

© 2006 by Chetan Shiva Shankar. All rights reserved.

POLICY-BASED PERVASIVE SYSTEMS MANAGEMENT USING  
SPECIFICATION-ENHANCED RULES

BY

CHETAN SHIVA SHANKAR

B.E., Bangalore University, 2000  
M.S., University of Illinois at Urbana-Champaign, 2003

DISSERTATION

Submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy in Computer Science  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

# Abstract

Pervasive computing enables a new paradigm that integrates digital and physical entities into a unified programmable system. Physical entities such as electrical appliances can be augmented with sensors and actuators that interact with digital devices to facilitate remote interactions and control. These entities have spatial-awareness of their environments, which enables several new applications and services.

Numerous prototype pervasive systems have been successfully built and deployed in various controlled environments such as classrooms and computing labs. Lately, there has been some interest in commercial deployment of these systems in offices, health care, laboratories and assisted-living facilities. When they are deployed in commercial organizations they need to follow the guidelines set by the organization. These guidelines dictate quality-of-service guarantees that should be provided to users, stipulate hours of operations of the system, specify system configuration that should always be maintained, constrain application behavior and mandate service offerings.

Policy-based management is an approach in which organization guidelines can be expressed as policies that are enforced by a management system. These rules specify the corrective actions that should be executed in different situations and are designed using the Event-Condition-Action (ECA) rule framework. The events and conditions express the situation in which the corrective action should be executed.

A management policy evolves over time by addition and removal of rules, policy composition, rule modifications and due to various system dynamisms. These changes may result in conflicts and cycles among rules. Multiple rules can become simultaneously eligible for enforcement in a situation and the order of enforcement may determine the final system state. Rule enforcement may fail necessitating an exception model for policies. Rules may contain long-running actions, which may cause conflicts with rules that are enforced at a later situation. The dynamism of pervasive systems with frequently changing configurations further complicates policy design. In order to address these issues, complex static and dynamic reasoning techniques have to be supported by management systems. The ECA rule framework is poorly suited for designing management policies for pervasive systems since it does not contain any information about the rule action. A rule action is initiated on the specified situation and there is no information about the effect

of the action on the system or whether the action completed execution successfully. This information is vital for policy analysis and for providing various guarantees. The above problems lead to non-determinism, which makes the management process unpredictable.

In this thesis, we propose a rule framework called Event-Condition-Precondition-Action-Postcondition (ECPAP) that contains axiomatic specifications of rule actions, for designing management policies. These specifications formally state the effect of an action using Hoare logic as pre- and post-conditions. This framework facilitates advanced conflict and cycle analysis, determines enforcement order when multiple rules are simultaneously triggered, supports policy exception handling and provides reasoning support for rules with long-running actions. We show how the ECPAP framework enables deterministic policy-based management.

The mobility of devices and applications in a pervasive system complicates policy design. Rules have to be added or revoked when the composition of a system changes. Therefore, we propose an extension to the ECPAP framework based on roles. This approach simplifies policy management in pervasive systems.

We propose algorithms for static and dynamic analyses, enforcement verification and monitoring and reasoning with long-running actions. We demonstrate the need for these algorithms on various distributed and pervasive systems and evaluate their performance. Our experiments show that the ECPAP framework leads to effective policy-based management and is a feasible approach.

*To my parents, Asha and Prof. Shivashankar*

# Acknowledgments

I would like to profoundly thank my advisor, Prof. Roy Campbell, for his support and guidance throughout my graduate studies. He gave me the freedom to pursue my ideas and constantly encouraged me by refining my research direction through his excellent evaluations and feedbacks. I am forever indebted to him.

My thesis committee members - Prof. Gul Agha, Prof. Mehdi Harandi and Prof. Klara Nahrstedt were very supportive of my ideas and constantly motivated me with valuable suggestions. I am very grateful to them. Anda Ohlsson was extremely helpful throughout my stay at the University of Illinois. It was due to her time management abilities that I was able to regularly interact with my ever-busy advisor, attend conferences and complete many administrative tasks on time. I am sure no Systems Software Research Group thesis can ever be complete without acknowledging Anda. My sincere thanks to her.

My colleagues and friends in the Systems Software Research Group were extremely supportive throughout my doctoral days. It was due to the invaluable discussions with Anand and his constructive comments that I was able to narrow down to my research problem. My interactions with Jalal and his advice on using the Active Spaces infrastructure contributed immensely to the refinement of my research problem. My other colleagues - Prasad, Zahid, Manuel, Chris, Suvda, Geta, Apu, Dulcinea, Jeff Naisbitt, Francis, Ellick, Dongyun, Jeff Carlyle and Sundeep helped me at different phases of my doctoral research. I am extremely thankful to all of them.

My special thanks to Sandeep Uttamchandani who introduced me to the world of policy-based management. Discussions with him motivated me to pursue policy research for my doctoral work.

The folks at Hewlett Packard Laboratories - Vanish, Yuan, Subu, Dejan, Keith and Akhil require special mention. Regular interactions with them helped shape my thesis and opened up several new research directions. I am grateful to them for these discussions and providing me financial support during the last few months of my doctoral research. I am also thankful to National Science Foundation for funding my research work.

My long-time friends – Rohith and Harsha and my roommates - Narendra, Badri, and Hemant encouraged me and morally-supported me during my doctoral days. My sincere thanks to them.

I am very grateful to my brother-in-law, Sathya, who strongly advised and encouraged me to pursue my doctoral education and my sister, Chitra, for her moral support. Without their encouragement and support I would not have completed my doctoral work so comfortably. My wife, Sindhu, brought discipline to my student life during the last few days by making sure that I worked diligently on writing my dissertation. My special thanks to her for her support.

It is very hard to find words to express my gratitude to my parents, Asha and Prof. Shivashankar. They stood by me every step of the way and morally-supported me during the “crests” and “troughs” of my doctoral studies. It is unlikely that I would have completed my doctoral research without their unwavering support and encouragement. I dedicate this thesis to them.

# Table of Contents

<b>List of Figures</b> . . . . .	<b>xi</b>
<b>List of Abbreviations</b> . . . . .	<b>xiii</b>
<b>Chapter 1 Introduction</b> . . . . .	<b>1</b>
1.1 Problem . . . . .	3
1.2 Approach . . . . .	6
1.3 Thesis Contributions . . . . .	7
1.4 Roadmap . . . . .	8
<b>Chapter 2 Background</b> . . . . .	<b>9</b>
2.1 Policy-based Management . . . . .	9
2.2 Policy Classification . . . . .	10
2.3 Active Space Policy Model . . . . .	12
2.3.1 Event Reception Model . . . . .	12
2.3.2 Policy Evaluation . . . . .	13
2.4 Pervasive Computing . . . . .	14
2.4.1 Gaia and Active Spaces . . . . .	15
<b>Chapter 3 Problem Statement</b> . . . . .	<b>17</b>
3.1 Context . . . . .	17
3.1.1 Active Space Dynamism . . . . .	17
3.1.2 Multiple Administrators . . . . .	18
3.1.3 Erroneous System Components . . . . .	18
3.1.4 Long-running Actions . . . . .	18
3.2 Problem . . . . .	18
3.2.1 Policy Conflicts . . . . .	19
3.2.2 Policy Cycles . . . . .	19
3.2.3 Rule Enforcement Order . . . . .	20
3.2.4 Exception Model . . . . .	21
3.2.5 Long-running Actions . . . . .	22
3.2.6 Policy Design and Management . . . . .	22
3.3 Solution Space . . . . .	22
3.4 Thesis Statement . . . . .	23
3.5 Success Criteria . . . . .	24
<b>Chapter 4 Specification-enhanced Policy Framework</b> . . . . .	<b>25</b>
4.1 Requirements . . . . .	25
4.2 Policy Framework . . . . .	26
4.2.1 Syntax and Semantics . . . . .	26
4.2.2 Action Specifications . . . . .	28
4.2.3 ECPAP Examples . . . . .	29

4.3	Policy Enforcement Process . . . . .	30
4.4	Conclusion . . . . .	30
<b>Chapter 5</b>	<b>Conflict and Cycle Analysis . . . . .</b>	<b>33</b>
5.1	Policy Conflicts . . . . .	33
5.1.1	Conflicts Due to Action Effects . . . . .	34
5.2	Policy Conflict Detection . . . . .	35
5.2.1	Static Detection . . . . .	37
5.2.2	Dynamic Detection . . . . .	38
5.2.3	Conflict Analysis with the $\sqcup$ Operator . . . . .	39
5.2.4	Conflict Resolution using Resolution Policies . . . . .	39
5.3	Policy Cycles . . . . .	41
5.4	Evaluation . . . . .	43
5.5	Conclusion . . . . .	45
<b>Chapter 6</b>	<b>Ordering Rule Enforcement . . . . .</b>	<b>46</b>
6.1	The Rule Ordering Problem . . . . .	46
6.2	Analyzing Action Dependencies . . . . .	47
6.3	Enforcement Semantics . . . . .	54
6.4	Petri net Workflow Execution . . . . .	55
6.5	Evaluation . . . . .	56
6.6	Conclusion . . . . .	58
<b>Chapter 7</b>	<b>Policies with Long-running Actions . . . . .</b>	<b>60</b>
7.1	Long-running Actions . . . . .	60
7.2	Temporal Logic Description of Long-running Actions . . . . .	61
7.3	Conflict Analysis . . . . .	62
7.4	State Model . . . . .	63
7.4.1	Model Updation . . . . .	63
7.4.2	Garbage Collection . . . . .	63
7.5	Evaluation . . . . .	63
7.6	Conclusion . . . . .	64
<b>Chapter 8</b>	<b>Enforcement Monitoring and Policy Exception Model . . . . .</b>	<b>65</b>
8.1	Policy Errors . . . . .	65
8.2	Policy Exception Model . . . . .	66
8.3	Verification System . . . . .	67
8.4	Monitoring System . . . . .	68
8.5	Exception Generation System . . . . .	69
8.6	Handling Exceptions with Ordered Rule Enforcement . . . . .	69
8.6.1	Workflow Execution . . . . .	70
8.7	Conclusion . . . . .	71
<b>Chapter 9</b>	<b>Towards deterministic policy-based management . . . . .</b>	<b>75</b>
9.1	Non-determinism of the ECA framework . . . . .	75
9.2	Formal Proof of Determinism . . . . .	75
9.3	Discussion . . . . .	79
<b>Chapter 10</b>	<b>Role-based Management . . . . .</b>	<b>81</b>
10.1	Complexity of Policy Design . . . . .	81
10.2	Role-based Design . . . . .	82
10.3	Role-based ECA Rules . . . . .	83
10.4	RBM Models . . . . .	84
10.4.1	Role Hierarchies . . . . .	85

10.4.2 Role Constraints . . . . .	85
10.5 Role Manager . . . . .	87
10.6 Policy Analysis . . . . .	87
10.6.1 Conflict Analysis . . . . .	88
10.6.2 Cycle Analysis . . . . .	88
10.7 Conclusion . . . . .	88
<b>Chapter 11 Architecture and Implementation . . . . .</b>	<b>89</b>
11.1 Architecture . . . . .	89
11.2 Policy Compiler . . . . .	90
11.3 Policy Enforcement System . . . . .	91
11.4 System Evaluation . . . . .	95
11.5 Case Studies . . . . .	96
11.5.1 Monitoring System Management on PlanetLab . . . . .	96
11.5.2 Cluster File Server Management . . . . .	100
11.6 Conclusion . . . . .	103
<b>Chapter 12 Related Work . . . . .</b>	<b>105</b>
12.1 Static Policy Analysis . . . . .	105
12.2 Dynamic Analysis . . . . .	106
12.3 Policy Verification/Monitoring . . . . .	106
12.4 Model-based Management . . . . .	107
12.5 Role-based Management . . . . .	107
12.6 Active Database Rules . . . . .	108
12.7 Other Projects . . . . .	108
<b>Chapter 13 Future Work . . . . .</b>	<b>110</b>
13.1 Policy Profiling and Debugging . . . . .	110
13.2 Policies with Composed Events . . . . .	110
13.3 Policies and Models . . . . .	111
13.4 Incorrect Specifications . . . . .	111
13.5 Behavioral Specifications . . . . .	111
13.6 Policy Interpretation . . . . .	112
13.7 Constraint Language for Role-based Management . . . . .	112
13.8 Empirical Validation of Approaches . . . . .	112
<b>Chapter 14 Conclusion . . . . .</b>	<b>113</b>
14.1 Summary . . . . .	113
14.2 Contributions . . . . .	114
<b>Appendix I Policy Language Grammar . . . . .</b>	<b>115</b>
<b>References . . . . .</b>	<b>123</b>
<b>Author's Biography . . . . .</b>	<b>128</b>

# List of Figures

2.1	Policy Classification . . . . .	10
2.2	Event Reception Model . . . . .	14
2.3	Gaia Architecture . . . . .	15
3.1	Conflict due to Action Effects . . . . .	19
3.2	Policy with a Cycle . . . . .	20
3.3	Policy to Restart Hibernated Active Space . . . . .	20
4.1	Actor modeling of rule evaluation . . . . .	28
4.2	Flowchart of Policy Enforcement . . . . .	31
5.1	Policy with Conflicting Actions . . . . .	33
5.2	Policy with Conflicting Action Effects . . . . .	34
5.3	Policy Rules in the ECPAP Framework . . . . .	35
5.4	Static Conflict Detection Algorithm . . . . .	38
5.5	Dynamic Conflict Detection Algorithm . . . . .	39
5.6	Dynamic Conflict Resolution Algorithm . . . . .	41
5.7	A Policy containing a Cycle . . . . .	42
5.8	Trigger Graph . . . . .	42
5.9	Trigger Graph Construction Algorithm . . . . .	43
5.10	Cycle Detection Algorithm . . . . .	44
5.11	Conflict and Cycle Analysis Overhead . . . . .	45
6.1	Policy to demonstrate the need for Rule Ordering . . . . .	47
6.2	Policy in the ECPAP Framework . . . . .	48
6.3	Petri net Workflow for Triggered Rules . . . . .	49
6.4	Trivially-enabled Action Analysis . . . . .	49
6.5	Enablement Analysis . . . . .	50
6.6	Partial-sets Determination . . . . .	51
6.7	Petri net Workflow Construction . . . . .	52
6.8	Workflow Execution Algorithm . . . . .	57
6.9	Petri net Workflow Generation Overhead . . . . .	58
7.1	Policy with Conflicting Long-running Actions . . . . .	62
7.2	Dynamic Conflict Detection with Persistent State Predicates . . . . .	63
7.3	State Model Response Times . . . . .	64
8.1	Policy with Exception Rules . . . . .	67
8.2	Verification Algorithm . . . . .	68
8.3	Predicates to Exceptions Mapping Interface . . . . .	70
8.4	Petri net Workflow for Triggered Rules . . . . .	70
8.5	Workflow Execution Algorithm . . . . .	73

8.6	Token Passing Sequence . . . . .	74
8.7	Reconstructed Workflow . . . . .	74
9.1	Policy with Sink Rules . . . . .	79
10.1	Typical Active Space Rule Templates and Instantiated Rules . . . . .	84
10.2	Role-based Management Models . . . . .	85
10.3	Partial Active Space Role Hierarchy . . . . .	86
11.1	Management System Architecture . . . . .	90
11.2	Policy Compilation . . . . .	91
11.3	Policy Design Tools . . . . .	92
11.4	Rule Evaluation Algorithm . . . . .	92
11.5	Petri net Workflow Data Structures . . . . .	93
11.6	Role Manager Interface . . . . .	95
11.7	Rule Evaluation Overhead . . . . .	96
11.8	Ganglia Management Policy . . . . .	98
11.9	Petri net Workflow . . . . .	98
11.10	Ganglia Monitoring System Configuration . . . . .	99
11.11	Cluster File Server Configuration . . . . .	100
11.12	Cluster File Server Management Scripts . . . . .	101

# List of Abbreviations

ANTLR	Another Tool for Language Recognition
BIPN	Boolean Interpreted Petri net
CORBA	Common Object Request Broker Architecture
ECA	Event-Condition-Action
ECPAP	Event-Condition-Precondition-Action-Postcondition
PDL	Policy Description Language
PMAC	Policy Management for Autonomic Computing
RBAC	Role-based Access Control
RBM	Role-based Management

# Chapter 1

## Introduction

Technological advances, over the last decade, in computing, communications and devices have ushered in an era where large numbers of heterogeneous mobile and immobile devices can be combined to form a programmable system. These systems, called *pervasive systems*, blend digital and physical infrastructures and enable “smart” homes, “aware” offices [KOA<sup>+</sup>99], sentient spaces [ACH<sup>+</sup>01] and other responsive environments [PJKF03, RHC<sup>+</sup>02]. The goal of pervasive computing is to create a programmable system from everyday digital and physical devices such as desktop computers, handheld devices, sensors, actuators and electrical appliances. These systems create infrastructures that simplify usage of such devices and facilitate remote interactions with them. These systems can be used in offices, classrooms, laboratories and various other venues to aid users in performing everyday tasks.

The vision of pervasive computing is to create a user-centric non-intrusive system that aids the user with little or no distraction [Wei94]. Users should spend more time using the system than configuring it. Pervasive systems consist of diverse computing and communication devices. These devices communicate through a myriad of technologies such as WiFi, Bluetooth, Infrared, wired networks and so on. This multitude of technologies necessitates a software infrastructure that integrates these devices with minimal intervention. The infrastructure should also manage the lifetime of this device-ensemble and provide services that unify the resources of various entities of the system. Middleware-based technologies have been very successful in enabling a pervasive computing system. Several research projects have developed prototype pervasive systems and deployed them in various controlled environments such as classrooms [Abo99], laboratories [GDL<sup>+</sup>04] and homes [KOA<sup>+</sup>99].

The Gaia project [RHC<sup>+</sup>02] takes an operating system approach to create a pervasive system, called *Active Spaces*, by identifying and separating essential pervasive services from applications. An active space is a physically-bounded collection of devices, applications and services [RSC04]. Typical examples of active spaces include “smart” rooms, interactive meeting rooms and collaborative classrooms. The active space operating system has a services layer containing various services for discovery, naming, eventing and location-awareness that can be used by any application running in the active space. These services provide support

for discovering new devices and services, integrating services of mobile devices with that of the active space, migrating applications and data across devices and for various other functionalities.

The success of many pervasive system architectures has encouraged their deployments in many commercial settings. *Aware-home* systems have been deployed in homes [KOA<sup>+</sup>99], *IBM Everywhere display systems* have been deployed in shopping malls [PPK<sup>+</sup>03] and *Ubisense location systems* have been deployed in offices [SG05]. When pervasive systems are deployed in commercial organizations, they need to follow the guidelines set by the organizations. Organizations may stipulate hours of operation of the system, the quality of service to be provided to different users, protocols for authorization, policies for data and application management and so on.

Traditionally, organization guidelines are expressed as administrative tasks that are enforced by system scripts. These scripts are executed on timers, signals or other notifications. For example, data backup on servers every night is a good example of a timer-based administrative task. Initiating virus checking when a virus signature is detected in an executable file is an example of a notification-based task. These management activities are typically designed in an ad-hoc manner by various administrators. Each administrator designs management scripts for specific aspects of the system such as file systems, memory, process and so on. As a result, scripts may conflict with each other; they may fail and leave the system in an undesired state or their concurrent executions may adversely affect the functioning of various applications on the system. Script-based management systems provide no means for reasoning about concurrent administrative activities.

Policy-based management provides a framework for specification and reasoning about concurrent management activities and is a natural extension of script-based management. Many system-administration activities can be viewed as reactions to system situations by execution of corrective actions. For example, initiation of virus checking process on a system is a reaction to a *virus – sense* situation. Policy-based management systems view administrative tasks as sets of situation-action pairs that are specified as rules. System administrators encode organization guidelines as sets of rules, called *policies*, and these policies are enforced by a management system. Rules can be analyzed to determine conflicts, cycles and various other undesired behaviors.

Most management policy rules are designed using the Event-Condition-Action (ECA) rule framework [LBN99, SRC05, Slo94]. This framework enables specification of situations as Event-Conditions. When an event occurs, the management system enforcing the policy rules receives the event and determines which rules in the policy needs to be enforced. It verifies if the conditions of those rules are satisfied and if so, executes the associated action. A typical ECA rule is shown below:

```
on(NodeFail(Node n))
if(n.type == "computeNode")
do(startnewFailOverNode());
```

This rule is enforced when a *NodeFail* event is observed and if the failed node is of type *computeNode*. The action of the rule starts a new failover node from a pool of nodes.

A management policy consists of several ECA rules. Two or more rules may be based on the same event-conditions and therefore, can get triggered on the same situation. These rules may conflict with each other confusing the management system. Action of a rule may trigger another rule, whose action may trigger the first rule through a chain of rules causing a non-terminating cycle of rule enforcements. Even when rules do not conflict, order of enforcement of multiple rules may determine the final system state. Typically, when multiple rules are simultaneously triggered, existing policy systems enforce rules in a random order. Therefore, no guarantees are provided regarding the final system state. Rule actions may fail and leave the managed system in an invalid state. Rules can have long-running actions that can cause problems with rules that are triggered on events in a future situation. These issues result in non-deterministic system management.

In this thesis, we argue that existing ECA rule framework is poorly suited to address the above problems and should be extended with formal specifications of actions. We propose a new rule framework called Event-Condition-Precondition-Action-Postcondition (ECPAP) for designing management rules. This framework combines axiomatic specifications of actions with management rules and enables complex static and dynamic reasoning. We show how the ECPAP framework can be used for conflict analysis, cycle detection, determining enforcement order when multiple rules are triggered, designing exception models for policies and for reasoning about rules with long-running actions. We formally prove that the ECPAP framework enables deterministic policy-based management.

## 1.1 Problem

In this section, we detail the problem that was briefly discussed in the last section. Most policy-based management systems use the event-condition-action framework for rule specification. Application of the framework to design active space policies leads to numerous problems. Though the problems are not specific to active spaces, the dynamism and heterogeneity of an active space exacerbates them. Some of these problems have been noticed in policies designed for other distributed systems such as Grids and computational

clusters.

### **Policy Rule Conflicts**

A rule conflict occurs when two or more rules may need to be simultaneously enforced in the system but their actions conflict with each other. Typical examples of conflicting rules include rules with actions to turn on and off a device, to increase and decrease service quality simultaneously, to migrate a process to a device while simultaneously stopping the process and so on. These conflicts can be detected by the action constraint model [CLN00] that captures conflicting actions as a conjunction of predicates. The set of actions that are in conflict are stated explicitly by the system administrator. If two or more rules with those actions are triggered, the system detects a conflict.

While this model can detect conflicting actions that are stated explicitly, the model fails to detect conflicts due to side-effects of actions. For example, a rule that migrates a process to a target device conflicts with a rule that shuts down the device, since a device that is not running cannot host the migrated process. Similarly, a rule that starts an application conflicts with a rule that hibernates the host device. These conflicts are not normally identified by an administrator and therefore, are not explicitly specified. Rather, they occur due to side-effects of the original rule actions. In the first example, the inability of the stopped device to receive a migrating process is a side-effect of the shutdown process. In order to detect such conflicts, the effects of the actions on the system should be stated explicitly.

### **Policy Cycles**

Policies may have a set of rules that trigger each other continuously in sequence. Action of a rule may generate events that trigger another rule whose action may trigger the first rule through a sequence of zero or more intermediate rules. This leads to a non-terminating sequence of rule enforcements causing a policy cycle. Though some cycles may lead to desired system behavior, most cycles may not, as they cause oscillations in system states. Therefore, cycles should be detected and reported to the policy designer. In order to detect cycles, the set of events generated by an action execution must be known. The ECA framework contains no such information and therefore, provides no means to detect policy cycles.

### **Rule Enforcement Order**

In certain situations, multiple non-conflicting rules may need to be enforced and the order of enforcement of these rules can determine the final system state. For example, in our active space system, when a device is brought into the space, the location system generates an *ObjectEnter* event that triggers two rules. The

first rule authenticates the device with the space and the second rule mounts the device file system onto the active space file system. The mounting operation succeeds only after the device has authenticated itself with the space. If the first rule is enforced before the second rule, the file system is successfully mounted. But if the enforcement order is reversed the mounting operation fails. Therefore, the policy system should define semantics of enforcement when multiple rules are simultaneously triggered. Existing policy-based management systems do not reason about concurrent rule enforcements and define no enforcement ordering.

### **Policy Exceptions**

Actions of rules may fail due to policy errors, wrong parameters or various other system conditions. Existing policy-based systems do not verify action execution and assume that rule enforcement was successful. The success of rule enforcement should be verified and corrective actions should be taken on rule failure.

A policy defines an abstraction level by capturing situation-action pairs as rules. Errors in policy enforcement should be addressed at this abstraction level by the policy designer and therefore, an exception model for policies is required. For example, if a rule to migrate an application to a target device fails, there should be mechanisms in the policy system to detect the situation and handle the failure.

### **Long-running Actions**

Some rules may contain actions that persist through out the running time of the system or for significantly long periods of time such that they interfere with rules that are enforced in future situations. Such rules include rules that initiate heartbeat messages on devices for fault detection, force entities to be in certain states and maintain certain QoS. When these rules are enforced, the actions of these rules can cause conflicts with rules that are triggered on a different event in future. For example, a rule that initiates heartbeats on a device conflicts with a rule that turns off the device in a future situation since a device cannot send out heartbeats in the *off* state. In order to detect such conflicts formal specification of action behavior is required and therefore, the ECA framework is poorly suited.

Rules with long-running actions may need to be monitored through out their enforcement duration. In our active space, heartbeat messages are sent by devices periodically to indicate device membership. If heartbeat messages stop, the device is considered to be no longer part of the active space. When the messages stop, the rule that initiated the heartbeats should be considered to have failed and corrective actions should be taken. Therefore, rules with long-running actions should be monitored throughout their enforcement duration.

## Complexity of policy design due to active space dynamism

An active space is a dynamic system with devices entering and exiting the space, applications migrating across devices and system configuration changing frequently. This complicates policy maintenance since rules should be added and removed quite frequently. Each time a policy is modified, it has to be analyzed for conflicts and cycles. When devices exit a space, rules should be removed to avoid unnecessary enforcement failures due to absence of devices. These issues cause significant overhead to manage policies.

## 1.2 Approach

In order to address the above problems, rules require information about the effects of actions. The ECA framework does not contain any such information and therefore, fails to address the above problems. In this thesis, we propose an extended version of ECA framework, called Event-Condition-Precondition-Action-Postcondition (ECPAP) for designing policy rules. This framework combines axiomatic and behavioral specifications of actions with ECA rules as pre- and post-conditions. The pre-condition represents the state the system should be in before the action can be executed and the post-condition represents the system state once the action successfully completes execution.

The extra knowledge about action effects provided by the ECPAP framework enables analyzing conflicts due to side-effects of actions. The post-conditions of actions describe the effect of the action on the system and therefore, the system can analyze if two actions lead to conflicting post-conditions.

The post-condition of an action contains the set of events that are generated during action execution. This information enables the system to determine if there is a set of rules that trigger each other in a cycle. This facilitates policy analysis to determine termination of rule enforcement.

When multiple rules are simultaneously triggered, the action specifications can be used to build a dependency graph of triggered rule actions. Rules are enforced based on dependencies. This enables the management system to determine dependent and confluent rules and provide enforcement guarantees.

The post-condition of an action can be used to determine if a rule action successfully executed. If the post-condition is not satisfied, the rule is assumed to have failed and corrective actions can be taken. This facilitates an exception model for policies, which was unavailable with the ECA framework.

The post-conditions can also contain behavioral specification of long-running actions in temporal logic. These specifications facilitate reasoning about long-running action conflicts and monitoring their execution.

Finally, the ECPAP framework can be combined with role-based management approaches to simplify policy design for active spaces. Policy rules are designed with roles and an entity, such as a device or an

application, that is assigned to a role should enforce the rule associated with the role. This simplifies policy design and analysis in a dynamic system such as active spaces.

### 1.3 Thesis Contributions

This research work has developed and implemented the proposed framework and demonstrated its applicability and usefulness in active spaces and two other distributed systems. The specific contributions of this thesis are listed below:

**Specification-enhanced rule framework:** This framework combines action specifications with ECA rules. Action specifications can contain system states, generated events and some behavioral information.

**Static and dynamic policy analysis techniques:** This thesis has developed various analysis techniques for conflict analysis, termination detection and dependency analysis. These techniques enable the system to provide better enforcement guarantees.

**Definition of enforcement semantics for policies:** This thesis also introduces the notion of enforcement semantics, which defines the order of enforcement of rules when multiple non-conflicting rules are simultaneously triggered.

**Exception model for policies:** This thesis presents an exception model for policies. The model can be used to detect rule enforcement failures and take corrective actions.

**Support for rules with long-running actions:** The thesis shows how the ECPAP framework can be used to analyze rules with long-running actions. This approach uses state models to store system states such as heartbeats and entity states.

**Role-based management framework:** The thesis demonstrates how role-based approaches to designing policies can reduce the complexity of managing policies in dynamic environments such as active spaces. It introduces role-based management models similar to RBAC [SCFY96] and introduces analysis techniques for policies with roles.

**Deterministic policy-based management:** The ECPAP framework enables deterministic policy-based management. This thesis presents a formal proof of determinism.

**Implementation and evaluation:** Finally, the thesis provides implementation and evaluation details of the framework in our active space pervasive system. It also describes our experiences in managing the Ganglia monitoring system [MCC] on PlanetLab clusters [PCAR02] and designing management policies for a cluster file server using the new framework.

## 1.4 Roadmap

In chapter 2, we discuss policy-based approaches to system management and introduce the active spaces pervasive system. We present the problem addressed in this thesis in detail in chapter 3 and discuss our approach. Chapter 4 presents the ECPAP framework with a discussion on the policy enforcement process. Chapter 5 details conflict and cycle analyses using the ECPAP framework. We describe the problem of rule ordering in chapter 6 and present our solution. Chapter 7 demonstrates the ability of the ECPAP framework to support long-running actions. We present our enforcement monitoring techniques and exception handling model for policies in chapter 8. We formally prove that the ECPAP framework enables deterministic management in chapter 9. We extend the ECPAP framework with roles and show how it simplifies policy design and management in dynamic environments in chapter 10. We describe the architecture and implementation details of our system and present case studies on using the system on two other distributed systems in chapter 11. We compare our work to other research efforts on policies in chapter 12 and present possible extensions to our work in chapter 13. We finally conclude in chapter 14 with a brief summary of our work.

# Chapter 2

## Background

In this chapter, we set the stage to explain this research work by describing policy-based approach to systems management and our active spaces pervasive system, which was used as a test bed system to validate the results of this work. In section 2.1, we discuss policy-based management of systems with a brief description of different functional areas. We classify policies based on objectives and frameworks in section 2.2. We present the active spaces policy model in section 2.3 and conclude with a discussion of pervasive systems and active spaces in section 2.4.

