2009

TOWARDS AUTOMATING POLICY- BASED MANAGEMENT SYSTEMS

Abdelnasser Hassan Ahmed Ouda
Western University

TOWARDS AUTOMATING POLICY-BASED MANAGEMENT SYSTEMS

(Spine title: Towards Automating Policy-Based Management Systems)

(Thesis format: Monograph)

by

Abdelnasser Hassan Ahmed Ouda

Graduate Program
in
Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO
SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Joint-Supervisor	Examiners
Dr. Hanan Lutfiyya	Dr. Michael Katchabaw
Joint-Supervisor	Dr. Roberto Solis-Oba
Dr. Michael Bauer	Dr. Nicole Haggarty
Supervisory Committee	Dr. Jerome Rolia

The thesis by

Abdelnasser Hassan Ahmed Ouda

entitled:

Towards Automating Policy-Based Management Systems

is accepted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Date_____

_____
Chair of the Thesis Examination Board

Abstract

The goal of distributed systems management is to provide reliable, secure and efficient utilization of the network, processors and devices that comprise those systems. The management system makes use of management agents to collect events and data from managed objects while policies provide information on how to modify the behaviour of a managed system. Systems as well as policies governing the behaviour of the system and its constituents can change dynamically. The aim of this work is to provide the services and algorithms needed to automatically identify and deploy management entities and be able to respond automatically to both changes to the system itself as well as to changes in the way the system is to be managed, i.e., changes to the set of management policies or sets of management agents.

One significant challenge in the use of policy-based management systems is finding efficient mechanisms to address and simplify the gap between expressing and specifying policies and an actual configuration of a management system that realizes and makes use of policies. Little work has been done to define how the monitoring operations are to be configured and updated according to the policies. This Thesis proposes a general architecture for a policy-based management system for distributed systems which allows for expressing and automating the deployment of a wide range of management policies. The proposed solution is based on the matching between the management operations that are carried out by the management agents and the policies. The matching process relies on the attributes that the agents can monitor and the extracted attributes from the components of the policies. One major contribution of this Thesis is to build the policy model and services on existing management services found in commercial management systems. The work of this Thesis also focuses in finding87 strategies for selecting and configuring agents to be used to keep the time of a policy deployment low.

The Thesis introduces the Policy-Management Agent Integrated Console (PMagic) prototype. The PMagic prototype has been implemented to provide a practical validation of the policy based management system model proposed. The approach, architecture and prototype have demonstrated that it is possible to create a more autonomic management system, particularly one that can instantiate agents to react to changes in sets of policies.

Keywords: Distributed Systems Management, Management Agents, Policy-Based Management, Management Configuration, Events Monitoring.

Epigraph

Dedication

to my family

wife
Nesrin Ouda,

sons
Hassan ,
Ahmed ,
Ibrahim,
&
Majd Ouda

Acknowledgments

I am most grateful to my supervisors, Prof. Hanan Lutfiyya and Prof. Michael Bauer. They were always ready with support, advice, encouragement and valuable feedback.

Thanks to David Wiseman, the Computer Science Department Systems Administrator, for his great help specially the installation and trouble shooting of the SNMP work. Thanks to all of the systems group members; Dave Martin, Scott Feeney, Bruce Richards and Jim Thorsley for their timely support. Thanks should also go to the administrative staff; Janice Wiersma, Cheryl McGrath, Angie Kramp and Dianne McFadzean for their friendly treatment, energetic and hard work.

I would also like to express my appreciation to the examination board for their willingness to examine my work in this Thesis; Dr. Michael Katchabaw and Dr. Roberto Solis-Oba of the Computer Science department, University of Western Ontario, Dr. Nicole Haggarty of Ivey School of Business, University of Western Ontario and Dr. Jerome Rolia of Hewlett Packard Laboratories (HP-Labs).

Last but in many ways most, I must also express a great thanks to my wife, Nesrin Ouda and my sons Hassan, Ahmed, Ibrahim and Majd Ouda. They have always stood by my side during the good times and the bad. I owe them everything. This acknowledgment would not be completed without a word of appreciation to my parents for their continuous encouragement throughout my studies.

Table of Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 Policy-Based Management Systems

Enterprise computing systems consist of thousands of heterogeneous computers and devices connected through communication networks to allow devices, services and applications to communicate with each other. Management entails the operation, administration and maintenance of a computing system so that the system behaves as expected with respect to availability, performance and security. Systems management includes monitoring of the run-time behaviour of a system, analysis of monitored data, and determining actions to modify the behaviour of the system [140].

A management system may make use of policies. Policies are one source of information which influences the behaviour of objects within a system [100]. The use of policies in management is called Policy-Based Management (PBM) [15,33,34,100,106,141,156]. Using policies facilitates the management system to be adaptable to changes in management strategies

without requiring the recoding of the management system. A policy typically consists of one event and one or more rules. These policies can be represented by high-level policy languages. The definition of policy used in this Thesis is presented in Definition 1.1.

Definition 1.1: A *policy* is an event-triggered, a set of condition-actions rules [34] i.e., an event triggers the evaluation of a set of rules of the form **if** *condition* **then** *actions*.

This form of policy is often refereed as an Event Condition Action (ECA) rule [13]. The event is referred to as the *policy event*, while the set of the condition-actions rules is referred to as a *policy rule*.

Definition 1.2: An *event* is defined as a message of notification of a change in system state that is of interest.

The condition in the rule is used to determine the actions to be executed. It is possible for there not to be a condition which indicates that for the event the actions specified in the policy are always executed. It is possible for an event to have several rules associated with it. An example policy is the following:

Example 1.1: **if** *a login session is idle for more than 20 minutes* **then** *close the session.*

System state is characterized by a set of attributes. The idle time of a login session is an attribute. A change of interest occurs when the idle time exceeds 20 minutes. Thus, the event of the policy specified in Example 1.1 occurs if *a login session is idle for 20 minutes.* The notification message may include values of attributes of interest; in this case these might be the user identifier, the device used, time of login and session process identifier. There is no condition that needs to be evaluated to determine if the policy action should be taken. This means that the action is taken when the event is detected.

Example 1.2: **if** *a login session is idle for more than 20 minutes* **then** *notify the system administrator* **if** *login session owner = AAA.*

The event specified in Example 1.2 is the same as in Example 1.1 but with a different rule. In this case, the system administrator is only notified if the condition on the *login session owner* is true. It should be noted that if both policies found in Examples 1.1 and 1.2 are applied and the login session owner is equal to *AAA* then two actions are taken otherwise only one action is taken.

A policy applies to an entity being managed i.e., *managed object*. Examples of managed objects include workstations, routers and web servers. The set of managed objects that a specific policy applies to is called a *domain*. The policy presented in Example 1.1 applies to a set of host machines. Many of the policy specification languages provide constructs for defining the domain and associate it with policies to be applied to that domain. This work uses domains, but the examples only describe events and rules.

A policy is said to be *enforced* if the actions specified in the policy are taken when the event occurs and the condition (if specified) is true. This requires that the attributes used in the specification of a policy be monitored and evaluated and that actions can be carried out. For the policy in Example 1.1, this requires that the management system monitor login sessions on each of the machines in the domain that the policy applies to. Monitoring is done by management agents.

Definition 1.3: *A management agent* is defined as a logical entity that provides a single interface and performs management operations (i.e., monitor and collect data, analyze data collected, carry out control actions) on managed objects and emits notifications on behalf of managed objects.

There may be multiple policies applied to the same managed object. Example 1.3 specifies a policy based on attributes of a login session not used in Example 1.1.

Example 1.3: **if** *a login session is from the IP address xxx.xxx.xxx.xxx* **then** *notify the system administrator* **if** *the time of occurrence of the login is between 12:00 AM and 5:00 AM.*

The IP address initiating the login session is an attribute. The event occurs if *a login session is from the IP address xxx.xxx.xxx.xxx.* The action *notify the system administrator* is taken only if the condition *the time of occurrence of the login is between 12:00 AM and 5:00 AM evaluates to true.*

1.2 Problem Statement

There are several languages for specifying policies as defined in Definition 1.1. Techniques for analyzing policies (e.g., conflict analysis) have been developed. These languages are considered independent of any management system. One significant challenge in the use of policy-based management systems is to configure a management system to monitor the attributes in the policies, generate the events specified in the policies and match rules with the events. There has been little previous work in finding efficient mechanisms to address and simplify the gap between expressing policies in a high-level specification language and an actual implementation of a management system that makes use of these policies [117-120]. Existing management systems do not provide facilities to automate the efficient deployment of management entities i.e., finding, initiating and deploying management agents that monitor, analyze and control the managed system to support policies. Such activities still fall under the responsibilities of the system administrator. A key element of this Thesis work is policy deployment, which is defined in Definition 1.4.

Definition 1.4: *Policy deployment* is defined as the mapping of policies to a configuration of the management system (e.g., identifying and configuring of management agents and providing executables rules) so that this management system enforces the policy.

For example, to deploy the policy specified in Example 1.1, a management agent must be found that can monitor attributes associated with login sessions and then the management agent must be instantiated to monitor the login sessions and generate events. This Thesis addresses the problem of automatically mapping policies to elements of existing management systems, including selecting and deploying agents to ensure the enforcement of those policies.

The state of the art in management tools provides functionality that includes monitoring, software distribution, event generation, event analysis and determining control actions through rules. Despite the significant contributions made towards the development of management tools that monitor and control distributed systems, little has been done to address issues such as optimizing the execution of management functions. Efficient operation of management functions is important since uncontrolled use could increase the load on the systems at the wrong time [1]. For example, if there is currently an instance of a management agent monitoring attributes associated with login sessions for the policy in Example 1.1, then it may be possible to reuse this same management agent instance to monitor the login sessions for the policy specified in Example 1.2, thus reducing the number of agent processes.

