An active space may have more than one administrator controlling different aspects of the system resulting in different policies that need to be merged. Policies may be designed for various system granularities such as domains, devices and applications and therefore multiple policies may simultaneously apply. Such policy operations create several inconsistencies among rules such as conflicts [2], dominance [12] and insufficient coverage [24]. In addition, multiple rules may need to be enforced on the occurrence of a single event and the management system should determine the enforcement order of these rules. Existing policy-based management systems execute rule actions once an event is received and condition is verified. If multiple rules are triggered by a single event the order of execution of the rule actions determines the behavior of the system. For example, policy rule R_1 may state "if a device enters an active space and the space has stopped, restart space" and rule R_2 may state "if a device enters an active space, authorize device". When a device enters the active space that is not running, both rules are triggered. If rule R_1 is enforced before R_2, then the active space and authorization application are both started. But if R_2 is enforced before R_1 the authorization application fails to start because the space services are not running. Current policy-based systems do not provide any guarantees to the order of enforcement of the management rules. While several research projects have addressed conflict detection [2, 3, 11], dominance checks [12] and coverage checks [24] no project on policy-based management has addressed the problem of ordering rule enforcement, to the best of our knowledge.

In order to reason about the order of enforcement, management systems require explicit specifications of rule actions and therefore policy rules based on ECA framework are unable to address the above problem. We have developed a specification-enhanced rule framework called Event-Condition-Precondition-Action-Postcondition (ECPAP) for specifying management rules for active spaces [1, 2]. ECPAP rules contain axiomatic specification of rule actions in first-order predicate logic as pre- and post-conditions. The pre-condition specifies the partial system state before execution of rule action while post-condition specifies the partial system state once the action has successfully executed. Note that rule condition is different from pre-condition because the rule condition is specified by the policy designer while the pre-condition is specified by the action developer (programmer). We have used the ECPAP rule framework for conflict detection and resolution and monitoring of rule enforcement in [2] and analyzing policy cycles in [1]. In this paper, we show how the ECPAP framework can be used to reason about enforcement order of rules. A typical policy containing ECPAP rules is shown in figure 2.

When multiple rules are triggered by a single event the management system detects conflicts and resolves those using priorities [1, 2]. The system then analyzes dependencies between different rule actions using pre- and post-conditions and constructs a Petri net-based workflow that defines the enforcement order of rules.

The problem of guaranteeing certain enforcement order of rules in management systems is roughly analogous to the problem of providing transaction guarantees in databases. Transaction semantics defines the order in which concurrent transactions need to be executed to meet certain correctness criteria. In policy-based management systems, we need to define similar semantics of enforcement to define the way in which multiple triggered rules need to be enforced to meet certain criteria. Therefore, we define *maximum rule enforcement semantics* for policy-based systems that guarantees that when multiple rules are concurrently triggered the system successfully enforces as many rules as possible. We formally prove that our algorithms for Petri net workflow construction guarantee the above semantics.

Our extension of the ECA framework with action specifications follows naturally from current research efforts in autonomic computing. There has been widespread interest lately on using planning techniques from AI for programming and managing pervasive and distributed systems with encouraging results [21, 22, 23]. These research works have shown that annotating actions with simple pre- and post-condition specifications provides numerous benefits such as raising the programming abstraction level and automating system management. Based on the success of these efforts we have extended management policies with action specifications and introduced the ECPAP framework [1, 2].

In section 2, we discuss our management system based on the ECPAP rule framework and present the policy structure. In section 3, we present algorithms for constructing the Petri net workflow and define our idea of enforcement semantics. We discuss the architecture and implementation details of our system in section 4. In section 5, we evaluate the system overhead by determining the algorithmic complexities and use those to explain empirical results. Section 6 discusses the feasibility of specification-enhanced programming and management and argues the viability of extending management policies with action specifications. In section 7 we relate our work to research in policy-based management and finally conclude the paper.

2. ECPAP Management System

Our management system is currently implemented as a service and receives events from other services and applications. Figure 1 shows the flowchart of policy enforcement. A policy is compiled and checked for conflicts and cycles using static analysis techniques [1, 2]. An object file is generated if the policy is free of static conflicts and loaded into the management system. The management system subscribes to events in the policy rules and waits for events to occur. Once an event is received the management system determines the set of triggered rules. It analyzes the set for dynamic conflicts [2] and resolves those using priorities. It determines the enforcement order of rules and constructs a Petri net workflow. This workflow is executed by a workflow execution engine and the system waits for further events. Currently we process each event separately and if subscribed events are generated during the workflow execution they are cached in the event reception system for processing in a queue. We are currently investigating approaches to evaluate policy rules based on