### 2.1 Policy-based Management

Policy-based management has been successfully used for well over a decade as an administrative interface for managing various systems such as network switches [BLK00], content distribution networks [VCA02] and distributed systems [Slo94]. Policies provide a convenient means of guiding the behavior of a system using a set of rules. These rules dictate the corrective actions that should be taken in different circumstances; set limits and constraints to states reached by various system components; prioritize one admin task over another and define access rights of entities. In this thesis, we restrict our focus to obligation rules, which are rules that are enforced in reaction to certain situations and are specified using reaction rule framework.

Many network components and several distributed systems have used policies to manage system configurations, faults, performance and security. The Open Systems Interconnection (OSI) Reference model has identified five functional areas for systems management (ISO/IEC 7498-4, 1989) [ISO89]. These functional areas, commonly associated with the acronym FCAPS, are as follows [Lup98]:

**Fault Management** includes generation, correlation and distribution of alarms, diagnosis of failures and corrective actions.

**Configuration Management** includes configurations of network and computational resources, software resources and components of distributed applications.

**Accounting Management** includes issues with billing, logging and modeling of accountable resources.

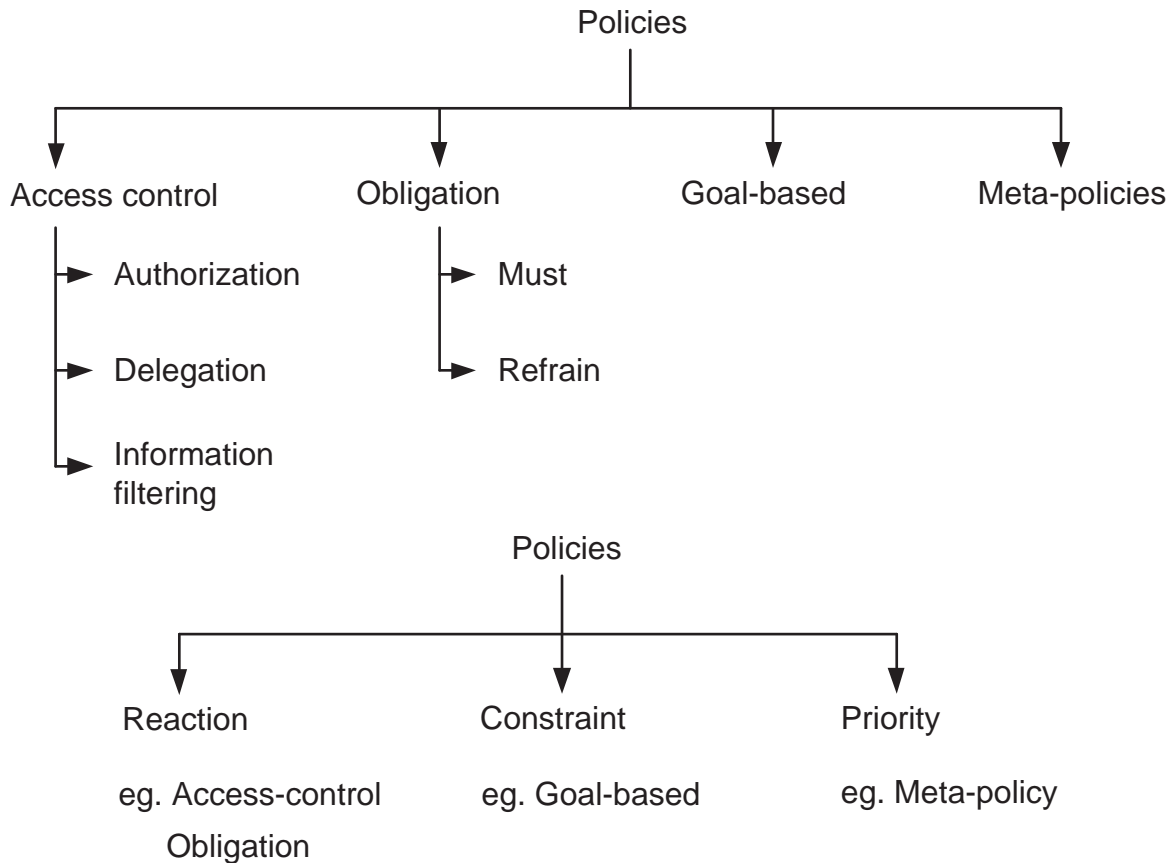


Figure 2.1: Policy Classification

**Performance Management** is concerned with performance of the system and associated actions such as monitoring, load-balancing to maintain quality of service and capacity planning.

**Security Management** covers security of the system, authorization, authentication, access control and privacy issues.

The scope of our thesis is restricted to fault and configuration management in active spaces. Obligation rules can be also used for accounting and performance management though we do not address them in this thesis. Security management in active spaces has been addressed in prior research works [AM05, Sam05].

## 2.2 Policy Classification

Policies can be broadly classified into access-control, obligation, goal-based and meta policies based on their purpose. They can also be classified into reaction, constraint and priority policies based on the rule framework used. Figure 2.1 illustrates the two classifications.

Access control policies are concerned with limiting the activities of legitimate users who have been successfully authenticated [M.D93, SS97]. These policies specify what actions entities *can* or *cannot* perform in a system [Slo94]. Access control policies can be further classified into authorization, delegation and information filtering policies. Authorization policies define what activities a member of the subject domain can perform on the set of objects in the target domain [Dam02]. Delegation policies transfer access rights from one entity to another. This transfer can either be temporary or permanent. Information filtering policies transform data and are normally used to implement privacy by data obfuscation. For example, the location information of a person can be reported with lesser accuracy to prevent the exact position from being revealed using information filtering policies.

Obligation policies are concerned with enforcing specified activities in a system in specific circumstances. These policies specify what actions entities *must* or *must not* perform in a system [Slo94]. Typically obligation policies are used for fault and configuration management, file system management, executing maintenance jobs such as backing up and checkpointing and so on. These policies can be classified into *must* policies and *refrain* policies. Must policies dictate the actions that must be enforced by the system while refrain policies specify the actions that must not be enforced. Refrain policies are normally implemented as negative authorization policies by restricting access to those actions.

Goal-based policies are a declarative means of specifying the desired system behavior. These policies specify the final system state that should be reached from a given state. Planning-based techniques are employed to determine the set of actions that need to be executed for the state transition [AHS05, RC04]. Meta-policies are policies that guide the behavior of the management system. These policies are used to resolve conflicts, modify policies at runtime and change various parameters of the management system.

As mentioned above, policies can also be classified based on the rule framework used. Reaction policies are specified using some form of the event-condition-action (ECA) framework. These policies facilitate expression of situation-action pairs and provide an imperative means to specify the behavior of the system. Access-control and obligation policies are examples of reaction policies.

Constraint policies restrict the states that entities can attain. These policies are used to specify performance bounds, desired system states in different situations and in role-based management for constraint-based inheritance [SCFY96]. Goal-based policies are examples of constraint policies. Constraint policies are normally specified as a conjunction of predicates.

Priority policies are used to choose an approach in a range of possible approaches. These include policies for choosing a rule among a set of conflicting rules and ordering enforcement of multiple triggered rules based on some metrics. Priority policies do not use explicit rule frameworks and have been specified in

different ways in different projects. IBM PMAC priority policies assign priority values to rules that are used for conflict resolution [ACG<sup>+</sup>05]. The Rei language uses priority policies to define priority relationships between rules [Kag04]. We use implications as priority rules for conflict resolution [SC05]. Some meta-policies are examples of priority policies.

## 2.3 Active Space Policy Model

The active space management system uses the policy model and language presented in [CLN00, LBN99]. The language, called Policy Description Language (PDL), uses the ECA rule framework for rule specification. We briefly describe the syntax of the language to aid in the explanation of the model.

A policy is a set of event-condition-action rules of the form

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

This rule is read as: “If the *event* occurs in a situation where the *condition* is true, then the *action* is executed”.

PDL consists of three basic classes of symbols: primitive event symbols, action symbols and constant symbols. Formally, a policy is a finite collection of ECA rules, having event, condition and action 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 or 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  $a(t_1, \dots, t_m)$  where  $a$  is an action symbol of  $m$  arguments and each  $t_i$  is a variable that appears in the event or condition parts of the rule or a constant. The condition part of an ECA rule is an expression of the form  $p_1 \ \&\& \ p_2 \ \&\& \ \dots \ \&\& \ p_k$  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  $\theta$  is a relational operator.

A finite set of event instances is termed as an *epoch*. The notations  $event(r)$ ,  $condition(r)$  and  $action(r)$  are used to denote the event, condition and action parts of a rule  $r$ , respectively.

### 2.3.1 Event Reception Model

Our policy framework views a situation as a set of correlated events and evaluates the policy based on the events in the set. Event correlation is a well-researched problem and numerous models have been proposed to group events corresponding to a change [NYGS, KYY<sup>+</sup>95, HPO]. Since the focus of our work is on policy evaluation and enforcement, we use a simple event correlation model based on epochs proposed by Chomicki et al. [CLN00] for policy evaluation. In this model, the event reception time axis is divided into discrete

intervals. All events received in an interval, by the management system, are assumed to be correlated and correspond to the same situation. In figure 2.2(a) events  $e_1$ ,  $e_2$  and  $e_3$  are assumed to be correlated. This model is useful in systems with little dependencies between events.

Assume a time interval,  $\delta$ .  $C_i : (e_1, \dots, e_n)$  represents the sequence of events,  $e_1, \dots, e_n$ , received by the event reception system corresponding to change  $C_i$ .  $time(e)$  represents the time at which event  $e$  was received by the reception system. The model can be described formally as:

$$C_1 : (e_1, \dots, e_n) \Rightarrow time(e_n) - time(e_1) < \delta$$

In addition, we considered a variant of the epoch model that groups events based on “gaps” in the event reception time axis. A set of events is considered to be correlated, if no event is received for a minimum duration ( $\delta$ ) after the last event in the set was received.

$$C_1 : (e_{11}, \dots, e_{1n}) < C_2 : (e_{21}, \dots, e_{2m}) \Rightarrow time(e_{21}) - time(e_{1n}) \geq \delta$$

$$C : (e_1, \dots, e_k) \Rightarrow \forall e_i e_{i+1}, time(e_{i+1}) - time(e_i) < \delta$$

where  $C_1 < C_2$  denotes that change  $C_1$  occurs before change  $C_2$ .

The model is illustrated in figure 2.2(b). This model assumes that events occurring due to a change exhibit bursty nature – all events corresponding to a change are received in one burst and a minimum delay exists between events corresponding to two changes.

Our policy evaluation system accepts a set of events as input and therefore, the evaluation of the policy is unaffected by the event correlation model used. The epoch model can be replaced by more appropriate correlation models without affecting policy evaluation.

### 2.3.2 Policy Evaluation

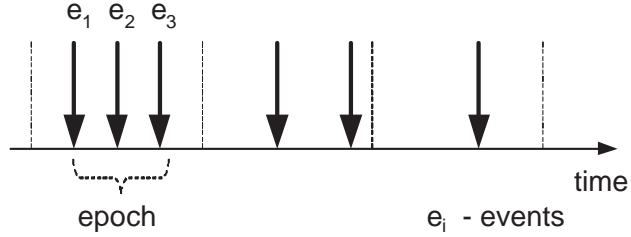
The semantics of each rule  $(e, c, a)$  is defined as

$$occ(e) \wedge c \rightarrow exec(a)$$

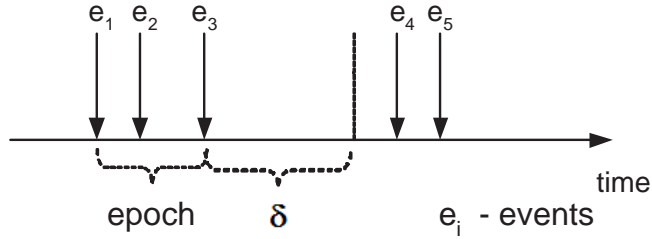
where  $occ(e)$  is *true* when event  $e$  is received by the evaluation system.  $exec(a)$  is the initiation of the execution of action  $a$ .

Evaluation of a policy,  $P$ , is a mapping from a set of events,  $E$  and conditions,  $C$  to a set of actions,  $A$ :  
 $eval(E, C) \rightarrow_P A$

**Definition 2.1:** A policy rule,  $r$ , is said to be *triggered* in an epoch  $E$ , if  $event(r) \in E$  and  $condition(r)$  is true.



(a)



(b)

Figure 2.2: Event Reception Model

## 2.4 Pervasive Computing

Pervasive computing extends traditional distributed and mobile computing with knowledge of the deployed physical environment. This knowledge enhances the power of this new computing paradigm by facilitating better resource management, introducing more applications and enabling better integration of the physical and digital worlds. Information about available devices in the vicinity of a mobile device enables the latter to better optimize communication. Physical devices such as door locks, lights and various other appliances can communicate with digital devices to assist users in performing various everyday tasks.

Numerous prototypes for pervasive computing have been developed in industry and academia. Some projects have taken an operating system approach by identifying essential services and creating complex architectures [RHC<sup>+</sup>02, PJKF03] while others have chosen ad-hoc approaches [KSK06, RBKI04]. Gaia [RHC<sup>+</sup>02] and iROS [PJKF03] view a physical space of devices as a pervasive system and have operating systems to program them. The Pervasive Information Communities Organization (PICO) [KSK06] project supports architectures for integrating ad-hoc device clusters for just-in-time communication and proactive collaboration. Ravi et al. [RBKI04] use ad-hoc Java based architectures for integrating devices and supporting pervasive applications.

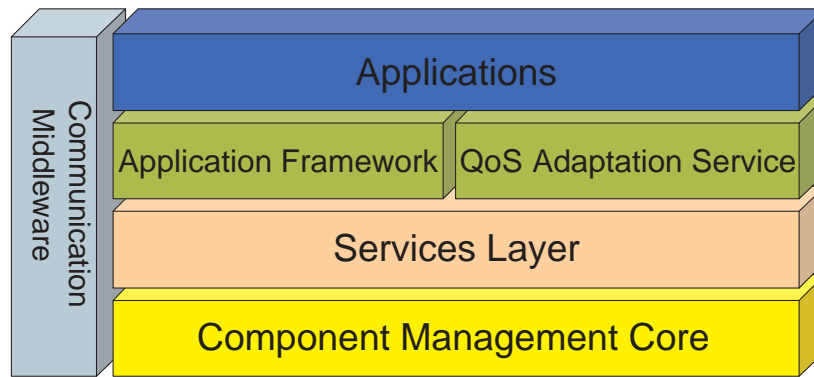


Figure 2.3: Gaia Architecture

With the success of many architectures, there is a move towards commercialization of these systems. IBM everywhere displays have already been deployed in airports and retail environments[PPK<sup>+</sup>03]. Ubisense location systems are being deployed in offices. When pervasive systems are deployed, they require management systems to enforce organization guidelines. Lately, there have been some efforts in developing policy-based management systems for pervasive systems [KFJ03]. But existing policy-based systems suffer from several limitations that we have detailed in the next chapter. These issues have to be addressed before these systems can be deployed in commercial settings.

### 2.4.1 Gaia and Active Spaces

The vision of the Active Spaces project is to create a programmable system from a collection of heterogeneous mobile and immobile devices. An active space provides frameworks to program the system, add new services dynamically and create multi-device applications. It can be deployed in classrooms, offices, homes, healthcare and various other assistive environments.

An active space is enabled by a distributed operating system called Gaia. Gaia consists of a collection of distributed components each representing a device, service or an application. Various services are provided for fault tolerance, location awareness, security and event handling for use by pervasive applications. Figure 2.3 illustrates the architecture of Gaia.

An application framework based on the model-view-controller framework [KP88] enables applications to be distributed across multiple heterogeneous devices [RC]. The various components of Gaia communicate through CORBA remote procedure calls. We discuss some of the services of Gaia that are used by the management system.

**Event Service :** The Gaia event service builds on the CORBA event service, which is a publish-subscribe based event communication system [HV99]. The CORBA event service supports both push and pull based delivery. It does not provide support for parameterized events and any information associated with an event has to be transferred as the payload of the event.

The Gaia event service extends the CORBA event service by providing mechanisms to store event logs in a repository called persistent repository [Bor02] and supporting parameterized events, which are events containing typed parameters. Parameterized events are useful to transfer additional information about an event. For example, *ObjectEnter(Device d)* is a parameterized event where *ObjectEnter* is the event name and *d* is the event parameter of type *Device*.

**Presence Service :** The Gaia presence service is used to detect membership of an entity, such as device, service or an application, in the active space. Each entity sends out heartbeat messages at periodic intervals to announce its presence in the active space. This service is used to detect entity failures and device exits from an active space. Each such situation generates an event on a well-known event channel.

**Location Service :** The Gaia location service provides position information to mobile entities in an active space [RAS<sup>+</sup>04]. This service uses a spatial database to store spatial information of the environment and maps the location of an entity in the database. In our active spaces, the location of an entity can be detected by different means such as Ubisense [SG05], RFID tags, finger-print readers and so on. The location service supports sensor fusion techniques to combine information from different sensing technologies and arrives at an approximate location estimation of an object.

The location service also supports spatial triggers, which are events generated due to changes in the spatial database. Triggers can be programmed to be generated when objects enter or exit a region, objects are brought close to each other or when an object is in line-of-sight with another object. The spatial database is implemented using PostgreSQL [Sto87] that supports complex geometric functions.

# Chapter 3

## Problem Statement

This chapter presents the problems addressed by this thesis and briefly explains our approach to solving them. Section 3.1 describes the context in which this research was performed. We discuss the problem addressed by the thesis in detail in section 3.2 and present our solution in section 3.3. We state our thesis in section 3.4 and conclude with a listing of the success criteria for this work.

### 3.1 Context

Complex systems such as pervasive systems require administrative interfaces to guide their behaviors according to well-defined guidelines. While policy-based management has been successfully applied to manage various simple systems, the complexity and dynamism of active spaces poses several novel challenges. Though this thesis addresses these challenges in the context of active spaces, the challenges and the approach presented are applicable in the wider context of complex distributed systems. In this section, we identify various characteristics of active spaces that complicate policy-based management.

#### 3.1.1 Active Space Dynamism

An active space is a highly dynamic distributed system with mobile devices, users and applications entering and exiting a space. The frequent addition and removal of devices from the system changes the composition and configuration of the system, repeatedly. Each device has its own set of applications and services that are implicitly integrated with the active space when the device is added to the system. This dynamism complicates policy design that traditionally assumes static set of entities as found in network switches [BLK00], content distribution networks [VCA02] and distributed systems [Slo94].

Devices may have their own management policies guiding the behavior and these policies may interfere with those of the active space. In addition, the active space dynamism necessitates frequent policy rule additions, removals and modifications.

### **3.1.2 Multiple Administrators**

An active space may have multiple administrators managing different aspects of the space. Some administrators may be concerned with management of services such as file systems, devices, sensors and actuators while other administrators may focus on application management. In addition, active spaces may be organized into a hierarchical system of spaces [AMSRC04] with each set of spaces being managed by an administrator. Each administrator authors his or her own policy that is enforced by a common management system. When different policies are combined, they may interfere with each other causing various problems such as policy conflicts, cycles and dominance.

### **3.1.3 Erroneous System Components**

An active space consists of heterogeneous devices, services and applications from several different vendors. One of the primary goals of active spaces was to develop a system that can dynamically integrate diverse components with minimum user intervention [Wei93]. These components may not be rigorously tested and may be incompatible with other components of an active space, thus causing failure of management actions. This complicates the management process.

### **3.1.4 Long-running Actions**

Like many distributed systems, an active space contains actions that execute for significantly long periods of time or throughout the running time of the active space. For example, most services of an active space start when the space is booted and execute till the space is switched off. Many applications such as text-to-speech converters and speech recognizers execute continuously in the background in our active space. Some of these applications are initiated by rules on certain situations. Rules that are triggered in a different situation in future may cause conflicts with these long-running processes.

## **3.2 Problem**

In this section, we present the policy-related problems caused by the dynamism of complex distributed systems, in general and active spaces, in particular. We present several policies that demonstrate the issues faced by policies for complex systems.

### 3.2.1 Policy Conflicts

A policy conflict occurs when a situation triggers two or more rules whose actions cannot execute concurrently [CLN00]. For example, rule  $R_1$  may specify –“if a user enters an active space, turn active space lights on”. Rule  $R_2$  may state that “after 5pm, turn off all active space lights”. If a user enters an active space after 5pm, both rules are triggered.  $R_1$  tries to turn the space lights on, while  $R_2$  mandates that all space lights be off. This causes a conflict between the two rules.

Existing conflict detection techniques [CLN00, DIR02] detect conflicts based on conflicting actions. In the above example, the administrator specifies that the action to turn lights on and the action to turn lights off conflict. So when two rules containing these actions are simultaneously triggered, the rules are flagged as conflicting by the management system. While this approach can detect a significant number of conflicts, certain conflicts may arise from effects of actions, which may be hard to detect. For example, consider the policy shown in figure 3.1. Both rules  $R_3$  and  $R_4$  are triggered when a person enters an active space.  $R_3$  starts the authorization application while  $R_4$  reboots the active space, thus killing the authorization application. Shutting down the application is a side-effect of the reboot process. In order to detect such conflicts, information about the effects of the action must be known. Existing policy-based systems do not provide any techniques to determine such conflicts.

```
 $R_3$ :  on (ObjectEnter(Person p)) //if user enters an active space, start authorization application
      if (true)
      do(startApp(“AuthApp”));
 $R_4$ :  on (ObjectEnter(Person p)) //if user enters an active space, reboot space
      if (true)
      do (rebootSpace());
```

Figure 3.1: A Policy containing conflicts due to action effects

### 3.2.2 Policy Cycles

A policy cycle occurs when a set of rules trigger each other continuously [SC05]. Policy cycles may lead to a non-terminating enforcement of a set of rules. Figure 3.2 shows a policy with cycles. Rule  $R_5$  states that “when a person exits the space, suspend all applications”. This triggers rule  $R_6$  that starts the checkpoint application when any application is suspended. Starting an application triggers rule  $R_7$  that checks to see if the space is empty and if so suspends the application being started. In this policy, a cycle exists between rules  $R_6$  and  $R_7$ , if the space is empty.

Policy cycles can cause oscillations in system states. The cycle in the policy in figure 3.2 can cause the

```

R5:  on (ObjectExit(Person p)) //if user exits, suspend applications
      if (true)
      do(suspend_apps());

R6:  on (suspendApp(Application App))
      if (true)
      do (startApp("checkpoint_app", App)) ;

R7:  on (startApp(Application App))
      if(statusSpace() == "empty")
      do (suspendApp (App));

```

Figure 3.2: A Policy containing a cycle between rules  $R_6$  and  $R_7$

checkpoint application to be repeatedly started and suspended. This can consume valuable system resources and confuse users. Though some cycles may be normal behavior of a system, it would be useful if policy designers are notified of cycles as warnings.

### 3.2.3 Rule Enforcement Order

When multiple non-conflicting rules are triggered, the order of enforcement of the rules can determine the final system state. For example, consider the policy shown in figure 3.3.

```

R8:  on (ObjectEnter(Device d, Space s))
      if (statusSpace(s) == "stopped")
      do(restartSpace(s));

R9:  on (ObjectEnter (Device d, Space s))
      if (roleDevice(d) == "guest")
      do(authorizeDevice(d, s));

R10: on (ObjectEnter (Device d, Space s))
      if (deviceType(d) == "laptop" && roleDevice(d) == "guest")
      do (mountFileSystem(d, s));

R11: on (ObjectExit (Device d, Space s))
      if (deviceType(d) == "laptop")
      do(unmountFileSystem(d, s));

```

Figure 3.3: Policy to Restart Hibernated Active Space

Rule  $R_8$  restarts the active space (and its various services) if it has stopped when a device enters the space. Rule  $R_9$  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  $R_{10}$  mounts a laptop’s file system onto the active space file system [HC03] when the laptop is brought into the active space and  $R_{11}$  unmounts it when the laptop leaves the space. 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  $R_8$ ,  $R_9$  and  $R_{10}$ . The order of

enforcement of the rules determines the behavior of the space. A rule is said to be *enforced* when its action is executed. If  $R_9$  is enforced before  $R_8$ ,  $R_9$  fails since all services of the active space are stopped. Similarly, if  $R_{10}$  is enforced before  $R_8$ ,  $R_{10}$  fails since the active space file system is not running. But if  $R_8$  is enforced before  $R_9$  and  $R_9$  is enforced before  $R_{10}$ , the active space successfully restarts, 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 predictable. 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 [SC06].

### 3.2.4 Exception Model

Actions in policy rules may fail due to errors in policies, bugs in actions or due to system errors. Most policy enforcement systems do not verify if execution of a rule action was successful. They ignore execution errors or assume that errors are handled by exception handlers that may be programmed within actions. An ECA policy enables an abstraction by separating the specifications of situation-action pairs from their implementations. This abstraction hides the intricacies of observing events in the system, verifying system conditions and initiating actions. Each abstraction level requires its own exception model since models of lower abstraction, such as exception handlers in actions, may not handle exceptions favorably. For example, a rule  $R_{12}$  may state “If a device owned by a guest user physically enters an active space, authorize using credentials”. The rule would be represented as:

```

R12:  on (ObjectEnter (Device d, Space s))
        if (roleDevice(d) == “guest”)
        do(AutoAuthorizeDevice(d, s));

```

When a device enters the active space, the location system generates an *ObjectEnter* event. The management system receives this event, verifies if the device is owned by a guest user and if so, executes the *AutoAuthorizeDevice* action. Device authorization may fail due to wrong credentials, credentials being absent or inability to communicate with the device. In such circumstances, the administrator of the active space may want to detect the failure and use a different means of authorization such as passwords or fingerprint verification. But the *AutoAuthorizeDevice* action may internally handle the exception by logging the error message, which would be unacceptable to the administrator. Therefore, an exception model is required to detect errors and specify corrective actions at the policy abstraction level. The ECA rule framework is

poorly suited for verifying execution of an action since it does not contain action specifications and therefore, relies on the action to report execution errors. Actions may not propagate the exception to the enforcement system and may handle exceptions unfavorably. Therefore, the enforcement system incorrectly perceives this as successful action execution. A policy-based management system should support techniques to verify action execution and take corrective actions on failure, for effective management.

### **3.2.5 Long-running Actions**

Active spaces contain certain management actions that execute throughout their lifetimes or for significantly long periods of time. These include actions that trigger heartbeats, maintain certain QoS, enforce specific states on entities and so on. The persistent nature of these actions interfere with the enforcement of rules that are triggered in a different situation in future. For example, an active space has a rule that initiates heartbeats in devices that are brought into the space. Heartbeats are “I’m alive” messages periodically sent to the presence service to indicate membership and proper functioning of the device in the active space. This action executes as long as the device is part of the active space. If a rule to turn off the device is enforced in a different situation in future, the rule conflicts with the rule to initiate heartbeats, since turning off the device stops heartbeats. In order to detect such conflicts, information about the behavior of the action is required. ECA rules do not contain behavioral information of actions and are unable to detect such conflicts.

Rules with long-running actions need to be monitored to detect failures during their execution lifetimes. Existing policy systems do not provide any mechanisms to monitor enforcement of such rules. Once initiated, the rules are assumed to execute without any failures.

### **3.2.6 Policy Design and Management**

The dynamism of an active space affects the management policy. When devices and applications are added to an active space, rules to manage these new entities should be added. When these entities are taken out of the active space, corresponding rules must be removed. Policies may need to be composed when entities have their own rules. These issues cause additional overhead to the management system.

## **3.3 Solution Space**

From the above set of problems, we realized that the ECA framework is poorly suited for designing management rules for active spaces since very poor static and dynamic reasoning can be performed with the framework.

The ECA framework has been used for designing active database triggers [BW00], authorization rules [SNC02] and management rules [CLN00]. In the field of active database triggers, the framework has been used to perform various kinds of analyses such as conflicts analysis, confluence analysis and termination analysis [BW00]. Similarly, in the domain of access control, the ECA framework enables conflict analysis [KMPP02], rights-amplification analysis [Bla03] and so on. But very poor analyses and guarantees are provided by ECA-based management systems. We realized that for active database and access control rules, the set of actions used is well-defined. Normally, trigger rules use the database operations *insert*, *delete* and *update*, while access control rules use *authorize*, *deny* and *delegate*. But actions in management rules can range from simple atomic operations to complex scripts and therefore, are not well-known. Since actions are well-defined in the first two kinds of rules, implicitly their effect on the system is known. For example, an *insert* operation increases the number of records in a database by 1. Similarly, the *authorize* operation grants access rights to an entity. In order to enable similar reasoning with management rules, information about the effects of an action on a system must be *explicitly* provided.

Therefore, we introduce an extended framework of ECA called *Event-Condition-Precondition-Action-Post condition (ECPAP)* for designing management rules for active spaces. ECPAP rules contain axiomatic specifications 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 and post-condition specifies the partial system state once the action has successfully executed. The 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).

The ECPAP framework has been successfully used to detect conflicts due to effects of actions [SRC05], policy cycles [SC05], determining enforcement order of rules [SC06], policy exception handling and reasoning about policies with long-running actions. We describe each of these in detail in the rest of the thesis.

### 3.4 Thesis Statement

From the description of the context, problem and solution space, we state the thesis as follows:

*Management rules for active spaces require specifications of management actions to be stated explicitly to enhance static and dynamic reasoning. Extending management rules with specifications of actions facilitates conflict detection, cycle detection, definition of enforcement semantics, confluence analysis, enforcement monitoring and reasoning with long-running actions. This new framework enables deterministic policy-based management.*

### 3.5 Success Criteria

This thesis addresses a range of problems with policy-based management systems using the ECPAP framework. The solution to each problem is evaluated based on its algorithmic complexity and overhead imposed on the system. In addition, we propose the following criteria to evaluate this thesis :

- Can this framework detect a larger class of conflicts than current conflict detection techniques ?
- Can this framework be used for detecting cycles in a set of policy rules ?
- If multiple rules are triggered in a situation, can this framework enable reasoning to determine a predictable order of rule enforcement ?
- Can this framework enable an exception model for policies ?
- Can this framework support reasoning for policies with long-running actions ?
- Does the framework enable determinism in management ?
- Is the overhead of policy reasoning acceptable in practical systems ?

In addition, we study the feasibility of using the ECPAP framework on two distributed systems. These studies provide empirical evaluations of the correctness of the analysis techniques and demonstrate the usefulness of the framework on complex distributed systems.

## Chapter 4

# Specification-enhanced Policy Framework

In this chapter, we describe the policy framework based on Event-Condition-Precondition-Action-Postcondition (ECPAP) rules. In section 4.1, we discuss the requirements of the policy system. We present the policy framework in section 4.2 with examples. In section 4.3 we discuss the policy enforcement process and conclude the chapter.