Both the managed objects and the policies are dynamic. Examples of the dynamic nature are the following:

(i) The set of machines to which the policies specified in Examples 1.1, 1.2 and 1.3 are applied to may change over time;

(ii) A policy may be changed. The policies in Examples 1.1 and 1.2 may be changed so that the event occurs when the session is idle for 10 minutes long and not 20 minutes;

(iii) Policies may be added at different times. The policies specified in Examples 1.1 and 1.2 do not have to be added at the same time;

(iv) Policies may be activated and deactivated at different times.

Management systems are usually assumed to be static [47] i.e., configured at start-up. Management systems do provide some ability to be changed but there is little work that looks at the automation of changing the management system configuration in response to changes in the managed objects or the policies. For example, although most management systems allow for changes in the set of policies, there is relatively little work in having the management system reconfigure itself to support the changes in policies, much less do this in an optimal or semi-optimal fashion.

This Thesis addresses the problem of automating policy deployment. This work aims to provide the services and algorithms needed to identify and deploy management entities and be able to respond automatically to both changes to the system itself, as well as to changes in the way the system is to be managed (i.e., changes to the set of management policies or sets of management agents).

1.3 Thesis Statement

It is possible to provide automated policy-based management (PBM) systems based on a general model that links the management services and the management policies. The attributes that describe the system states and specified in the management policies, may be used both to build such links and to automatically help in finding and configuring the appropriate management services to support deploying management policies.

1.4 Thesis Contributions

The Thesis makes the following significant research contributions:

- A general model for specifying and automatically deploying a wide range of management policies for PBM systems is defined. Key features of the model are to identify and deploy management entities and the ability to respond automatically to both changes to the system itself as well as to changes in the way the system is to be managed.

- The Thesis shows how to build a PBM model on existing management services found in commercial management systems.
- A prototype implementation is described. This implementation confirms the validity and provides a means to evaluate the concepts of the proposed PBM system model.
- Several experiments are conducted to demonstrate the successful application of the model, the prototype and the deployment of different policies into domains with different numbers of hosts.
- The thesis addresses an alternative deployment approach for optimization, namely one that utilizes management agents for policy deployment.
- Management agents, together with the algorithms needed to handle and processes events, are introduced, designed and implemented

1.5 Thesis Outline

The Thesis is organized as follows: Chapter 2 introduces, discusses, and assesses a comprehensive background research and related work to provide the background for this Thesis. In Chapter 3, an information model used to specify policies is presented and discussed. Chapter 3 then emphasizes the semantics of several event operators used to express events. In Chapter 4, we introduce the proposed policy-based management system (PBM) architecture. Chapter 4 draws the roadmap for the rest of the Thesis. Chapter 5 presents and discusses an information model used to define management agents and agent design. Chapter 6 shows a template-based approach to map the high-level specified policy elements to components of a management system. Chapter 7 introduces the Policy-Management Agent Integrated Console (PMagic) software. Chapter 8 describes the experiments conducted to illustrate the execution, validity and evaluation of our PMagic policy model. The Thesis contributions, conclusions and future work are presented in Chapter 9.

Chapter 2

BACKGROUND AND RELATED WORK

This Chapter introduces, discusses, and assesses related work. The Chapter starts with a discussion of elements commonly found in existing management systems, and a review of common system management concepts. This is followed by a discussion of policy specification and deployment. Next, an overview of the efforts related to the work of policy refinements and policy conflicts is highlighted. Thereafter, the issue of management systems performance will be addressed. This Chapter concludes by looking at existing commercial management systems tools.

2.1 Management Systems

Several commercial management systems, such as IBM Tivoli [151], HP OpenView [61], CA-Unicenter [23], Microsoft SMS and MOM [105], support the management of distributed systems. This Section briefly describes the common elements currently provided by these systems. These common elements are graphically depicted in Figure 2.1. These elements include management agents, information repositories, management applications, and event handlers.

Management Agents

Management agents are used to monitor data, analyze monitored data and emit notifications on behalf of managed objects. A managed object is any distributed system component (e.g., workstations, routers, web servers) that is to be controlled by a management system.

Information Repository

An information repository is used for the storage and retrieval of monitored information, information about the managed system (e.g., definitions of managed objects, abstractions of management agents, domains, definitions and interfaces of management applications) and management information (e.g., policies, events).

Management Applications

A management system may make use of one or more management applications that analyze monitored data and determine control actions. A management application may analyze data collected from multiple management agents over a period of time and determine control actions that are to be carried out by management agents on managed objects. A management application is also referred as a *manager.* An example of a management application often found in management systems is an application that provides software distribution. Such an application distributes, configures/reconfigures, and updates software applications, system patches and management agents. Another manager is an *event-driven rule-based engine* in which events are associated with rules. For example, the event in Example 1.2 is associated with the rule *notify the system administrator* **if** *login session owner = AAA.* An example of an event-driven rule-based engine is found in Tivoli Enterprise Console (TEC) [151]. TEC not only associates an event with a rule but also supports the detection of a pattern of events based on a time operator. Management applications often present graphical representations of the system and monitored information. For example, a graphical display of the status of each router in the network.

Management Protocols

Standards for manager and agent communication have been defined. The Simple Network Management Protocol (SNMP) [65,103,144], Common Management Information Protocol (CMIP) [70,70], Application Response Measurement (ARM) [121] and Web-Based Enterprise Management (WBEM) on top of HTTP [116], are examples of standard management protocols.

Event Handlers

The notification of an event is a message consisting of a set of attributes and values that provide information about the change of the state. For example, the management agent that monitors the user logins to the system could fire an event whenever user *AAA* tries to login to the system, and provide information such as the user identifier, device used and time of login. Events may be of interest to multiple managers. This requires that for each event, the event handler keeps track of the managers interested in the event. Management systems often include one or more event handlers to deal with the collection and distribution of events to other management components. There is a distinction between an event-driven rule-based engine and an event handler. An event-driven rule-based engine would register its interest in specific events with the event handler. An event received by the event handler would be forwarded to the event-driven rule-based engine if it is of interest.

2.2 Policy Management Frameworks

The general idea of Policy-Based Management (PBM) is not new [33,64,100,134,140]. Every management system provides and utilizes management polices to some extent. This Section describes frameworks that use a specific policy language for a specific management system. We can identify two general aspects that need to be addressed in the design and implementation of any PBM system [120].

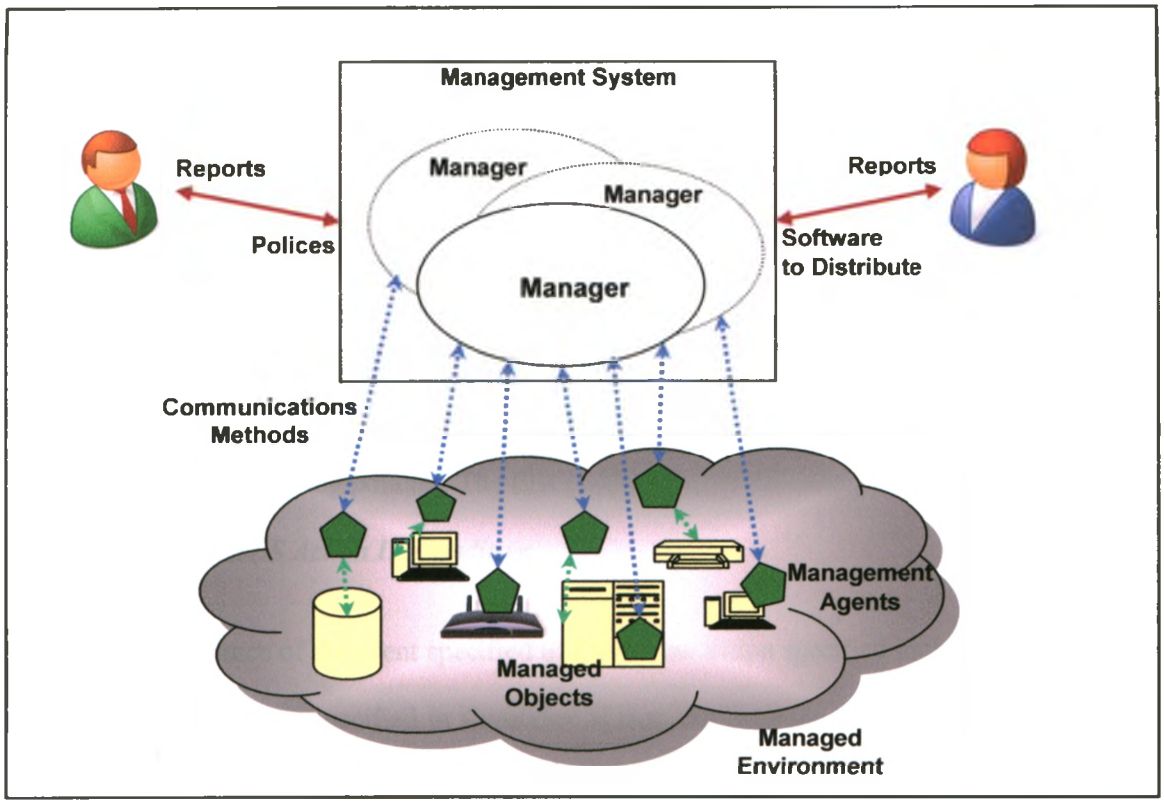


Figure 2.1: Common Management Architecture

The first aspect is policy specification. Policy specification deals with the definition of policies. The most commonly used forms of policies are those that are defined in Definition 1.1 or just the rule part [13,51]. Significant contributions in policy specification include: PDL [86,154], Ponder [34,92] , PFDL [110,111], Tower [57], Power [109], SNAP [78], Logic [145] , Rei [76], KAoS [17,37,57,155], PPL [146], Cfengine [22] , ACPL [62].

The second aspect is policy deployment. Policy deployment can make use of a variety of technologies, including expert-systems, programmable rules, agent-based, and mobile-agents or some combination of these technologies.

This Section discusses policy-based management modules with respect to two aspects: policy specification and policy deployment. In this Section we describe four policy management

modules or frameworks often cited in the literature. We then describe policy deployment within these frameworks along with a discussion of the limitations. These four modules are representative of much of the work in policy specification and deployment.