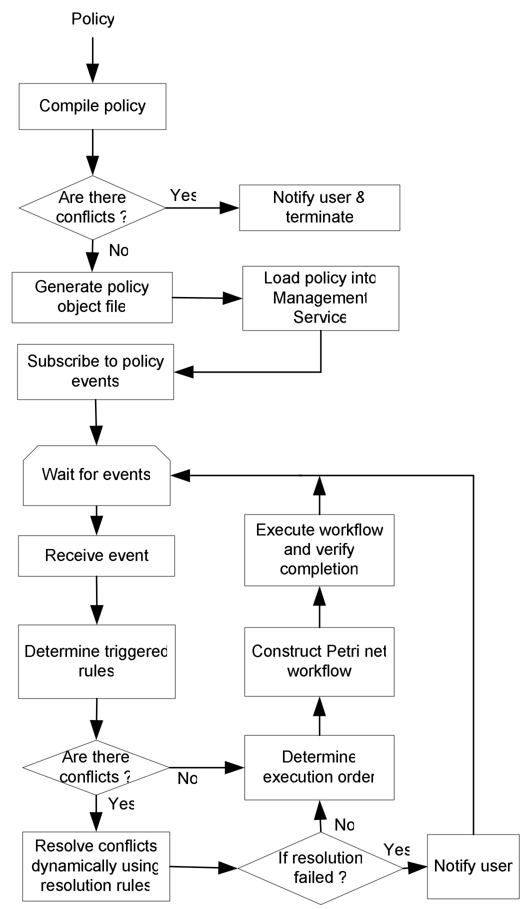


Figure 1. Flowchart of policy enforcement

complex events that are formed from composition of simple events and so we do not consider complex events in this paper.

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 execute action". The action is a call to a method in a library of actions where each action is annotated with a pre-condition and a post-condition by the action developer (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-precondition-action-postcondition (ECPAP), although pre-conditions and post-conditions are not specified as part of the rules. An action may be invoked by multiple rules in the policy and this format avoids listing the specifications at multiple places. We represent an ECPAP rule as $(e, c, p) \rightarrow (a, s)$ where e denotes the rule event, c denotes the condition of the rule, p is the pre-condition of the rule action, a, and s is the action post-condition. Our policy rule framework extends that of Policy Description Language (PDL) [13] by adding axiomatic specifications as "extension"s 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 that are generated by the location system when any object physically enters or exits a geographic region, respectively. An event is a primitive event symbol or a term of the form $e(T_1\ t_1, \ldots, T_n\ t_n)$, where e is a primitive event symbol of n arguments and each t_i is a variable of type T_i. T_i can be a simple type such as *int*, *float*, *char* or a complex type consisting of a set of attributes of simple or other complex types. The condition part of an ECPAP rule is an expression of the form $p_1\ \&\&\ p_2\ \&\&\ \ldots\ \&\&\ p_n$ where each p_i is a predicate of the form $x_1 \theta x_2$ and each x_i is a constant, a variable that appears in the event part of the rule or a function and θ is a relational operator. Each action symbol denotes the name of a procedure that can be invoked in the system. An action is of the form $proc(t_1, \ldots, t_n)$ where $proc$ is an action symbol and t_is are parameters. For example, *restartSpace(s)* is an action where s denotes an active space. Actions are defined in an action library that also contains pre- and post-conditions of actions. Pre-condition of an action is a first-order predicate logic formula of the form $p_1\ \&\&\ \ldots\ \&\&\ p_n$, where each p_i is a first-order predicate of the form $Q_1 t_1 \ldots Q_n t_n\ pred(t_1, .., t_n)$: Q_i is an optional quantifier and each t_i is a constant or a variable. Post-condition of an action is a predicate logic formula of the form $Q_1 y_1 \ldots Q_n y_n\ (events(E)\ \&\&\ P)$, where P is of the form $(p_1\ \&\&\ \ldots\ \&\&\ p_n)$ and each p_i is a predicate, E is of the form $(e_1\ \&\&\ \ldots\ \&\&\ e_m)$ where each e_i is a primitive event and Q_is are optional quantifiers and y_is are variables that appear in the E and P parts of the formula. E represents the conjunction of events that are produced by the action and P represents the condition that exists after all events in E have been observed (*events(E)* is true). We use the *events* keyword to identify the event part from the condition part. Post-conditions provide an action specification by listing all observed events produced by the action and the perceived partial state of the system after all events have been observed. Our active space uses asynchronous communication and therefore we use this approach, based on runtime verification techniques, to monitor action execution [2].

A policy, P is a finite set of ECPAP rules and is formally defined as $P = \{r \mid r \text{ is an ECPAP rule}\}$. The management system enforcing the policy expects as input an event, e and its occurrence is represented by *occ(e)*. The semantics of each rule, $(e, c, p) \rightarrow (a, s)$ in the policy is specified by the implication,

$$\mathrm{occ}(e) \wedge c \wedge p \rightarrow \mathrm{exec}(a)$$
$$\mathrm{exec}(a) \rightarrow \Diamond s$$

where exec(a) represents the initiation of the execution of action a. $\Diamond$ is the eventually temporal operator [14] and $\Diamond s$ means that s becomes true after a few execution steps. We are using a monitoring framework to monitor policy action execution. We interpret $\Diamond s$ as *bounded eventually* implying that s becomes true in a bounded number of execution steps or the action is assumed to have failed. The number of steps is system dependent and is independent of the ECPAP rule framework.