### 4.1 Requirements

An active space contains several heterogeneous entities, each having its own set of management guidelines. This heterogeneity necessitates a policy language that supports a rich set of events, data types and operations. The language should be extensible to accommodate new events, types and operations as entities are introduced into the active space. Existing policy languages [LBN99, KFJ03] are fairly static with regard to supported events and types.

The policy framework should support a specification language that allows actions to be described axiomatically and behaviorally. Action specifications contain the system states that are reached and events generated due to action execution. The language should contain an easy-to-specify methodology and a rich set of operators for formal specifications of actions.

Policy errors should be detected statically so that policy designers can be notified before enforcement. The policy framework should provide support for debugging and profiling of policies. Existing policy systems provide little or no support for policy error detection.

The management policy gets modified when entities are added or removed from an active space. The policy system should provide interfaces to support rule modifications and policy composition.

## 4.2 Policy Framework

### 4.2.1 Syntax and Semantics

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

on  $(ruleid, event)$  if  $(condition)$  do  $(action)$

A policy rule is read as: “When *event* occurs in a situation where *condition* is true, then *action* is executed”. *ruleid* is an identifier that uniquely identifies a rule in the policy. 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). For the purpose of analysis, 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. We make the distinction between an action developer and a policy designer. An action developer is responsible for developing the management action of an active space. These actions are part of the services that are developed for a managed entity. For example, a mobile device contains services for device hibernation; battery power estimation; secondary storage access; application initiation and so on. These are actions developed by the vendor of the device and is used by the device user. When the device is used in an active space, it needs to be managed according to the guidelines set by the organization. A policy designer encodes these guidelines as rules. So this distinction is important.

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) [LBN99] by adding axiomatic specifications as “extension”s to the rule.

Similar to PDL, 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  $e(T_1 t_1, \dots, T_n t_n)$ , where *e* is a primitive event symbol of *n* arguments and each *t<sub>i</sub>* is a variable of type *T<sub>i</sub>*. *T<sub>i</sub>* 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  $p_1 \ \&\& \ p_2 \ \&\& \ \dots \ \&\& \ p_m$  where each *p<sub>i</sub>* is a predicate of the form  $x_1 \theta x_2$  and each *x<sub>i</sub>* 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  $proc(t_1, \dots, t_w)$  where *proc* is an action symbol and *t<sub>i</sub>*s are parameters. For example,

$restartSpace(s)$  is an action where  $s$  denotes an active space. Pre- and post-conditions of an action are first-order predicate logic formulas of the form  $p_1 (\&\&/||p_k)^{k=2..l}$ , where each  $p_i$  is a first-order predicate of the form  $Q_1 t_1 \dots Q_v t_v \text{ pred}(t_1, \dots, t_v)$ :  $Q_i$  is an optional quantifier and each  $t_i$  is a constant or a variable. The policy grammar is presented in appendix I.

A policy,  $P$  is a finite set of ECPAP rules. 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,

$$\begin{aligned} occ(e) \wedge c \wedge p &\rightarrow exec(a) \\ exec(a) &\rightarrow \diamond s \end{aligned}$$

where  $exec(a)$  represents the initiation of the execution of action  $a$ .  $\diamond$  is the *eventually* temporal operator and  $\diamond s$  means that  $s$  becomes true after a few execution steps. 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.

The post-conditions of actions can also specify the events generated using the *event* predicate. The predicate takes in a string expression denoting the event and evaluates to *true* when the event has been observed. For example, the post-condition  $\bigwedge_{\forall App \in Applications} event(suspendEvent(< App >)) \wedge statusApp(< App >, "suspended")$  is a predicate expression specifying the events that are generated and the status of the applications once all events have been observed. Variables are specified using angle brackets –  $<>$ . These variables are replaced by their values when the predicate expression is converted to a propositional expression at runtime. If the set  $Applications = \{authapp, checkpointapp, demoapp\}$  at runtime, the expression is expanded to

$$\begin{aligned} &(event(suspendEvent("authapp")) \wedge statusApp("authapp", "suspended")) \wedge \\ &(event(suspendEvent("checkpointapp")) \wedge statusApp("checkpointapp", "suspended")) \wedge \\ &(event(suspendEvent("demoapp")) \wedge statusApp("demoapp", "suspended")). \end{aligned}$$

The ordering of events can also be specified using temporal operators. This ordering enables more complex evaluations and aids monitoring of rule enforcement. Currently, we support the *next* temporal operator represented as  $\bigcirc$ .  $event(p) \wedge \bigcirc event(q)$  implies that event  $q$  is received after event  $p$  by the event reception system.

Gaia uses asynchronous communication and therefore, can be modeled using actors [Agh86]. Each service in Gaia is represented as an actor. The rule enforcement system can be considered to be a special actor that receives asynchronous events from the Gaia system. Rules in the actor get evaluated and messages are generated that invoke actions on the other actors. The modelling is pictorially illustrated in figure 4.1.

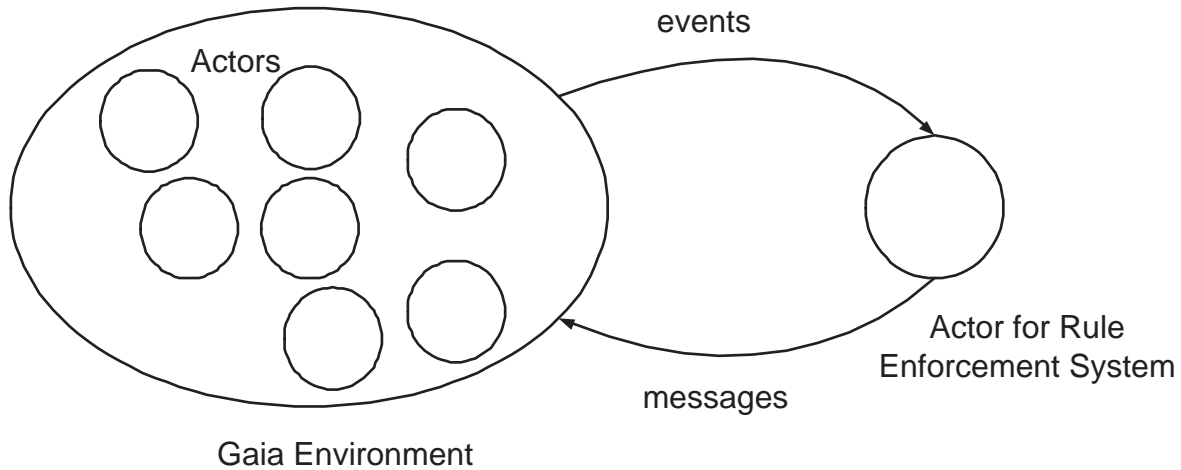


Figure 4.1: Actor modeling of rule evaluation

In addition, we also introduce an operator called *choice* to list the possible set of states that can be taken after an action execution. Only one of the listed states is reached after action execution. This operator is denoted by  $\sqcup$ .  $s_1 \sqcup s_2 \sqcup \dots \sqcup s_n$  is read as “choice of  $s_1$  or  $s_2$  or ...  $s_n$ ”. A *choice-set* of a *choice-expression*,  $p_1 \sqcup \dots \sqcup p_n$ , is the set  $\{p_1, \dots, p_n\}$ . This operator is useful when the state reached after an execution is known to belong to a set of states but cannot be predicted in advance. We present some examples below:

$\{true\}$

`toggleLight(); //toggle state of light`

$\{statusLight(on) \sqcup statusLight(off)\}$

$\{true\}$

`x = random(n); //generate random number between 0 and n-1`

$\{\sqcup_{y=0..n-1} x=y\}$

The semantics of the  $\sqcup$  operator differs with the analysis. We discuss in each analysis how the operator is treated.

## 4.2.2 Action Specifications

The ECPAP framework extends the ECA framework by using the Hoare triple [Hoa69]. A Hoare triple represented as  $\{P\} C \{Q\}$  describes how an action  $C$  changes the state of computation from a state where  $P$  is true to a state where  $Q$  is true.  $P$  and  $Q$ , expressed as first-order predicate logic expressions, are pre- and post-conditions of  $C$ , respectively and are called *axiomatic specifications*. The pre-condition specifies the system state that should exist before  $C$  can be executed and the post-condition specifies the system

state that exists once  $C$  completes execution.

Specification-enhanced programming and system management have recently gained prominence as important approaches to reducing programming and management efforts of complex systems [SC05, SRC05, SC06, AHS05, RC04, Sri05]. In [SC05, SRC05] we showed how extending actions with specifications enabled advanced conflict and termination analysis for policy-based management systems. In [SC06] we demonstrated the advantages of the ECPAP framework for ordering rule enforcement. Andrzejak et al. [AHS05] have used actions with pre- and post-conditions for planning complex workflows from simple actions for system management. Anand et al. [RC04] use specification-enhanced actions, expressed as pre-conditions and effects, for programming pervasive computing environments. The ABLE project [Sri05] 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.

In this thesis, we assume that the specifications are correct and complete to the extent required for analysis. The specifications describe the changes that happen to the system due to action execution. Dealing with incorrect specifications is an orthogonal problem, which we briefly discuss in chapter 13.

### 4.2.3 ECPAP Examples

In this section, we present some example ECPAP rules with short descriptions. The pre- and post-conditions of actions are specified above and below each rule action for ease of reading and are not specified as part of the rules in the actual policy.

```

R41: on(DevicePlugin(Device d))
      if(d.type == "compute")
      { authorizationstatus(d, authorized) }
      do(queryResourceList(d));
      { resourceListReceiveStatus(d, received) }

```

$R_{41}$  gets triggered when a device is plugged into the active space. If the device is of type “compute” and has been authorized, the system queries the list of resources available on the device. If the action is successful, the resource list is received.

```

R42: on(ClusterPlugin(Cluster c))
      if(true)
      { true }
      do(authorizeDevicesinCluster(c));
      { forall node .in. c, authorizationStatus(node, authorized) }

```

$R_{42}$  gets triggered when a cluster of nodes is added to an active space. All nodes of the cluster are authorized with the space. This example demonstrates the usage of first order expressions in the action post-condition.

```

 $R_{43}$ : on(ObjectEnter(Person p))
    if(true)
    {true}
    do(lightsStatus("on"));
    {.henceforth. statusDevice("Lights", "on")}

```

$R_{43}$  turns on the lights in an active space when a person enters the space. The post-condition specifies that the lights remain *on* henceforth. This expression uses the henceforth temporal operator to describe the long-running action *lightsStatus*. Rules with long-running actions are described in chapter 7.

### 4.3 Policy Enforcement Process

The policy enforcement process is illustrated in Figure 4.2 as a flowchart. A policy is compiled and checked for conflicts and cycles using static analysis techniques [SRC05, SC05]. 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 [SRC05] and resolves them using priorities. It determines the enforcement order of rules and constructs a Petri net workflow [SC06]. This workflow is executed by a workflow execution engine. The management system verifies action execution after each action completes. If an action fails, the system generates exception events and determines the exception handlers in the policy triggered by the events. It reconstructs the workflow with the new actions and executes it. 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. The different phases of enforcement are described in detail in the rest of the thesis.

### 4.4 Conclusion

This chapter presented the ECPAP framework that we have used in this thesis for policy design. The framework supports axiomatic specification of actions. Some behavioral aspects of the actions can also be

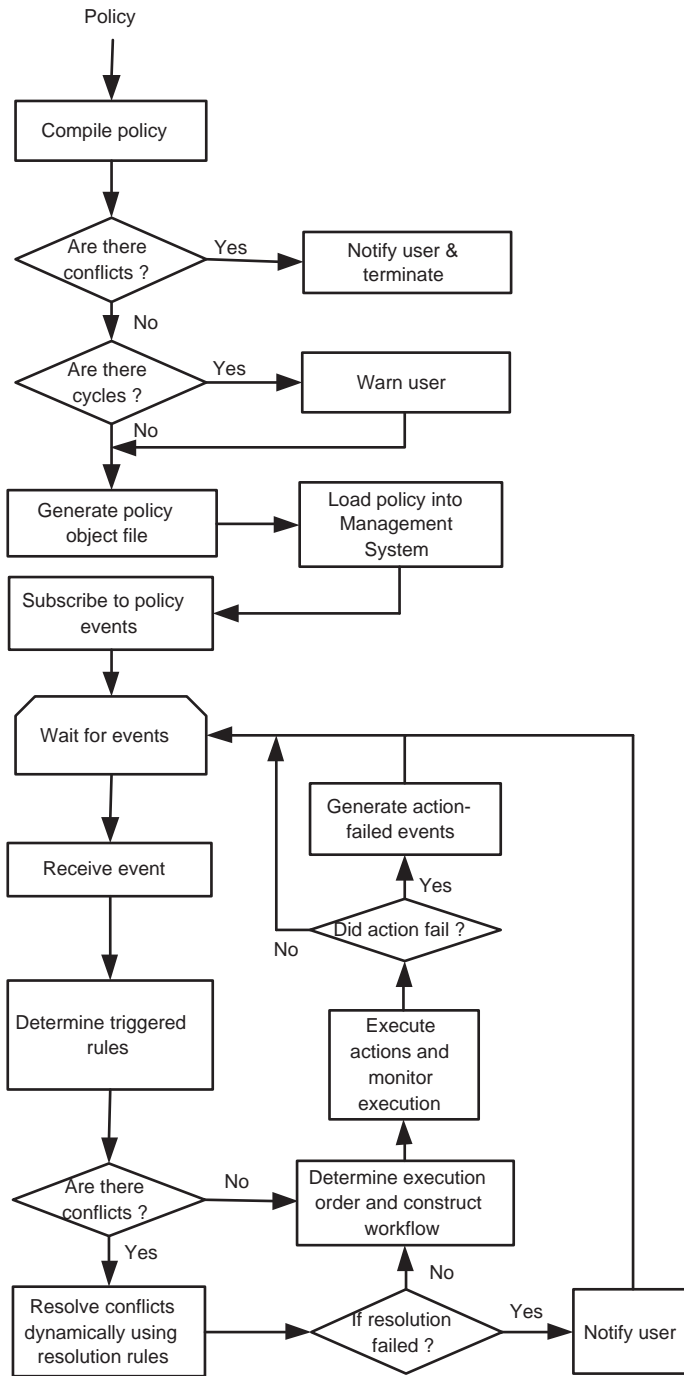


Figure 4.2: Flowchart of Policy Enforcement

described using temporal operators. The management system built with the ECPAP framework offers enough flexibility to add new events, data types and operations dynamically. The framework is being extended to support more complex behavioral expressions as discussed in chapter 13.

# Chapter 5

## Conflict and Cycle Analysis

In this chapter, we explore the conflict analysis and resolution problem. Prior research works on conflict analysis address rule conflicts that occur due to conflicting actions. Section 5.1 presents examples of rule conflicts that occur due to conflicting action effects and shows how the ECA framework is unable to detect such conflicts. We describe the ECPAP approach to conflict and cycle analysis in sections 5.2 and 5.3, respectively. We evaluate the algorithms in section 5.4 and conclude the chapter.

### 5.1 Policy Conflicts

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 [SRC05]. For example, consider the policy shown in figure 5.1.

Rule  $R_{51}$  mounts a device’s file system onto the active space file system, when the device is brought into the active space. Rule  $R_{52}$  restarts all active space services when an administrator’s device is brought into the active space. In order to mount a device file system onto that of the active space, the file system of the active space should be running. Therefore, the actions of the two rules conflict with each other. When a “compute” device of an administrator is brought into the space both rules are triggered and this leads to a rule conflict.

Previous research works on conflict detection for obligation policies define conflicts as violations of action constraints [CLN00, Slo94, KFJ03]. Action constraints specify the set of actions that cannot occur together

```
 $R_{51}$ : on(r51, ObjectEnter(Device d))  
      if(d.type == "compute")  
      do(mountFileSystem(d));  
  
 $R_{52}$ : on(r52, ObjectEnter(Device d))  
      if(d.owner == "admin")  
      do(restartActiveSpaceServices());
```

Figure 5.1: Policy containing rule conflicts due to conflicting actions

```

R53: on(r53, ObjectEnter(Device d))
    if(d.type == "compute")
    do(startAuthorization(d));
R54: on(r54, ObjectEnter(Device d))
    if(true)
    do(stopAllApps(d));
R55: on(r55, ObjectExit(Person p))
    if (p.role == "owner")
    do(startApp("logApp"));
R56: on(r56, ObjectExit(Person p))
    if(person_count() == 0) //space is empty
    do(suspendSpace());

```

Figure 5.2: Policy containing rule conflicts due to conflicting action effects

as a predicate:  $\neg(a_1 \wedge \dots \wedge a_n)$  where  $a_i$  is an action term. In the above example, the action constraint would be specified as  $\neg(\text{mountFileSystem}(\text{Device}) \wedge \text{restartActiveSpaceServices})$ . When two or more rules are triggered, the enforcement system checks to see if the set of rules satisfies the action constraint. The actions of the triggered rules are asserted into a Prolog knowledge base. The action constraint is queried to check if it is satisfied.

$$\begin{aligned}
 &(\text{mountFileSystem}(\text{Device}) \wedge \text{restartActiveSpaceServices}) \neq \\
 &\quad \neg(\text{mountFileSystem}(\text{Device}) \wedge \text{restartActiveSpaceServices})
 \end{aligned}$$

Since the above policy violates the constraint when  $R_{51}$  and  $R_{52}$  are triggered, the resulting situation is flagged as a conflict.

### 5.1.1 Conflicts Due to Action Effects

While the action constraint approach to conflict detection can detect modality conflicts – inconsistencies arising from actions being permitted and prohibited [LS99], conflicts due to temporal events and roles [DIR02] and conflicts due to opposing obligation and authorization policies [LBN99], they cannot detect conflicts arising from effects of actions. For example, consider the policy in figure 5.2.

The action of  $R_{53}$ , *startAuthorization*, authorizes a device when it is brought into an active space, while the action of  $R_{54}$ , *stopAllApps*, stops all applications running on the device. A device is authorized by the authorization application. When both rules are triggered, *startAuthorization* action starts the authorization application, while *stopAllApps* action stops all applications resulting in stopping the authorization application too. This causes a conflict between the two rules. Stopping of the authorization application by the *stopAllApps* action is an effect of the action. In order to detect such a conflict, each set of actions has to

```

R53: on(r53, ObjectEnter(Device d))
      if(d.type == "compute")
        {true}
        do(startAuthorization(d));
        {statusApp(authApp, running)}

R54: on(r54, ObjectEnter(Device d))
      if(true)
        {true}
        do(stopAllApps(d));
        {statusApp(logApp, stopped) && statusApp(authApp, stopped)}

R55: on(r55, ObjectExit(Person p))
      if(p.role == "owner")
        {statusSpace(running)}
        do(startApp("logApp"));
        {statusApp(logApp, running)}

R56: on(r56, ObjectExit(Person p))
      if(person_count == 0) //space is empty
        {statusSpace(running)}
        do(suspendSpace());
        {forall app .in. spaceApps(), statusApp(app, stopped)}

```

Figure 5.3: Policy Rules in the ECPAP Framework

be manually analyzed by the system administrator to determine if they conflict with each other. This may be infeasible as the system administrator may not know the complete effect of the action on the system. In addition, this approach does not scale well with the large number of actions in an active space.

Similarly, a conflict exists between rules  $R_{55}$  and  $R_{56}$  since suspending the space suspends all applications including *logApp* that checkpoints application data into the active space file system.

Typical examples of conflicts due to action effects include rules that turn off a device and start an application on the same device, increase file system quota for a user while simultaneously reducing the service quality (which decreases the file system quota for the user) and start two applications that need a common non-sharable resource.

## 5.2 Policy Conflict Detection

In order to detect policy conflicts due to effects of actions, we need to express these rules using the ECPAP framework and redefine policy conflicts as violations of post-condition constraints [SRC05]. The rules in the ECPAP framework are shown in figure 5.3. A post-condition constraint is a predicate that expresses a state that the system should not reach and is an expression of the form  $\neg(p_1 \wedge \dots \wedge p_m)$ , 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 [CLN00]. 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 5.1:**  $K$  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  $S \models ac$  [CLN00]. If  $S \not\models ac$ , then  $S$  is said to be a conflict in the policy.

**Definition 5.2:** Given an action constraint  $ac = \neg(a_1 \wedge \dots \wedge a_n)$ , we define an equivalent post-condition constraint,  $epc$  of  $ac$ , as a predicate  $\neg(p_1 \wedge \dots \wedge p_n)$  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 5.1:** 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  $K = \{p_i | \forall a_i \in S, p_i \text{ is the post-condition predicate of } a_i\}$

$$\begin{aligned} S \not\models ac &\Rightarrow \exists a_1, \dots, \exists a_s \in S | \{a_1, \dots, a_s\} \not\models ac \\ &\Rightarrow \{p_1, \dots, p_s\} \not\models pc, \text{ where } p_i \text{ is the post-condition predicate of } a_i \text{ and } pc =_e ac \\ &\Rightarrow K \not\models pc \text{ (since } p_i \in K \text{)} \end{aligned}$$

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

The active space 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. The problem of determining if two relational expressions are equal can even be undecidable. We employ dynamic conflict detection techniques to detect remaining conflicts at runtime (when we can evaluate all the conditions), and use meta-rules to resolve them. Some policy-based systems [CLN00, KFJ03] employ resolution policies to resolve all policy conflicts. This requires specification of resolution rules for all possible conflicts. Detecting and resolving as many conflicts as possible, statically (before the policy is actually loaded into the system), reduces the number of dynamic 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.

### 5.2.1 Static Detection

Static detection is used to detect 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. For example, in the policy in figure 5.1, event terms of rules  $R_{51}$  and  $R_{52}$  match. Static matching can be extended with unification to match a variable with possible values that the variable can take based on its data type.  $ObjectEnter(Person\ p)$  matches  $ObjectEnter("Tom")$ , since the event symbols match and "Tom" is an instance of data type *Person*. A conjunctive 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  $C_2$ .

Figure 5.4 shows the algorithm for static conflict detection. The algorithm is initialized with the post-condition constraint set,  $PC$ , and the policy specified as ECPAP rules. Each rule,  $r$ , in the policy is evaluated against other rules in the policy. If the event, conditions and pre-conditions of rule  $r$  match those of another rule  $s$ , the post-condition of rule  $s$  is added to a set,  $K$ . 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 called XSB [XSB] 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 the 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, conditions and pre-conditions of rule  $r$ . If the union set is inconsistent, the corresponding rules are flagged as being in conflict and their identification numbers are displayed.

Applying this algorithm to the policy in figure 5.3 with the post-condition constraint

```

//initialize

PC          : post-condition constraint set
P           : the Policy
event(r)    : event of rule r
id(r)       : rule identifier of rule r
condition(r) : condition of rule r
precond(r)  : pre-condition 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)
  p := precond(r)
  K := {}
  R := {} //rule id set
  for each rule s in P and s ≠ r
    if(match(event(s), e) && match(condition(s), c) && match(precond(s), p))
      K := K ∪ post(s)
      R := R ∪ id(s)
    endif
  end for
  K := K ∪ post(r)
  if (K ∪ PC) is not consistent
    print R
  endif
end for

```

Figure 5.4: Static Conflict Detection Algorithm

$P_1 : \neg(status(App, running) \wedge status(App, stopped))$

detects that rules  $R_{53}$  and  $R_{54}$  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  $R_{55}$  and  $R_{56}$  is not detected since their condition expressions are not found to match statically.

### 5.2.2 Dynamic Detection

Once the user resolves the above conflict, the policy compiler generates a policy object file. This is loaded into the management service. 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  $R_{55}$  and  $R_{56}$ . 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  $R_{55}$  and  $R_{56}$  are both added

to the activation set. The activation set is analyzed to determine any conflicts. The algorithm for detecting the conflicts is presented in figure 5.5.

```

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

for each rule r in AS
    K := K ∪ {post(r)}
end for

if (K ∪ PC) is not consistent
    resolve conflict

```

Figure 5.5: Dynamic Conflict Detection Algorithm

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 concluded and the rules are sent to a conflict-resolver for resolution. For the example in figure 5.2, the algorithm detects a conflict between rules  $R_{55}$  and  $R_{56}$  since *logApp* 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.

### 5.2.3 Conflict Analysis with the $\sqcup$ Operator

The  $\sqcup$  operator lists the set of possible post-conditions reached by a system. Only one post-condition among the set may cause a conflict with another triggered rule. We take a conservative approach by considering the possibility of the system reaching the conflicting post-condition. Therefore, the system analyzes each post-condition predicate with that of the other triggered rules for conflicts. Since these post-conditions are not guaranteed to occur, the possible conflicts are listed as warnings to the user.

**Definition 5.3:** An action with post-condition  $(p_1 \sqcup \dots \sqcup p_n)$  conflicts with an action with post-condition  $x$  if  $\bigvee_{i=1..n} ((p_i \wedge x) \not\models pc)$  where  $pc$  is the post-condition constraint.

### 5.2.4 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  $postcondition$

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 [LBN99, CLN00, KFJ03] 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 5.3 is

$M_1$  : if ( $event(ObjectExit) \wedge can\_reach(status(log\_app, running)) \wedge can\_reach(status(log\_app, stopped))$ )  
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,  $M_1$  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 5.4:** A post-condition predicate  $p$  is *preferred* by a resolution policy  $M = \{m_1, \dots, m_n\}$  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 presented in figure 5.6 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 [BW00] and is discussed in 6. Therefore, the resolution algorithm chooses rule  $R_{55}$ . The management system executes the

```

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 ∪ r
    end if
end for
if cardinality(N) > 1
    notify user
else
    return N

//function preferred
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

```

Figure 5.6: Dynamic Conflict Resolution Algorithm

rule action of  $R_{55}$  and ignores  $R_{56}$ .

### 5.3 Policy Cycles

A policy may contain a non-terminating sequence of rule executions where action of one rule may trigger another rule in a cycle. Figure 5.7 shows a policy with cycles. Rule  $R_{57}$  states that “when a person exits the space, suspend all applications”. This triggers rule  $R_{58}$  that starts the checkpoint application when any application is suspended. Starting an application triggers rule  $R_{59}$  that checks to see if the space is empty and if so suspends the application being started. In this policy, a cycle exists between rules  $R_{58}$  and  $R_{59}$ , if

```

R57: on(r57, ObjectExit(Person p)) //if user exits, suspend applications
      if (true)
        {true}
        do(suspend_apps())
        {forall App.in. Applications, event(suspendApp(App)) @@@ statusApp(App, suspended)}

R58 : on (r58, suspendApp(Application App))
      if (true)
        {true}
        do (startApp(checkpoint_app(App)))
        {event(startApp(checkpoint_app)) @@@ statusApp(checkpoint_app, suspended)}

R59 : on (r59, startApp(Application App))
      if(statusSpace() == "empty")
        {true}
        do (suspendApp (App))
        {event(suspendApp(App)) @@@ statusApp(App, suspended)}

```

Figure 5.7: A Policy containing a Cycle

the space is empty.

Annotating rule actions with axiomatic specifications enables us to reason about the policy to detect cycles. The policy compiler analyzes the policy and reports any cycles.

A rule  $r_1$  triggers another rule  $r_2$  in a policy if the execution of action of  $r_1$  generates an event that matches event of rule  $r_2$ . We use static analysis techniques to determine if an event specified in the post-condition matches an event specified in the rule. When the policy is compiled, a trigger graph is created that specifies which rule triggers which other rules in the policy. A trigger graph of a policy is a directed graph whose vertices represent rules and edges represent the *triggers* relation. The trigger graph of the policy in figure 5.7 is shown in figure 5.8.

**Definition 5.4** : A trigger graph for a policy  $P$ , denoted by  $\text{Trig}(P)$  is an ordered pair  $(P, E)$  such that  $E = \{(r_1, r_2) \mid \forall r_1, r_2 \in P, r_1 \text{ triggers } r_2\}$ .

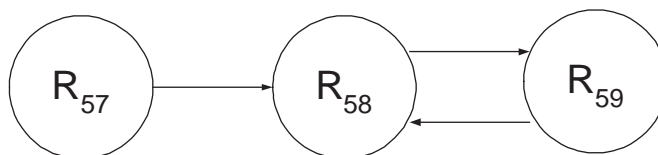


Figure 5.8: Trigger graph for policy in figure 5.7

The trigger graph construction algorithm is shown in figure 5.9. A policy contains a cycle only if a cycle exists in the trigger graph of the policy. Note that a cycle in a trigger graph does not imply a cycle in the policy because the rule condition and action pre-condition may prevent action execution and thus break the

cycle. Therefore, a cycle in a trigger graph is a necessary but not a sufficient condition for a policy cycle.

```

P           : policy: {r1 , ..., rn}
E           : edge set : initialized to {}
Event-Set(r) : set of events in post-condition of action of rule r
Event(r)    : event of rule r

for each rule r in P
  E = Event-Set (r);
  for each event e in E
    for each rule k in P
      if Event(k) matches e
        E = E (r, k);
      end if
    end for
  end for
end for

```

Figure 5.9: Trigger Graph Construction Algorithm

The trigger graph is analyzed for cycles using a simple breadth-first search algorithm shown in figure 5.10. Cycles are reported by the policy compiler as warnings. For the policy in figure 5.7, the policy compiler reports that rules  $R_{58}$  and  $R_{59}$  may have a possible cycle.

Similar to conflict detection with the  $\sqcup$  operator, we consider the possibility of the system generating the event that can cause a cycle. Therefore, we construct the trigger graph by checking if any of the events listed with the  $\sqcup$  operator triggers any other rule in the policy. An event with post-condition  $(e_1 \sqcup \dots \sqcup e_n)$  triggers a rule with event  $x$  if  $\bigvee_{i=1..n} e_i$  matches  $x$ .

## 5.4 Evaluation

For the static conflict detection algorithm (figure 5.4), 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  $O(q^2t)$  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  $O(n(b + q^2t))$  where  $n$  is the number of rules in the policy. The consistency checking has  $O(nc)$  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  $O(de)$  queries to the Prolog reasoner, where  $d$  is the number of post-condition constraints and  $e$  is

**Algorithm:** Cycle Detection

```

detectCycle(E: edge set)

pick (a,b) in E
enqueue(a)
mark a

while (q!= empty)
  x = dequeue()
  for all i such that (x,i) in E
    if i is marked
      warn about cycle
      exit
    enqueue(i)
    mark i
  end for
end while

```

Figure 5.10: Cycle Detection Algorithm

the average number of predicates per post-condition. Therefore, the complexity of consistency verification is  $O(nc + de)$ . We do not consider the complexity of Prolog queries and assume it to be  $O(1)$ . Since the outer loop iterates  $n$  times, the final complexity is  $O(n^2(b + q^2t) + n^2c + nde)$ .