2.2.1 Policy Description Language

The Policy Description Language (PDL) is a declarative policy definition language from Bell-Labs [80,86], originally developed for specifying network management policies. PDL has two constructs for specifying policies. The first construct is the following:

Event ***CAUSES*** *Action* ***IF*** *Condition*

On the occurrence of the event specified in *Event,* the action specified in the *Action* clause is executed if the condition specified in *Condition* is true. An example of the use of this policy is presented in Figure 2.2, where *SessionIdleEvent* is the name of the event that causes the action specified in *SendEmail* to be carried out only if the condition *SessionIdleEvent.userid = AAA* evaluates to true. The second construct used to specify policies is the following:

Event ***TRIGGERS*** *pde* (a_1=v_1, ..., a_n=v_n) ***IF*** *Condition*

On the occurrence of the event specified in *Event* the action is the notification of the event denoted by *pde.* The symbol *pde* represents an event symbol, a_i is an event attribute and v_i is its value. Figure 2.3 shows the policy described in Example 2.1 which is specified using the second PDL construct.

Example 2.1: **if** *the number of failed login attempts under a specific login name exceeds 3* **then** *lock the account used in logins.*

The event specified in Example 2.1 is *failed login attempts under a specific login name exceeds 3.* There is no condition that needs to be evaluated to carry out the policy action *lock the account used in logins.* The specification of the above policy as seen in Figure 2.3 defines two

events, the *LoginFailed* event which triggers when any user fails to login to a host in the System Labs, the second event is *ThreeFailedLogins* which triggers when the *LoginFailed* event triggered three times from the same user.

```
Events: SessionIdleEvent: system event          //event definition
Actions: SendEmail                              //action definition
Policy Description:                             //Policy of Example 1.2
SessionIdleEvent causes SendEmail("admin",SessionIdleEvent.userid)//parameters
if (SessionIdleEvent.userid = "AAA")                //condition used
```

Figure 2.2: Policies Written in PDL First Construct

```
Events: LoginFailed: system event ,          //event definition
        ThreeFailedLogins : pde              //policy define event
Actions: LockAccount                         //action definition
Policy Description:                          // Policy of Example 2.1
LoginFailed triggers ThreeFailedLogins (userid=LoginFailed.userid)
            if ( Count(LoginFailed.userid) = 3 )
ThreeFailedLogins causes LockAccount
```

Figure 2.3: Policies Written in PDL Second Construct

The system described in [129], which was implemented for Lucent switching products, represents one example of the deployment of PDL policies. There are three main components [129]: the Policy Enabling Point (PEP), the event handler and the policy engine. The implementation is in Java. PDL rules are compiled to Java classes. The Directory Server analyzes the policies to extract the different components. Events found in the policies are then registered with the event handler and rules are registered with the policy engine. The policy engine is an expert system. When a PEP receives an event from a Lucent switch component, the PEP sends the event to the event handler. The event handler correlates events from multiple PEPs to detect more events that may be specified in the policy. The event manager then sends the detected events to the policy engine. The rules in PDL are translated into a form that can be understood by the expert system. This system assumes a predefined set of events that are

generated only for network devices. This work does not address the issue of reconfiguring monitoring operations for triggering different events.

2.2.2 Ponder

Ponder is a declarative object-oriented language developed primarily to support security and management policies [34]. Ponder has four basic policy types: authorizations, obligations, refrains and delegations. There are four composite policy types that are used to compose policies: groups, roles, relationships and management structures. The Ponder language syntax allows for the specification of the following:

- Domains: The specification of a set of managed objects.
- Subjects: This refers to management system entities that upon receiving the notification of an event carry out the action if the specified condition is true.
- Targets: This refers to managed objects to which a policy applies.

An example of a Ponder obligation policy is seen in Figure 2.4. A Ponder obligation policy can be used to specify the example policies presented in Chapter 1. The management policy in Figure 2.4 represents the Ponder specification of the policy described in Example 1.2. The policy specifies that the domain is the *Syslab*. The policy in Figure 2.4 states that the automated manager CSD/*TEC* will execute the action *sendemail*, when the event *SessionEvent* is triggered, and when the specified condition *SessionEvent.sessionidle>20 and SessionEvent.userid="AAA"* is true. Ponder also introduces the notion of policy types, i.e., parameterized policy templates that can be instantiated multiple times with different parameters to create new policies. New policy types can be inherited from existing policy types [41]. For example, Figure 2.6 shows the Ponder specification of the policy stated in Example 2.1. The policy is triggered when there are *n* repeated login failures from the same user identifier. In Figure 2.5, the policy is instantiated with the automated manger */CSD/TEC* (specified using the subject clause). The target */CSD/users* (of

type <*userT*>), specifies the domain. The action *lock()* disables the account of a user identifier with three failures in logging in. Policy conflicts are addressed in [36,41].

```
inst oblig SessionIdlePolicy
{  on SessionEvent (userid);
   subject s = /CSD/TEC;
   target t =/CSD/Syslab/UNIXHosts;
   do sendemail("admin",userid);
   when SessionEvent.sessionidletime > 20 and
        SessionEvent.userid = "AAA";
}
```

Figure 2.4: Ponder Obligation Example Policy

```
type oblig RepeatedLoginFailure
          (subject s, target <userT> t, int number)
{  on number*LoginFailed(userid);
   subject s;
   target  <userT> t;
   do t.lock(userid);
}
inst oblig Three_RepeatedLoginFailure = RepeatedLoginFailure
                                        (/CSD/TEC, /CSD/users, 3);
```

Figure 2.5: Ponder Policy Type Example Policy

The work in [33] presents a management architecture that assumes Ponder is the policy specification language. This architecture includes three supporting services: a policy service, a domain service and an event service (essentially an event handler). The Policy Service compiles a policy to a Java class, stores these classes, and creates new policy objects. The Domain Service manages and maps the name of a domain to the set of target objects that it applies to. The Event Service receives events and sends these events to the interested management application (specified as subjects). More details about these services are described in [33].

A Policy Management Agent (PMA) is given a Java object generated from a policy. It is used to carry out actions i.e., it is a subject. The PMA registers with the event service to receive

events of interest (as specified in the policy). The occurrence of an event is sent to the PMA through the event service.

Events are specified using attributes that represent system and application behavior. These attributes need to be monitored. The PMA needs to evaluate a condition which is also specified using attributes representing system and application behavior. It is assumed that the monitoring is done. The architecture does not specify how the monitoring entities are initialized or configured. There is no discussion of how monitoring entities are reconfigured in response to changes in policies.

2.2.3 Policy Framework Definition Language

The three main policy models used in industry are the Directory Enabled Networks (DEN) model, the Internet Engineering Task Force (IETF) model and the Distributed Management Task Force (DMTF) model [110,111]. The IETF and DMTF refined the DEN model. The DMTF model has additional infrastructure not found in the IETF or DEN models. The DEN-ng model [148] is derived from the DEN and IETF models [92]. In the DEN-ng and DEN models, a policy is a set of rules that are evaluated when a specified event occurs. The IETF and DMTF do not specify a condition characterizing the event that triggers evaluation of a rule. This makes it difficult to have interoperable policy-based management systems [147]. Information models are used to specify the components of the policies and the relationship between these components. A conflict occurs when at least two conditions in the rules associated with a policy are satisfied but the actions cannot be executed at the same time. IETF addresses this with the assignment of priorities to each rule used in the policy. The rule with the highest priority is executed. The three models have a concept similar to that of domain. PFDL uses the notion of *Policy Role*, which is used to represent a collection of managed resources that share a common policy role. Generally, a role is a type of property that is used to select one or more policies for a set of entities and/or components from among a much larger set of available policies [37,39]. By using

roles, the administrator may assign each resource to one or more roles and specify policies to be associated with a role.

The IETF/DMTF developed a standard architecture to be used as a guideline for PBM implementations called the Policy Management Framework (PMF). It uses the Common Open Policy Service Protocol (COPS) [45,110,147]. The COPS protocol is used between Policy Enforcement Point (PEP) (agents) and a Policy Decision Point (PDP) (manager) to exchange the information needed for policy enforcement as will described in this Section. There are two 'flavors', or models of COPS: The Outsourcing Model and the Provisioning Model.

- In the Outsourcing Model, all policies are stored at the PDP. Whenever the PEP needs to make a decision, it sends all relevant information to the PDP. The PDP analyzes the information, makes a decision about an action to be taken, and sends the result of the decision to the PEP. The PEP then carries out the action.
- In the Provisioning Model (COPS-PR), the PDP downloads relevant policies to the PEP. The PEP makes decisions based on these policies. The Provisioning Model uses the Policy Information Base-PIB, defined in RFC 3159, as a repository of the policies.

More information about COPS and Policy-Based Network Management (PBNM) can be found in RFC-2748 [66] and RFC-3084 [68]. The COPS architecture, as shown in Figure 2.6, contains, in addition to PDPs and PEPs, the following components:

- Policy Management Tool (PMT): This manages the policies i.e., create, distribute, activate, deactivate, modify, check conflicts, etc.
- Policy Repository: This stores policies so that the policies can be accessed and retrieved by the one or more PDPs.