A typical management policy used in our active space is shown in figure 2. The policy language uses terms defined by services in our active space. For example, *ObjectEnter(Device d, Space s)*, is an event term that represents an event that is fired by the location system when a device is physically brought into an active space. Events contain data that map to arguments when the event is received by the management system. *Device* and *Space* are user-defined data types and the variables d and s contain values of the identifiers of device and space, respectively, when the event is received. Rule R_1 restarts the active space (and its various services) if it has stopped when a device enters the space. Rule R_2 authorizes a device of role *guest* if it enters the space. We have assigned roles to mobile devices to differentiate between devices of different users. Rule R_3 mounts a laptop's file system onto the active space file system [18] when the laptop is brought into the active space and R_4 unmounts it when the laptop leaves the space. The pre- and post-conditions of the actions are shown italicized in braces, above and below each rule action. When a guest user with a laptop enters an active space that is not running, the location system generates an *ObjectEnter* event that triggers rules R_1, R_2 and R_3. A rule is said to be *triggered* when its event has been observed and its condition has been evaluated to *true*. A rule is said to be *enforced* when its action is executed. The order of enforcement of the rules determines the behavior of the space. If R_2 is executed before R_1, R_2 fails since all services of the active space are stopped. Similarly, if R_3 is enforced before R_1, R_3 fails since the active space file system is not running. But if R_1 is enforced before R_2 and R_2 is enforced before R_3, the active space successfully restarts the space, authorizes the device and if successful mounts the laptop's file system. Therefore, when multiple rules are simultaneously triggered, the order of enforcement of rules determines the final system state. A policy-based management system must provide guarantees when multiple rules need to be concurrently enforced so that the system behavior is deterministic. Existing policy-based management systems based on ECA rules do not contain specifications of actions required for reasoning and so do not provide guarantees which can lead to unpredictable system states. Since our ECPAP rules contain action specifications we can reason about rule ordering and provide enforcement guarantees. In this paper, we show how the ECPAP framework is used for guaranteeing that when multiple rules are simultaneously triggered, the system enforces rules in an order that maximizes the number of rules successfully enforced.

R_1: on (ObjectEnter(Device d, Space s))
if (statusSpace(s) = = "stopped")
{statusService(spaceRepository(s), not_running}
do(restartSpace(s))
{events(endRestartSpace(s)) &&
statusSpace(s, running)}

R_2: on (ObjectEnter (Device d, Space s))
if (roleDevice(d) = = "guest")
{statusSpace(s, running)}
do(authorizeDevice(d, s))
{events(endAuthorizeDevice(d, s)) &&
authorizationStatus(d, authorized)}

R_3: on (ObjectEnter (Device d, Space s))
if (deviceType(d) = = "laptop" &&
roleDevice(d) = = "guest")
{statusSpace(s, running) && authorizationStatus(d, authorized)}
do (mountFileSystem(d, s))
{events(endMountFileSystem(d, s)) &&
statusFileSystem(d, mounted)}

R_4: on (ObjectExit (Device d, Space s))
if (deviceType(d) = = "laptop")
{statusFileSystem(d, mounted)}
do(unmountFileSystem(d, s))
{events(endUnMountFileSystem(d, s)) &&
statusFileSystem(d, unmounted)}

Figure 2. A typical active space policy

3. Ordering Management Action Execution

A management policy evolves over time by addition and deletion of rules, rule modifications and compositions. Therefore, each rule is generally enforced independent of other rules in the policy. This implies that when multiple rules are simultaneously triggered it is desirable that all rules are successfully enforced. As demonstrated in the previous section, order of enforcement of rules determines if a rule action successfully executes. Therefore, we define a notion called *enforcement semantics* that provides certain guarantees about rule enforcement. Enforcement semantics of a policy-based management system dictates the way rules are to be enforced when multiple rules are simultaneously triggered. Since our goal is to successfully execute as many rules as possible, we call the enforcement semantics of our management system as *maximum rule enforcement semantics*. This semantics guarantees that the management system enforces rules in an order that ensures as many rules are successfully enforced as possible, provided no other errors cause rule enforcement to fail.

When a set of rules is triggered, we determine the execution order of the rule actions by constructing a workflow that expresses dependencies between different actions. The pre- and post-conditions of actions are used to determine which action enables which other actions. An action is said to *enable* another action if the post-condition of the former satisfies the pre-condition of the latter. For example, in the policy in figure 2, when rules R_1 and R_2 are simultaneously triggered, execution of the action of R_1 brings the active space to a *running* state as indicated by the corresponding post-condition. This satisfies the pre-condition of action of rule R_2 and thus enables R_2's action. Therefore, enforcing R_1 before R_2 successfully enforces both rules.

The workflow of rule actions is represented as a Boolean Interpreted Petri net (BIPN) [16]. A Boolean Interpreted Petri net is a Petri net [19] whose transitions are assigned Boolean functions. A transition can fire only when all of its input places are marked and its Boolean function evaluates to *true*. We assign a place to each action and each transition is assigned the pre-condition of the action that is connected by a directed edge from the transition as the Boolean function. The Petri net for the triggered rules (R_1, R_2 and R_3) of figure 2 when the *ObjectEnter* event is received is shown in figure 3. The action of rule R_i is represented as A_i.