By similar analysis, the complexity of the dynamic conflict detection algorithm (figure 5.5) is found to be  $O(n + n^2c + nde)$ . For the resolution algorithm (figure 5.6), the *preferred* method has a complexity of  $O(m(q^2t)) + O(k)$ , 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  $O(pc(m(q^2t) + k))$ , where  $p$  is the size of the conflict rule set.

Building the trigger graph (figure 5.9) has a worst-case complexity of  $O(n^3)$ , where  $n$  is the number of rules, if every rule triggers every other rule. Given a graph, a cycle can be detected with a time complexity of  $O(n + e)$  where  $e$  is the number of edges [Tar72]. For a fully connected graph,  $e = n^2$  and therefore, cycle detection has a complexity of  $O(n + n^2) = O(n^2)$ . Therefore, the entire policy cycle detection algorithm has a complexity of  $O(n^3) + O(n^2) = O(n^3)$ .

Figure 5.11 shows the overhead of conflict and cycle analysis. The policy compiler was executed on a Pentium(M) 1.7 GHz machine with 1.0GB RAM. For a policy containing 400 rules, the compiler takes about 1 sec to perform the analyses.

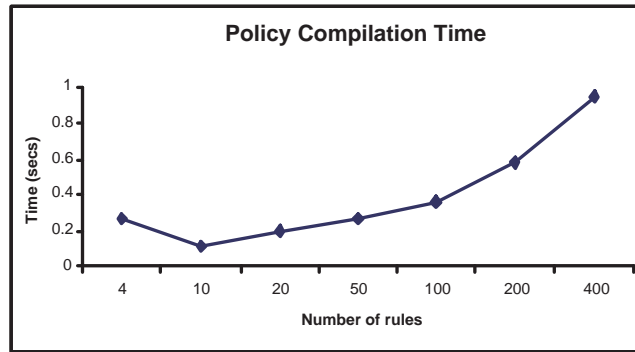


Figure 5.11: Conflict and Cycle Analysis Overhead

## 5.5 Conclusion

In this chapter, we presented techniques to detect and resolve conflicts due to effects of actions using the ECPAP rule framework. Static analysis cannot detect all conflicts since many conflicts occur due to runtime system conditions. Therefore, our approach uses a combination of static and dynamic conflict analysis techniques. Static conflicts are resolved by the policy designer while dynamic conflicts are resolved using a resolution framework. This framework uses priorities to choose one rule from a set of conflict rules. We also presented techniques to detect cycles in policies and presented algorithms for cycle analysis. The complexity analysis revealed that the overhead from these algorithms is not very significant.

## Chapter 6

# Ordering Rule Enforcement

In this chapter, we present the rule ordering problem and our proposed solution. In a single situation, multiple rules can get simultaneously triggered. The order of enforcement of the rules may determine the final system state. In section 6.1, we present the rule ordering problem with an example. We discuss how the ECPAP framework enables reasoning about rule ordering in section 6.2. In section 6.3, we present a notion called enforcement semantics that dictates the way rules should be enforced when multiple rules are simultaneously triggered. Section 6.4 describes ordered rule enforcement. We evaluate the algorithms in section 6.5 and conclude in section 6.6.

### 6.1 The Rule Ordering Problem

A management policy evolves over time by addition and deletion of rules, rule modifications and policy compositions. Policies authored by different administrators may be merged to form the final system management policy. These operations cause various problems such as conflicts and cycles that we discussed in the previous chapter. Even in a policy without conflicts and cycles, multiple rules may be based on a common event-condition and so may be triggered in the same situation. Since the management system enforces any set of non-conflicting rules, all triggered rules are sequentially or concurrently enforced. The order in which the management system enforces the rules can determine the system behavior.

Consider the policy shown in figure 6.1. Rule  $R_{61}$  restarts the active space (and its various services) if it has stopped when a device enters the space. Rule  $R_{62}$  authorizes a device of role *guest* if it enters the space. We have assigned roles to mobile devices to differentiate between devices of different kinds of users. Rule  $R_{63}$  mounts a laptop's file system onto the active space file system [Hes03] when the laptop is brought into the active space and  $R_{64}$  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 in figure 6.2. 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  $R_{61}$ ,  $R_{62}$  and  $R_{63}$ . A rule is said to be *triggered* when its event has been observed and its

```

R61 : on (r61, ObjectEnter(Device d, Space s))
      if (statusSpace(s) == "stopped")
      do(restartSpace(s));
R62 : on (r62, ObjectEnter (Device d, Space s))
      if (roleDevice(d) == "guest")
      do(authorizeDevice(d, s));
R63 : on (r63, ObjectEnter (Device d, Space s))
      if (deviceType(d) == "laptop" && roleDevice(d) == "guest")
      do (mountFileSystem(d, s));
R64 : on (r64, ObjectExit (Device d, Space s))
      if (deviceType(d) == "laptop")
      do(unmountFileSystem(d, s));

```

Figure 6.1: Policy to demonstrate the need for Rule Ordering

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  $R_{62}$  is executed before  $R_{61}$ , authorization fails since none of the services of the active space are running. Similarly, if  $R_{63}$  is enforced before  $R_{61}$ , mount operation fails since the active space file system is not running. But if  $R_{61}$  is enforced before  $R_{62}$  and  $R_{62}$  is enforced before  $R_{63}$ , 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 predictable. 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 ECPAP rules contain action specifications we can reason about rule ordering and provide enforcement guarantees.

## 6.2 Analyzing Action Dependencies

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 ECPAP framework enables analyzing these dependencies. Figure 6.2 shows the policy in the ECPAP framework.

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 6.2, when rules  $R_{61}$  and  $R_{62}$  are simultaneously triggered, execution of the action of  $R_{61}$  brings the active space to the *running* state as indicated by the corresponding

```

R61 : on (r61, ObjectEnter(Device d, Space s))
      if (statusSpace(s) == "stopped")
        {statusService(spaceRepository(s), not_running)}
        do(restartSpace(s));
        {statusSpace(s, running)}

R62 : on (r62, ObjectEnter (Device d, Space s))
      if (roleDevice(d) == "guest")
        {statusSpace(s, running)}
        do(authorizeDevice(d, s));
        {authorizationStatus(d, authorized)}

R63 : on (r63, ObjectEnter (Device d, Space s))
      if (deviceType(d) == "laptop" && roleDevice(d) == "guest")
        {statusSpace(s, running) && authorizationStatus(d, authorized)}
        do (mountFileSystem(d, s));
        {statusFileSystem(d, mounted)}

R64 : on (r64, ObjectExit (Device d, Space s))
      if (deviceType(d) == "laptop")
        {statusFileSystem(d, mounted)}
        do(unmountFileSystem(d, s));
        {statusFileSystem(d, unmounted)}

```

Figure 6.2: Policy of figure 6.1 in the ECPAP framework

post-condition. This satisfies the pre-condition of action of rule  $R_{62}$  and thus enables  $R_{62}$ 's action. Therefore, enforcing  $R_{61}$  before  $R_{62}$  successfully enforces both rules.

The workflow of rule actions is represented as a Boolean Interpreted Petri net (BIPN) [Rou94]. A Boolean Interpreted Petri net is a Petri net [Rei85] 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 ( $R_{61}$ ,  $R_{62}$  and  $R_{63}$ ) of figure 6.2 when the *ObjectEnter* event is received is shown in figure 6.3. The action of rule  $R_i$  is represented as  $A_i$ .

**Definition 6.1 :** Formally, the BIPN of a set of actions  $A = \{a_1, \dots, a_n\}$  is a 1-safe marked Petri net [Rou94] represented as a triple  $B = (P, T, F)$ ,  $P = \{place(a) | \forall a \in A\} \cup \{Start\}$ , where  $place(a)$  is the place representation of action  $a$ .  $T = \{t_{K,pre(a)} | \forall x \in K, t_{K,pre(a)} \in x \cdot \wedge place(a) \in t_{K,pre(a)} \cdot\}$ ,  $K$  is a set of places and for  $x \in P \cup T$ ,  $x \cdot = \{y | yFx\}$  is called the *input set* of  $x$  and  $x \cdot = \{y | xFy\}$  is called the *output set* of  $x$  and the flow relation,  $F \subseteq (P \times T) \cup (T \times P)$  such that  $dom(F) \cup codom(F) = P \cup T$ .  $pre(a)$  represents the pre-condition of action  $a$ .

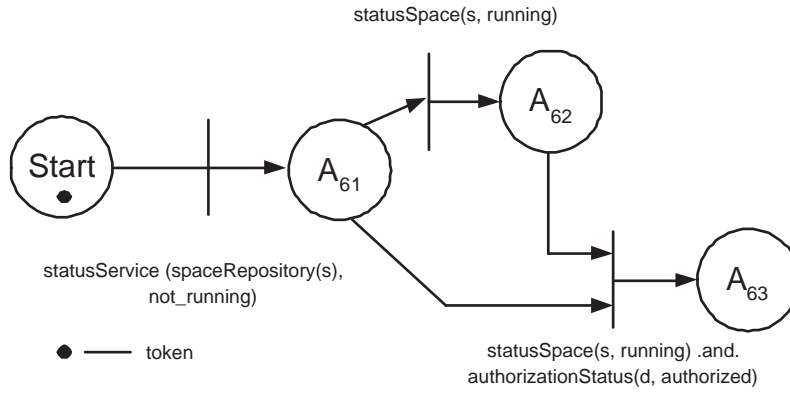


Figure 6.3: Petri net workflow for triggered rules of policy in figure 6.2

### 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 6.2 :** 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  $R_{61}$  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,  $A_{61}$  can be executed independent of  $A_{62}$  and  $A_{63}$ . The algorithm to determine trivially-enabled actions is shown in figure 6.4.

$V = \{\}$  : set of trivially-enabled actions  
 $A$  : set of actions of triggered rules

for each action  $a$  in  $A$   
  if `pre(a)` evaluates to true  
     $V = V \cup a$

Figure 6.4: Algorithm for trivially-enabled action analysis

$\text{Enable}(a) = \{\}$  : set of actions enabled by action  $a$   
 $V$  : set of trivially-enabled actions from algorithm 6.4  
 $A$  : set of actions of triggered rules

for each action  $a \in A$   
   for each action  $b \in A-V$   
     if  $\text{post}(a) \models \text{pre}(b)$   
        $\text{Enable}(a) = \text{Enable}(a) \cup b$

Figure 6.5: Enablement Analysis

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

**Definition 6.3:** An action  $a_1$  is said to *enable* action  $a_2$  if  $\text{post}(a_1) \models \text{pre}(a_2)$  where  $\text{post}(a_1)$  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 in figure 6.5. This algorithm determines that executing  $A_{61}$  enables  $A_{62}$ .

Post-conditions of some actions may satisfy part of the pre-condition of another action. For example, post-condition of  $A_{61}$  -  $\text{statusSpace}(s, \text{running})$  satisfies a conjunct of the pre-condition of  $A_{63}$ . Similarly, post-condition of  $A_{62}$  satisfies a part of the pre-condition of  $A_{63}$ . Therefore,  $A_{61}$  and  $A_{62}$  must be executed to enable  $A_{63}$ . We say that each action  $A_{61}$  and  $A_{62}$  *partially-enables*  $A_{63}$ . The variables  $s$  and  $d$  in predicates  $\text{statusSpace}(s, \text{running})$  and  $\text{authorizationStatus}(d, \text{authorized})$  are bound to values of the active space and device during evaluation, respectively, and thus form propositions whose satisfiability checks are decidable.

**Definition 6.4:** An action  $a_1$  is said to *partially-enable* action  $a_2$  if  $\text{post}(a_1) \models \text{partial} - \text{pre}(a_2)$ , where  $\text{partial} - \text{pre}(a_2)$  is a conjunction of some proper subset of conjuncts of  $\text{pre}(a_2)$ . 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  $\text{partial} - \text{sets}(a)$ . In the above example,  $\text{partial} - \text{sets}(A_{63}) = \{A_{61}, A_{62}\}$ . The algorithm in figure 6.6 determines the partial-sets.

The algorithm determines the set of actions that collectively enable every non-trivially-enabled action,  $a$ . If the set contains only one action, then it implies that a single action enables  $a$  and therefore, is already determined by the enablement analysis in figure 6.5. Therefore, the algorithm in figure 6.6 only considers

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
  end for
  for each subset s of S
    if (cardinality(s) ≥ 2)
      p = true
      for each action c
        p = p ∧ post(c)
        if p ⊨ pre(a)
          Partial-sets(a) = Partial-sets(a) ∪ s
        end if
      end for
    end if
  end for
end for
end for
  
```

Figure 6.6: Partial-sets Determination

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 algorithm of figure 6.5, we separate the two algorithms since partial-enablement analysis has a much higher complexity as detailed in section 6.5.

Once we determine the partial-sets, we construct the workflow as a Petri net using algorithm in figure 6.7. The Petri net is represented as an adjacency set of places and transitions.

This algorithm constructs a BIPN using the results from algorithms in figures 6.4 - 6.6. 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 6.7 for action set *A* is represented as  $B = (P, T, F)$  where

$A$  : set of actions of triggered rules  
 $V$  : set of trivially-enabled actions  
 $\text{Enable}(a)$  : set of actions enabled by action  $a$   
 $\text{Partial-sets}(a)$  : set of all partial-sets of action  $a$   
 $P = \{\text{Start}\}$  : set of Petri net Places initialized to Place called Start  
 $T = \{\}$  : set of Petri net Transitions  
 $\text{place}(a)$  : Place for action  $a$   
 $\text{adj}(x)$  : adjacency set of  $x$ ,  $x \in P \cup T$   
 $\text{trans}(p, f)$  : Transition with function  $f$  connected by edges from places in set  $p$

for each action  $a \in A$

$P = P \cup \text{place}(a)$

$t = \text{trans}(\{\text{Start}\}, \text{true})$

$\text{adj}(\text{Start}) = \text{adj}(\text{Start}) \cup t$

$T = T \cup t$

for each action  $a \in V$  //trivially-enabled actions

$\text{adj}(t) = \text{adj}(t) \cup \text{place}(a)$

for each action  $a \in A$  //enable

for each action  $b \in \text{Enable}(a)$

$t = \text{trans}(\{\text{place}(a)\}, \text{pre}(b))$

if  $t \notin T$

$T = T \cup t$

$\text{adj}(\text{place}(a)) = \text{adj}(\text{place}(a)) \cup t$

end if

$\text{adj}(t) = \text{adj}(t) \cup \text{place}(b)$

end for

end for

for each action  $a \in A-V$  //partially-enable

for each set  $s \in \text{Partial-sets}(a)$

$t = \text{trans}(s, \text{pre}(a))$

$T = T \cup t$

$\text{adj}(t) = \text{adj}(t) \cup \text{place}(a)$

for each action  $b \in s$

$\text{adj}(\text{place}(b)) = \text{adj}(\text{place}(b)) \cup t$

end for

end for

Figure 6.7: Petri net Workflow Construction

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

$T = \{t_{i,j} | (i = \{Start\}, j = true) \wedge (i = \{place(a)\}, j = pre(b) | \forall a, b \in A, (post(a) \models pre(b)) \wedge (pre(b) \not\models true)) \wedge (i = s, j = pre(b) | \forall b \in A, (\forall s \in 2^{P - \{Start\}}, \bigwedge_{\forall k \in s} post(action(k))) \models pre(b))\}$ ,  $action(k)$  represents the action in set  $A$  assigned to place  $k$ .

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

The three conjuncts in the definition of  $T$  correspond to the transitions resulting from algorithms 6.4 - 6.6. The transitions are labeled  $t_{i,j}$  where  $i = t_{i,j}$  and  $j$  is the assigned Boolean function. The flow relation,  $F$ , represents the various edges of the Petri net.

**Theorem 6.1:** For a set of actions  $A = \{a_1, \dots, a_n\}$ , the Petri net generated by algorithm 6.7 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 [Rei85] 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 trivially-enabled analysis algorithm from figure 6.4. 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 from figure 6.5,  $t_{\{place(a)\}, pre(a_1)} \in T$  and  $\{(place(a), t_{\{place(a)\}, pre(a_1)}), (t_{\{place(a)\}, pre(a_1)}, place(a_1))\} \subset F$ . Therefore, there is a reachable path from  $place(a)$  to  $place(a_1)$  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 from figure 6.6. Therefore,

$t_{\{place(x)|\forall x \in A'\},pre(a_1)} \in T$  and  $\{(place(a)|\forall a \in A', t_{\{place(x)|\forall x \in A'\},pre(a_1)}), (t_{\{place(x)|\forall x \in A'\},pre(a_1)}, place(a_1))\} \subset F$ . Therefore, there is a reachable path from  $place(x), \forall x \in A'$  to  $place(a_1)$  through the transition  $t_{\{place(x)|\forall x \in A'\},pre(a_1)}$ . 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  $place(a_1)$ .  $\square$

Post-conditions with *choice* operators are treated as follows:

$$(p_1 \sqcup \dots \sqcup p_n) \models p_1 \vee \dots \vee p_n$$

A choice-expression satisfies any of the individual predicates in its choice-set. We adopt a liberal view in order to ensure progress of action executions. For example,  $(statusLight(on) \sqcup statusLight(off))$  satisfies both  $statusLight(on)$  and  $statusLight(off)$  predicates. Since the light can be in either the *on* state or in the *off* state but not in both, an action with the former post-condition satisfies actions with either of the latter pre-conditions. If the former action turns light *off*, it still enables an action that requires the light to be *on*. But since we use a Boolean Interpreted Petri net, the Boolean function associated with the transition ensures that the second action does not get executed.

$$(p_1 \sqcup \dots \sqcup p_n) \not\models p_1 \wedge \dots \wedge p_n$$

The above is trivially true by the definition of the  $\sqcup$  operator. The system can be in one of the states in the choice-set.

$$(p_1 \sqcup \dots \sqcup p_n) \models (p_1 \sqcup \dots \sqcup p_n)$$

An choice-expression satisfies itself since there is a probability of an action reaching a state that enables an action with a matching pre-condition. If the post-condition and pre-condition do not match, the Boolean function associated with the transition corresponding to the latter will not be satisfied, at runtime, and so the action is not executed.

### 6.3 Enforcement Semantics

An active space is managed by multiple policies, each designed for different aspects of the system. When they are loaded into the management system, rules from different policies may be triggered in a single situation. Since rules are created independent of rules in other policies, it would be desirable to have all rules successfully enforced. It would be unintuitive to a user if a rule failed to execute due to the triggering of a rule from

a different policy. 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.

Once dependencies among triggered rule actions have been determined, the enforcement semantics of the adaptation system specifies the execution order of actions. We have identified three different enforcement semantics for policy-based adaptation systems.

**Random:** This semantics executes rule actions in a random order. Most policy systems follow this semantics, implicitly, since they do not reason about ordering multiple triggered rules. This is the weakest of all three semantics and does not require the action workflow to be constructed. This semantics can be used when dependency among rule actions is low and very few rules are triggered by a single change.

**All-or-none:** The all-or-none semantics specifies that the rule actions in the workflow must be executed only if all actions can eventually execute. This implies that even if one action in the workflow cannot be enabled then the entire workflow should be discarded. In order to enforce this semantics, the BIPN workflow is analyzed to see if all places can be reached using a reachability algorithm [Rei85]. The all-or-none semantics provides the strongest guarantee and is useful in policies that have high dependency among rule actions.

**Maximum Rule:** The maximum rule 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. The difference between all-or-none and maximum rule enforcement semantics is that in the latter if any place in the workflow can be reached from the *Start* place it will be executed. If a place cannot be reached, the workflow is not discarded as in the all-or-none semantics. The active space management system uses the maximum rule enforcement semantics.

## 6.4 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 [Rei85]. 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 to traverse the net and execute actions. The workflow execution algorithm is shown in figure 6.4. 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.

## 6.5 Evaluation

Trivially-enabled action analysis (figure 6.4) has a linear complexity of  $O(n)$  pre-condition checks for  $n$  actions. Enablement analysis (figure 6.5) does a pair-wise satisfiability check of actions and therefore, has a quadratic complexity of  $O(n^2)$ . Partial-enablement analysis (figure 6.6) 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 an action, for all actions, the final complexity is  $O(n^2 2^n)$ . Currently, partial sets determination algorithm 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. Finally, the Petri net workflow generation algorithm (figure 6.7) 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 of Petri net workflow generation is shown in figure 6.9. The management system was executed on a Pentium(M) 1.7GHz machine with 1.0GB RAM. Figure 6.9(a) shows the overhead with varying number of triggered rules. The 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 6.9(b) illustrates the performance overhead of the system. The x-axis indicates the average number

```

in(x)           : set of incoming transitions/places of place/transition x
out(x)          : set of outgoing transitions/places of place/transition x
post(a)         : post-condition of action a
action(p)       : action of place p
bf(t)           : Boolean function of transition t
create(x, tok)  : create execution thread for transition/place x and assign token tok
send(x, tok)    : send token tok to transition/place x
recvToken(x)   : receive token from transitions/place x

begin //workflow execution
    create(Start, regular) //assign regular token to Start place
    executePlace(Start)
end

executePlace(p)
    tok = token(p)
    //execute action
    execute action(p)

    for all trans in out(p)
        if trans exists
            send(trans, tok)
        else
            create(trans, tok)
            executeTransition(trans)
        end if
    end for
end

executeTransition(trans)
    tok = token(trans)
    wait for all p in in(trans) to send tokens
        t = recvToken(p)
    end wait

    //check for Boolean function satisfaction
    if bf(t) != true
        exit
    end if

    for all p in out(trans)
        if p exists
            send(p, tok)
        else
            create(p, tok)
            executePlace(p)
        end if
    end for
end

```

Figure 6.8: Workflow Execution Algorithm

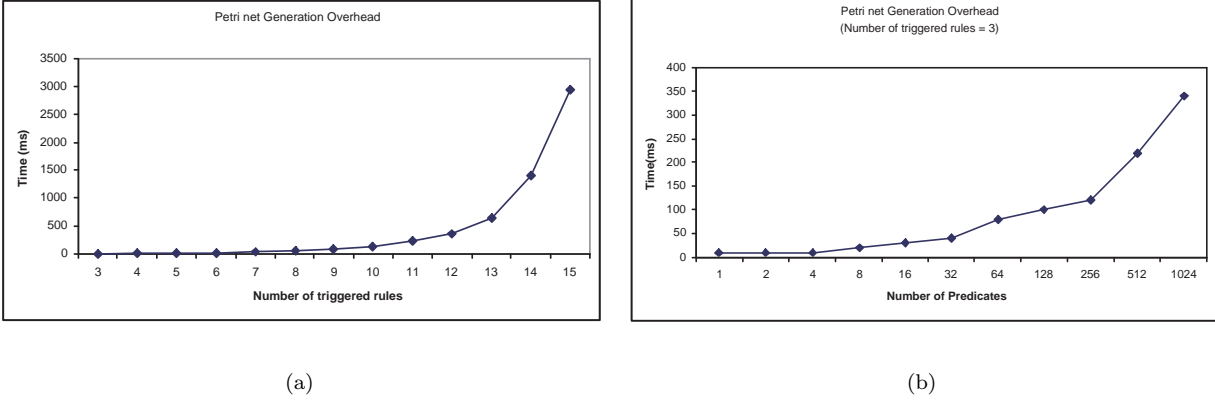


Figure 6.9: Petri net Workflow Generation Overhead (a) Overhead vs. Number of Triggered Rules (b) Overhead vs. Average Number of Predicates

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.

## 6.6 Conclusion

Existing policy-based systems enforce rules as they are triggered. When multiple rules are triggered in a single situation, these systems enforce rules in a random order, generally depending on the order of evaluation of the rules. We demonstrated in this chapter that the order of rule enforcement can determine the system behavior and therefore, randomly enforcing simultaneously triggered rules leads to unpredictable behavior. Using the ECPAP framework, we proposed approaches to determine dependencies among triggered rule actions by analyzing their pre- and post-conditions. This analysis generates a graph that captures dependencies among rule actions.

We proposed a new notion called enforcement semantics that determines how simultaneously triggered rules should be enforced and identified three different semantics that can be used in management systems. We have found the maximum rule enforcement semantics to be suitable in our active spaces. Though the initial evaluation of the semantics was encouraging more empirical evaluations should be performed to determine the efficacy of the proposed semantics.

We evaluated the performance overhead of the analysis algorithms and found one of the algorithms to have exponential complexity in the number of rules that are analyzed. For active spaces, we found that the number of triggered rules in a given situation is generally small and the overhead is negligible. But in large

enterprise systems, this overhead can be significant and so we propose model-based optimization techniques to reduce this overhead. We discuss this approach in chapter 13.

# Chapter 7

## Policies with Long-running Actions

In this chapter, we discuss the applicability of the ECPAP model for reasoning about rules with long-running actions. In section 7.1, we present our definition of long-running actions with examples. We discuss the policy problems caused by rules with long-running actions and show the shortcomings of expressing these rules with ECA framework. In section 7.2, we extend the ECPAP framework for representing rules with long-running actions. Section 7.3 presents techniques for conflict analysis with this kind of rules. We discuss the use of state models for reasoning about long-running actions in section 7.4. The chapter concludes with an evaluation of the approach.

### 7.1 Long-running Actions

Actions of certain rules execute throughout the running time of the system or for a significantly long time. These actions include actions that initiate heartbeat signals in devices, maintain certain quality-of-service, force entities to be present in certain states for an indefinite period of time and so on. It is not possible to describe these actions using Hoare logic since that requires the action to complete execution before the post-condition can be satisfied [Hoa69]. Therefore, we describe these actions using behavioral specifications using temporal logic expressions.

The semantics of the ECPAP rule  $(e, c, p) \rightarrow (a, s)$  is modified slightly as follows:

$$occ(e) \wedge c \wedge p \rightarrow exec(a)$$

$$exec(a) \rightarrow s$$

where  $exec(a)$  is the initiation of the execution of action  $a$ . The temporal expression  $s$  holds immediately after the initiation of the action  $a$ . Some example rules with long-running actions are shown below:

```

R71: on(r71, ObjectEnter(Device d))
    if(true)
    {true}
    do(InitiateHeartbeats(d, 50));
    {  $\square (occ(TimerEvent) \rightarrow ((CurrentTime - hbeatRecvTime) < 50)) \wedge \square (statusDevice(d, running))$  }

```

$R_{71}$  initiates heartbeats with an interval of 50 time units on a device that is brought into an active space. The post-condition describes that, henceforth a heartbeat is observed within every 50 time units and that the device is always in *running* state.

## 7.2 Temporal Logic Description of Long-running Actions

We use simple temporal logic representations to describe the behavior of long-running actions. Currently, we support the *henceforth* and *eventually* temporal operators [MP92]. The *henceforth* operator, represented as  $\square p$ , describes that the predicate  $p$  is always satisfied in future. The *eventually* operator, represented as  $\diamond p$ , describes that after a finite number of (possibly 0) steps  $p$  becomes true.

The syntax of the post-condition is extended to  $p_1 (\&\&/||p_k)^{k=2..m}$ , where each  $p_i$  is a first-order predicate of the form  $Q_1 t_1 \dots Q_n t_n Mpred(t_1, \dots, t_n)$ :  $Q_i$  is an optional quantifier,  $M$  is a temporal operator from the set  $\{\square, \diamond\}$  and each  $t_i$  is a constant or a variable. The keywords *.henceforth.* and *.eventually.* are supported in the post-condition expressions.

In order to support timed temporal operations, we have extended the operators to have timing constraints. These constraints specify the time in milliseconds that the operators will be applicable. For example,  $\square_{t < 6} p$ , describes that  $p$  will be satisfied for a period of 6 milliseconds. Similarly,  $\diamond_{t < 6} p$  describes that within 6 milliseconds  $p$  will be satisfied. A typical behavioral specification of an action to turn a heating device on would be

```

{true}
startHeater();
{.henceforth.(1000000) statusDevice("Heater", "on") &&.eventually.(7000) exceed("temperature", 65)}

```

This specification describes that the device *Heater* will be in *on* state for a period of 1000 seconds and within 7 seconds the temperature will exceed 65 units.

```

R72:  on(r72, ObjectEnter(Device d))
      if(true)
      {true}
      do(InitiateHeartbeats(d, 50));
      {  $\square (occ(TimerEvent) \rightarrow ((CurrentTime - hbeatRecvTime) < 50)) \wedge \square (statusDevice(d, running))$  }

R73 :  on(r73, PowerlowEvent(Device d))
      if(true)
      {true}
      do(hibernateDevice(d));
      {statusDevice(d, suspended)}

```

Figure 7.1: Policy with Conflicting Long-running Actions

### 7.3 Conflict Analysis

The ECPAP framework enables conflict analysis of policies with long-running actions. Long-running actions may affect rules that are fired in a different situation, in future, and therefore, the action constraint approach [CLN00] cannot be used to detect conflicts. Since the ECPAP framework can also describe the behavior of the action, the future effect of the action can be determined and can be used for conflict detection.

For example, consider the policy shown in figure 7.1.  $R_{72}$  initiates heartbeats on a device while  $R_{73}$  hibernates a device if the battery power goes below a certain threshold. The actions of the two rules conflict with each other since a hibernated device cannot send heartbeats. The rules are triggered on different events and therefore, traditional conflict detection techniques would fail to detect this conflict.

With the ECPAP framework, we use state models to store the state of the system as temporal logic expressions. A state model represents a system state as a set of predicates. This set is stored in a database that is updated periodically by polling and by system events. We have implemented the state model as a database with the schema  $\langle operator, time, predicate \rangle$ . An example record in the database would be  $\langle .henceforth., 3000, statusDevice(d, running) \rangle$ .

When a rule with a long-running action is enforced, the post-condition of the action is added to the state-model. The state-model supports ageing and garbage-collects predicates that have expired. The predicates in the state model are also used to monitor the action. Enforcement monitoring is discussed in detail in chapter 8.

When a rule is triggered, the predicate in the state model is used with the post-condition of the action to check for satisfiability of the post-condition constraint. The algorithm for conflict analysis is shown in figure 7.2.

This algorithm extends algorithm in figure 5.5 by including persistent state predicates from the state-model while checking any violations of post-condition constraints.