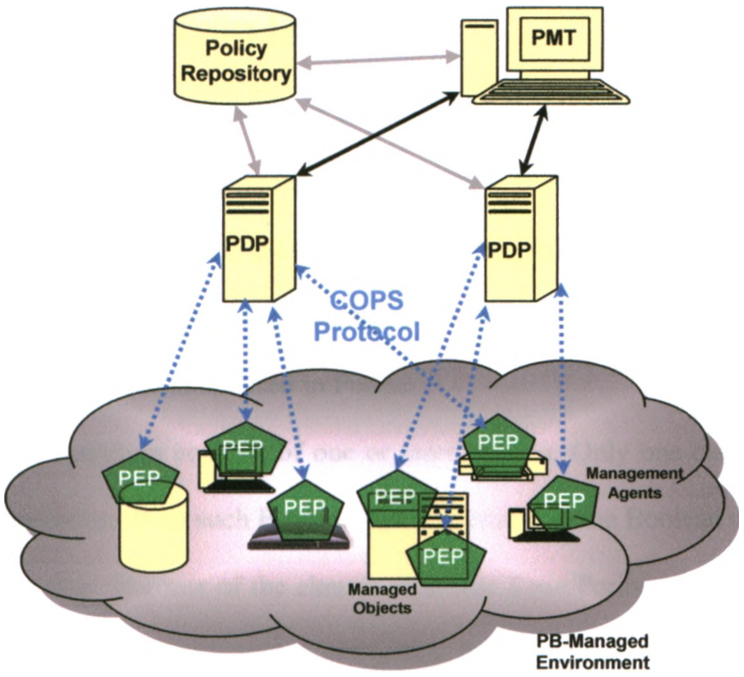


Figure 2.6: COPS Architecture [45]

Examples of the use of the IETF architecture can be found in management tools for Quality of Service (QoS) and Virtual Private Network (VPN) management (e.g.,[54,96]). In [54] deployment refers to the sending of a policy to a PDP from the PMT. The work in [54] investigates the requirements and implementation of a PBNM system to ensure that a policy is deployed on all target PEPs. The work in [54] does not address the issue of identifying and configuring monitoring entities. The work in [96] focuses on mapping objectives specified in policies to system configurations. This is based on previous observations. A change in an objective in the policy is mapped to a configuration based on a comparison of the objective in previous observations. This is used to determine configuration parameters which are carried out by existing management entities.

2.2.4 Automatic Computing Policy Language

Automatic Computing Policy Language (ACPL) [62] is used for writing policy rules for Policy Middleware Automatic Computing (PMAC) from IBM [64]. ACPL is XML-based. Policies specified using ACPL include the following components:

- Scope: This specifies the set of the managed resources that the policy applies to. This is similar to the domain concept used in Ponder.
- Condition: A condition consists of one or more clauses. Only one clause can express a Boolean expression. If no such Boolean clause exists, then the Boolean clause is assumed to be always true. The rest of the clauses represent Time-Period elements. If there is no Time-Period clause in the Condition of a policy, then the policy is always active.
- Decision: This specifies how a policy is to be enforced. Decisions include the following:
    - Result: This returns a set of required monitoring information from the managed resources.
    - Action: Invokes operations on the managed resources.
    - Configuration profile: Applies both Result and Action.
- Business value: This specifies the priority of a policy. This is used to enable the manager to decide which policy should be enforced when multiple policies can be applied. This allows for the handling of policy conflicts.

ACPL requires the specification of the policy name, decision name (for result and configuration decisions), the policy version, and the policy description. Figure 2.7 shows the structure of the policy described in Example 1.2 written in ACPL that utilizes the components described earlier in this Section. More about ACPL and its use in web services is described in [62,64]. PMAC also provides a Policy Analysis Toolkit that can be used to identify conflicts in policy specifications.

```
<!-- Policy Meta Data -->
  <acpl:Policy policyEnabled="true"
               policyName="Check_Idle_Logines Sessions ">
      <!--the above expression would include other Meta data -->
  <acpl:Description>Policy Description</acpl:Description>

<!-- Policy Condition -->
  <acpl:Condition>
      <exp:And>
          < exp:Greater>
              <exp:PropertySensor propertyName="SessionIdleTime" />
              <IntegerConstant>
                  <Value>20</Value>
              </IntegerConstant>
          </exp:Greater>
          <exp:Equal>
              <exp:PropertySensor propertyName="UserId" />
              <exp:StringConstant>
                  <Value>AAA</Value>
              </exp:StringConstant>
          </exp:Equal>
      </exp:And>
  </acpl:Condition>

<!-- Policy Decision -->
  <acpl:Decision>
      <acpl:Action>
          <acpl:WS_Operation operationname="SendEmail"
                     portType="MailManagerResource" />
      </acpl:Action>
  </acpl:Decision>

<!-- Policy Business Value -->
  <acpl:BusinessValue>
      <Importance>10</Importance>
  </acpl:BusinessValue>

<!-- Policy Scope -->
  <acpl:Scope>
      <acpl:StringScope>
          <Value>SysLab/UNIXHosts</Value>
      </acpl:StringScope>
  </acpl:Scope>

</acpl:Policy>
```

Figure 2.7: ACPL Example Policy

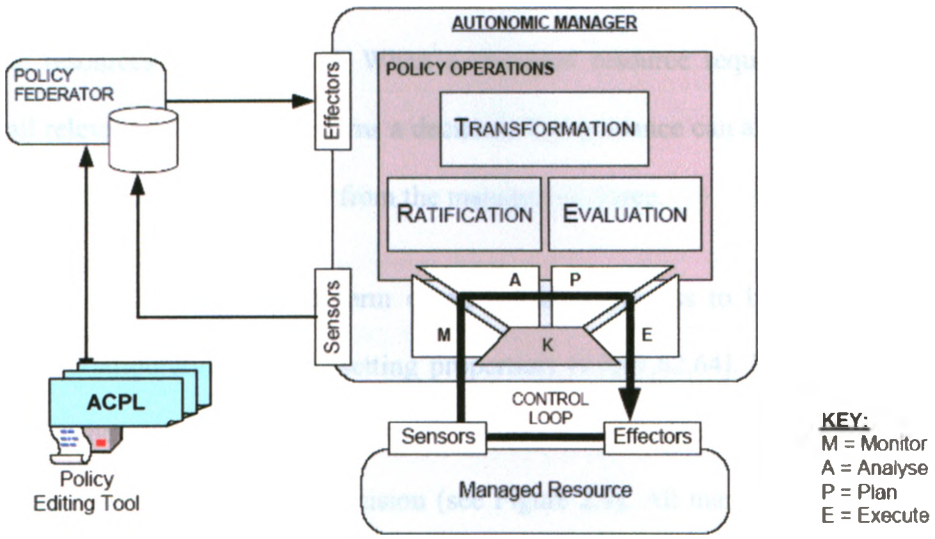


Figure 2.8: PMAC Architecture [64]

In PMAC, the Autonomic Managers (AMs), which are equivalent to the concept of PDP, manage the system by using the machine-readable policies specified using ACPL. The PMAC framework consists of a policy editing tool, a federator, an autonomic manager and managed resources (see Figure 2.8). Edited policies (using ACPL) are saved and distributed to the relevant autonomic managers by the use of federator which acts as a publish/subscribe hub. PMAC uses the ECA rule paradigm that follows the XML schema (which tends to be quite verbose). The Simple Policy Language (SPL) [5] can also be used by PMAC to support the specification of ECA rules. SPL is internally mapped to ACPL which can be understood by the PMAC policy federator.

Sensors and effectors are the interfaces of the managed resources. Sensors (representing management agents) provide management information when triggered by get or subscribe commands while effectors (also representing management agents) apply the management decisions. Since AMs are considered as managed resources, AMs also have sensors and effectors to be used by other AMs. PMAC provides support for policy analysis and conflict resolution using a process called Policy Ratification [6,35]. The autonomic manager (AM) is central to the

PMAC infrastructure. The AM is responsible for providing the control-loop (Figure 2.9) to manage the resources assigned to it. When a managed resource requests guidance, the AM evaluates all relevant policies and returns a decision. The guidance can also be initiated from the autonomic manager without a request from the managed resource.

These decisions can be in the form of an action (a process to be run on the managed resource) or a configuration profile (setting properties) [4,5,47,62,64]. When a change of state occurs (event), the managed resource notifies the AM-Event Monitoring Subcomponent which triggers requests for guidance on a decision (see Figure 2.9). All management information that represents system states also passes to and is retrieved from the Data-Gathering-Subcomponent. The Rules-Expression-Engine then evaluates the policy rules, which reside in the Runtime-Configuration-Cache, according to the triggered request, and selects the policy corresponding to this request. The Policy-Actuator-Subcomponent will enforce the decision made by invoking the effectors on managed-resources. The WSRF (Web Service Resource Framework) interface implements, supports and supplies the resource properties specifications inside the AMs by the means of web service standards.

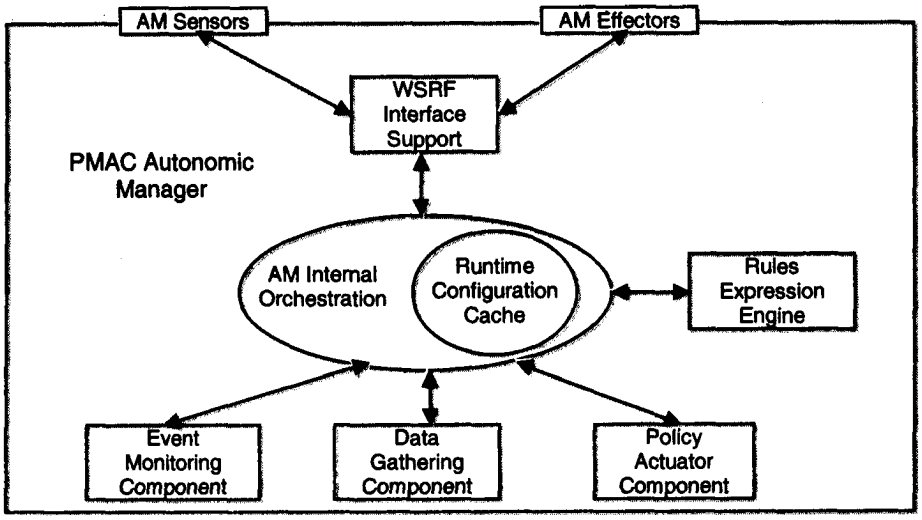


Figure 2.9: PMAC Autonomic Manager [47]

To summarize, the PMAC framework provides a language for specifying policy and a policy enforcement architecture. Policy analysis and refinement operations are also addressed to some extent [64]. However, the management services, which are needed to support and supply the management information, are assumed to be already defined and executing in the managed resources. Moreover, there is no indication how PMAC would initiate new services at the managed resources, i.e., how to configure and/or map the required management operations to the sensors.