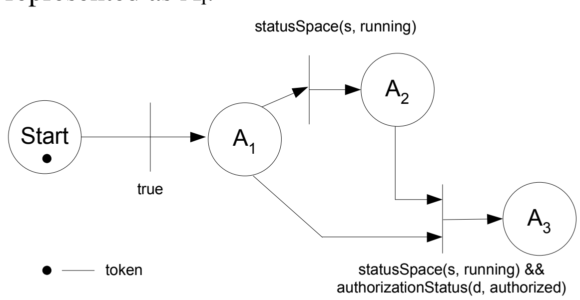


Figure 3. Petri net workflow for triggered rules of policy in figure 2

Definition 1. Formally, the BIPN of a set of actions $A=\{a_1,\ldots,a_n\}$ is a 1-safe marked Petri net [16] represented as a triple B = (P, T, F) where

$P = \{place(a) \mid \forall a \in A\} \cup \{Start\}$, where *place(a)* is the place representation of action *a*.

$T=\{t_{K,pre(a)} \mid \forall x \in K,\ t_{K,pre(a)} \in x\bullet \wedge\ place(a) \in t_{K,\,pre(a)}\bullet\}$, where K is a set of places and for $x \in P \cup T$,

$\bullet x = \{y \mid yFx\}$ is called the *input set* of x and

$x\bullet = \{y \mid xFy\}$ is called the *output set* of x

and the flow relation, $F \subseteq (PxT) \cup (TxP)$ such that $dom(F) \cup codom(F) = P \cup T$. *pre(a)* represents the pre-condition of action *a*.

3.1 Petri net Workflow Construction

A Petri net workflow expresses dependencies between different actions and therefore to construct a workflow we analyze each pair of actions to determine if one enables the other. Pre-conditions of certain actions are satisfied by the current system state and therefore these actions are called *trivially-enabled actions*.

Definition 2. An action *a* is said to be *trivially-enabled* if the current state of the system, *I*, satisfies its pre-condition. It is represented as $I \models$ *pre(a)*, where $\models$ is the *satisfies* symbol.

Intuitively, trivially-enabled actions are independent of other actions and can be executed as the first set of actions in the workflow. For example, the pre-condition in rule R_1 is *statusService(spaceRepository(s), not_running)*. *spaceRepository(s)* returns the identifier of an active space service called *Space Repository* that contains information about applications and devices in the active space. If the space repository is not running, it implies that there are currently no running applications in the active space and so it is safe to restart the space. If the active space is not running, the pre-condition evaluates to *true* and therefore A_1 can be executed independent of A_2 and A_3. The algorithm to determine trivially-enabled actions is shown below.

Algorithm 1: Trivially-enabled action analysis

V = { }: set of trivially-enabled actions
A : set of actions of triggered rules
for each action a in A
    if pre(a) evaluates to true
        $V = V \cup a$

Once trivially-enabled actions have been identified, we check to see which action enables which other actions through *enablement analysis*.

Definition 3. An action a_1 is said to *enable* action a_2 if $post(a_1) \models pre(a_2)$ where $post(a_1)$ represents the post-condition of action a_1 and a_2 is not trivially-enabled. This implies that execution of a_1 would satisfy the pre-condition of a_2 and so a_2 can be executed after a_1. Since trivially-enabled actions are already enabled by current system state we do not check to see if any actions enable them. The algorithm for enablement analysis is shown below.

Algorithm 2: Enablement analysis

```
Enable(a) = { }  : set of actions enabled by action a
V      : set of trivially-enabled actions from algo. 1
A      : set of actions of triggered rules
for each action a ∈ A
   for each action b ∈ A-V
      if post(a) ⊨ pre(b)
         Enable(a) = Enable(a) ∪ {b}
```

This algorithm determines that executing A_1 enables A_2. Post-conditions of some actions may satisfy part of the pre-condition of another action. For example, post-condition of A_1 – *events(endRestartSpace(s)) && statusSpace(s,running)* – satisfies a part of the pre-condition of A_3 (*statusSpace(s, running)*). Similarly, post-condition of A_2 satisfies a part of the pre-condition of A_3. Therefore, A_1 and A_2 must be executed to enable A_3. We say that each action A_1 and A_2 *partially-enables* A_3. Note that the variables *s* and *d* in predicates *statusSpace(s,running)* and *authorizationStatus(d, authorized)* are bound to values of the active space and device during evaluation, respectively, and thus form propositions whose satisfiability checks are decidable.

Definition 4. An action a_1 is said to *partially-enable* action a_2 if $post(a_1) \vDash partial\text{-}pre(a_2)$, where $partial\text{-}pre(a_2)$ is a conjunction of some proper subset of conjuncts of $pre(a_2)$. A set of partially-enabling actions of an action *a* that together enable *a* is called a *partial-set* of *a*. An action may have multiple partial-sets and therefore, the set of all partial-sets of *a* is denoted by *partial-sets(a)*. In the above example, $partial\text{-}sets(A_3) = \{ \{A_1, A_2\} \}$. The following algorithm determines the partial-sets.