```

PC – post-condition constraint set
AS – Activation Set
S – state model
K := {}

for each rule r in AS
    K := K ∪ post(r)
end for

for each predicate s in S
    K := K ∪ s
end for

if (K ∪ PC) is not consistent
    resolve conflict

```

Figure 7.2: Algorithm for Dynamic Conflict Detection with Persistent State Predicates

## 7.4 State Model

The state model is a repository of configuration and behavioral information of various system components. The configuration information is stored as key-value pairs while the behavioral information is stored as temporal predicates. Currently, the model supports henceforth and eventually temporal operators.

### 7.4.1 Model Updation

The state model is updated by different components of the system. The enforcement verification system adds temporal predicates to the model; events from the system update the configuration information and the garbage collector checks for the validity of the temporal predicates and removes expired ones.

### 7.4.2 Garbage Collection

A garbage collector runs periodically checking for expired temporal predicates and purges them out. The garbage collector runs every 5 seconds and checks for the validity of each predicate. Once a predicate is removed from the state model, it is no longer monitored by the enforcement monitoring system.

## 7.5 Evaluation

Figure 7.3 shows the response times of the state model for query and garbage collection. Performance of queries is important since the model is queried during conflict analysis and therefore, contributes to the

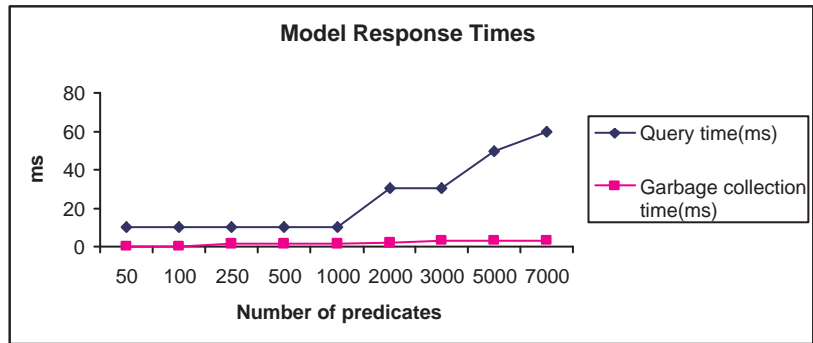


Figure 7.3: State Model Response Times

overall response time of the management system. On a Pentium(M) 1.7GHz system with 1.0GB RAM, the state model takes about 60ms for querying a model of 7000 predicates.

Garbage collection has a very low overhead. We found that to verify 7000 predicates the garbage collector takes about 5ms.

## 7.6 Conclusion

Long-running actions can affect the management process by conflicting with rules that are fired in a future situation. Axiomatic specifications describe the state of the system after an action completes execution and therefore, cannot describe long-running actions accurately. In this chapter, we extended the ECPAP framework to describe actions using temporal logic operators. This extension enables complex conflict analysis using state models. Behavioral description is also used in the next chapter for monitoring enforcement of rules with long-running actions.

The current framework only supports henceforth and eventually temporal operators. It can easily be extended with more operators for better action description and analysis.

## Chapter 8

# Enforcement Monitoring and Policy Exception Model

In this chapter, we present an exception model for policies that handles rule enforcement failures. Section 8.1 discusses policy errors. Section 8.2 presents our exception model. Verification and monitoring systems are discussed in sections 8.3 and 8.4. The exception generation system is discussed in section 8.5. We describe how exceptions are handled with ordered rule enforcement in section 8.6 and conclude with a brief summary.

### 8.1 Policy Errors

Policies, like program code, can have various errors that affect system management. These errors can occur due to wrong mapping of event-conditions with actions, failure of system components and buggy action code. Policy errors can lead the system to undesirable states and severely affect the functioning of the various system components.

Management policies separate the specification of situation-action pairs from their implementations. A policy rule describes the action that needs to be executed on a specific event-condition. The sensing of the situation and the execution of the action is implemented by the management system enforcing the policy. Therefore, a policy defines a different abstraction level. An administrator, who authors a policy, need not be concerned about the enforcement of the policy and can focus completely on describing the organization guidelines in the ECPAP framework.

Every abstraction level requires its own exception model to handle errors at that level. For example, machine level programs have exception models based on processor interrupts, programming languages support exception models using continuations [FHW01], some databases use triggers to handle exceptions and so on.

Different exception models are required at different levels since the exception handlers of lower abstraction levels may not adequately handle exceptions at higher levels. For example, programming language exceptions cannot be handled by processor interrupt handlers since programmers may not have knowledge of the processor interruption generation and may not be able to modify interrupt handler code. Therefore, the abstraction level defined by policies also necessitates an exception model since exception handlers of actions

may not handle policy exceptions appropriately. For instance, if a rule action fails, the internal exception handler of the action may log the exception, while the administrator would want to enforce an alternative action.

Bettini et al. [BJWW02] propose an approach for monitoring obligation policy enforcement. Obligation policies in their work focuses mainly on message transfers. They adopt an approach in which the monitoring system verifies the presence of a receive message for every send message in the message logs. Actions in active spaces can range from simple atomic actions to complex long-running actions. Therefore, the approach proposed in [BJWW02] would not suffice and more complex monitoring approaches have to be employed.

## 8.2 Policy Exception Model

Our management system provides an exception model for policies based on program verification techniques. Once a rule action is executed, the post-condition of the action is tested to see if it is true. If the post-condition is false, the verification system generates events that denote the exception. These events, called *exception events*, are similar to regular system events except that they are generated by the verification system and contain the enforcement context as one of their parameters.

The policy enforcement system exhibits an *iterative control behavior* - if action of one rule triggers another rule, action of the former rule is completed before that of the latter rule is initiated. The size of the control context remains constant. The context of the failure contains only the rule that failed and is independent of other rules that could have triggered the failed rule. This is in contrast to exception models of some programming languages such as C++ and Java that exhibit *recursive control behavior*. When an exception occurs, the control context contains the call stack and this necessitates propagation of exception across the stack. An iterative behavior obviates the need for propagating exceptions and this simplifies the exception model. Our policy exception model defines an exception state as a situation in which the post-condition of an executed rule action is not satisfied.

**Definition 8.1:** A policy exception,  $e$ , is a triple,  $(N, R, C)$  where  $N$  is the exception name,  $R$  is a set of exception parameters and  $C$  is a set of arguments containing the enforcement context of the failed rule.

An exception handler is designed as an ECPAP rule and is part of the management policy. In the policy in figure 8.1, rules  $R_{84}$  and  $R_{85}$  are exception handler rules. When an action fails, the verification system detects the failure and generates exception events. These events trigger exception handler rules, similar to system events triggering management rules. The exception handling system consists of the verification,

```

R81 : on(r81, ObjectEnter(Device d, Space s))
      if(statusSpace(s) == "stopped")
        {statusService(spaceRepository(s), not_running)}
        do(restartSpace(s));
        {statusSpace(s, running)}

R82 : on(r82, ObjectEnter(Device d, Space s))
      if(roleDevice(d) == "guest")
        {statusSpace(s, running)}
        do(AutoAuthorizeDeviceAndInitiateHBeats(d, s));
        {authorizationStatus(d, authorized) && heartbeatStatus(d, started)}

R83 : on(r83, ObjectEnter(Device d, Space s))
      if(deviceType(d) == "laptop" && roleDevice(d) == "guest")
        {statusSpace(s, running) && authorizationStatus(d, authorized)}
        do(mountFileSystem(d, s));
        {statusFileSystem(d, mounted)}

R84 : on(e1, authorizationFailed(Device d, RuleContext rc))
      if(roleDevice(d) == "guest" && rc.id != "e1")
        {true}
        do>PasswordAuthorizeDevice(d);
        {authorizationStatus(d, authorized)}

R85 : on(e2, heartbeatStartFailed(Device d, RuleContext rc))
      if(rc.id != "e2") //avoids loop
        {authorizationStatus(d, authorized)}
        do(InitiateHeartbeats(d, 100));
        {heartbeatStatus(d, started)}

```

Figure 8.1: Example policy with exception rules

monitoring and exception generation systems.

### 8.3 Verification System

The verification system checks if the post-condition of an action is true after the action completes execution. Since the post-condition is an expression containing propositions, some or all of the propositions may be false. The verification system determines the failed propositions and forwards them to the exception generation system along with the enforcement context of the failed rule. The verification algorithm is shown in figure 8.2. This algorithm verifies if each proposition of a post-condition expression is satisfied. If a proposition is false, it is added to the *FailedProps* set. The set is finally forwarded to the exception generation system that maps the propositions to exceptions. For example, in the policy in figure 8.1, when a “guest” laptop is brought into the active space, *myspace*, that is running, rule  $R_2$  is triggered and enforced. This causes the action *AutoAuthorizeDeviceAndInitiateHBeats*(*#laptop*, *#myspace*) to be executed. Note that *#laptop* and *#myspace* are used to indicate the argument values and do not represent

$\text{post}(a)$  : post-condition expression of action  $a$  expressed in propositional logic  
 $\text{FailedProps} = \{\}$  : set of failed propositions  
 $\text{props}(k)$  : set of propositions of a propositional expression,  $k$

for each proposition  $x \in \text{props}(\text{post}(a))$   
   if  $x$  is false  
      $\text{FailedProps} = \text{FailedProps} \cup x$

if  $\text{FailedProps} \neq \{\}$   
   send  $\text{FailedProps}$  to exception generation system

Figure 8.2: Verification Algorithm

the actual arguments. Upon completion of the action, the post-condition is verified. Since authorization of the device should be successful before heartbeats can be initiated, if authorization fails, both propositions  $\text{authorizationStatus}(\#laptop, \text{authorized})$  and  $\text{heartbeatStatus}(\#laptop, \text{started})$  are false. These propositions are sent to the exception generation system.

Expressions with the *choice* operator are evaluated as disjunctions. For example, an choice-expression  $(\text{valueOf}(x, 1) \sqcup \text{valueOf}(x, 2))$  is evaluated as  $(\text{valueOf}(x, 1) \vee \text{valueOf}(x, 2))$ . This approach of verification cannot detect exceptions if the choice-expression lists all possible states that can be reached, since the system is always in one of the states regardless of success or failure of action execution. In such situations, the choice-expression should be modified to capture some behavioral aspects of the action. For example, consider the action

```

{true}
toggleLightState();
{(statusLight(on)  $\sqcup$  statusLight(off))}

```

Since the status of the light is always *on* or *off*, failure of the *toggleLightState* action cannot be detected. The action post-condition should then contain some more information that lists the state of the light before the action was executed.

$$\{((\text{previousStatusLight}(\text{on}) \wedge \text{statusLight}(\text{off})) \sqcup (\text{previousStatusLight}(\text{off}) \wedge \text{statusLight}(\text{on})))\}$$

## 8.4 Monitoring System

The monitoring system is used to monitor long-running actions by checking the satisfaction of temporal logic expressions in the action post-condition. When a temporal logic expression is added to the active space

state model, the monitoring system interprets the expression and creates monitors that periodically verify the expression.

*Henceforth* expressions are monitored by periodic verification of the expression. The polling rate is system dependent and is of the order of a few seconds in our active spaces. If the expression is false, the enforcement system concludes that as an exception.

*Eventually* expressions are monitored by periodic verification but once the predicate is satisfied monitoring is terminated. For example, the expression *.eventually.(7000)exceed("temperature", 65)* is monitored by periodic polling for 7 seconds till the *exceed("temperature", 65)* is satisfied. Once this predicate becomes true, monitoring is terminated.

Bounded-temporal expressions are monitored for the period specified in the bound. Once the predicate expires, it is automatically removed by the garbage collector. The garbage collector notifies the monitoring thread about the expiry of the expression. If the thread is monitoring a *.henceforth.* expression, it simply terminates. If the expression is an *.eventually.* expression and is not satisfied, the predicate is assumed to have failed. In the above example, if the temperature of the heater does not reach 65 units before 7 seconds, the action *startHeater* is assumed to have failed and appropriate exception events are generated.

## 8.5 Exception Generation System

The exception generation system contains a map of predicates to exceptions. For example, *authorizationStatus(#laptop, authorized)* and *heartbeatStatus(#laptop, started)* are mapped to exceptions *authorizationFailed(Device d)* and *heartbeatStartFailed(Device d)*, respectively. The exception generation system further appends the rule context as a parameter to the exceptions and creates exception events. In the above example, the exception events *authorizationFailed(#laptop, rc)* and *heartbeatStartFailed(#laptop, rc)* are generated where the rule context *rc* contains the identifier of the failed rule - *r<sub>2</sub>*, event name - *ObjectEnter* and event parameters - *#laptop, #myspace*. A partial map of predicates to exceptions is shown in figure 8.3. Mappings can be added and removed dynamically through a user interface to the management system.

## 8.6 Handling Exceptions with Ordered Rule Enforcement

In chapter 6, we showed how the ECPAP framework can be used to analyze dependencies among multiple triggered rule actions and construct a Boolean Interpreted Petri net (BIPN) workflow. This Petri net represents dependencies and facilitates enforcement of rules according to some semantics. The workflow for

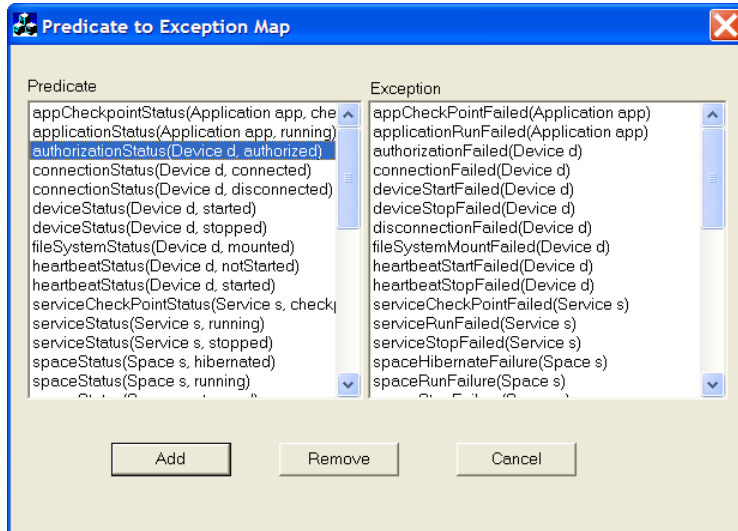


Figure 8.3: Predicates to Exceptions Mapping Interface

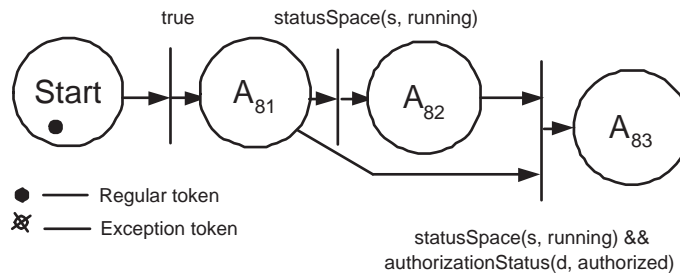


Figure 8.4: Petri net Workflow for Triggered Rules of Policy in figure 8.1

the example in figure 8.1 when rules  $R_{81}$ ,  $R_{82}$  and  $R_{83}$  are triggered is shown in figure 8.4. Place  $A_i$  contains the action of rule  $R_i$ . The pre-condition of action of  $A_{81}$  is satisfied by current system state as the space repository service will not be running in an active space that has stopped. Post-condition of action of  $A_{81}$  -  $statusSpace(s, running)$  satisfies the pre-condition of action of  $A_{82}$  and therefore, action of  $A_{82}$  can be executed after that of  $A_{81}$ . Once actions of  $A_{81}$  and  $A_{82}$  are executed, the pre-condition of action of  $A_{83}$  is satisfied and so it can be executed.

### 8.6.1 Workflow Execution

The workflow execution algorithm extends the algorithm presented in figure 6.8 by adding two kinds of tokens – *regular* and *exception*. A place generates a regular token when action associated with the place successfully completes execution. The place generates an exception token when its action fails during execution. When a transition receives an exception token, it indicates that a preceding place in the Petri net workflow generated

an exception. Since all places that follow this transition in the Petri net need to be notified of this exception, the transition immediately fires and propagates the exception token to all connected places. The algorithm is shown in figure 8.5.

Similarly, when a place receives a regular token, it executes the corresponding action, since its reception indicates that at least along one path from the *Start* place in the Petri net all actions have been successfully executed. But when an exception token is received by a place, it waits for all incoming transitions to notify exceptions before forwarding the exception. For example, the token passing sequence when action  $A_{82}$  fails is shown in figure 8.6. The *Start* place of the Petri net is initialized with a regular token. This token is forwarded to place  $A_{81}$ . The action in the place is executed and verified. Since the action was successful, a regular token is generated and forwarded to place  $A_{82}$ . Action of  $A_{82}$  is executed and verified. Since it fails, an exception token is generated and forwarded to  $A_{83}$ . On receiving the exception token, the action of  $A_{83}$  is not executed and the workflow execution terminates.

Once all places in the Petri net have been marked with tokens, the net is analyzed to see if there were any exceptions. Exception handlers are determined and the Petri net is reconstructed with the handler actions and the unexecuted actions in the original Petri net. In our example, failure of action  $A_{82}$  generates *authorizationFailed(#laptop, rc)* and *heartbeatStartFailed(#laptop, rc)* exception events that trigger rules  $R_{84}$  and  $R_{85}$  in the policy. Action  $A_{83}$  in the original Petri net was unexecuted. So the Petri net is reconstructed with actions of rules  $R_{84}$  and  $R_{85}$  (actions of  $A_{84}$  and  $A_{85}$ ) and the unexecuted action  $A_{83}$ . The workflow is reconstructed using the algorithms mentioned in the previous section. The reconstructed workflow is shown in figure 8.7. This workflow is executed again using the execution engine. Action  $A_{84}$  authorizes the device using password authentication and if successful, actions  $A_{85}$  and  $A_{83}$  concurrently start device heartbeats and mount device file system, respectively. The actions may fail and generate exception events and these are handled as above. Currently, we do not analyze the policy to determine if the exception generation terminates, though an approach similar to policy cycle detection can be used.

## 8.7 Conclusion

Exceptions in policies occur due to several reasons such as mistakes in policy design, wrong parameters and system errors. These exceptions should be detected and appropriate corrective actions must be taken for effective policy-based management. Since policies expose a different abstraction level, an exception model is required that enables a policy designer to provide corrective measures when policy enforcement fails. Existing policy-based systems do not offer any means of detecting and handling policy exceptions.

```

in(x)           : set of incoming transitions/places of place/transition x
out(x)          : set of outgoing transitions/places of place/transition x
post(a)         : post-condition of action a
action(p)       : action of place p
create(x, tok)  : create execution thread for transition/place x and assign token tok
send(x, tok)    : send token tok to transition/place x
recvToken(x)   : receive token from transitions/place x

begin           //workflow execution
    create(Start, regular) //assign regular token to Start place
    executePlace(Start)
end

executePlace(p)
begin
    tok = token(p)
    if (tok == exception)
        wait for all trans in in(p) to send tokens
        t = recvToken(trans)
        if (t == regular)
            tok = regular
            break
        end if
    end wait
end if
//execute action only if at least one token is a regular token
if (tok == regular)
    execute action(p)
    verify post(action(p))
    if exception
        determine false predicates
        tok = exception
    endif
endif
for all trans in out(p)
    if trans exists
        send(trans, tok)
    else
        create(trans, tok)
        executeTransition(trans)
    end if
end
end

```

```

executeTransition(trans)
begin
  tok = token(trans)
  if (tok == regular)
    wait for all p in in(trans) to send tokens
    t = recvToken(p)
    if (t == exception)
      tok = exception
      break
    end if
  end wait
end if
for all p in out(trans)
  if p exists
    send(p, tok)
  else
    create(p, tok)
    executePlace(p)
  end if
end

```

Figure 8.5: Workflow Execution Algorithm

In this chapter, we proposed an exception detection and handling scheme using the ECPAP framework. The post-condition of the action, available to the management system, enables verification of action execution. If the action fails, the post-condition predicates are used to generate exception events that contain the context of the rule corresponding to the failed action. This method has been extended to monitor execution of long-running actions using runtime monitoring techniques.

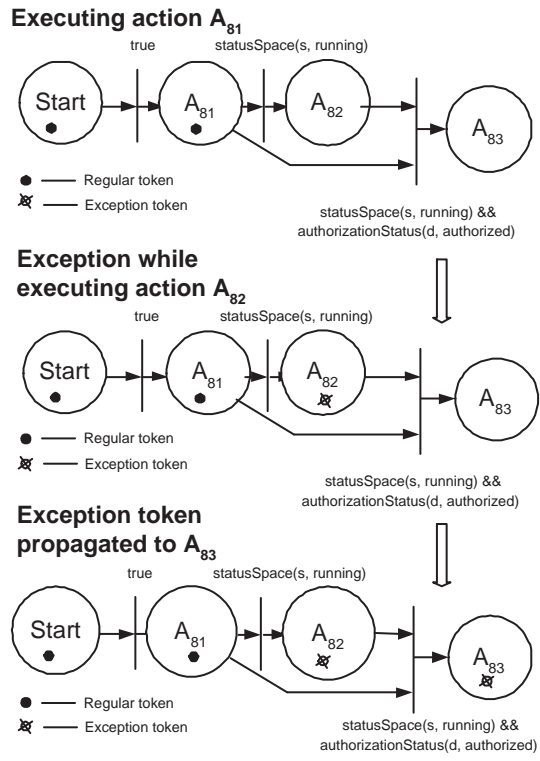


Figure 8.6: Token passing sequence with failure of action  $A_{82}$

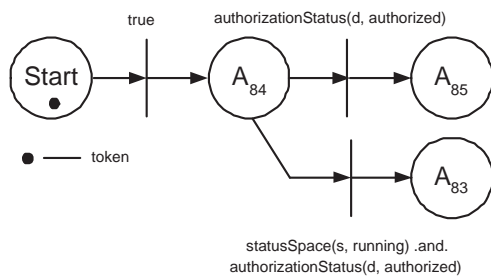


Figure 8.7: Workflow reconstructed with exception handler actions

## Chapter 9

# Towards deterministic policy-based management

Event-Condition-Action framework leads to non-determinism and therefore, provides no guarantees about the final system state that is reached when multiple rules are simultaneously triggered. This chapter discusses the non-determinism of the ECA framework in section 9.1. We prove that the ECPAP framework leads to deterministic management in section 9.2. We conclude the chapter with a discussion of the policy requirements to enable deterministic management.

### 9.1 Non-determinism of the ECA framework

As discussed in the previous chapters, the ECA framework poses several problems for policy-based management. Its inability to detect conflicts and cycles, order multiple triggered rules and handle enforcement failures leads to non-determinism. The rule evaluation process of an ECA-based management system can be modeled as a non-deterministic finite automaton (NFA). When two or more rules are triggered, the system enforces rules in a random order. Similarly, if enforcement fails, the system may reach an invalid state. Cycles can cause non-termination of the enforcement process, which cannot be modeled as a finite automaton.

Non-determinism leads to unpredictable states and can confuse the user. A management system should exhibit deterministic behavior for effective management.

### 9.2 Formal Proof of Determinism

In this section, we formally prove that ECPAP framework enables deterministic management. We show how the ordered rule enforcement and exception handling techniques can be represented as a finite state machine.

Since the policy enforcement process consists of a rule ordering phase and an exception handling phase, we prove the determinism of these phases separately. We show that the Petri net execution algorithm without exceptions is deterministic. We then introduce a new type of rule, called *sink rule*, that guarantees

termination of rule enforcement. We then argue that acyclic policies with sink rules enable deterministic enforcement.

**Theorem 9.1:** Exception-free workflow execution algorithm is deterministic.

The Petri net execution algorithm presented in figure 8.5 is deterministic if no exceptions are generated during execution. We show this by structural induction on the Petri net.

**Basis:**

The Petri net is represented as (P,T,L) where P is the set of places, T is the set of transitions and L is the flow relation.

case 1:  $P = \{Start, a^\psi\}, T = \{t_1\}, L = \{(Start, t_1), (t_1, a^\psi)\}$ , where  $a^\psi$  represents the Petri net place corresponding to action  $a$ .

The algorithm maps the above Petri net into a DFA,  $D = (Q, \Sigma, \delta, q_0, F)$  as follows:

$$Q = \{exec(Start), exec(a), Stop\}, \Sigma = \{end\}, q_0 = exec(Start), F = \{Stop\}$$

$$\delta : \{(exec(Start), end) \rightarrow exec(a), (exec(a), end) \rightarrow Stop\}$$

where  $exec(a)$  represents execution of action  $a$  and  $exec(Start)$  represents the start state of the DFA.

case 2:  $P = \{Start, a_1^\psi, \dots, a_n^\psi\}, T = \{t_1\}, L = \{(Start, t_1)\} \cup \{(t_1, a_i^\psi)\}_{i=1..n}$

The DFA for this Petri net is

$$Q = \{exec(Start), exec(\{a_i\}_{i=1..n}), Stop\}, \Sigma = \{end\}, q_0 = exec(Start), F = \{Stop\}$$

$$\delta : \{(exec(Start), end) \rightarrow exec(\{a_i\}_{i=1..n}), (exec(\{a_i\}_{i=1..n}), end) \rightarrow Stop\}$$

$exec(A)$  represents the concurrent execution of the actions in set  $A$ . Since for a place  $p$ ,  $(p)$ . represents the set of confluent actions, the order of execution of the actions does not affect the final system state and so can be considered to be deterministic.

case 3:  $P = \{Start\} \cup \{a_i^\psi\}_{i=1..n} \cup \{b^\psi\}, T = \{t_1, t_2\}$

$$L = \{(Start, t_1)\} \cup \{(t_1, \{a_i^\psi\}_{i=1..n}), (\{a_i^\psi\}_{i=1..n}, t_2)\} \cup \{(t_2, b^\psi)\}$$

This Petri net is mapped to the following DFA.

$$Q = \{exec(Start), exec(\{a_i\}_{i=1..n}), exec(b), Stop\}, \Sigma = \{end\}, q_0 = exec(Start), F = \{Stop\}$$

$$\delta : \{(exec(Start), end) \rightarrow exec(\{a_i\}_{i=1..n}), (exec(\{a_i\}_{i=1..n}), end) \rightarrow exec(b), (exec(b), end) \rightarrow Stop\}$$

**Hypothesis:** The algorithm maps a Petri net  $R = (P, T, L)$ , into a DFA

$$Q = \{exec(actions(p)), \forall p \in 2^P\} \cup \{exec(Start)\} \cup \{Stop\} \Sigma = \{end\} q_0 = exec(Start) F = \{Stop\}$$

$$\delta : \{\forall X \in places(Q - F) | (X \cdot) \cdot \neq \phi, (exec(actions(X)), end) \rightarrow exec(actions((X \cdot) \cdot)), \forall X \in places(Q - F) | (X \cdot) \cdot = \phi, (exec(actions(X)), end) \rightarrow Stop\}$$

where  $actions(K) = \{action(k)\}_{\forall k \in K}$  and  $places(R) = \{a^\psi | \forall r \in R, r = exec(a)\} \cup \{Start | \exists r \in R, r = Start\}$

**Proof:**

case 1: Extend the Petri net with a new action  $b$ , such that

$$\exists p \in P | (p \cdot) \cdot = \phi$$

$$p \cdot = p \cup t_{new}$$

$$t_{new \cdot} = t_{new \cdot} \cup b^\psi$$

The algorithm maps the Petri net into

$$Q' = Q \cup \{exec(b)\} \cup \{Stop'\}, \Sigma' = \{end\}, q_0' = exec(\{Start\}), F' = \{Stop'\},$$

$$\delta' = \delta \cup \{(Stop, end) \rightarrow exec(b)\} \cup \{(exec(b), end) \rightarrow Stop'\} \text{ which is a DFA.}$$

case 2: Extend the Petri net with a set of places  $\{b_1^\psi, \dots, b_m^\psi\}$  such that

$$\exists p \in P | (p \cdot) \cdot = \phi$$

$$p \cdot = p \cup t_{new}$$

$$t_{new \cdot} = t_{new \cdot} \cup_{i=1..m} b_i^\psi$$

Since, concurrent places in the Petri net represent confluent actions, according to the Petri net construction algorithms from chapter 6, the algorithm maps the Petri net into the DFA,

$$Q' = Q \cup \{exec(\{b_i\}_{i=1..m})\} \cup \{Stop'\}, \Sigma' = \{end\}, q_0' = exec(Start), F' = \{Stop'\},$$

$$\delta' = \delta \cup \{(Stop, end) \rightarrow exec(\{b_i\}_{i=1..m})\} \cup \{exec(\{b_i\}_{i=1..m}), end\} \rightarrow Stop'\}$$

case 3: Extend the Petri net with an action  $b$  such that

$$\exists P' \subset P | \forall p \in P', (p \cdot) \cdot = \phi$$

$$p \cdot = p \cup t_{new}$$

$$t_{new \cdot} = t_{new \cdot} \cup b^\psi$$

Since, actions in places of  $P'$  represent concurrent actions that terminate the workflow, there is a transition in the DFA of the Petri net

$$\delta(exec(actions(P')), end) = Stop$$

The algorithm replaces this transition as follows:

$\delta' = (\delta - \{(exec(actions(P')), end) \rightarrow Stop\}) \cup \{(exec(actions(P')), end) \rightarrow exec(b)\} \cup \{(exec(b), end) \rightarrow Stop\}$ , which is a DFA.  $\square$

If exceptions are generated during action execution, the verification system generates exception events. The policy should contain exception handler rules, for deterministic behavior of the system. If exception handler rules are not present, the system may reach an unpredictable state. A policy should contain exception handler rules for all possible exceptions that might be generated during action execution.

**Definition 9.1:** A policy  $P$  is said to be *exception-complete*, if  $\forall r \in P, \forall p \in post(action(r)), \exists e_x \in P|event(e_x)$  is the exception event of  $p$ .

An exception-complete policy contains exception handlers rules corresponding to each predicate in the various action post-conditions. Since actions in exception-handler rules can themselves fail, we define a type of rule called *sink rule* whose action generates no exceptions.

**Definition 9.2:** A *sink rule* is a rule, whose condition and action pre-condition evaluate to true when the rule is triggered and action post-condition evaluates to true after the action execution.

A sink rule is necessary to terminate the exception handling process. The enforcement system provides an action called  $logException(RuleContext rc)$  that logs the exception. Since this action is an internal action of the management system, it does not fail. Rules can use this action to prevent further exception generation. Figure 9.1 shows a subset of the policy of figure 8.1 extended with sink rules. This policy is also exception-complete. Rules  $R_{94}$  and  $R_{95}$  are sink rules. When  $R_{92}$  or  $R_{93}$  is triggered,  $R_{94}$  or  $R_{95}$  is also triggered, respectively. Actions of the sink rules log the exceptions and do not affect the enforcement process.

**Theorem 9.2:** The workflow execution algorithm for a non-cyclic exception-complete policy with sink rules for every exception event is deterministic.

**Proof:** Since the policy is exception-complete, the exception handling process maps one Petri net workflow into another by substituting one or more exception handler actions for every failed rule.

Let  $P/E$  represent the set of triggered rules of policy  $P$  on reception of events in  $E$  and  $ee(x)$  represent the exception events for predicates in  $x$ .

The exception handling process can be represented as a DFA  $(Q, \Sigma, \delta, q_0, F)$  where  
 $Q = \{P/E | \forall E \in 2^{Events(P)}\} \cup \{Stop\}, \Sigma = \{succ, fail\}, q_0 = \{P/E | E \subset Events(P)\}, F = \{Stop\}$   
 $\delta : \{(P/E, succ)_{\forall E \in 2^{Events(P)}} \rightarrow Stop, (P/E, fail) \rightarrow P/E' | E' = \{ee(x) | \forall r \in P/E, r \text{ is unenforced} \wedge x = post(r)\}\}$