2.3 Policy Refinements and SLAs

Policy refinement is the process of mapping high level management policies to an appropriate set of policies rules. A Service Level Agreement (SLA) is typically a written agreement between a service provider and a customer about the quality of service (QoS) [14,27,79,81,88,94,95,99,130-132,153]. An SLA is one source of the management goals. The management application that handles SLAs is often called the Service Level Manager (SLM). As discussed in [89], PBM systems are often considered closely aligned to the SLM. Furthermore, the work in [75] clearly defines the relationship between SLA and policies, and discusses how SLAs can be enforced by policies. The goals defined in an SLA must be mapped to a form that can be understood by the management system. The process of the automation of policy refinement is a challenging problem. Certain aspects of policy refinement can be achieved when the problem is constrained to a well-defined functional area [77], such as extracting the QoS policies from given SLAs in order to provide policy management rules to reconfigure network routers to achieve the QoS goals. The work in [77] sketches the relationship between the generality of the refinement technique used and the amount of automation. As illustrated in Figure 2-10, refinement techniques that are more domain specific have more opportunities for automation.

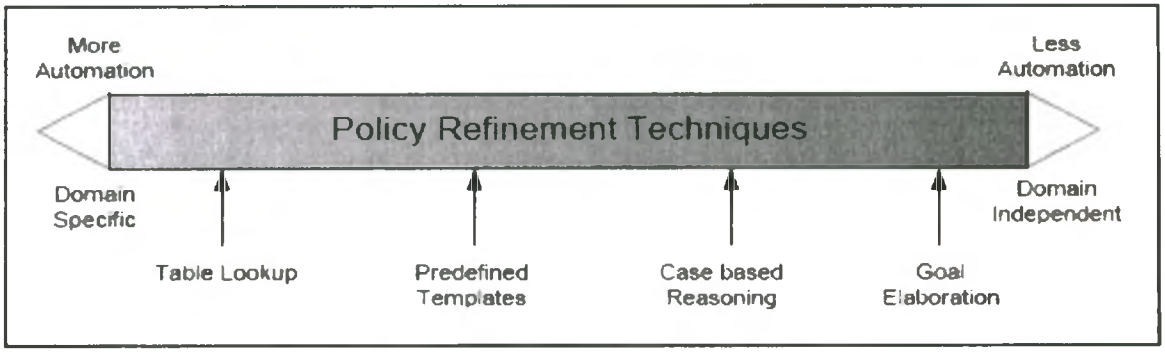


Figure 2.10:Relationship between policy refinement techniques [77]

The work in [11] identifies two main objectives of a policy refinement process as follows:

- Determining the resources that are needed to satisfy the requirements of the policy, i.e., mapping abstract entities defined as part of a high-level policy to concrete objects/devices that make up the underlying system. For instance, determining the specific routers that need to be configured to handle the traffic for "WebServices applications on the eCommerce Server" [11].
- Refinement of high-level goals into operations, supported by the concrete objects/devices, that when performed will achieve the high-level goal. For instance, the set of operations, supported by the specified routers that will meet the objective of "Gold QoS for WebServices Applications on the eCommerce Server" [11].

There is substantial, ongoing work, both commercially and academically, addressing strategies for expressing, refining and implementing SLAs using policies [9,11,37,38,52,75,77,89,123,137,152], where several refinement techniques are used. In [123,143], an approach to refinement is presented that is based on hierarchical architectures that correspond to different abstraction levels of the management functions or policy expressions. A role-based mechanism is used in [57] for specifying role-based access control (RBAC) policies, where the policy is refined into the basic structures of RBAC such as, users, roles, objects, permissions or privileges. The work in [38] presents an approach using the Common Information

Model (CIM) [39] for the SLA-driven management of distributed systems. The SLA is viewed as a set of services selected and aggregated from provider services matrices. Although this approach seems to be a realistic approach, such an approach is applicable only for a service provider whose management system is based on CIM.

The work in [36,37,57,152] introduces SLA decomposition and classification approaches for deriving low level patterns or system thresholds from service level objectives (SLOs) specified in SLAs. A mapping technique is used in [137] to map firewall rules into policy-rules for better firewall rule editing and to act as an advisor for anomaly discovery. The work in [52,77] introduces an approach for modeling and formulating QoS policies, in which the refinement process exploits the use of integrity constraints within abduction reasoning. The report in [77] shows how the integration of abduction reasoning with constraint solving can help to increase automated support for policy refinements. Integrity constraints are rules that specify the conditions under which the formal model of the system being analysed is inconsistent [77], and is typically used for modeling firewall policies. The policy wizard tool "POWER" [109] handles policy refinement through the use of policy templates. The administrator or the system expert specifies a set of policy templates, expressed as Prolog programs, and then uses the provided policy-engine to interpret these programs. Policy templates then guide the user in selecting the policy elements from an information model.

2.5 Policy Conflicts

As introduced, a policy is a set of rules; these rules consist of an event, conditions and actions. As in all event-driven rule-based systems, major issues in policy-based management systems are the analysis, detection and resolution of the conflicts between rules. An example of two conflicting rules is the following: 1) the rule that has the action to prevent connections from the source where intrusion was detected will only block connections from that source and 2) the

rule that has the action to shutdown the hacked system. Deploying such two conflicting rules may cause an unstable system. The automation of conflict analysis, detection and resolution are challenging problems.

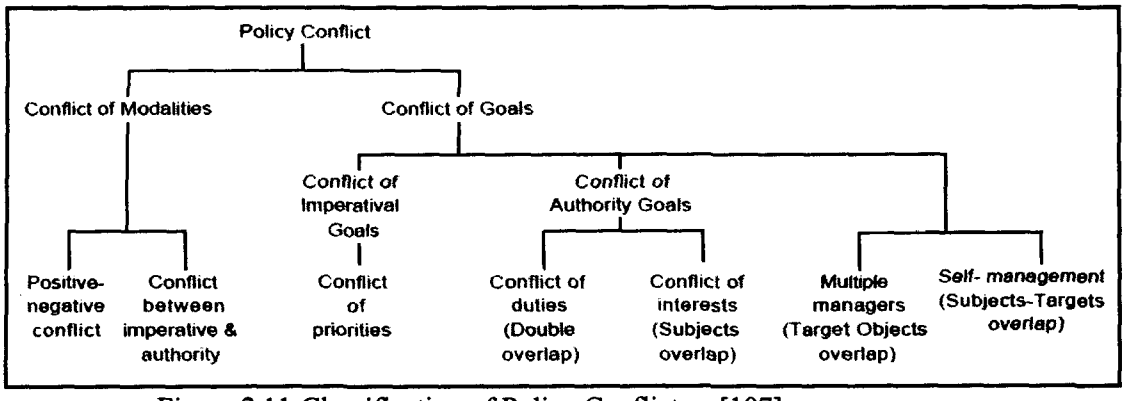


Figure 2.11:Classification of Policy Conflicts [107]

The work in [91,107,142] have considered different cases and types of conflicts by examining several examples of policy conflicts. Policy conflicts can be classified into two types as in Figure 2.11 [91,107]:

- Conflicts arising from inconsistencies in policy specification, where the same event and condition are in two different policies which have conflicting actions, e.g., allow or prevent the user "*AAA*" access to host "*HHH*". This conflict type is also called a *conflict of modalities* [107].
- Conflicts arising in the policy enforcement, even though there is consistency in the set of policies being used. These conflicts cannot be determined directly from the policy specifications, since additional information that relies on the state of the system, is needed to specify the conditions which result in conflicts. This also known as a *semantic conflict* or *conflict of goals* [107]. For example, there is a conflict between two policies that result in the same packet being placed on two different queues [142].

The work in [44,91] addresses the two forms of conflict analysis:

- Static analysis is typically performed during policy specification and before deployment. Section 2.2 discussed some of the different techniques used by the policy specification languages to handle policy conflicts. However, few conflicts can be detected using the static analysis, as it is not possible to automatically determine and evaluate policy conditions which depend on run-time state values.
- Dynamic conflict detection is typically performed at runtime. There is no easy solution to dynamically relate conflicts, and any solution would have tradeoffs. For instance, an improved dynamically conflicts resolution technique would potentially be a more computationally cost approach.

Some conflict resolution techniques include specifying priorities or precedence of policy-rules, as in [64,142]. This approach is not scalable to large systems with a large number of rules specified by different administrators. Other work uses action constraints, as in [28,147]. The efficiency of this approach depends on understanding the semantics of the policy actions, which itself is another challenge. Some other work is based on translating policies into event calculus for better conflict analysis as in [12,17]. This methodology is impractical and does not scale if there are a large number of rules. The model provided in [46] for conflict handling tried to be independent of the syntax of the policy-specification language used. It assumes that subjects, targets and action concepts are included in, and common to most policy languages. The model in [46] relies on defining an intensive relationship among the policy objects to classify any possible conflicts, which would be very hard for a management environment with a large number of management policies. The use of meta-policies, which are policies to handle the conflicts among other policies, is explored in [34,76]. Although this approach has had some success for the detection of dynamic conflicts, it is a computationally expensive approach. A more successful approach was presented in [42-44], where the use of multiple techniques for conflict detection

and resolution in PBMSs were explored. Generally, there has been little research into the performance overhead associated with the operations of monitoring, detecting and resolving conflicts.

2.6 Commercial Tools

Many vendors provide management systems with some policy-based management capability. The majority of these solutions are directed towards the configuration and security management of network devices. Examples of such PBM solutions are: Cisco's QoS Policy Manager (QPM) [30] and Cisco's CiscoAssure [32], IBM Tivoli Security Policy Manager [151], Nortel's Optivity [115], Computer Associates' eTrust Solutions [23], Hitachi's JP1-PolicyXpert [59], Lucent technologies RealNet policy rules [87], Hewlett-Packard's OpenView [61], and Tivoli Access Manager-NetView [151].