Algorithm 3: Partial-sets determination

```
Partial-sets(a) = { }    : set of partial-sets of action a
A         : set of actions of triggered rules
V         : set of trivially-enabled actions
S         : temporary set
for each action a ∈ A-V
   S = { }
   for each action b ∈ A-{a}
        if b partially-enables a
           S = S ∪ {b}
   for each subset s of S
        if (cardinality(s) > 1)
           p = true
           for each action c ∈ s
               p = p ∧ post(c)
           if p ⊨ pre(a)
             Partial-sets(a) = Partial-sets(a) ∪ {s}
        end if
   end for
end for
```

The above algorithm determines for every action *a* that is not trivially-enabled, which set of actions collectively enable *a*. If the set contains only one action, then it implies that a single action enables *a* and therefore is already determined by algorithm 2. Therefore, algorithm 3 only considers sets having more than one element. In addition, the algorithm does not test an action with itself as this might lead to a deadlock.

Though the algorithm for partial-enablement analysis can replace enablement analysis of algorithm 2, we separate the two algorithms since partial-enablement analysis has a much higher complexity as detailed in section 5.

Once we determine the partial-sets, we construct the workflow as a Petri net using algorithm 4. The Petri net is represented as an adjacency set of places and transitions.

Algorithm 4: Petri net Workflow Construction

```
A                 : set of actions of triggered rules
V                 : set of trivially-enabled actions
Enable(a)         : set of actions enabled by action a
Partial-sets(a)   : set of all partial-sets of action a
P  = {Start}      : set of Petri net Places –
                    initialized to Place called 'Start'
```

```
T  = { }            : set of Petri net Transitions
place(a)            : Place for action a
adj(x)              : adjacency set of x,  x ∈ P ∪ T
trans(p, f)         : Transition with function f
             connected by edges from places in set p

for each action a ∈ A
    P = P ∪ {place(a)}
t = trans({Start}, true)
adj(Start) = adj(Start) ∪ {t}
T = T ∪ {t}

for each action a ∈ V     //trivially-enabled actions
    adj(t) = adj(t) ∪ {place(a)}
for each action a ∈ A                //enable
     for each action b ∈ Enable(a)
          t = trans({place(a)}, pre(b))
          if t ∉ T
               T = T ∪ {t}
               adj(place(a)) = adj(place(a)) ∪ {t}
          end if
          adj(t) = adj(t) ∪ {place(b)}
     end for
end for
for each action a ∈ A-V            //partially-enable
     for each set s ∈ Partial-sets(a)
          t = trans(s, pre(a))
          T = T ∪ {t}
          adj(t) = adj(t) ∪ {place(a)}
          for each action b ∈ s
               adj(place(b)) = adj(place(b)) ∪ {t}
      end for
end for
```

This algorithm constructs a BIPN using the results from algorithms 1-3. It initializes the Petri net by assigning a place to every action. The *Start* place is connected to each place representing trivially-enabled actions through a transition with the *true* Boolean function. We assign the Boolean function *true* to the transition since the pre-condition of all trivially-enabled actions evaluate to true. For each action *a* enabling action *b* a transition is created with the Boolean function *pre(b)* that connects *place(a)* to *place(b)*. Finally, for every set of actions *s* enabling an action *a*, a transition with Boolean function *pre(a)* is created that connects places representing actions in *s* to *place(a)*.

The Petri net generated from algorithm 4 for action set A is represented as B = (P, T, F) where

$$P = \{place(a) \mid \forall a \in A\} \cup \{Start\}$$

$$\begin{aligned} T = \{t_{i,j} \mid\ & (i = \{Start\}, j = true) \wedge \\ & (i = \{place(a)\}, j = pre(b) \mid \forall a,b \in A, (post(a) \models pre(b)) \wedge (pre(b) \not\models true)) \wedge \\ & (i = s, j = pre(b) \mid \forall b \in A, (\forall s \in 2^{P-\{Start\}}, \Lambda_{\forall k \in s}\, post(action(k))) \models pre(b)) \}, \end{aligned}$$

action(k) represents the action in set *A* assigned to place *k*.

$$\begin{aligned} F = \{ (x,y) \mid \forall t_{i,j} \in T, \forall x \in i, y=t_{i,j}\} \cup \{(x,y) \mid \\ \forall t_{i,j} \in T, (x= t_{i,j} \wedge y=place(k), \forall k \in A \mid j=pre(k))\} \end{aligned}$$

The three conjuncts in the definition of T correspond to the transitions resulting from algorithms 1-3. The transitions are labeled $t_{i,j}$ where $i = {\bullet}t_{i,j}$ and j is the assigned Boolean function. The flow relation, F, represents the various edges of the Petri net.

Theorem 1. For a set of actions $A = \{a_1, \dots , a_n\}$, the Petri net generated by algorithm 4 enables maximum number of actions starting from the current system state I.

Proof. To prove the above theorem, it is sufficient to prove that for every action $a \in A$, if $I \Rightarrow_k a$, then there is a reachable path [19] in the Petri net from the *Start* place to *place(a)*, where $I \Rightarrow_k a$ means that starting from the current system state I, successful execution of k actions of A enables a. $X \rightarrow a_1$ implies execution of all actions of set X enables a_1.

We prove this by structural induction on the Petri net.

Basis: $I \Rightarrow_0 a$

pre(a) is satisfied by current system state and so a is trivially-enabled by algorithm 1. Therefore, $t_{\{Start\}, true} \in T$ and $\{(Start, t_{\{Start\}, true}), (t_{\{Start\}, true}, place(a))\} \subseteq F$. Therefore, there is a reachable path from S to *place(a)* through the transition labeled $t_{\{Start\}, true}$.