```

R91 : on(r91, ObjectEnter(Device d, Space s))
      if(roleDevice(d) == "guest")
      {statusSpace(s, running)}
      do(AutoAuthorizeDeviceAndInitiateHBeats(d, s);
      { authorizationStatus(d, authorized)  $\mathcal{E}\mathcal{E}$  heartbeatStatus(d, started)})

R92 : on(e1, authorizationFailed(Device d, RuleContext rc))
      if(roleDevice(d) == "guest" && rc.id != "e1")
      {true}
      do>PasswordAuthorizeDevice(d);
      { authorizationStatus(d, authorized)}

R93 : on(e2, heartbeatStartFailed(Device d, RuleContext rc))
      if(rc.id != "e2") //avoids loop
      { authorizationStatus(d, authorized)}
      do(InitiateHeartbeats(d, 100);
      {heartbeatStatus(d, started)})

R94 : on(s1, authorizationFailed(Device d, RuleContext rc))
      if(true)
      {true}
      do(logException(rc));
      {true}

R95 : on(s2, heartbeatStartFailed(Device d, RuleContext rc))
      if(true)
      {true}
      do(logException(rc));
      {true}

```

Figure 9.1: Example policy with sink rules

Since the policy is non-cyclic with sink rules for every exception event, for every subset  $E \subseteq Events(P)$  there is a finite sequence of enforcement failures  $P/E \rightsquigarrow_{fail} P/E'$ , such that  $P/E'$  is a set of sink rules. Since actions of sink rules do not fail, enforcement eventually terminates.

The exception handling process maps one Petri net into another with exception handling rules. Since the exception handling process for each action in the Petri net eventually terminates, there is a finite number of Petri nets constructed for every set of events received. Since from theorem 9.1 the Petri net execution algorithm is deterministic, the workflow execution algorithm for non-cyclic exception-complete policy with sink rules for every exception event is also deterministic.  $\square$

### 9.3 Discussion

This chapter formally proves that the ECPAP framework enables deterministic policy-based management. The framework prescribes a methodology of policy design that ensures that the system reaches a well-defined predictable state after rule enforcement. If the required exception handling rules or sink rules are not present

in the policy, no guarantees can be provided by the framework. Previous policy enforcement systems were non-deterministic even if the policy contained rules for all possible exception situations.

Currently, our proofs assume the presence of exception handling rules and sink rules for all possible situations. This requirement can be relaxed if certain exception rules can be mapped to regular system events. In addition, the number of sink rules can be greatly reduced if the exception generation system can have some state information that generates pre-specified events when actions of exception handlers fail.

# Chapter 10

## Role-based Management

The dynamism of active spaces due to changing entities and configuration complicates policy design. Policies have to be modified when entities enter or exit an active space. Designing policies using roles alleviates some of these problems and simplifies policy design. In this chapter, we present a role-based management approach to designing active space policies. In section 10.1, we discuss the complexity of policy design in dynamic environments. We describe role-based policy design in security and management systems in section 10.2. We present our framework for role-based management in section 10.3 and discuss the various models in section 10.4. We present the design of the role manager in section 10.5 and conclude.

### 10.1 Complexity of Policy Design

An active space is a dynamic system with changing system components and configuration. Entities such as devices, applications and users are constantly added or removed from the system. Configuration changes necessitate frequent policy redesign to accommodate new entities and modified configurations. This increases the overhead associated with managing policy additions and removals. For example, devices and applications in an active space need to send periodic heartbeat messages on a well-defined channel to announce their presence in the space. If the device is owned by a guest user, the device should report its location periodically to the active space. Therefore, when a new device is brought into the space, the device must be configured to fulfill the above obligations. Rules must be designed for the above obligations making administration of an active space a laborious process. Similarly, when a device leaves the space these rules must be removed to avoid exceptions.

Rules can also be modified in an active space. For example, if the number of devices in an active space crosses a certain threshold, the overhead of heartbeats becomes significant. In such circumstances, the heartbeat initiation rules are modified to reduce the frequency of heartbeats. The policy system should determine the heartbeat initiation rules for the various entities and modify them to reflect the new guidelines.

Entities in an active space share a set of common obligations – all entities should send out heartbeat

messages; mobile devices should report their locations periodically to the location system [SACM05]; guest devices should authenticate themselves when they enter an active space; applications should checkpoint periodically and so on. This property can be exploited to simplify policy design and management.

## 10.2 Role-based Design

Though active space entities change frequently, the types of entities present in a space remain fairly unchanged. For instance, cell phones, PDAs and laptops are added and removed from an active space as users enter and leave the space but the entity type *Mobile Device* is always present in the set of types supported by an active space. Role-based management is an appropriate technique of decoupling entities from policy specification by introducing a logical separation using roles. A role groups related entities based on certain common characteristics. For example, all mobile devices have a common characteristic of being mobile – they have location sensors attached to them, run on battery power, normally communicate using wireless technologies and have limited resources. Similarly, all applications have common characteristics - they use the Gaia application framework [Rom03]; they can be migrated across devices; and they checkpoint their state periodically. Since many entities grouped under a single role have common obligations to fulfill, policies can be designed for roles instead of individual entities. When an entity is assigned to a role, the entity takes up the obligations associated with that role. For instance, when a laptop is added to the *MobileDevice* role, it has to fulfill the various obligations listed above.

Therefore, we introduced the notion of roles, based on entity types, for active space management. Role-based management was proposed in [Lup98] for managing distributed systems. Roles were created for different kinds of administrative positions and obligation policies were assigned to these roles. Users and automated agents assigned to these roles were required to fulfill the role obligations. We extend this notion of roles to entity types such as devices, applications, services and so on by creating a role for each type. Obligation policies and entities are assigned to these roles and entities are required to fulfill role obligations.

Role-based access control (RBAC) [SCFY96] is used extensively in computer security to assign access permissions to users using authorization policies. RBAC uses a notion of subjects, which are processes running on behalf of users [FCK95], and provides a one-to-many mapping from users to subjects. Permissions assigned to users percolate to subjects. Role-based management (RBM) extends the role concept to obligation policies. While authorization policies are based on users, obligation policies should be based on both users and entities since different entities have different obligations to fulfill even though they all belong to the same user. For example, all devices should mount their file systems onto the active space file system

when the devices are brought into the active space. This obligation cannot be assigned to the *user* role, since some devices such as mobile phones may not have a file system. In [Lup98], Lupu et al. propose a role-based management model entirely based on user roles. Obligations are assigned to a user role and all entities owned by the user must fulfill these obligations. Their model does not create roles for entities based on their capabilities. The heterogeneity of active spaces necessitates a role-based management model that groups entities not only on the obligations that they need to fulfill but also on the capabilities that they possess. Therefore, active space roles are based on entity types and user types. In this thesis, we generalize our discussion by only considering entity roles since users can also be viewed as entities with regard to roles.

Roles classify entities into groups and differ from data types. In many instances, roles can be substituted for types and therefore, it is important to distinguish between the two. We make the following distinction between roles and types :

- Organization guidelines specify the way a system should be managed and therefore, define the different management activities that should be performed by various system components. In role-based management, entities that have common characteristics and fulfill similar obligations are grouped into roles. Therefore, roles are primarily based on organization guidelines and dependent on the deployment.

Types define the set of operations that should be supported by an entity. For example, the type *Space* refers to entities that support active space operations such as *reboot*, *hibernate*, *deployService*, *stopApplication* and so on. Types are based on the managed system and are independent of the organization guidelines.

- Entities can be associated with multiple unrelated roles but with a single type. For example, a device can be in the *Mobile*, *Guest* and *IdentificationDevice* roles but can only be of type *Device*.
- The role associated with an entity can dynamically change but the type of the entity cannot.
- Roles can have constraints associated with them. These constraint can dictate the maximum number of entities that can be assigned to the role, the obligations that are inherited across role hierarchies and so on. These are discussed in more detail in later sections. Types do not support such constraints.

### 10.3 Role-based ECA Rules

Roles are created based on entity types and assigned rule templates. Rule templates are ECA rules designed with reference to roles. The keyword *role* is used in the template to denote the entity reference. When an entity is assigned to a role, the rule template gets instantiated using the entity reference. The reference

### Entity Role:

<code>on(TimerEvent())</code>		<code>on(TimerEvent())</code>
<code>if(true)</code>	<code>=====&gt;</code>	<code>if(true)</code>
<code>do(sendHeartbeat(role));</code>		<code>do(sendHeartbeat("handheld01"));</code>

### Device Role:

<code>on(ObjectEnter(role, Space s))</code>		<code>on(ObjectEnter("laptop-21", Space s))</code>
<code>if(s.type == "research-lab")</code>	<code>=====&gt;</code>	<code>if(s.type == "research-lab")</code>
<code>do(authenticate(role, s));</code>		<code>do(authenticate("laptop-21", s));</code>
<code>on(TimerEvent())</code>		<code>on(TimerEvent())</code>
<code>if(true)</code>	<code>=====&gt;</code>	<code>if(true)</code>
<code>do(reportLocation(role));</code>		<code>do(reportLocation("tablet-3105"));</code>

### Application Role:

<code>on(TimerEvent())</code>		<code>on(TimerEvent())</code>
<code>if(true)</code>	<code>=====&gt;</code>	<code>if(true)</code>
<code>do(checkpoint(role));</code>		<code>do(checkpoint("editapp"));</code>

Figure 10.1: Typical Active Space Rule Templates and Instantiated Rules

is substituted for the *role* keyword. Figure 10.1 shows a set of rule templates for various roles and the corresponding rule instantiations.

Roles are managed by an active space service called *Role Manager*. A rule template can be added and removed from a role dynamically when the active space is running using the role manager interfaces. When an entity is instantiated or brought into an active space, it is assigned to a role. Currently, we use manual techniques for role assignment. For example, when an application is started in an active space, the user chooses the role for the application through a user interface to the role manager. The role manager instantiates the rule templates and adds them to the management system. In addition, the role manager maintains a list of entities assigned to a role for all roles and is used for revocation of rules.

## 10.4 RBM Models

We define four different role-based management models that correspond to the models of RBAC [SCFY96]. The base model  $RBM_0$  contains only roles.  $RBM_1$  supports role hierarchies and enables policy inheritance.  $RBM_2$  supports role constraints that enable specification of restrictions on roles.  $RBM_3$  supports both role hierarchies and constraints. Figure 10.2 shows the relationship between different RBM models.

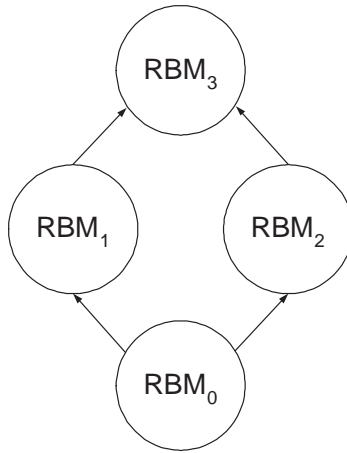


Figure 10.2: Role-based Management Models

### 10.4.1 Role Hierarchies

Roles may contain some or all obligations of another role. For example, both devices and applications need to send heartbeats to periodically announce their presence. This is a common obligation shared between the roles *Device* and *Application*. Mobile devices in addition have to periodically send their location information in addition to sending out heartbeat messages. Therefore, the obligations of the role *MobileDevice* is a super set of that of *Device*. It would be convenient if common obligations can be specified once and reused by multiple roles. Organizing roles hierarchically enables obligation reuse and makes policy management simpler. A role hierarchy defines roles that have unique obligations and may implicitly include the obligations associated with another role [FCK95]. The role hierarchy is defined using a partial order relation called *contains*. Role  $r_1$  contains role  $r_2$  if the set of obligations of  $r_1$  is a superset of that of  $r_2$ .  $r_1$  is called a *sub-role* of  $r_2$  and  $r_2$  is called a *super-role* of  $r_1$ . Sub-roles inherit obligations from super-roles. A partial hierarchy of roles used in our active space is shown in figure 10.3. The role *ActiveSpaceEntity* is a super-role for roles *Device* and *Application*. Therefore, the common obligation to send heartbeats can be assigned to the role *ActiveSpaceEntity*.

### 10.4.2 Role Constraints

Role constraints define restrictions on roles and role hierarchies. RBAC<sub>3</sub> uses constraints for separation of duty, limiting the number of users assigned to a role and enforcing various other organizational requirements [SCFY96]. Constraints can be used very effectively in role-based management to specify what rule templates can be inherited and overridden. Constraints can also be used to limit the number of entities and type of obligations assigned to a role and to specify role-based priorities for conflict resolution.

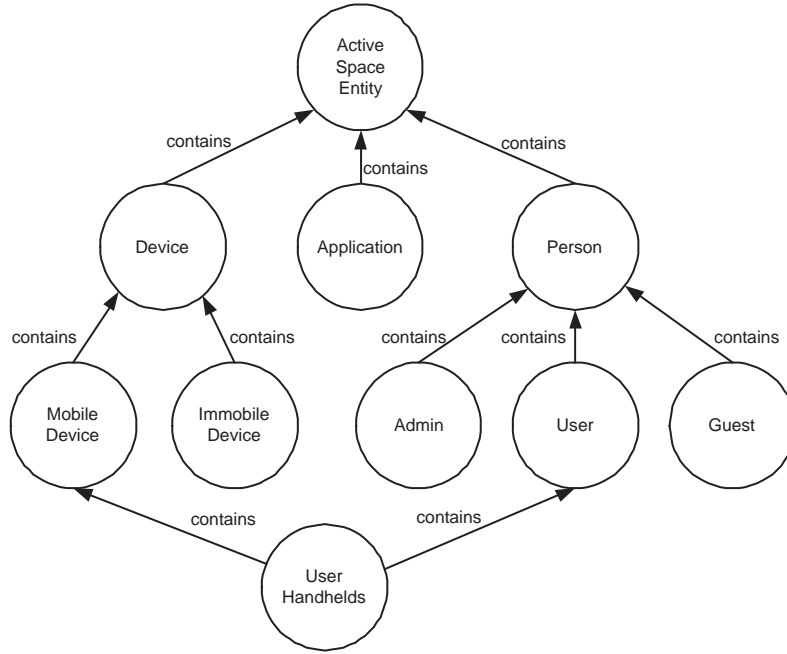


Figure 10.3: Partial Active Space Role Hierarchy

RCL2000 was proposed as a constraint language for Role-based Access Control [AS00]. RCL2000 uses first-order predicate logic expressions to specify various types of constraints on roles, users, objects and operations. To the best of our knowledge, there are no constraint languages for role-based management. In this thesis, we use a restricted form of RCL2000 to specify constraints. This restricted form limits the constraints that can be specified. A formal constraint language for role-based management is necessary and is discussed in detail in chapter 13.

The language we consider contains the following basic elements and functions:

$R$  = a set of roles  $\{r_1, \dots, r_n\}$

$E$  = a set of entities  $\{e_1, \dots, e_m\}$

$RT$  = a set of rule templates  $\{rt_1, \dots, rt_q\}$

$RH \subseteq R \times R$  is a partial order on  $R$  called the role hierarchy or role dominance relation, written as  $\preceq$

$EA \subseteq E \times R$ , a many-to-many entity-to-role assignment relation

**OBLIGATIONS**:  $R \times 2^{RT}$ , a one-to-many role-to-rule template relation

$entities(r) = \{e | (e, r) \in EA\}$

$roles(e) = \{r | (e, r) \in EA\}$

$obligations(r) = \{o | (r, o) \in OBLIGATIONS\}$

$inherit : OBLIGATIONS \times \{true, false\}$ , a one-to-one obligation-to-Boolean relation that defines if an

obligation can be inherited by a sub-role

With this language, we can express simple constraints to limit number of entities and obligations assigned to a role, stop obligation from being inherited and mandate assignment of an entity to role  $A$  if it is assigned to role  $B$ .

(a) maximum number of entities in a role

$$|entities(r)| \leq n$$

(b) maximum number of obligations assigned to a role

$$|obligations(r)| \leq n$$

(c) Stop obligation from being inherited

$$inherit(o) = false$$

(d) If entity is assigned to role  $A$ , it should also be assigned to  $B$

$$A \in role(e) \rightarrow B \in role(e)$$

## 10.5 Role Manager

Role manager is a Gaia service that is responsible for managing the life cycle of a role. It stores the various roles of an active space in a repository along with role inheritance graphs and constraints. Each role stores the rule templates associated with that role. The role manager provides interfaces to add and remove roles, assign roles to entities, add rule templates and constraints to roles and modify the inheritance graph.

An entity is assigned to a role by providing its unique reference to the role manager. Active space components are CORBA objects, which are identified by a unique reference called Interoperable Object Reference (IOR). An IOR is an encoding of the hostname, port and various other information that is used to communicate with an object. When the entity is assigned to the role, the role manager traverses the role hierarchy and collects the rule templates. At each role, the constraints associated with that role are evaluated. All rule templates are instantiated with the object's reference and loaded into the management system.

## 10.6 Policy Analysis

Rule templates can be statically analyzed to determine any conflicts or cycles. Since rule templates are instantiated into rules before enforcement, normal dynamic analysis is performed at enforcement time. In

this section, we focus on static analysis of rule templates.

### 10.6.1 Conflict Analysis

Conflict analysis on rule templates is performed similar to that discussed in chapter 5. Static matching of events and conditions are performed by treating roles as data types. For the purposes of analysis, two roles are considered equivalent if one role is a sub-role of another or there is a common sub-role for the two roles. For example, in figure 10.3 roles *Device* and *ImmobileDevice* are equivalent since the latter is a sub-role of the former. Similarly, roles *MobileDevice* and *User* are equivalent since they share the common sub-role *UserHandhelds*.

In order to match two event signatures, the analysis algorithm compares their event names and their parameters. If the parameters refer to roles, then equivalence between two roles is checked. Once events and conditions match, the algorithm performs conflict analysis similar to that described in section 5.2.

### 10.6.2 Cycle Analysis

The cycle detection algorithm is similar to that presented in section 5.3 except that equivalence checking of parameters is performed on the event signatures to generate the trigger graph. A rule triggers another rule if the event listed in the action post-condition of the former rule matches the event of the latter rule by the parameter equivalence approach defined in the previous subsection.

## 10.7 Conclusion

Role-based management is a suitable approach for managing dynamic systems such as active spaces. Roles provide a logical separation between entities and policies. Rules are designed for roles instead of entities and any entity assigned to a role should fulfill the obligations associated with that role. We have proposed a few models for role-based management similar to that of role-based access control. We have developed a simple language for specifying role constraints based on RCL2000. Initial experiences with role-based management of active spaces seems encouraging.

# Chapter 11

## Architecture and Implementation

In this chapter, we discuss the architecture and implementation details of the management system. Section 11.1 details the overall architecture and how the different pieces of the system fit together. Section 11.2 presents the policy compiler. Section 11.3 presents the runtime system and provides details about the various enforcement components. We evaluate the system in section 11.4 and present case studies of the usage of the management system on two distributed systems in section 11.5. We finally conclude the chapter with an identification of the salient features of our system.

### 11.1 Architecture

The management system consists of two main parts - static analyzer and runtime enforcement system as depicted in figure 11.1. The static analyzer is responsible for conflicts and cycles. It uses a policy compiler that takes an ECA policy as input and generates an object file. The object file is a set of Java classes corresponding to the different policy rules. Policy compilation is discussed in detail in section 11.2. The object file is compiled and loaded into the runtime system by the policy loader. The policy compiler interacts with a Prolog engine, called XSB [XSB] for propositional reasoning. A repository of actions, called *action library*, contains a set of actions that can be invoked by policy rules. The library also contains pre- and post-conditions of actions. The compiler queries the library for these specifications during static analysis.

The runtime system is made up of a number of components. The enforcement coordinator is responsible for managing the policies in the policy store, receiving events, evaluating rules, forwarding rules for dynamic analysis and finally forwarding the workflow to the execution engine and verifier. The runtime system consists of an event composer that composes primitive events into a complex event. This system is useful for sensing situations based on multiple events. For example, failure of a cluster of nodes is a combination of failures of all individual nodes. The event composer generates a cluster failure event from a set of node failure events. The policy and model store contain the rules and state models loaded into the enforcement system. The dynamic reasoning system is responsible for dynamic conflict and dependency analysis. It uses

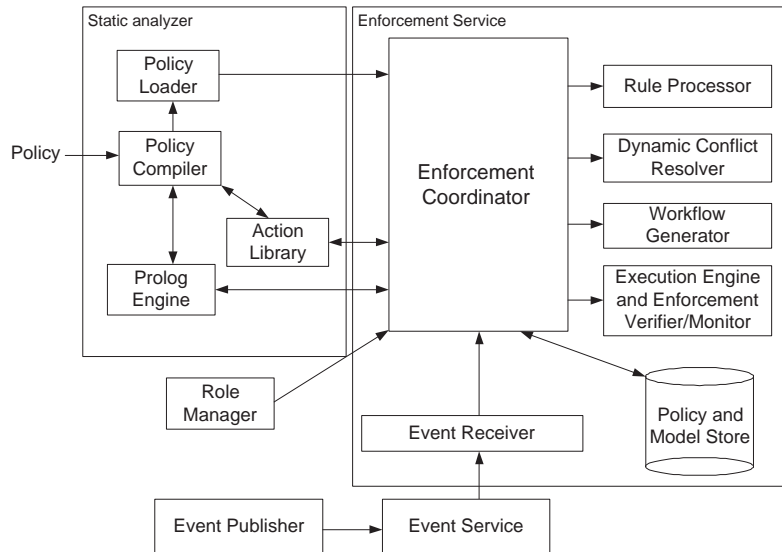


Figure 11.1: Management System Architecture

a prolog engine for analysis. The workflow executor and enforcement verifier are responsible for execution of the constructed action workflow and subsequent verification of execution, respectively.

## 11.2 Policy Compiler

The policy compiler performs static conflict and cycle analysis. It translates a policy into a set of Java classes. Each ECA rule is converted into a Java class. The event part of the rule is mapped into a method in the class, the condition part into an ‘if’ block and the action part into a set of instructions that creates an action object. Figure 11.2 shows an ECA rule and its equivalent Java class.

As shown in the figure, the event *AggregatorFail(Node n)* is mapped into a method of the same signature. The condition ( $n.id \neq \text{“FailOver”}$ ) is mapped to an *if* structure with the same relational expression. The action *useNodeAsAggregator(“FailOver”)* is mapped into a sequence of statements that create an object containing the action object and its specifications. This approach of conversion of a policy into native language code enables reuse of language types and simplifies event matching and condition expression evaluation. The Java classes are compiled and the class files are loaded into the policy enforcement system by the policy loader. The policy compiler has been designed in Java. We used the ANTLR [Par] parser generator to generate the lexical analyzer and parser from the policy grammar. The policy grammar is presented in appendix I.

The policy language reuses many of the Java types and some types specifically designed for active spaces.

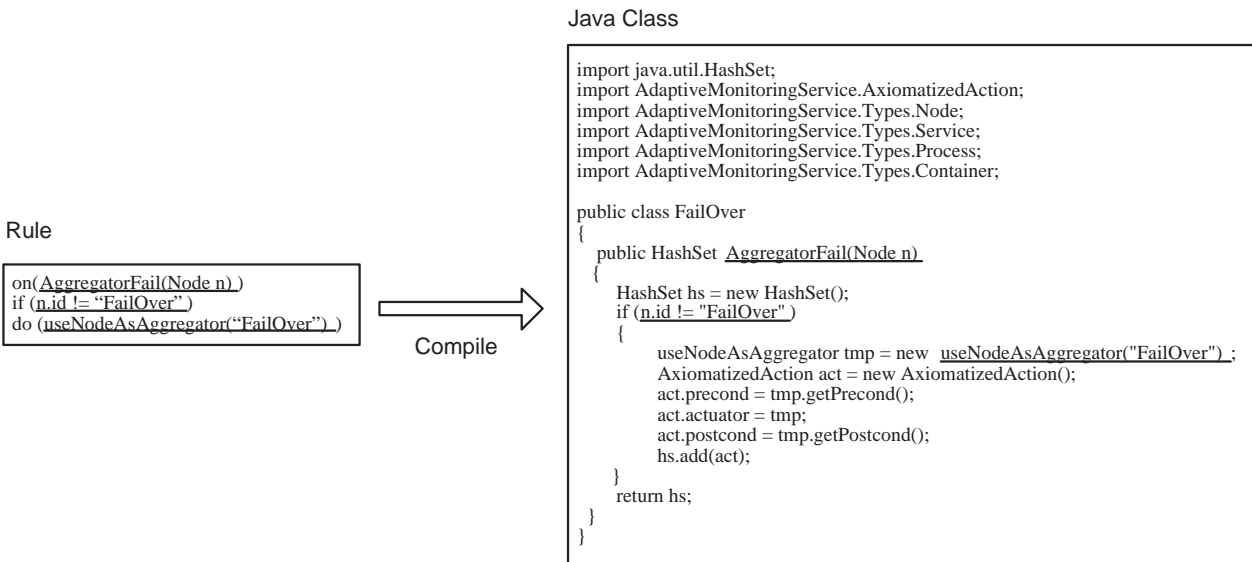


Figure 11.2: Policy Compilation

These active space types represent entities of a pervasive system such as devices, applications, services and users. These types are presented in detail in [RSAM<sup>+</sup>05].

The policy compiler uses the XSB prolog engine [XSB]. XSB is a prolog interpreter for logic programming. It supports propositional reasoning and is used during both static and dynamic analyses.

The action library is a set of actions that can be invoked by policy rules. Methods of several services or applications in the system may not expose appropriate interfaces for invocation by the policy enforcement system. The action library contains wrappers for such methods and these wrappers forward the invocations to the actual services or applications through different means such as CORBA RPC, Java RMI and socket calls.

We have provided a few tools for designing policies. The event catalog lists the various events supported in the system along with their parameters [Bor02]. The action library supports a look-up interface that lists the supported actions along with their specifications. Snapshots of the interfaces to these tools are shown in figure 11.3.

### 11.3 Policy Enforcement System

The policy enforcement system has been implemented in Java. As depicted in figure 11.1, the enforcement coordinator is central to the enforcement system and orchestrates the functioning of the other components. When a policy is loaded, the Java class objects are stored in the policy store. The event composer listens

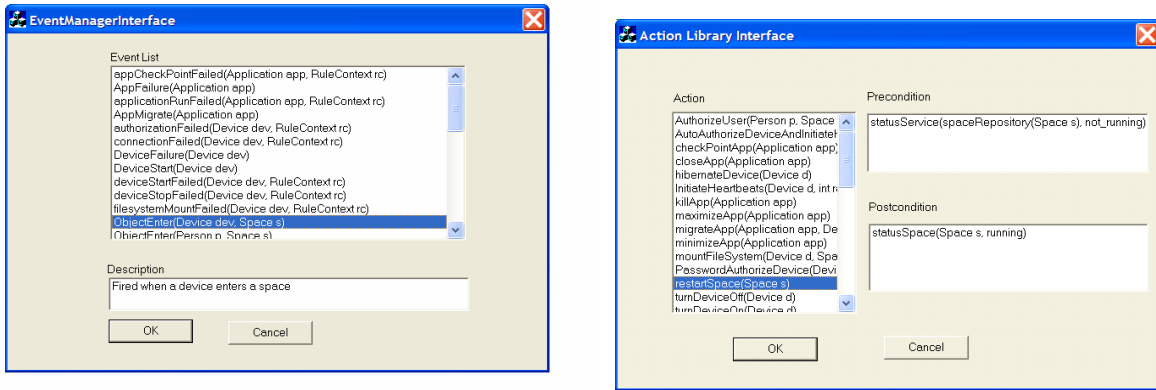


Figure 11.3: Policy Design Tools

Events : set of received events  
 Pstore : policy store  
 ActionSet = {} : set of triggered rule actions

```

forall e in Events
  forall p in Pstore
    ret := p.invoke(e)
    if (ret != fail)
      ActionSet = ActionSet ∪ ret
    endif
  endfor
endfor
  
```

Figure 11.4: Rule Evaluation Algorithm

on a well-defined event channel called *MgmtChannel*. The system assumes that all events of interest are sent on this channel. When an event is received, the event composer forwards this event to the coordinator, which constructs a method signature from it. This method is invoked on all class objects in the policy store. If the invocation is successful, the condition is evaluated. The rule object returns an object containing the action with its specifications. This object is added to an action set. The algorithm for rule evaluation is shown in figure 11.4.

The action set is sent to the dynamic analyzer that performs conflict analysis using the XSB propositional reasoning engine and constructs a workflow of actions as described in chapter 6. This workflow is executed by the workflow executor and the enforcement verification system verifies action execution by checking the post-condition or by initiating a monitoring thread.

## Rule Processor

The rule processor is a component in the enforcement service that evaluates the policy rules and maps a set of events to a set of triggered rule actions. The rule processor processes event sets in sequence and consists of an event queue to buffer events that are received when an event set is being processed.

## Dynamic Conflict Resolver

The dynamic conflict resolver detects and resolves conflicts that are found at runtime using the algorithms described in chapter 5. Resolution rules are loaded into the resolver through the enforcement coordinator interface. The resolver filters conflicting rules based on priorities and returns a non-conflicting set of actions to the enforcement coordinator.