There are several commercial management systems that provide additional management functionality, such as event handling, event-driven rule-based engines, logfile analysis, software distribution, and more. Examples of such systems include: IBM's Tivoli Framework and related products [151], Hewlett-Packard's OpenView [61], Computer Associates' Unicenter [23], Microsoft's SMS and MOM [105], Hitachi's JP1 Integrated Management [59], Sun's N1 Datacentre [150]. Such commercial management systems are often derived from traditional networked environments, where devices use relatively static configurable management software to perform rather simple management operations, e.g., collect management information and enforce management actions.

Even with the management operations, such commercial management systems still need to have dedicated skillful administrators. It is also hard to handle the increased heterogeneity of resources in large distributed systems. A common feature in commercial tools is a graphical user interface (GUI), which typically allows the administrator to construct management rules. These rules are later compiled and loaded into the system manager. In these systems, there is a lack in

the definition of the link between these rules and the management monitoring services (agents) that execute in order to collect events of interest from managed objects. This relationship between rules and agents is typically defined and configured by the system administrator. Management policy can be changed dynamically, which may require a change to the management services to support these changes. Management systems should be able to provide a dynamic adaptation to changes in polices.

2.8 Summary

This Chapter has described several policy-based management systems, focusing primarily on policy specification languages and policy deployment systems. Policy deployment can make use of a variety of technologies, including expert-systems, programmable rules, agent-based, mobile-agents or some combination of these technologies. An expert system might use the policies as rules within its knowledge base to validate and enforce policies. Management based on expert systems usually focuses on a single type of managed object or application, e.g., access control in network routers, and does not provide appropriate knowledge representations for use across different application domains. Policy rules could be pre-programmed through high-level programming languages to provide decision making of runtime execution. This is static and can be difficult to adapt in response to changes in policies. Management systems often use management agents to enable policy deployment.

While there has been some work on automation of certain aspects of policy-based management systems, there is clearly a need for more work on the automation of the mapping of policies to management elements (e.g. agents, rules), the configuration of those management elements, the efficient runtime use and reuse of those elements, and the efficient reconfiguration of those elements in response to changes in the system being managed or in policies. This research focuses on the means for a management system to automatically identify and efficiently deploy management operations and management system configurations for deploying policies.

Chapter 3

A Policy Information Model

This Chapter presents an information model for policies. An information model for policies is an abstraction and representation of the components of policies. For each component this includes the definition of attributes, operations and relationships. An important aspect of the information is an event. This Chapter starts with a discussion of events including a definition of an event, introduction of event operators and semantics of event operators. This is followed by a discussion of the policy information model. The information model allows for the specification of different type of policies in a vendor and device independent way. The specification of policies is done in a modular fashion i.e., policies are specified and assembled from other components (e.g., event, condition, action).

3.1 Events

An important component of the policy information model is an event. This Section defines events, event operators and the semantics of those event operators.

3.1.1 Event Definition

There are several different definitions of events found in the distributed system management literature [19,98,118,127,134,138,160]. The definition of an event for this work (as described in Definition 1.2) is the following: *An event is defined as a message of notification of a change in system state that is of interest.* Examples of events include:

a) router R1 is down;

b) server A is not responding;

c) printer LP1 is out of paper;

d) the cpuload of server B is greater than 85;

e) the average of the cpuload of server A and B is greater than 80 over a 10 minute period;

f) the cpuload of server A is greater than 80 after user K is logged in;

g) user R failed to use su login as root user 2 times;

h) user S has more than 10 sessions open on server A and 3 sessions on server B;

i) user S failed to login to server A three consecutive times within 2 minutes;

j) send the number of current users of a machine to the manager every hour.

As can be seen from these examples, a change in state does not necessarily cause an event. For example, a change in the CPU load from 60 to 70 does not generate an event. A change from 70 to 90 does generate an event, since the policy may only be concerned with CPU loads at that level. Hence, an event is generated when the change in state is of interest. The elapse of a certain amount of time can also cause an event e.g., the passing of one hour is an event that causes the number of current users to be sent (example j). Events can be classified into primitive and composite events [97].

Definition 3.1: A ***primitive event*** is characterized by *a condition on attributes of one or more managed objects*. The logical expression representing the condition uses standard logical operators.

Events described in a) to d) are primitive events. The logical expression that represents the condition characterizing the event presented in example d is (*cpuload>85*). The *cpuload* is the attribute that needs to be monitored on host *B*.

Definition 3.2: A ***composite event*** is characterized by a condition composed of multiple events (which may be primitive or other composite events) using event operators.

Events described in e) to i) are composite events.

3.1.2 Event Attributes

Each event E_i has a well-defined set of attributes Att_i . A subset of Att_i includes a set of attributes that uniquely identifies the event e.g., event identifier, time of occurrence, source that generates the event. The set of attributes Att_i may also contain attributes that characterize the state of a managed object, e.g., for a host machine, attributes characterizing its state include cpu load and memory usage. It is assumed that the subset of Att_i that uniquely identifies the events is used to uniquely identify the second set of attributes.

3.1.3 Event Operators

The event operators considered in this work are described in this Section. The semantics of these operators are defined in Section 3.1.4. Events specified using event operators are in the form of E_i *eop* E_j or *eop* E_j. The event operators described are E-SEQ, E-AND, E-OR, E-NOT and E-COUNT (other event operators are possible). It is assumed that events occur in a specific time window. The event operators are briefly described as follows.

- E_i E-SEQ E_j: The generated event occurs when an instance of E_i occurs before an instance of E_j.
- E_i E-AND E_j: The generated event occurs when instances of both E_i and E_j occur.
- E_i E-OR E_j: The generated event occurs when either instance of E_i or E_j occurs, or both occur.
- E-NOT E_j: The generated event occurs when an instance of E_j did not occur.
- E-COUNT E_j [n]: The generated event occurs when instances of E_j occurs n times.

Example 3.1: if *the total number of user logins is greater than 5* **followed by** *the CPU load is greater than 90 and the total number of processes running is greater than 35,* **then** *block any new user logins.*

This policy consists of a composite event E_i E-SEQ E_j where

1) E_i is a primitive event that is characterized by the condition the *total number of logins is greater than 5*. The logical expression that represents this condition is *userloginstotal>5*.
2) E_j is a primitive event which is characterized by the condition *the CPU load is greater than 90 and the total number of processes running is greater than 35*. The logical expression which represents this condition is *cpuload>90 && cpuprocesstotal >35*.

3.1.4 Event Semantics

This Section describes the semantics of event operators based on first-order predicate logic.

Definition 3.3: A predicate *occ* is defined as follows: $occ(E, t, [t_s,t_e])$ is true if E occurs at time t and $t_s \leq t \leq t_e$, where t_s and t_e are the time window start and end points of time respectively.

With the E-OR operator applied to E_i, E_j, an event is generated upon detection of either E_i or E_j. The timestamp t of the resulting composite event instance is either the occurrence time of the instance of the events E_i or E_j (see Figure 3.1 (1)).

$$occ(E_i \text{ E-OR } E_j, t, [t_s, t_e]) \equiv ((\exists t_i \mid t_s \le t_i \le t_e, t=t_i)\ occ(E_i, t_i, [t_s, t_e])) \vee ((\exists t_j \mid t_s \le t_j \le t_e, t=t_j)\ occ(E_j, t_j, [t_s, t_e]))$$

The attribute set of the resulting composite event is Att_i if either of the following two cases occurs: (i) $occ(E_i, t_i, [t_s, t_e])$ is true but $occ(E_j, t_j, [t_s, t_e])$ is not true; (ii) $occ(E_i, t_i, [t_s, t_e])$ is true and $occ(E_j, t_j, [t_s, t_e])$ is true but $t_i < t_j$. Otherwise the attributes set of the resulting composite event is Att_j.

With the E-AND operator applied to E_i, E_j, an event is generated upon detection of both E_i and E_j. The timestamp t of the resulting composite event instance is the occurrence time of the last detected instance of the events E_i and E_j (see Figure 3.1 (2)).

$$\begin{aligned} occ(E_i \text{ E-AND } E_j, t, [t_s, t_e]) \equiv\ & ((\exists t_i, t_j \mid t_s \le t_i \le t_j \le t_e, t=t_j) \\ & (occ(E_i, t_i, [t_s, t_e]) \wedge occ(E_j, t_j, [t_s, t_e]))) \vee \\ & ((\exists t_i, t_j \mid t_s \le t_j \le t_i \le t_e, t=t_i) \\ & (occ(E_i, t_i, [t_s, t_e]) \wedge occ(E_j, t_j, [t_s, t_e]))) \end{aligned}$$

The attribute set of the resulting composite event is the union of all attributes of both Att_i and Att_j, i.e., the set $Att_i \cup Att_j$.

With the E-SEQ operator applied to E_i, E_j, an event is generated upon detection of E_i followed by a detection of E_j. The timestamp t of the resulting composite event instance is the occurrence time of the detected instance of event E_j (see Figure 3.1 (3)).

$$\begin{aligned} occ(E_i \text{ E-SEQ } E_j, t, [t_s, t_e]) \equiv\ & (\exists t_i, t_j: t_s \le t_i < t_j \le t_e, t=t_j) \\ & (occ(E_i, t_i, [t_s, t_i]) \wedge occ(E_j, t_j, [t_i, t_e])) \end{aligned}$$

The attribute set of the resulting composite event would be the union of all attributes of both Att_i and Att_j, i.e., the set $Att_i \cup Att_j$.

With the E-NOT operator applied to E_j, an event is generated upon no detection of E_j in time window T, i.e., $[t_s, t_e]$. The timestamp t of the resulting composite event instance is the end time point, t_e.

$$occ(\text{E-NOT } E_j, t_e, [t_s, t_e]) \equiv (\forall t : t_s \le t < t_e) \neg occ(E_j, t, [t_s, t_e])$$

The attribute set of the resulting composite event is $\varnothing$.