Hypothesis: Assume if $I \Rightarrow_k a$ there is a reachable path from *Start* to *place(a)*. We need to prove that if $I \Rightarrow_{k+1} a_1$ there exists a reachable path from *Start* to $place(a_1)$.

Since $I \Rightarrow_k a$ from our inductive hypothesis, there is a set of actions $A' \subset A$ such that $\forall x \in A'$, $I \Rightarrow_{l \leq k} x$ and $A' \rightarrow a_1$. Therefore, there is a reachable path from *Start* to *place(x)* for all $x \in A'$. There are two cases to consider.

Case 1: $A' = \{a\}$

Since a is found to enable a_1 from enablement analysis in algorithm 2, $t_{\{place(a)\},\ pre(a1)} \in T$ and $\{(place(a),\ t_{\{place(a)\},\ pre(a1)}),\ (t_{\{place(a)\},\ pre(a1)},\ place(a_1))\} \subset F$. Therefore, there is a reachable path from *place(a)* to *place(a₁)* and since by hypothesis there exists a reachable path from *Start* to *place(a)*, by transitivity, there is a reachable path from *Start* to $place(a_1)$.

Case 2: $Cardinality(A') > 1$

Actions in A' are found to enable a_1 from partial-enablement analysis in algorithm 3. Therefore, $t_{\{place(x) \mid \forall x \in A'\},\ pre(a1)} \in T$ and $\{(place(a) \mid \forall a \in A',\ t_{\{place(x) \mid \forall x \in A'\},\ pre(a1)}),\ (t_{\{place(x) \mid \forall x \in A'\},\ pre(a1)},\ place(a_1))\} \subset F$. Therefore, there is a reachable path from $place(x),\ \forall x \in A'$ to $place(a_1)$ through the transition $t_{\{place(x) \mid \forall x \in A'\},\ pre(a1)}$. Since there is a reachable path from *Start* to $place(x),\ \forall x \in A'$ from our hypothesis, by transitivity, there is a reachable path from *Start* to $place(a_1)$. $\square$

3.2 Petri net Workflow Execution

Once the workflow is constructed, the actions are executed using our Petri net workflow execution engine. The engine analyzes the Petri net for any deadlocks using the deadlock detection algorithm described in [19]. If a deadlock is found the execution engine does not execute any action in the workflow. Currently, we do not resolve deadlocks and abandon the workflow. If the Petri net is deadlock-free, the engine uses a simple Petri net traversal algorithm based on Breadth-First Search (BFS) to traverse the net and execute actions. The transition states of the Petri net act as synchronization points in the workflow. When multiple places lead to a single transition, the engine waits for the completion of all actions in the places before executing actions of places leading out of the transition. At each transition, the engine verifies the Boolean function for satisfaction before executing the following action.

4. Architecture and Implementation

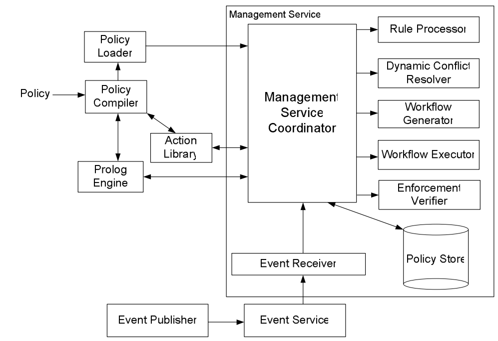


Figure 4. Management Service Architecture

Figure 4 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 an object file. The action library contains a library of actions that can be invoked from the action part of the policy rule. The management service uses dynamic invocation to invoke actions and this enables actions to be dynamically added into the action library. In addition, the action library contains action specifications as pre- and post-conditions. These specifications are used for static and dynamic conflict detection and resolution [2], termination analysis [1] and for reasoning about enforcement order when multiple rules are simultaneously triggered. 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 rule processor to test the rule condition expressions. If a condition evaluates to true the rule is added to a triggered action set. The dynamic conflict resolver determines and resolves any conflicts among rules in the action set [2]. The workflow generator constructs a Petri net workflow of actions that is executed by the workflow execution engine.

Enforcement verifier detects the end of each rule action by monitoring the events in the post-condition and informs the workflow executor.

Our active space consists of various devices such as plasma displays, tablet PCs, desktops, laptops, cameras and sensors and Gaia operating system [17] 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 used for Windows 2000/XP is Orbacus. The management system is implemented in Java. It uses a Prolog reasoner called XSB and uses JNI 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. Location of an object is sensed by Ubisense Location System [25] installed in our active space lab. Ubisense uses ultra wideband (UWB) sensing technology to detect location of Ubisense badges. Each device is associated with a badge and the location of the badge indicates the location of the device.