```
class Place
{
    String placeName;
    AxiomatizedAction aa; //actions with specifications
    HashSet<Transition> connectedToTransitions; //set of Transitions that this Place connects to
    HashSet<Transition> connectedFromTransitions; //set of Transitions that connect to this Place
    Boolean triviallyEnabled = false; //flag to indicate trivial satisfaction
    int executionStatus = 0; //indicates if this place action was successfully executed once entire
                                //Petri net executes. used for exception handling.
                                //0 - unexecuted, 1 - successfully executed, 2 failed

    HashSet<RuleException> reset = null; //rule exception set - contains rule exceptions if action failed
}

class Transition
{
    String transName;
    HashSet<Place> connectedToPlaces; //set of Places that this Transition connects to
    String[] predicateSet; //a transition node contains a set of predicates that represents the
                            //pre-condition of one Petri net place (action).
                            //Each predicate in this set is a conjunct of the pre-condition
    HashSet<Place> connectedFromPlaces;
}
```

Figure 11.5: Petri net Workflow Data Structures

## Workflow Generator

The workflow generator analyzes dependencies and constructs a workflow of rule actions using the algorithms described in chapter 6. The Petri net workflow is represented as an adjacency set of places and

transitions. The Petri net data structures are shown in figure 11.5. Each *Place* structure contains a list of transitions that it connects to and a list of transitions that connect to the place. The former list is required to know the transitions that should be processed after the action in the place is executed. The latter list is required to know how many tokens are expected before the action in the place is executed. Since our workflow execution system uses two kinds of tokens – regular and exception, an action in a Petri net place fires if at least one regular token is received. The action is not executed if all transitions to the place send exception tokens. Therefore, the list of transitions that lead to the place is required. The *Place* structure also contains a flag to indicate if the place action is trivially-enabled and a flag indicating if the action was successful.

The structure *Transition* contains a list of places that lead to the transition and a list of places that the transition leads to. Since a transition fires only after all places leading to the transition fire, the former list is required. The latter list is used to forward the tokens after the transition fires. The *Transition* structure also contains a set that holds the conjuncts of the associated Boolean function. This Boolean function is evaluated before the transition fires.

### **Execution Engine and Enforcement Verifier**

The workflow executor uses the algorithm in figure 8.5 to execute the Petri net workflow. At the end of each action execution, the result from enforcement verifier determines the token flow in the Petri net. The enforcement verifier evaluates a predicate of the form  $p(x, a)$  by invoking a method call  $p(x)$  and comparing the return value with  $a$ . For example,  $statusSpace("myspace", "running")$  is interpreted as  $statusSpace("myspace") == "running"$ .

Temporal expressions are monitored by the enforcement monitoring system. Each expression has an associated monitoring thread that executes for the duration of validity of the expression. Expressions are monitored by polling or event reception. Polling is employed if the entity being monitored does not generate events on required changes. The frequency of polling is currently a few seconds but can be modified based on the monitored entity.

### **Policy and Model Store**

The policy store is a repository of rules that are loaded into the management system. Each rule is stored as a Java object and has a unique identifier that is used to access the rule. The store provides interfaces for adding, removing and querying rule objects.

The model store is a database of configuration information and persistent state predicates. The con-

```

module RoleManager
{
    interface RoleManagerService
    {
        string createRole(in string roleName);
        string connect(in string parentRole, in string childRole);
        string addRuleTemplate(in string roleName, in string ruleTemplate);
        string addRoleConstraint(in string roleName, in string constraintStr);
        short addRoleEntity(in string roleName, in string memberName);
        string assignEntity(in string roleName, in string entityId);
    };
};

```

Figure 11.6: Role Manager Interface

figuration information is stored as key-value pairs and the persistent state predicates are stored as a set. The model store is accessed by the enforcement monitoring system that creates threads to monitor each persistent state predicate.

### Role Manager

The role manager is also implemented as a database of roles, constraints and rule templates. It provides interfaces for role operations, rule template instantiation and constraint specification. The main interfaces to the role manager are shown in figure 11.6.

### Enforcement Log

The management service contains a log of the events received, triggered rules, action workflow and exceptions, if any. This log can be used for debugging policy errors and profiling the performance of the management system.

## 11.4 System Evaluation

The infrastructure of the active spaces prototype system consists of a set of desktop computers running 1.5GHz Pentium 4 CPUs with 1 GB of RAM each. These computers run the Gaia kernel with various services. A set of plasma displays and high definition screens act as interfaces to the active spaces system. Several mobile devices such as laptops, handhelds and mobile phones are part of the system. Location sensors, RFID detectors and finger print scanners have their own services that are part of the Gaia kernel services.

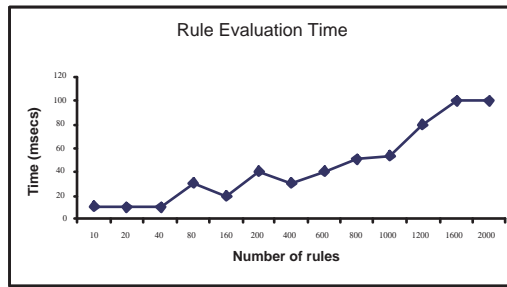


Figure 11.7: Rule Evaluation Overhead

As previously discussed, the management system consists of a set of services that are part of the Gaia kernel. The services execute on one of the kernel machines. Most of the services of the management system have been designed in Java. The other services of Gaia have been developed using C++. The management services interact with the Gaia services using CORBA communication middleware. Figure 11.7 shows the time required for rule evaluation. For a policy with 2000 rules the system takes about 100 msecs.

## 11.5 Case Studies

In this section, we present case studies of using the management system on two other distributed systems. These studies demonstrate the importance of using the ECPAP framework in non-pervasive settings. Subsection 11.5.1 presents our experience of using the system for managing a monitoring system, called Ganglia, on PlanetLab clusters. Subsection 11.5.2 demonstrates the advantages of the ECPAP framework for designing policies for a cluster file server.

### 11.5.1 Monitoring System Management on PlanetLab

In addition to using the management system for active space management, we used the system for managing a monitoring system, called Ganglia, for changes in configurations and faults.

Ganglia is a wide-area monitoring system that can be used to monitor the health of nodes in a distributed system [MCC]. Ganglia contains a data collection system and a data distribution system. The data collection system gathers performance information of a node using a daemon called *gmond*. An aggregation agent, called *gmetad*, aggregates information from multiple *gmond* daemons. The distribution system forms an overlay network of *gmetads* and *gmonds* that collectively process data and distribute it to interested clients. Ganglia has been deployed on Globus Grid [Fos05], PlanetLab nodes [STI<sup>+</sup>06] and various other commercial and academic systems.

Figure 11.10(a) shows the configuration of the Ganglia system deployed on PlanetLab nodes. A set of nodes *planetlabXX.cs.washington.edu* and *planetlabYY.millennium.berkeley.edu* form the monitored environment and host gmond daemons that collect performance data such as CPU load, memory used and so on. A node called *aggregator*, receives data from the monitored nodes and aggregates them by applying an aggregation function. This data is distributed to interested clients (archival store and performance visualizer in the figure). The configuration also contains a *Failover* node that replaces the aggregator node if the latter fails. This replacement is enforced by the policy shown in figure 11.8.

Fault detectors distributed in the system detect different kinds of failures and send out events. When the aggregator node fails, the system generates an *AggregatorFailure* event. Since the failure of the node kills all processes running on the system, the gmetad aggregation agent also fails. *AggregationAgentStopped* event is generated by the fault detectors. When the monitored nodes fail to connect to the crashed aggregator, they each send out a *DataSendFail* event. Similarly, when the archival store and visualizer nodes do not receive any data, they each send out a *DataReceptionFail* event. Therefore, a single change in the monitoring system causes 1 *AggregatorFailure*, 1 *AggregationAgentStopped*, 2 *DataSendFail* (one from each monitored node) and 2 *DataReceptionFail* (one from each client node) events.

The policy enforcement system receives the above events and evaluates the policy rules from figure 11.8. One instance each of rules  $R_{111}$  and  $R_{114}$  and two instances each of rules  $R_{112}$  and  $R_{113}$  are triggered. The system constructs the Petri net workflow shown in figure 11.9 by analyzing the dependencies. This Petri net is executed by the workflow execution system.

Figure 11.10(b) shows the output of the visualizer node during adaptation with and without ECPAP reasoning. The graphs show the CPU load and memory used over a 1 hour time window. In the first set of graphs (figure 11.10(b)(i)), when the aggregator node was stopped, multiple instances of the rules were triggered. The system did no reasoning to determine the order of rules and rules were enforced in a random order. The aggregation agent failed to start and a complete disruption in data reception was observed.

In the second set of graphs in figure 11.10(b)(ii), reasoning was enabled to determine dependencies among triggered rule actions. The enforcement of the rules was ordered based on the dependencies between actions. The figure shows a temporary disruption in data while the system recovered. The aggregation agent was successfully started and data reception resumed.

This study demonstrated the need for ordering rule enforcements when multiple rules are simultaneously triggered. The overhead of workflow construction was of the order of a few seconds which is a negligible overhead for a management system.

```

R111: on(r111, AggregatorFailure(Node n))
if(n.id != "FailOver")
{ statusNode("FailOver", running)
do(useNodeAsAggregator("FailOver"));
{ statusAggregator(running)}

R112 : on(r112, DataSendFail(Node n))
if(n.type == "MonitoredNode")
{ statusAggregator(running)}
do(reconnectToAggregator(n));
{ connectionStatusToAggregator(n, connected)}

R113 : on(r113, DataReceptionFail(Node n))
if(true)
{ statusAggregator(running)}
do(reconnectToAggregator(n));
{ connectionStatusToAggregator(n, connected)}

R114 : on(r114, AggregationAgentStopped(Node n))
if(true)
{ statusAggregator(running) ∧ ∀x ∈ MonitoredNodes, connectionStatusToAggregator(x, connected)
∧ ∀x ∈ ClientNodes, connectionStatusToAggregator(x, connected) }
do(restartAggregationAgent());
{ statusService("AggregationAgent", running)}

```

Figure 11.8: Ganglia Management Policy

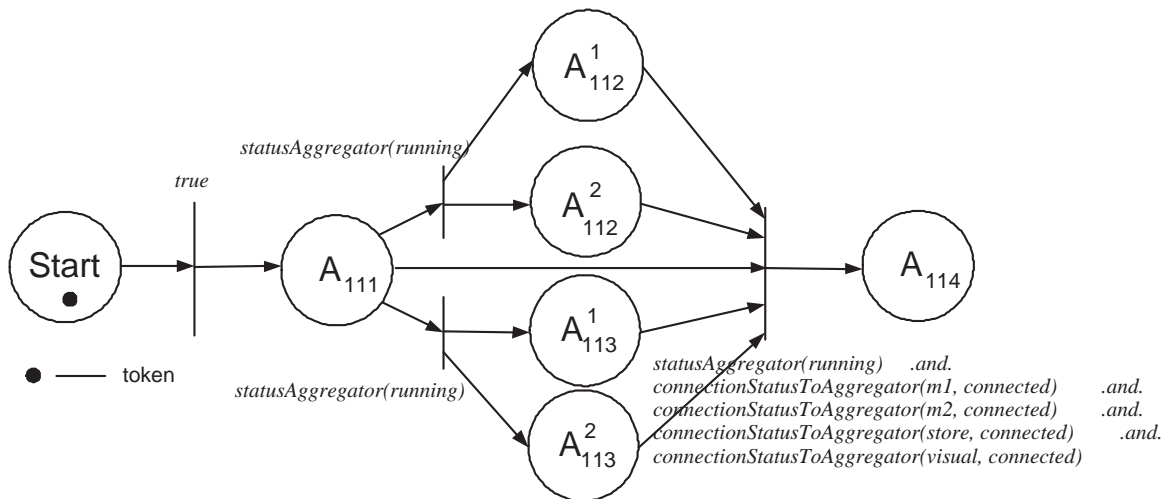
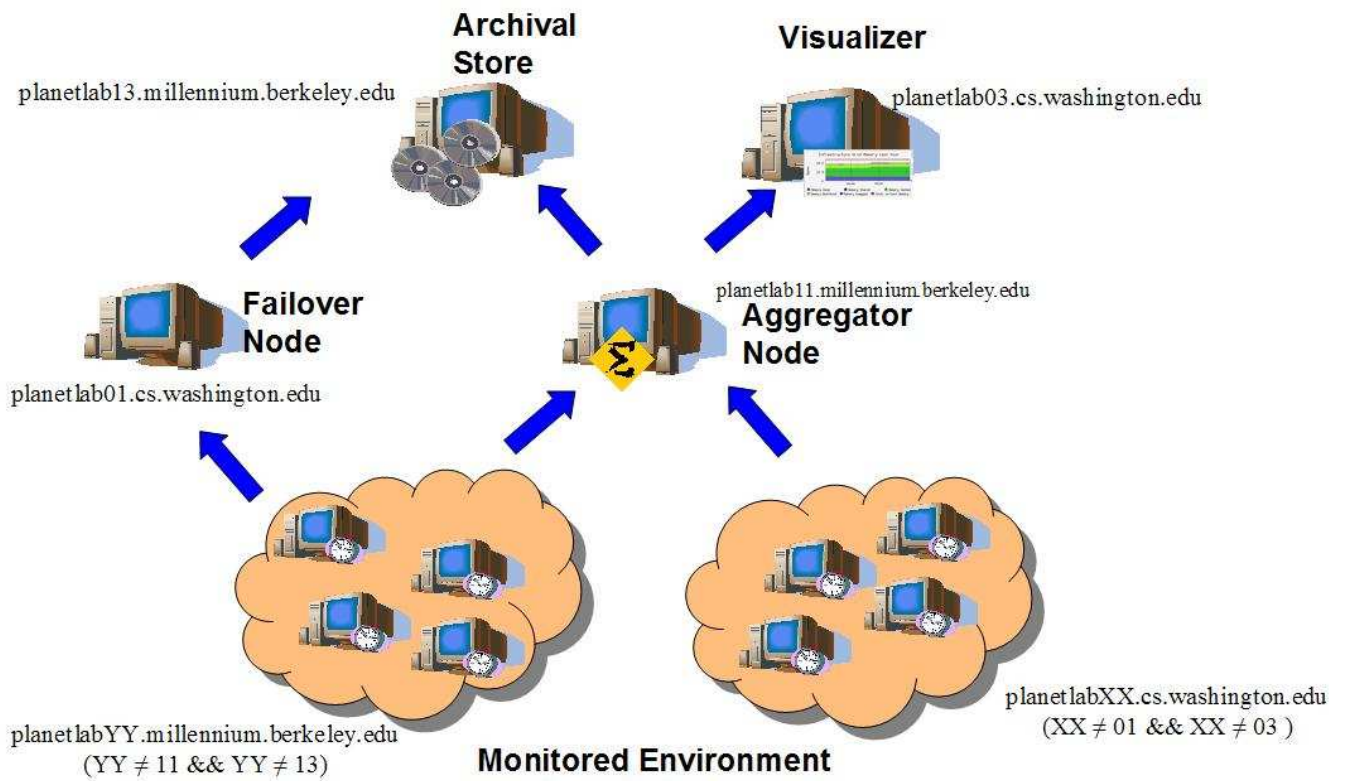
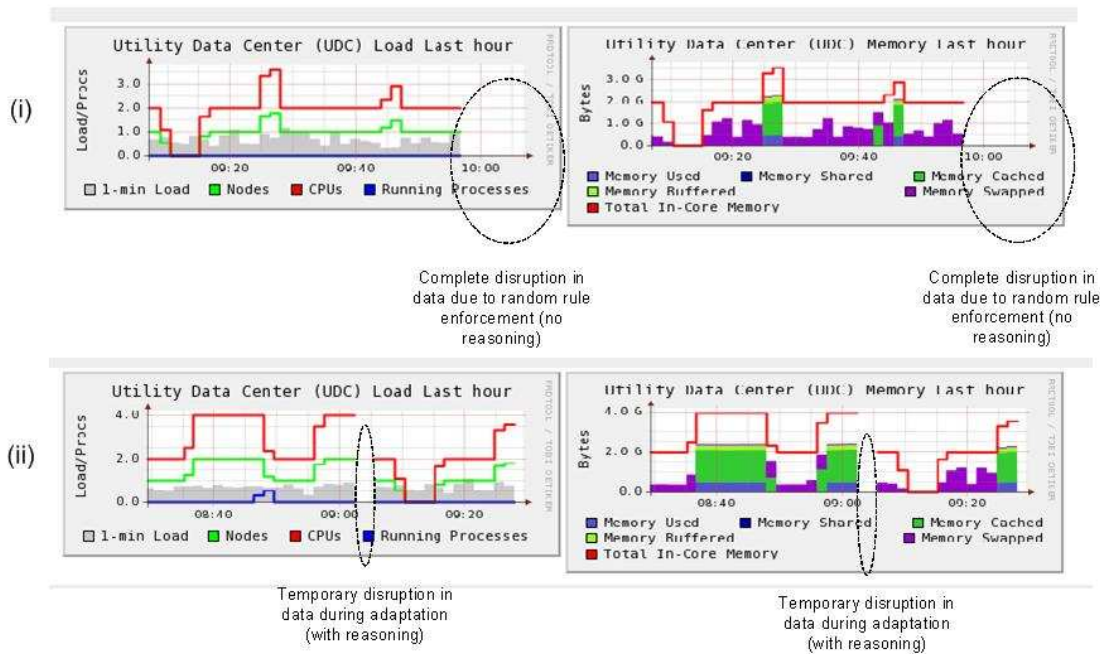


Figure 11.9: Petri net Workflow



(a)



(b)

Figure 11.10: (a) Ganglia monitoring system deployed on PlanetLab nodes (b) Disruption in monitored data during adaptation (as perceived by the visualizer node)

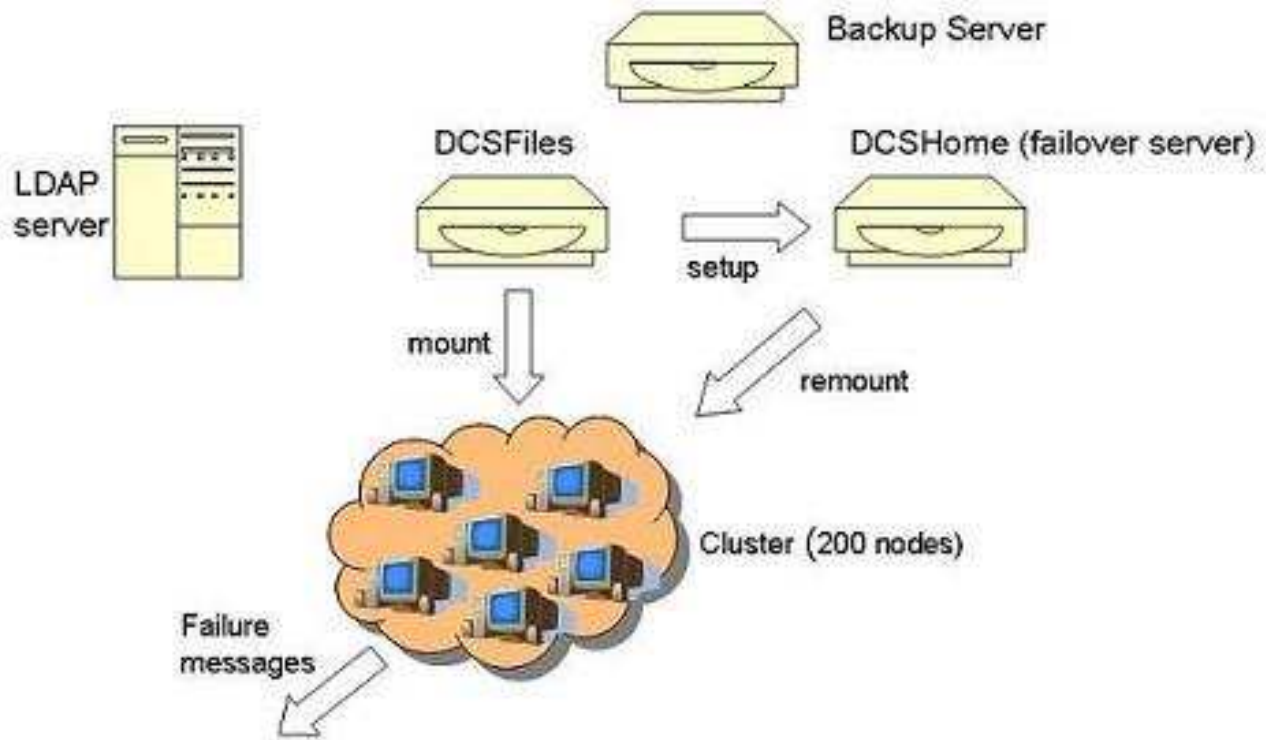


Figure 11.11: Cluster File Server Configuration

### 11.5.2 Cluster File Server Management

Script-based management has been traditionally used to manage computational resources, network components and file systems. System management scripts are programs that execute on system level triggers such as timers, file system failures, virus notifications and so on. They are seldom executed by individual users.

Policies provide a natural extension to scripts by capturing situation-action pairs as rules. Most scripts can be designed as policies. In addition, policies provide a centralized view of the different management activities of a system and thus enable complex reasoning. The problems faced by script-based management generally apply to policy-based systems as well. We studied how a cluster file system in the Computer Science department at UIUC is managed using scripts.

The file system serves a cluster of 200 nodes that run various applications for course assignments, class lectures and so on. The configuration of the system is shown in figure 11.11. The system consists of one main file server called *DCSFiles* that is mounted on all nodes of the cluster. Another file server called *DCSHome* acts as a fail over server for *DCSFiles*. A central LDAP server contains credentials to access the file server contents. Each of the nodes contact the LDAP server to obtain credentials prior to accessing

Management scripts		
Script	Execution Situation	Description
check_csil_linux	On demand	Logs into different machines using ssh and runs specified command
post-install	Triggered after package install	Turns smartd off. Turns sendmail off
update	Every night	Updates any new node additions to cluster
logwatch	Every night	Checks all logs and mails results
mapper	Every night	Maps www.cs.uiuc.edu/home/... to /dcs/grads/...
radiate	On demand	Collect anti-virus run status from various nodes
automount	On disk access failure	Automatically mounts fail-over file system onto client file system
setup	On over 100 node failure	Assigns DCSHome server as the new file server
transfer	On demand	Transfers mount points

Figure 11.12: Cluster File Server Management Scripts

*DCSFiles*. Periodically, the data in *DCSFiles* and *DCSHome* are backed up into the backup server.

When a node fails to access the file server, an email is sent out to the administrator. If the LDAP server fails, the administrator receives multiple emails, one from each node trying to access the file server. If the number of failure messages received is over 100, the file system is assumed to have failed and the backup server (*DCSHome*) is mounted as the new server on all nodes. The scripts used to achieve this task are listed in figure 11.12.

In the present system, the administrator coordinates between the different scripts and so avoids various potential conflicts and cycles among the scripts. If the administrator intervention has to be reduced, management should be more automated. This increases the possibility of occurrence of these problems. We discuss the various problems that might arise and how the ECPAP framework would be useful in those situations.

### Policy Conflicts

Currently, the file server contains two rules to handle failures:

**R<sub>115</sub>** : If number of failure messages is over 100 and *DCSFiles* is the file server, mount *DCSHome* as the file server

**R<sub>116</sub>** : If LDAP server fails, shutdown backup server

When the LDAP server fails, each node sends out a failure message as *DCSFiles* server is inaccessible. In addition, an LDAP failure message is also sent out. Therefore, both rules have to be enforced. When *DCSHome* is mounted as the new file server, all data on the server is treated as new and backed up. But the LDAP server failure shuts down the backup server causing a conflict between the two rules. Since

the administrator is currently involved, the rules are executed in sequence, with additional scripts executed manually by the administrator to restart the LDAP server and the backup server before rule  $R_{115}$  is enforced. If the management was automated, the two rules would conflict.

Designing the rules with the ECPAP framework would enable detection of such conflicts. The backing up of data of *DCSHome*, when it is mounted as the new file server, is an effect of the mounting action. The post-condition of the action would contain this information and the fact that the backup server is running during this operation. The post-condition of  $R_{116}$  would specify that the backup server would be stopped after the corresponding action executes. A simple post-condition constraint specifying that the backup server should not be in the running and stopped states simultaneously would detect this conflict.

### Policy Cycles

If a third rule is added to the above policy to assign *DCSFiles* as the file server if *DCSHome* failed, on LDAP failure, it would result in an oscillation.

$R_{117}$ : If number of failure messages is over 100 and *DCSHome* is the file server, assign *DCSFiles* as the file server.

When the LDAP server fails, either  $R_{115}$  or  $R_{117}$  is triggered depending on the current file server. When one of the servers is assigned as the new server, the nodes try to contact the newly mounted server and fail. This triggers the mounting of the other server. This oscillation takes place continuously. Currently, rule  $R_{117}$  is not part of the system policy but a cycle can result if the rule is added accidentally.

The ECPAP framework would be able to detect cycles in such scenarios. The fact that the nodes try to access the server on mounting is expressed as a post-condition of the mount operation. This action may cause a failure event and this can be expressed as the post-condition of the node-access operation. A trigger graph can be generated from this information and the cycle can be detected.

### Problems with policy/script-based management

In addition, there are many other issues with script-based management that are applicable to policy-based management systems as well. We summarize them below.

- **Script errors:** Many management scripts are developed by third-party vendors who may not have tested them well. These scripts may fail during management requiring human intervention. Some scripts may handle errors unfavorably making error detection a difficult task. A model for exception handling of scripts is required to reduce administrator intervention.
- **Automated situation analysis:** Concluding the cause of an error in a system is a very laborious

process. This requires automated situation analysis. In addition, management systems should support triggering rules based on combinations of different system events.

- **Ordered execution of scripts:** Many scripts have to be executed in a specific order to manage the system as specified by the organization guidelines. For example, kernel upgrades have to be made before applications and services are upgraded. This requires certain ordering of management activities. Currently, administrators determine the proper order and it would be desirable to automate this process.

This study revealed that many existing systems managed by scripts face some of the problems that we have addressed in this thesis. Currently, administrators frequently intervene and coordinate the management process. If total automation of management has to be achieved, a framework for complex reasoning about management activities should be designed. The ECPAP framework seems to address some of the issues and is a promising step towards automated management.

## 11.6 Conclusion

In this chapter, we presented the architecture and implementation details of our management system. Some of the salient features of the architecture that differentiates it from architectures of other existing policy-based management systems are :

- **Rule translation into native language code :** The policy compiler translates ECA rules into Java classes. This enables type reuse and simplifies event matching and condition expression evaluation.
- **Policy debugging support :** The management system provides support for policy debugging by logging events, triggered rules, order of actions executed in the workflow executor and results of enforcement verification. By extending the logging information with timing information, the performance of the management system can also be profiled.
- **Tools for policy design :** Our management system provides tools to look up events and actions that are supported by the active space. This greatly aids policy design.

The initial evaluation of the system revealed negligible overhead from rule evaluation and enforcement. The correctness guarantees provided by specification-enhanced rules justifies the extra overhead offered by the system for analysis. User studies have to be conducted to determine the effort required to write action specifications.

In addition, we presented case studies of employing the ECPAP framework for managing two other distributed systems. The studies demonstrated the benefits of using the framework for policy-based management.

# Chapter 12

## Related Work

This chapter positions our work in relation to other research efforts on policy-based management of pervasive and distributed systems. Various policy systems and frameworks have been developed in the past and these systems focus on different aspects of a management system. We identify past research results related to each component of our system and provide detailed comparisons.

### 12.1 Static Policy Analysis

Static analysis of a policy is performed prior to the policy being loaded into the management system. Many research projects have focused on static analysis of policies for conflict analysis [CLN00, LS99]. Chomicki et al. [CLN00] 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 that are free of 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. Lupu et al. [LS99] classify conflicts into modality and application-specific conflicts. Modality conflicts arise when two or more policies with opposite modalities refer to the same subjects, actions and targets. Application specific conflicts are situations that arise due to external criteria such as limited resources and are enforced using organization constraints. An example of this type of conflict is when a mobile device has to concurrently execute multiple resource intensive applications, but the constraint limits the number of resource intensive applications to 1. None of the above approaches to conflict detection detect conflicts that can occur due to side-effects of actions. Detecting such conflicts requires formal specifications of actions and therefore, the ECPAP framework is well suited for it.

While most research works on static policy analysis focus on conflict analysis and resolution, none of the research works deal with termination analysis. Determining a cycle in a set of policy rules requires information about events that are generated due to action execution and therefore, the ECA framework

is unable to detect such cycles. The ECPAP framework can list the set of events generated by the action execution in the action post-condition and this enables termination analysis.

Agrawal et al. [ACG<sup>+</sup>05] perform dominance and coverage checks on policies in their work. Dominance checks verify if situation of one rule is completely dominated by that of another such that the former is never triggered. Coverage checks verify if a set of rules enforce intended policies on the whole managed system. Both checks are based on ECA rules and therefore, can be directly applied on ECPAP rules. Dominance checks can be extended to verify if action of one rule completely dominates the action of another by verifying if post-condition of one action satisfied post-condition of another. Such checks are useful to eliminate the execution of one action when both actions are eligible for execution.

## 12.2 Dynamic Analysis

Dynamic analysis of policies refers to the analysis that is performed during policy enforcement. Most policy enforcement systems perform some kind of dynamic analysis. Dunlop et al. [DIR02] propose an approach to detect conflicts that occur due to temporal aspects of actions and roles. Conflicts in their system are defined as situations in which two rules have common subjects and targets and their actions affect common system components. The scope of their conflict detection technique is on common subjects and targets. Our work on conflict detection generalizes the approach and eliminates the need to specify subjects and targets explicitly, since many management rules do not specify them.

Policy management for autonomic computing (PMAC) [Kam05] project from IBM research uses priorities at runtime for conflict resolution [Ver05]. In addition, the project performs various analysis to determine dominance and coverage checks. We limit our research to dynamic conflict analysis and resolution using resolution rules, though our work can be extended with the above analyses.

While most policy research works focus on conflict analysis at runtime, none of the projects have addressed the problem of determining enforcement order when multiple rules are simultaneously triggered. To the best of our knowledge, our research work [SC06] is the first effort to address this important problem. As demonstrated in earlier chapters, determining enforcement order and providing enforcement guarantees are crucial and our work forms the first step in that direction.

## 12.3 Policy Verification/Monitoring

Verifying policy enforcement is important for effective policy-based management. Bettini et al. [BJWW02] propose an approach for monitoring obligation policy enforcement. Their system model uses obligations

based on message transfers and therefore, their monitoring system monitors if a send action exists for every receive action by inspecting the history of messages sent by the service. While their work focuses on monitoring communication of obligation policies, successful completion of action execution is not verified. Actions can fail due to various reasons and this should be verified and corrective actions should be taken for reliable policy enforcement. Our research work uses well-developed ideas from runtime verification to determine the success of an action execution and also proposes an exception model for policy-based management.

## 12.4 Model-based Management

Model-based management is an orthogonal approach to policy-based management that has been successfully used for performance, security and file system management [UVS<sup>+</sup>04, LSMK99]. A mathematical model of the desired state or behavior of the managed system is created and whenever the system deviates, corrective actions are initiated to restore the system to the desired state or behavior. Model-based management is a declarative approach to management, in that the desired system property is expressed but not the flow of actions required to reach the property.

Model-based management is feasible for systems whose behavior can be easily expressed as models. Behavior of complex systems, such as active spaces, with several dynamic and heterogeneous entities is hard to describe using models. Policy-based management is a suitable approach in those circumstances since it provides an imperative approach to specifying corrective actions in different situations.

Our research work combines some elements of model-based and policy-based management. We use models to store configuration and persistent state information. This simplifies policy evaluation and enables reasoning about rules with long-running actions.

## 12.5 Role-based Management

Role-based management has been introduced by Lupu et al. [Lup98] in the Ponder project. In their work, roles are assigned to users and all entities belonging to a user should fulfill the associated obligations. This approach is not suitable for pervasive systems with heterogeneous entities since different devices have different capabilities. For example, if a user in a guest role has to authorize on entry into an active space, all devices used by the user need to authenticate themselves. Some passive devices, such as location badges, may not have the computing/communication capability necessary for authentication. Therefore, active space roles should be based on entity and user types as we have presented in our work.

Role-based access control (RBAC) [SCFY96] is used extensively in computer security to assign access permissions to users using authorization policies. RBAC uses a notion of subjects, which are processes running on behalf of users [FCK95], and provides a one-to-many mapping from users to subjects. Permissions assigned to users percolate to subjects. Role-based management (RBM) extends the role concept to obligation policies. We extend the models of RBAC to RBM in our work and define inheritance and constraints for roles.

## 12.6 Active Database Rules

Active databases extensively use ECA-based frameworks for database triggers [DBB<sup>+</sup>88, BW00]. Database triggers are actions that are fired in response to database events such as record insertions, deletions and updates. Actions used in ECA frameworks are normally simple database operations whose effect on the system is well-known. For example, an action that inserts a record in a database increases the number of database rows by 1. When actions are simple and their effects are implicitly known, ECA-based frameworks are sufficient.

ECA frameworks for management systems containing complex actions whose effects are not implicit. This requires specifying the effects explicitly using some formal specification. Therefore, an ECPAP framework is well-suited in such circumstances.

Baralis and Widom [BW00] have done extensive research on confluence and termination analysis of database rules using ECA and condition-action (CA) rules. They have proposed properties to determine confluent database actions. The scope of their work is limited to database rules and does not cover management rules.

## 12.7 Other Projects

The Ponder project from Imperial College has contributed extensively to policy research . The project has designed the Ponder language for policy specification [DDL01], developed techniques for conflict analysis [LS99], proposed role-based management techniques [Lup98] and ways for policy refinement using abduction [BLMR04]. The problems addressed in their research differ considerably from that of ours. They do not address conflicts due to action effects, do not offer techniques for cycle and enforcement order analysis, do not support an exception model or support policies with long-running actions. Their work on role-based management is also not directly applicable to pervasive systems.

Kagal et al. [KFJ03] have developed a language called Rei for specifying pervasive system policies. The

focus of their work is on language design and conflict detection. They do not address many of the issues such as enforcement ordering, long-running actions and exception handling as we have addressed.

# Chapter 13

## Future Work

In this chapter, we discuss some future research directions of this work and suggest possible approaches. Our initial results in some of the suggested directions have been encouraging. We do not describe the results here as many issues are yet to be investigated in detail.

### 13.1 Policy Profiling and Debugging

Policy error detection and handling is a nascent topic and our exception model is one of the first efforts in that direction. Our work focuses on runtime error detection and handling. It would be more desirable to detect policy errors statically, at compile time where the policy designer can correct the errors. Similarly, policies need to be profiled to determine various parameters such as rule evaluation time, dynamic analysis overhead and other bottlenecks in policy enforcement.

The ECPAP framework can be used to develop profiling and debugging tools for policies. The action specifications enable determining cascading rule enforcements and final system states after enforcement of a set of rules. This can be possibly be used to determine policy errors and bottlenecks.

### 13.2 Policies with Composed Events

Our enforcement system accepts a set of events as a situation and evaluates the policy. The event correlation system assumes a simple correlation model based on epochs [CLN00]. This model cannot sense all situations and complex event correlation models should be developed based on event relations [OJC97, KYY<sup>+</sup>95] and cause-effect graphs [Gru98]. The effect of these models on policy enforcement is an open research issue that needs to be addressed.

### 13.3 Policies and Models

The performance of the evaluation system can be improved if information about the configuration or state of the system is known. For example, the current system queries individual nodes of a cluster to determine their states while evaluating the pre- and post-condition expressions. This process is expensive requiring multiple remote procedure calls to various nodes. If instead, the states of the entities can be cached in a state model of the system, the model can be queried to evaluate the expressions. The model can be updated independently. Similarly, if multiple entities trigger instances of the same rule on a change, information about the configuration of the system can be effectively used to reduce the number of rules that are evaluated during analysis. Therefore, policies and models can be effectively used to reduce the management response times and communication overhead.

Currently, we use models to store the persistent system state. This can be extended to store the entire system configuration and used during evaluation and analysis.

### 13.4 Incorrect Specifications

In this thesis, we assume that the action specifications are correct. This may not always be a reasonable assumption to make. Incorrect specifications can greatly affect the performance of the system by wrongly flagging conflicts, ordering action execution improperly and raising unwanted exceptions. Dealing with incorrect specifications is an important research problem that is being addressed in model-checking, planning and goal-based programming. Handling incorrect specifications is an open problem that should be addressed in the ECPAP framework.

### 13.5 Behavioral Specifications

In this thesis, we have mainly focused on actions with axiomatic specifications. Most reasoning techniques are applicable to actions with simple pre- and post-conditions. Actions with behavioral specifications were introduced in chapter 7, where we showed how behavioral information expressed using temporal logic enables reasoning about long-running actions. More complex reasoning can be performed and stronger guarantees can be provided if detailed behavioral specifications are provided. Currently, we only support simple temporal operators. Extending the specification with more temporal operators would enable better guarantees.

## 13.6 Policy Interpretation

Policy interpretation is the process of determining the goal a set of rules would achieve and is the inverse of policy refinement. Interpretation is required to determine if a set of rules conforms to the stated guidelines and would be more intuitive to an administrator who wants to know the result of enforcing the set of rules on a system. Interpretation is an unaddressed problem in policy research and would be an interesting direction to pursue.

## 13.7 Constraint Language for Role-based Management

In chapter 10, we presented a simplified version of RCL2000 for specifying constraints for roles. RCL2000 was designed for RBAC systems and is not sufficiently powerful to specify constraint for role-based management systems. A language is required to express constraints on entities, roles, rule templates and role hierarchies. A formal constraint model should be developed to study the properties and power of the language.

## 13.8 Empirical Validation of Approaches

Our evaluations of the various algorithms and approaches have been mainly theoretical analyses of the complexities and corresponding performance estimations. Our evaluations lack empirical justifications, which is important before these techniques can be employed in practical settings. For example, our ordering approach to rule enforcement is based on dependency analysis. Empirical investigations are required to determine if the proposed maximum enforcement semantics is the desired semantics in a system. Similarly, we propose an exception model that reconstructs the workflow on failure. Empirical validation is required to verify if this approach is appropriate. Nevertheless, our framework and techniques provide a groundwork for such validation and form an important initial step in that direction.

# Chapter 14

## Conclusion

In this thesis, we presented a framework for specifying management policies for complex distributed systems with particular emphasis on pervasive systems. We summarize the presented work and draw some final conclusions in this chapter.

### 14.1 Summary

Pervasive computing opens up interesting opportunities by extending the physical environment with the ability to access information and use computing resources anywhere and anytime. Electrical appliances and consumer devices form components of a spatially distributed system and can be programmatically controlled by users. The past few years have witnessed the development of many prototype pervasive systems and several architectures, services and applications were created for different scenarios. The success of these efforts has motivated the commercial deployment of these systems in homes, offices and other environments.

Deployment of these systems necessitates a management framework to enforce guidelines set by the host organization. Policy-based management is an approach of enforcing such guidelines using reaction rules. These rules specify the action to be executed in a given situation and are designed using the Event-Condition-Action (ECA) framework. This framework expresses a situation as a combination of event and condition and specifies the corrective action that should be executed when the event and condition are observed. Existing policy-based systems are based on the ECA framework.

Typically, policies are designed for different components of a pervasive system by different administrators. When they are combined by a policy enforcement system many issues may arise. Policy rules may conflict, cause cycles, trigger in a random order, fail during enforcement and so on. Formal specifications of rule actions are required to address these issues and so the ECA framework is poorly suited for designing policy rules for pervasive systems.

In this thesis, we presented a specification-enhanced rule framework called Event-Condition-Precondition-Action-Postcondition (ECPAP) for designing policy rules. This framework combines formal specifications

of actions with rules. We demonstrated the ability of this framework for conflict analysis, cycle detection, determining enforcement order when multiple rules are triggered and proposed an exception model for policies. We showed how the ECPAP framework can be used for reasoning about rules with long-running actions.

The ECA framework leads to non-deterministic policy enforcement. Determinism is essential for effective policy-based management. In this thesis, we showed how the ECPAP framework enables deterministic policy-based management and formally proved that the ECPAP rule enforcement, using the presented algorithms, can be modeled as finite state machines.

The dynamism of active spaces complicates policy design as rules have to be updated when entities of a space enter or exit the space. We extended the ECPAP framework with roles that makes policy design simpler. Policies are designed for roles and a rule gets instantiated when an entity is added to a role.

We evaluated the various algorithms presented in this thesis theoretically and empirically. In addition, we evaluated this framework on two other distributed systems and presented case studies. These evaluations demonstrated the benefits and feasibility of the framework.

## 14.2 Contributions

The main contribution of this thesis is a framework for developing management rules for complex distributed systems. The specific contributions are :

- A framework for policy rule specification that incorporates formal specifications of actions.
- Techniques for conflict, cycle and rule ordering analysis.
- Exception model for policies.
- Support for policies with long-running actions.
- Role-based management approach for pervasive systems.
- Evaluation of the framework and case studies of its usage on two distributed systems.

In addition, we identified various research directions that can be pursued with the ECPAP framework and suggested possible approaches, in this thesis.

# Appendix I

## Policy Language Grammar