With the E-COUNT operator applied to E_j with the argument n, an event is generated after E_j has been detected n times. The timestamp t of the resulting composite event instance is the occurrence time of the n^{th} detected instance of event E_j.

$$occ(\text{E-COUNT } E_j[n], t, [t_s, t_e])) \equiv (\forall t_i \ 1 \le i \le n, \exists t_i \mid t_s \le t_i \le t_e) \ occ(E_j, t_i, [t_s, t_e])$$

The attribute set of the resulting composite event is Att_j of the n^{th} instance of E_j.

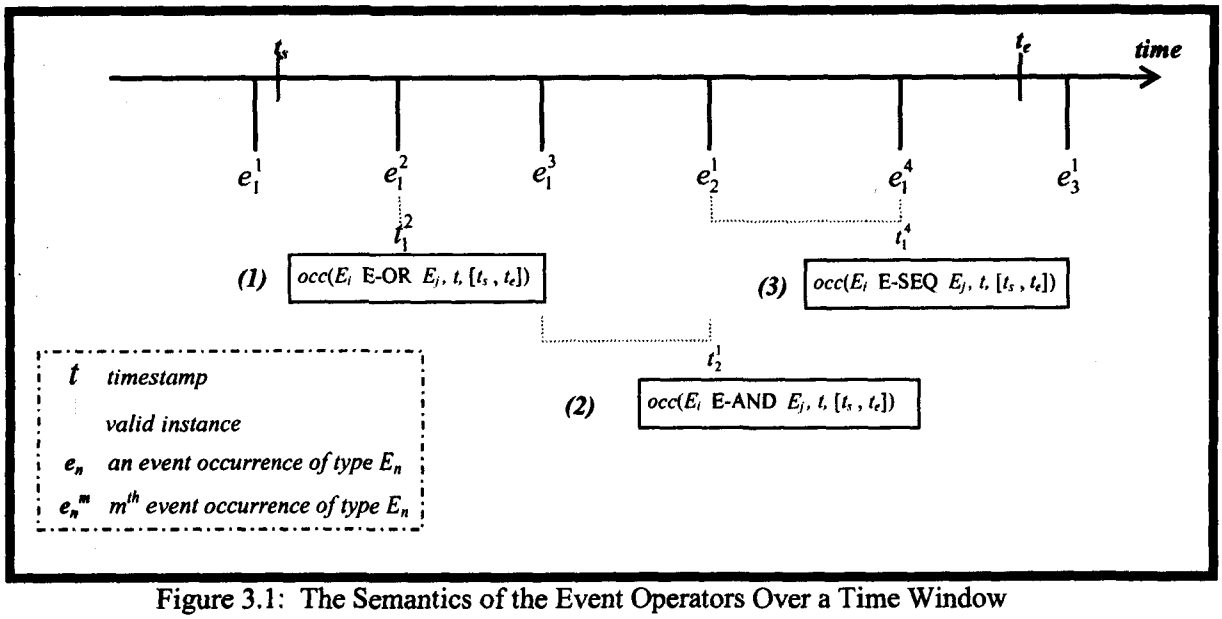


Figure 3.1: The Semantics of the Event Operators Over a Time Window

It is possible to express the event operators without explicitly stating a time window. In that case, the assumption is that the time window is defined by the system. The default time window defined by the system has t_s equal to the current time of the system clock of the device that generated the event, while t_e equals the current time plus some defined duration say d (e.g., 20 seconds). For example; if a time window in the E-OR event operator is not explicitly defined, then the semantics would be as follows:

$$occ(E_i \text{ E-OR } E_j, t) \equiv occ(E_i \text{ E-OR } E_j, t, [t_{current}, t_{current+d}])$$

3.2 Policy Information Model

The UML depiction of the policy information model is presented in Figure 3.2. A policy is modelled as an aggregation of an event (modelled by the *Event Expression* class) and a set of rules. Rules of the policy can be prioritized. Each rule (modelled by the *Rule* class) specifies a condition (modelled by the *Condition* class which represents a logical expression) which if true, specifies the actions (modelled by the *Action* class) to be executed. An action may have one or more parameters (modelled by the *Parameter* class). A policy is associated with a domain which is a set of computing resources. In this work, it is assumed that these computing resources are hosts (see Figure 3.3). If a subset of these computing resources is to have additional policies, then these are assumed to be a separate domain with its own set of policies. The work in [39] establishes a common conceptual framework that describes the managed computing resources. The information model in [39] represents an abstraction and representation of the elements in managed computing resources, a basic classification of these managed elements and associations of managed resources. The classification and relationship between managed objects are beyond the scope of this Thesis. This task will become easier when standard information models, e.g. CIM [39], are adopted.

Intervals that represent the date and time that a policy is active might be defined and associated at both the policy and rule levels. This is useful for refining the enforcement of the policy period. By associating an interval with a rule, typically a sub-interval of the policy interval, one can allow events to be detected within the policy interval and limit the enforcement of the associated rule with the detected event within a sub-interval of that of the policy. Evidently, It may be the case that the rules should be activated at different times of the day e.g., one rule for daytime hours and one rule for night time hours. By associating an interval with a rule, one can allow events to be detected within the policy interval and limit the enforcement of the associated rule with the detected event within a sub-interval of that of the policy. Static analysis of policies should validate that there is no conflict between the intervals i.e., between the policy interval and the intervals of its associated rules. Further discussion of several of these classes follows.

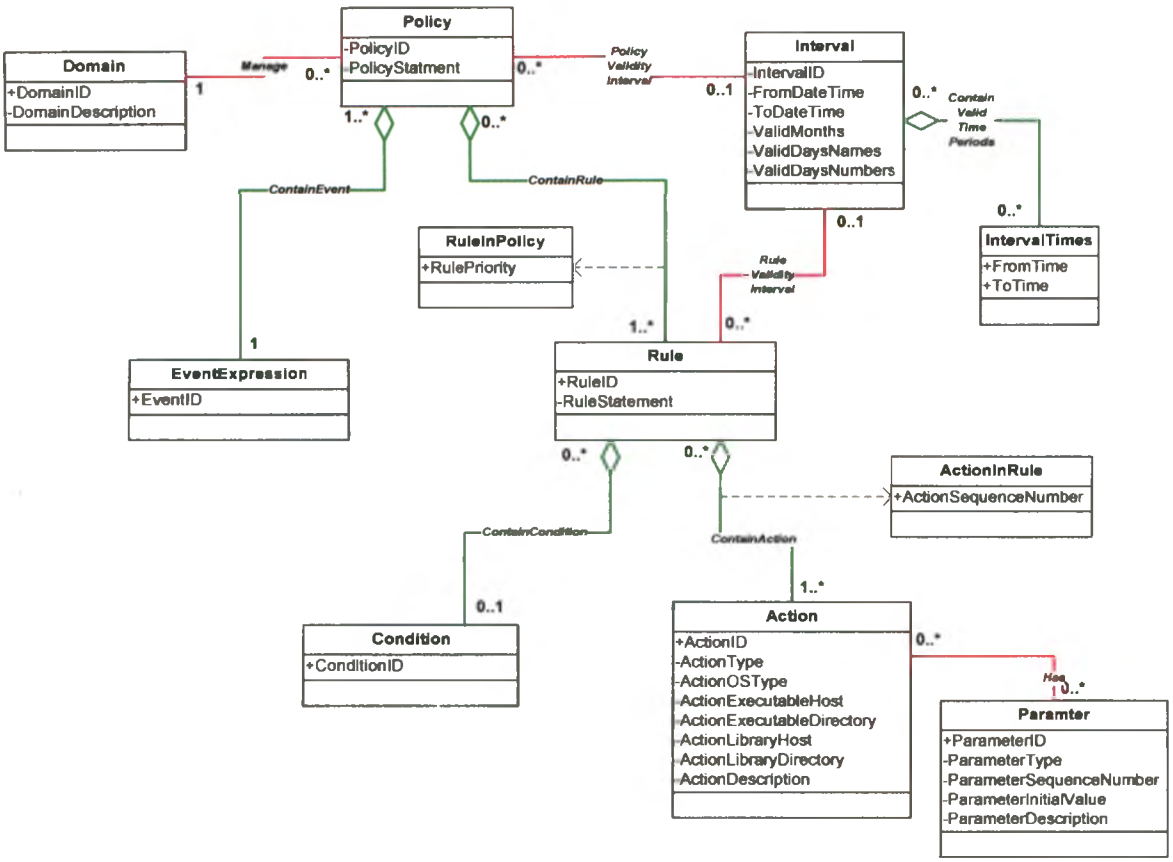


Figure 3.2: Policy Information Model

Figure 3.3: Domain Information Model

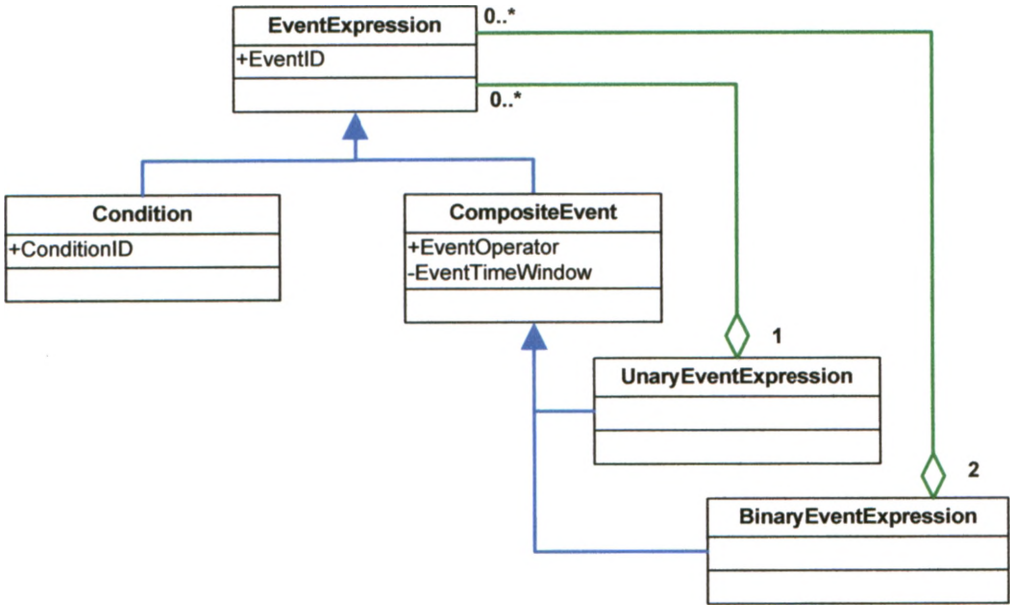


Figure 3.4: EventExpression Information Model

Action Class

This class represents information about an action. The attributes characterizing this class include the following:

- ActionID: This is the action identifier.
- ActionType: This represents the type of the action based on the type of file with the action code e.g., UNIX scripts, binary executable codes, Java classes, etc. This is referred to as an action executable. This field helps to construct the action calls that match its type.
- ActionOSType: This specifies the operating system on which this action can execute e.g., UNIX, Windows, Linux.