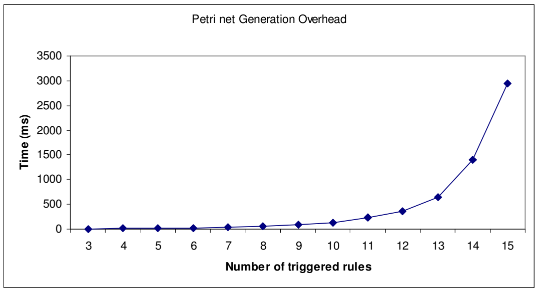
5. Evaluation

The management system is currently used for managing configuration of our active space when mobile devices are brought into the space; new services and applications are deployed in the space; applications are migrated across spaces and so on. The location service of Gaia [10] and presence service [17] are used for generating configuration change events. Location service generates location events when badges enter or exit a region while presence service generates events when new applications and services are started or stopped in the system. In this section, we will discuss the algorithmic complexities of the various algorithms presented in the paper and use them to explain the system performance that we have empirically measured.

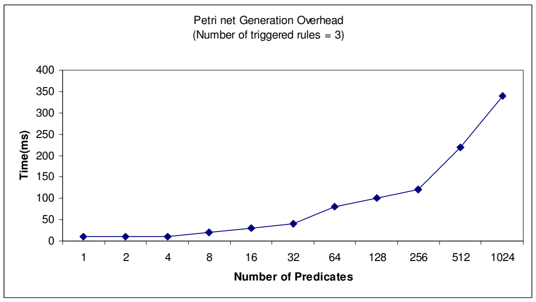
Trivially-enabled action analysis (algorithm 1) has a linear complexity of $O(n)$ pre-condition checks for n actions. Enablement analysis (algorithm 2) does a pair-wise satisfiability check of actions and therefore has a quadratic complexity of $O(n^2)$. Partial-enablement analysis (algorithm 3) analyzes for each action if it is enabled by a set of actions. Each action subset must be determined and this has an exponential complexity of $O(2^n)$. Since each subset is tested to see if it enables the action for all actions the final complexity is $O(n^2 2^n)$. Currently, algorithm 3 has a very high complexity but there are various optimizations that can be performed to reduce the value of n. For example, the enablement analysis algorithm reduces the number of rules to be verified during partial-enablement analysis. Since enablement analysis has a quadratic complexity the overall performance overhead is greatly reduced. In addition, the number of rules that are normally triggered on a single event is quite less (less than 5 rules per event in our active space policy) and so the overhead is tolerable. We are currently looking at static analysis techniques to determine dependencies between different rules at policy compilation time. Finally, the Petri net generation algorithm (algorithm 4) has a worst-case complexity of $O(n^2 2^n)$ since it uses results from partial-enablement analysis algorithm and so is bounded by the latter's complexity.

The performance overhead for Petri net workflow generation is shown in figure 5. The management system was executed on a Pentium(M) 1.7GHz machine with 1.0GB RAM. Figure 5(a) shows the overhead with varying number of triggered rules. Our test policy had multiple instances of the same rule since the focus was on testing the overhead of the system. As predicted from the algorithmic complexity described above the overhead is exponential with the number of triggered rules. For 15 triggered rules the overhead was found to be around 3 seconds. Normally, for a typical policy, the number of rules triggered on a single event can be expected to be much less than 15 and so the approach is feasible.

The number of predicates in pre- and post-conditions of actions influences the Petri net generation overhead. Therefore, we measured the overhead with varying number of predicates in action specifications. Figure 5(b) illustrates



(a)



(b)

Figure 5. Petri net Workflow Generation Overhead – (a) Overhead vs Number of Triggered Rules (b) Overhead vs Average Number of Predicates

the performance overhead of the system. The x-axis indicates the average number of predicates for each pre- and post-condition. The y-axis shows the overhead in milliseconds. The overhead is less than linear though the curve appears exponential in the graph due to the exponential increase in the values of x-axis.

6. Discussion

Specification-enhanced programming and system management have recently gained prominence as important approaches to reducing programming and management efforts of complex systems [1, 2, 21, 22, 23]. In [1, 2] we showed how extending actions with specifications enabled advanced conflict and termination analysis for policy-based management systems. Andrzejak et al [21] have used actions with pre- and post-conditions for planning complex workflows from simple actions for system management. Anand et al [22] use specification-enhanced actions, expressed as pre-conditions and effects, for programming pervasive computing environments. The ABLE project [23] uses axiomatic specifications of actions for goal-based autonomic computing. All of these approaches have shown that providing specifications for actions is a feasible extension that provides numerous benefits. Therefore, it would be fair to assume that actions for policy-based management can be annotated with specifications and is a viable approach.

Currently, in this work we assume that action specifications are correct and complete to the extent required for reasoning about rule enforcement. Dealing with incorrect and incomplete specification is an orthogonal problem and does not fall within the scope of our work.

7. 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 [6, 13, 15], detecting and resolving policy conflicts [2, 8, 11] and various other analyses [7, 12, 23]. To the best of our knowledge, no research work on policy-based management has addressed the problem of ordering management rules and providing enforcement guarantees as we have addressed in this paper. One of the well-addressed problems when multiple rules are triggered is conflict analysis. Dunlop et al [20] use temporal characteristics of policies to dynamically reason about policy consistency. Their approach detects a large class of conflicts that cannot be detected statically. The focus of their work is on conflict analysis and not on ordering rule enforcement as we have presented in this paper. In [1] we used the ECPAP framework to detect and resolve dynamic conflicts that occur due to side-effects of actions. In this paper, we use this work for conflict analysis prior to constructing the Petri net workflow. Sloman et al [5, 7, 8] have contributed extensively to research on policy-based management of distributed systems. They have developed the Ponder policy specification language and defined techniques for conflict analysis and role-based management. To the best of our knowledge, their work does not address the problem of ordering concurrently triggered rules. Many research projects in autonomic computing reason about action ordering [21, 22]. These projects are based on AI planning techniques where users specify high-level goals and the planning system determines the ordered set of actions to be executed to reach the desired goal state. The main difference between these projects and our work is that in goal-based approaches the final system state that needs to be reached is known and the system has to determine the actions to be executed to reach that state. In the problem that we have addressed, the final system state is unknown. When an event occurs, a set of rules get triggered and we need to reason about the execution order of the rule actions based on some enforcement semantics.