```
/*
 * Policy grammar
 *
 * Author: Chetan Shankar
 */
options {
    language="java";
}
//Parser grammar
class PolicyParser extends Parser;
options {
    buildAST = true;
}
startPolicy: policyNameDecl startRule;
policyNameDecl: "PolicyName" ^IDENT SEMI!;
startRule: (rule)* EOF!;
rule: "on" ^LEFT_PARAN! eventexpr RIGHT_PARAN!
    cond_expr
    "do" ^LEFT_PARAN! actionexpr RIGHT_PARAN! SEMI! ;
cond_expr: COND_EXPR ^;
eventexpr: IDENT ^(LEFT_PARAN! parameterDeclarationList RIGHT_PARAN!)? ;
// A list of formal parameters
parameterDeclarationList
    :      (parameterDeclaration (COMMA ^parameterDeclaration)* )?;
//A formal parameter
```

```

parameterDeclaration
    : IDENT ^IDENT;

actionexpr
    : dottedIdentifierList LEFT_PARAN! paramList RIGHT_PARAN!;

dottedIdentifierList
    : IDENT (DOT ^IDENT)*;

paramList
    : (param (COMMA ^param)*)?;

param
    : constant
    | dottedIdentifierList;

constant
    : STRING_LITERAL
    | CHAR_LITERAL;

class PolicyLexer extends Lexer;

options {
    k = 4; //four characters of lookahead
}

IDENT: ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')+;
SEMI: ';';
LEFT_PARAN: '(';
RIGHT_PARAN: ')';
LEFT_BRACE: '{';
RIGHT_BRACE: '}';
COMMA: ',';
DOT: '.';

// Whitespace ignored
WS    : (' '
        | '\t'
        | '\f'
        // handle newlines
        | (options {generateAmbigWarnings=false;}

```

```

:      "\ r \ n"
|      '\ r' // Macintosh
|      '\ n' // Unix (the right way)
)
{ newline(); }
)+
{ $setType(ANTLR_USE_NAMESPACE(antlr)Token::SKIP); };

// Single-line comments
SL_COMMENT
:      "//"
      ( ('\ n' | '\ r')* ('\ n' | '\ r'('\ n')?)?
      { $setType(ANTLR_USE_NAMESPACE(antlr)Token::SKIP); } ;

// multiple-line comments
ML_COMMENT
:      "/*"
      ( { LA(2)!= '/' }? '*' | '\ n' { newline(); } | ('*' | '\ n'))*
      "*/"
      { $setType(antlr::Token::SKIP); };

COND_EXPR
:      "if("
      (
      options {
          generateAmbigWarnings=false;
      }
      :
      '\ r' '\ n' {newline();}
      | '\ r' {newline();}
      | '\ n' {newline();}
      | ('\ n' | '\ r' | ')')
      )*
      ")";

// character literals

```

CHAR\_LITERAL

```
:      '\ ' ( ESC | ~('\ " | '\ n' | '\ r' | '\ \ ' ) ) '\ ' ;
```

// string literals

STRING\_LITERAL

```
:      "" (ESC | ~("" | '\ ' | '\ n' | '\ r'))* "" ;
```

protected

ESC

```
:      '\\ '
      ( '\ n' | '\ r' | '\ t' | '\ b' | '\ f' | '\ ' | '\ \ '
      | ('\ u')+ HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
      | '\ 0'..'3'
      (
          options {
              warnWhenFollowAmbig = false;
          }
      : '\ 0'..'7'
      (
          options {
              warnWhenFollowAmbig = false;
          }
      : '\ 0'..'7'
      )?
      )?
      | '\ 4'..'7'
      (
          options {
              warnWhenFollowAmbig = false;
          }
      : '\ 0'..'7'
      )?
      )
;

```

```

protected
HEX_DIGIT
    :      ('0'..'9' | 'A'..'F' | 'a'..'f');
class PolicyTreeWalker extends TreeParser;
options {
    k = 4;
}
policyname returns [ char *name]
{
}
    : #("PolicyName" pn:IDENT) {name = strdup(pn->getText().c_str()); };
paramdecl returns [char *retPList]
{
}
    : #(a:IDENT b:IDENT) {
        int len1 = strlen(a->getText().c_str());
        int len2 = strlen(b->getText().c_str());
        retPList = (char*) malloc(len1 + 1 + len2); //1 - for the comma
        strcpy(retPList, strdup(a->getText().c_str()));
        strcpy(retPList+len1, " ");
        strcpy(retPList+len1+1, strdup(b->getText().c_str()));
    };
paramdeclexpr returns [char *retPList]
{
    char *p, *prest;
}
    :      #(COMMA p=paramdecl prest=paramdeclexpr) {
        int len1 = strlen(p);
        int len2 = strlen(prest);
        retPList = (char*) malloc(len1 + 1 + len2); //1 - for the comma
        strcpy(retPList, p);
        strcpy(retPList+len1, ",");
    };

```

```

        strcpy(retPList+len1+1, prest);
    }
    |      (p=paramdecl) {retPList = strdup(p);};
eventexpr returns [ char *a]
{
    char * pdecl;
}
:      #(eventname:IDENT (pdecl=paramdecl)* ) {
        int len1 = strlen(eventname->getText().c_str());
        int len2 = strlen(pdecl);
        a = (char*) malloc(len1 + 2 + len2);
        strcpy(a, strdup(eventname->getText().c_str()));
        strcpy(a+len1, "(");
        strcpy(a+len1+1, pdecl);
        strcpy(a+len1+1+len2, ")");
    };
onexpr returns [ char *a]
{
    char * estr;
}
:      #("on" estr=eventexpr ) {a = strdup(estr);};
ifexpr returns [ char *a]
{
    char * x;
}
:      #("if" x=onexpr act:IDENT ) {a = strdup(act->getText().c_str());};
ifonexpr returns [ char *a]
{
    char * ifon;
}
:      #("if" ifon=onexpr IDENT ) {a = strdup(ifon); };
paramList returns [ char *retPList]

```

```

{
    char *paramrest1, *paramrest2, *paramVal3;
}

:      #(COMMA paramrest1=paramList paramrest2=paramList) {
        int len1 = strlen(paramrest1);
        int len2 = strlen(paramrest2);
        retPList = (char*) malloc(len1 + 1 + len2); //1 - for the comma
        strcpy(retPList, paramrest1);
        strcpy(retPList+len1+1, ",");
        strcpy(retPList+len1+1, paramrest2);
    }
|      (paramVal1:STRING_LITERAL) {retPList=strdup(paramVal1->getText().c_str());}
|      (paramVal2:CHAR_LITERAL) {retPList=strdup(paramVal2->getText().c_str());}
|      (paramVal3=dottedIdentList) {retPList=strdup(paramVal3);};
dottedIdentList returns[ char *retDIL]
{
    char * rest;
}

:      #(DOT rest=dottedIdentList id:IDENT ) {
        int len1 = strlen(rest);
        char *tmp = strdup(id->getText().c_str());
        int len2 = strlen(tmp);
        retDIL = (char *) malloc(len1 + 1 + len2);
        strcpy (retDIL, rest);
        strcpy(retDIL + len1, ".");
        strcpy(retDIL + len1 + 1, tmp);
    }
|      (idl1:IDENT) {retDIL= strdup(idl1->getText().c_str());};
params returns [ char *a]
{
    char * x, *act, *params;
}

```

```

:          #("do" x=newCondExpr act=dottedIdentList (params=paramList)?) {a = strdup(params); };
action returns [ char *a]
{
    char * x, *act,*p;
}

:          #("do" x=newCondExpr act=dottedIdentList (p=paramList)?) {a = strdup(act); };
cond returns [ char *a]
{
    char * condval,*d,*p;
}

:          #("do" condval=newCondExpr d=dottedIdentList (p=paramList)?) { a = strdup(condval);};
event returns [ char *a]
{
    char * onval,*d,*p;
}

:          #("do" onval=onexpr d=dottedIdentList (p=paramList)?) {a = strdup(onval); };
//===== new rules for modified grammar - treating if statement as a string
newCondExpr returns [char *a]
{
    char * estr;
}

:          #("on" estr=eventexpr x:COND_EXPR) {a=strdup(x->getText().c_str());};

```

# References

- [Abo99] Gregory D. Abowd. Classroom 2000: An Experiment with the Instrumentation of a Living Educational Environment. *IBM Systems Journal, Special issue on Pervasive Computing*, 38(7):508–530, October 1999.
- [ACG<sup>+</sup>05] Dakshi Agrawal, Seraphin Calo, James Giles, Kang won Lee, and Dinesh Verma. Policy Management of Networked Systems and Applications. In *Ninth IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*, May 2005.
- [ACH<sup>+</sup>01] Mike Addlesee, Rupert Curwen, Steve Hodges, Joe Newman, Pete Steggles, Andy Ward, and Andy Hopper. Implementing a Sentient Computing System. *Computer*, 34(8):50–56, 2001.
- [Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
- [AHS05] Artur Andrzejak, Ulf Hermann, and Akhil Sahai. FEEDBACKFLOW – An Adaptive Workflow Generator for Systems Management. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 335–336, Washington, DC, USA, 2005. IEEE Computer Society.
- [AM05] Jalal Al-Muhtadi. *An Intelligent Authentication Infrastructure for Ubiquitous Computing Environments*. PhD dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 2005.
- [AMSRC04] Jalal Al-Muhtadi, Chetan Shankar, Anand Ranganathan, and Roy Campbell. Super Spaces: A Middleware for Large-Scale Pervasive Computing Environments. *Perware 04: Proceedings of the Second Annual IEEE International Conference on Pervasive Computing and Communications Workshops*, 00:198, 2004.
- [AS00] Gail-Joon Ahn and Ravi S. Sandhu. Role-based authorization constraints specification. *Information and System Security*, 3(4):207–226, 2000.
- [BJWW02] Claudio Bettini, Sushil Jajodia, Xiaoyang Sean Wang, and Duminda Wijesekera. Obligation Monitoring in Policy Management, 2002.
- [Bla03] Matt Blaze. Cryptology and Physical Security: Rights Amplification in Master-keyed Mechanical Locks, March 2003.
- [BLK00] Randeep Bhatia, Jorge Lobo, and Madhur Kohli. Policy Evaluation for Network Management. In *INFOCOM (3)*, pages 1107–1116, 2000.
- [BLMR04] Arosha Bandara, Emil Lupu, Jonathan Moffett, and Alessandra Russo. A Goal-based Approach to Policy Refinement. In *POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*, page 229, Washington, DC, USA, 2004. IEEE Computer Society.
- [Bor02] Bhaskarjyothi Borthakur. Distributed and Persistent Event System for Active Spaces. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

- [BW00] Elena Baralis and Jennifer Widom. Better Static Rule Analysis for Active Database Systems, 2000.
- [CLN00] Jan Chomicki, Jorge Lobo, and Shamin Naqvi. A Logic Programming Approach to Conflict Resolution in Policy Management. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 121–132, San Francisco, 2000. Morgan Kaufmann.
- [Dam02] Nicodemos C. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD dissertation, Imperial College, London, 2002.
- [DBB<sup>+</sup>88] Umeshwar Dayal, Barbara T. Blaustein, Alejandro P. Buchmann, Upen S. Chakravarthy, Meichun Hsu, R. Ledin, Dennis R. McCarthy, Arnon Rosenthal, Sumil K. Sarin, Michael J. Carey, Miron Livny, and Rajiv Jauhari. The HiPAC project: Combining Active Databases and Timing Constraints. *SIGMOD Rec.*, 17(1):51–70, 1988.
- [DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. *Lecture Notes in Computer Science*, 1995, 2001.
- [DIR02] Nicole Dunlop, Jadwiga Indulska, and Kerry Raymond. Dynamic Conflict Detection in Policy-Based Management Systems. *Sixth International Enterprise Distributed Object Computing Conference (EDOC'02)*, page 15, 2002.
- [FCK95] David Ferraiolo, Janent Cugini, and Rick Kuhn. Role Based Access Control: Features and Motivations. In *Proceedings of the 11th Annual Computer Security Applications Conference (CSAC '95)*, 1995.
- [FHW01] Daniel P. Friedman, Christopher T. Haynes, and Mitchell Wand. *Essentials of Programming Languages (2nd ed.)*. Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.
- [Fos05] Ian Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *IFIP International Conference on Network and Parallel Computing*, pages 2–13, 2005.
- [GDL<sup>+</sup>04] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System Support for Pervasive Applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.
- [Gru98] B. Gruschke. Integrated Event Management: Event Correlation using Dependency Graphs. In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, 1998.
- [HC03] Christopher K. Hess and Roy H. Campbell. A Context-Aware Data Management System for Ubiquitous Computing Applications. *icdcs*, 00:294, 2003.
- [Hes03] Christopher Hess. *The Design and Implementation of a Context-Aware File System for Ubiquitous Computing Applications*. PhD dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 2003.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [HPO] Hewlett Packard OpenView Event Correlation Service.
- [HV99] Michi Henning and Steve Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [ISO89] ISO/IEC 7498-4:1989, Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 4: Management framework, 1989.

- [Kag04] Lalana Kagal. *A Policy-Based Approach to Governing Autonomous Behavior in Distributed Environments*. PhD dissertation, University of Maryland, 2004.
- [Kam05] David Kaminsky. An introduction to policy for autonomic computing. Technical report, 2005.
- [KFJ03] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment, 2003.
- [KMPP02] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. Conflict Detection and Resolution in Access Control Policy Specifications. In *Foundations of Software Science and Computation Structure*, pages 223–237, 2002.
- [KOA<sup>+</sup>99] Cory D. Kidd, Robert Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair MacIntyre, Elizabeth D. Mynatt, Thad Starner, and Wendy Newstetter. The Aware Home: A Living Laboratory for Ubiquitous Computing Research. In *Cooperative Buildings*, pages 191–198, 1999.
- [KP88] G. Krasner and S. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.
- [KSK06] Swaroop Kalasapur, Kumarvel Senthivel, and Mohan Kumar. Service Oriented Pervasive Computing for Emergency Response Systems. In *PERCOMW '06: Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops*, page 517, Washington, DC, USA, 2006. IEEE Computer Society.
- [KYY<sup>+</sup>95] S. Klinger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A coding approach to event correlation. In *Proceedings of the fourth international symposium on Integrated network management IV*, pages 266–277, London, UK, UK, 1995. Chapman & Hall, Ltd.
- [LBN99] Jorge Lobo, Randeep Bhatia, and Shamim A. Naqvi. A Policy Description Language. In *AAAI/IAAI*, pages 291–298, 1999.
- [LS99] Emil C. Lupu and Morris Sloman. Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering*, 25(6):852–869, November/December 1999.
- [LSMK99] Ingo Lück, Marcus Schönbach, Arnulf Mester, and Heiko Krumm. Derivation of Backup Service Management Applications from Service and System Models. In *DSOM '99: Proceedings of the 10th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, pages 243–256, London, UK, 1999. Springer-Verlag.
- [Lup98] Emil Lupu. *A Role-Based Framework for Distributed Systems Management*. PhD dissertation, Imperial College, London, 1998.
- [MCC] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation And Experience.
- [M.D93] M.D.Abrams. Renewed Understanding of Access Control Policies. In *Proceedings of 16th National Computer Security Conference*, Baltimore, Maryland, U.S.A, 1993.
- [MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.
- [NYGS] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Tolerating Correlated Failures in Wide-Area Monitoring Services. Technical Report IRP-TR-04-09, May.
- [OJC97] T. Oates, D. Jensen, and P. R. Cohen. Automatically Acquiring Rules for Event Correlation from Event Logs. Technical Report UM-CS-1997-014, , 1997.
- [Par] Terence Parr. ANTLR: Another Tool for Language Recognition.

- [PCAR02] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet, 2002.
- [PJKF03] Shankar R. Ponnkanti, Brad Johanson, Emre Kiciman, and Armando Fox. Portability, Extensibility and Robustness in iROS. In *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, page 11, Washington, DC, USA, 2003. IEEE Computer Society.
- [PPK<sup>+</sup>03] Claudio Pinhanez, Mark Podlaseck, Rick Kjeldsen, Anthony Levas, Gopal Pingali, and Noi Sukaviriya. Ubiquitous Interactive Displays in a Retail Environment. In *Proceedings of SIGGRAPH'03 Sketches*, San Diego, California, 2003.
- [RAS<sup>+</sup>04] Anand Ranganathan, Jalal Almuhtadi, Chetan Shankar, Roy Campbell, and M. Dennis Mickunas. MiddleWhere: a middleware for location awareness in ubiquitous computing applications. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 397–416, New York, NY, USA, 2004. Springer-Verlag New York, Inc.
- [RBKI04] N. Ravi, C. Borcea, P. Kang, and L. Iftode. Portable Smart Messages for Ubiquitous Java-enabled Devices, 2004.
- [RC] Manuel Roman and Roy H. Campbell. A Middleware-Based Application Framework for Active Space Applications.
- [RC04] Anand Ranganathan and Roy H. Campbell. Autonomic Pervasive Computing Based on Planning. *icac*, pages 80–87, 2004.
- [Rei85] Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
- [RHC<sup>+</sup>02] Manuel Roman, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: A Middleware Infrastructure to Enable Active Spaces. In *IEEE Pervasive Computing*, pages 74–83, 2002.
- [Rom03] Manuel Roman. *An Application Framework for Active Space Applications*. PhD dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 2003.
- [Rou94] B.N. Roussev. Self-checking Implementation of Boolean Interpreted Petri Nets. *IEEE Symposium on Emerging Technologies and Factory Automation*, 1994.
- [RSAM<sup>+</sup>05] Anand Ranganathan, Chetan Shankar, Jalal Al-Muhtadi, Roy H. Campbell, and M. Dennis Mickunas. Olympus: A High-Level Programming Model for Pervasive Computing Environments. *percom*, 00:7–16, 2005.
- [RSC04] Anand Ranganathan, Chetan Shankar, and Roy Campbell. Mobile Polymorphic Applications in Ubiquitous Computing Environments. *mobile*, 00:402–411, 2004.
- [SACM05] Chetan Shankar, Jalal Almuhtadi, Roy Campbell, and Dennis Mickunas. Mobile Gaia: a middleware for ad-hoc pervasive computing. In *IEEE Consumer Communications and Networking Conference (CCNC 2005)*, Las Vegas, January 2005.
- [Sam05] Geetanjali Sampemane. *Access Control for Active Spaces*. PhD dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 2005.
- [SC05] Chetan Shankar and Roy Campbell. A Policy-based Management Framework for Pervasive Systems using Axiomatized Rule-Actions. *nca*, 0:255–258, 2005.
- [SC06] Chetan Shiva Shankar and Roy H. Campbell. Ordering Management Actions in Pervasive Systems using Specification-enhanced Policies. In *PerCom*, pages 234–238, 2006.

- [SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.
- [SG05] Pete Steggle and Stephan Gschwind. The Ubisense Smart Space Platform: A ubisense white paper, 2005.
- [Slo94] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360, 1994.
- [SNC02] Geetanjali Sampemane, Prasad Naldurg, and Roy H. Campbell. Access Control for Active Spaces. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 343, Washington, DC, USA, 2002. IEEE Computer Society.
- [SRC05] Chetan Shiva Shankar, Anand Ranganathan, and Roy Campbell. An ECA-P Policy-based Framework for Managing Ubiquitous Computing Environments. *mobile*, 0:33–44, 2005.
- [Sri05] Biplav Srivastava. The Case for Automated Planning in Autonomic Computing. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 331–332, Washington, DC, USA, 2005. IEEE Computer Society.
- [SS97] R. Sandhu and P. Samarati. Authentication, Access Control, and Intrusion Detection. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 1929–1948. CRC Press, 1997.
- [STI+06] Chetan Shankar, Vanish Talwar, Subu Iyer, Yuan Chen, Dejan Milojicic, and Roy Campbell. Specification-enhanced Policies for Automated Management of Changes in IT Systems. In *LISA 06: 20th Large Installation System Administration Conference*, pages 73–84, Washington, DC, USA, December 2006.
- [Sto87] Michael Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of the 13th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Brighton UK*, 1987.
- [Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, pages 1:146–160, 1972.
- [UVS+04] Sandeep Uttamchandani, Kaladhar Voruganti, Sudarshan Srinivasan, John Palmer, and David Pease. Polus: Growing Storage QoS Management Beyond a “4-Year Old Kid”. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 31–44, Berkeley, CA, USA, 2004. USENIX Association.
- [VCA02] Dinesh C. Verma, Seraphin Calo, and Khalil Amiri. Policy based management of content distribution networks, 2002.
- [Ver05] Dinesh Verma. Simplifying Network Administration using Policy based Management, March 2005.
- [Wei93] Marc Weiser. Ubiquitous Computing. *IEEE Computer 'Hot Topics'*, 26(10), 1993.
- [Wei94] Marc Weiser. The world is not a desktop. *interactions*, 1(1):7–8, 1994.
- [XSB] *The XSB System Version 2.7.1, Volume 1: Programmers Manual*.

# Author's Biography

Chetan Shankar was born in Bangalore, India in 1978. He finished his undergraduate education in Computer Science from Bangalore University in 2000 figuring in the top 1% of students in the university. He completed his Master of Science in 2003 and Doctor of Philosophy in 2006, both in Computer Science from University of Illinois. He has published over 15 papers in reputed international journals, conferences and workshops.

Chetan worked as a Software Development Engineer at Microsoft India (R&D) in 2000. He was an intern at Hewlett-Packard Laboratories in the summers of 2002 and 2005 where he worked on Linux IA-64 kernel and Adaptive Monitoring Infrastructures.

His areas of interest include Distributed and Pervasive Systems, Operating Systems, Policy and Model-based management.