8. Conclusion and Future Work

Pervasive systems and services are gaining ubiquitous presence with commercial deployments in smart offices, aware homes and other establishments. Policy-based management is a feasible approach for enforcing organizational guidelines for usage of these systems. Policies are designed as sets of Event-Condition-Action rules that guide the behavior of these systems when certain events occur. Some events trigger multiple rules and the order of enforcement of these rules determines the system behavior. In this paper, we address this problem of ordering enforcement rules when they are concurrently triggered by a single event. We use a specification-enhanced rule framework called Event-Condition-Precondition-Action-Postcondition (ECPAP) for reasoning about enforcement order. The new rule framework enables us to construct a workflow of actions from triggered rules using Boolean Interpreted Petri nets. We define a new notion called maximum action enforcement semantics and show how this semantics provides enforcement guarantees for policy-based management of pervasive systems.

Currently, we are investigating several different enhancements to this work such as optimization of Petri net generation algorithms, defining correctness criteria and role-based approaches to managing pervasive systems.

Reference

[1] C.Shankar and R.Campbell, "A Policy-based Management Framework for Pervasive Systems using Axiomatized Rule Actions", *Fourth IEEE International Symposium on Network Computing and Applications (IEEE NCA05)*, MA, July 2005.

[2] C.Shankar, A.Ranganathan and R.Campbell, "An ECA-P Policy-based Framework for Managing Ubiquitous Computing Environments", *Mobiquitous 2005,* San Diego July 2005.

[3] R. Bhatia, et al., "Policy Evaluation for Network Management", *INFOCOM 2000*, pp.1107-1116.

[4] K. Amiri, et al., "Policy based management of content distribution networks," *IEEE Network Magazine*, 2002.

[5] 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.

[6] L. Kagal et al., "A Policy Language for a Pervasive Computing Environment", *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, June 2003.

[7] E. C. Lupu, "A Role-Based Framework for Distributed Systems Management", PhD Thesis, Imperial College, London.

[8] E. C. Lupu, et al., "Conflicts in Policy-Based Distributed Systems Management", *IEEE Transactions on Software Engineering*, Vol. 25, Nov 99, pp. 852-869.

[9] A. Ranganathan, et al., "Mobile Polymorphic Applications in Ubiquitous Computing Environments", *Mobiquitous 2004,* Boston, 2004.

[10] A.Ranganathan et al., "MiddleWhere: A Middleware for Location-Awareness in Ubiquitous Computing Applications", *Middleware 2004: ACM/IFIP/USENIX 5th International Middleware Conference*, Toronto, Canada, Oct. 2004.

[11] J.Chomicki et al., "Conflict Resolution Using Logic Programming", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, 2003.

[12] D.Verma, "Simplifying Network Administration using Policy based Management", *IEEE Network Magazine*, 2002.

[13] J. Lobo, et al., "A policy description language", in *Proc. of AAAI*, Orlando, FL, July 1999.

[14] Z.Manna and A.Pnueli, "The Temporal Logic of Reactive and Concurrent Systems", Springer-Verlag, 1995.

[15] N.Damianou et al., "The Ponder Specification Language", *Workshop on Policies for Distributed Systems and Networks (Policy2001)*, HP Labs Bristol, 29-31 Jan 2001.

[16] B.N. Roussev, "Self-checking Implementation of Boolean Interpreted Petri Nets", *IEEE Symposium on Emerging Technologies and Factory Automation*, 1994.

[17] M.Román et al., "Gaia: A Middleware Infrastructure to Enable Active Spaces", In *IEEE Pervasive Computing*, pp. 74-83, Oct-Dec 2002.

[18] C.Hess, "The Design and Implementation of a Context-Aware File System for Ubiquitous Computing Applications", PhD Thesis, UIUC, 2003.

[19] W.Reisig, "Petri Nets : An Introduction", Springer-Verlag, New York, 1984.

[20] N.Dunlop et al., "Dynamic Conflict Detection in Policy-Based Management Systems", *EDOC '02*, 2002.

[21] A.Andrzejak et al., "FeedbackFlow - An Adaptive Workflow Generator for System Management", *ICAC 2005*.

[22] A.Ranganathan et al., "Pervasive Autonomic Computing Based on Planning", *ICAC 2004*.

[23] B.Srivastava et al., "The Case for Automated Planning in Autonomic Computing", *ICAC 2005*.

[24] M. Beigi et al. "Policy Transformation Techniques in Policy-based Systems Management", *Policy 2004,* USA, 2004.

[25] Ubisense – http://www.ubisense.net/