- ActionExecutableHost: This represents the IP address of the host that stores the action executable.
- ActionExecutableDirectory: This represents the directory where the action executable file and any necessary configuration files are placed. The combination of ActionName, ActionExecutableHost and ActionExecutableDirectory is used to determine the location of a specific action executable.
- ActionLibraryHost: This represents the IP address of the host that has the libraries needed by the action for execution.
- ActionLibraryDirectory: This represents the directory where the libraries are located.
- ActionSequenceNumber: Since a rule may specify multiple actions, the association class ActionInRule, defines an attribute that represents the sequence order of an action within a rule.

Parameter Class

This class represents information about a parameter that an action may have. An action may have more than one parameter. The class is characterized as follows:

- ParameterID: This provides a parameter identifier.
- ParameterType: The type of the parameter e.g., string, integer, real, boolean.
- ParameterSequenceNumber: This represents the sequence order of a parameter within the action parameters.
- ParameterInitialValue: This represents the initial value of the parameter.
- ParameterDescription: This provides a textual description of the parameter.

Interval Class

This class represents information about an interval. The class is characterized by the following attributes:

- IntervalID: This provides an interval identifier.

- FromDateTime: This represents the start of an interval e.g., "2008/01/01 00:00:00".
- ToDateTime: This represents the end of an interval e.g., "2008/06/30 23:59:59".
- ValidMonths: This value, if specified, represents the valid months within the start and end dates of the interval. The default value is that all months are valid. This attribute and the next two attributes are used to filter the interval.
- ValidDayNames: This value represents the valid days according to their names or order in the week, e.g., SUNDAY, MONDAY, etc. or 1,2,..,7.
- ValidDayNumbers: This value represents the valid days based on the day of the month.
- The class IntervalTimes represents an additional filter by defining the valid period(s) of time within a day, e.g., FromTime=12:30:00, ToTime 17:29:59 will limit the policy to be active within this time period in each day of the policy interval.

EventExpression Class

This class represents information about an event. An event may be characterized by a condition or recursively defined from other event expressions using an event operator (see Figure 3.4). We use the composite design pattern from [47] to compose the tree structure of an event as follows:

- The Condition class (the logical expression) represents a leaf object in the composition. Leaves represent the primitive events in the composition.
- The CompositeEvent class is an abstract class that defines the attributes EventOperator and EventTimeWindow. The EventTimeWindow denotes a specific time preiod. More specifically, EventTimeWindow represents a time period delimited by two specific boundaries of time points, e.g., 10 minutes from current system time (now), or 10 seconds from the occurrence of either event of the event expression. In this time window the event expression should be evaluated.
- The UnaryCompositeEvent class extends the CompositeEvent class, to represent information about the unary composite event expression

- The BinaryCompositeEvent class extends the CompositeEvent class, to represent and store the components of the binary composite event expression, i.e., store two event component (EventID) which represent the operands in the event expression.

3.3 Chapter Summary

This Chapter presented an information model used to specify policies. The definition of events and event operators used in this Thesis were presented as well as the semantics of the event operators. The purpose of this Chapter is not to provide a unique set of event operators and semantics, but rather to present a model we assume for the work presented in this dissertation. The model and event operators semantics are derived from work found in [56,72,97,98,160]. The policy information model addressed in this Chapter is the basis for the implementation of the language used for expressing policies (see Appendix A). Event operators and their semantics are introduced in this thesis in order to model composite events and to facilitate the development of the algorithms used to handle composite event operators (see Appendix E).

Chapter 4

A Model for Policy Based Management Systems

This Chapter briefly describes a policy-based management system (PBM) architecture (see Figure 4.1) that can configure itself in response to changes to the system being managed and changes in policies. Chapters 5, 6 expand on the components of the architecture in more detail, while Chapter 7 describes an implementation of the architecture.

4.1 Proposed PBM System Architecture

Figure 4.1 illustrates an architecture for a PBM system model. The model's components are discussed in greater detail in the rest of this Section.

Policy Specification

This component is used to specify policies. The specified policies are stored in a repository. It is assumed that this component can transform the policy into a form understood by the other

components, e.g., the Agent Matcher and Mapping Mechanism components. The policy specification is based on the policy model defined in Section 3.2.1.

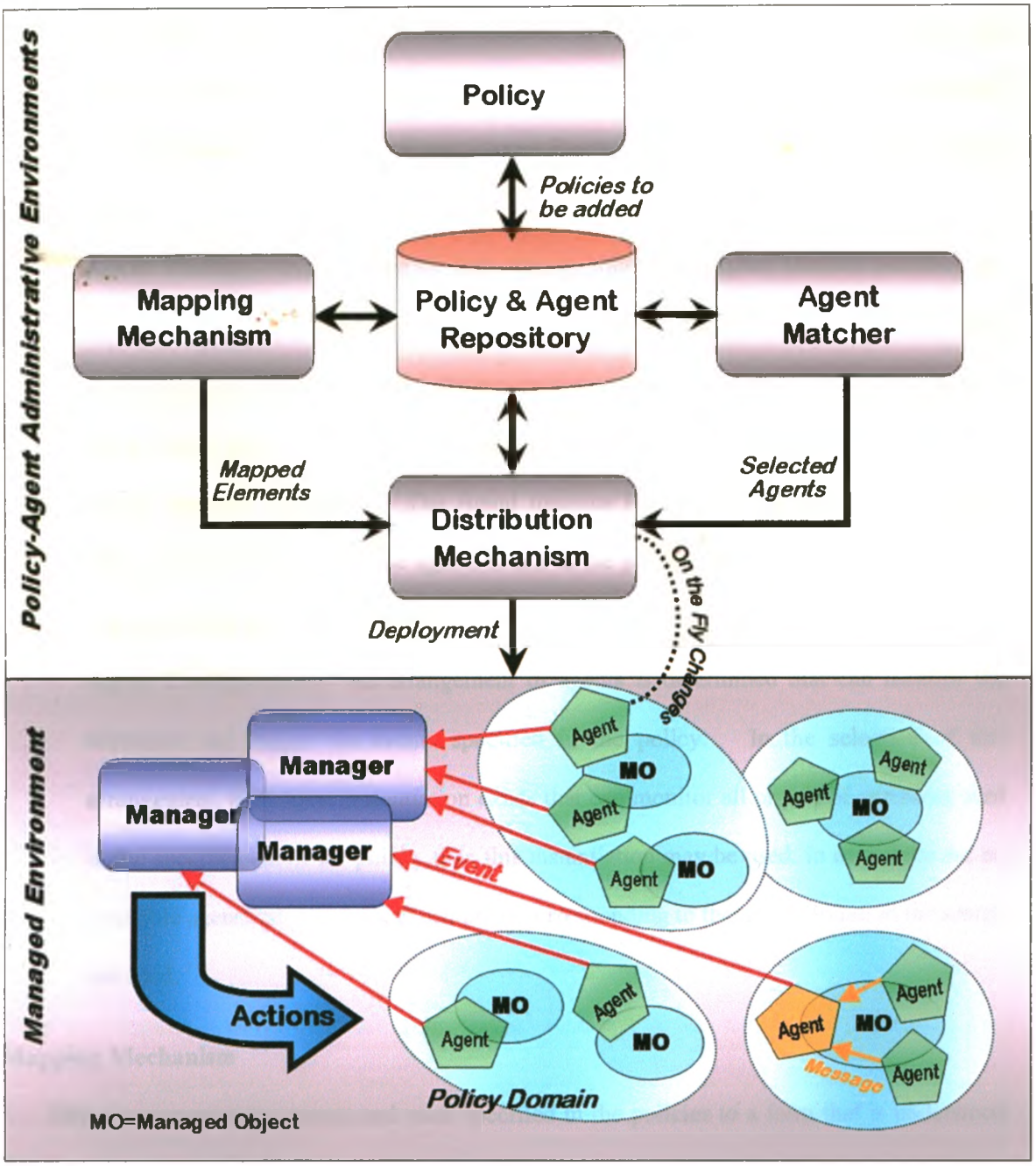


Figure 4.1: Proposed PBM System Architecture

Agent Matcher

When a policy is activated or an active policy is changed, the Agent Matcher is invoked. The agents that can be used to enforce the policy are determined and then any possible existing agent instances that might be used are identified as is or the identified agents can be used with some changes, e.g., by requesting that a specific condition characterizing an event be monitored. The three tasks carried out are the following: Agent Finding, Agent Instances Finding and Agent Configuration.

- **Agent Finding:** Upon adding or activating a policy, the Agent Matcher searches the agent repository to find a set of agents that can be used to monitor the attributes specified in the policy's event expression and the conditions used in the rules. A list of agents and the attributes that it can monitor is created (more on this in Section 5.2).
- **Agent Instance Finding:** The Agent Instance Finding task is used to determine if there are instantiations of these agents found in the previous step. It may be possible to use an existing agent instantiation.
- **Agent Configuration:** An arrangement of agents is determined that can monitor the attributes and trigger the events specified in the policy. In the selection of this arrangement, if an agent instantiation exists that can monitor all or part of attributes used in the specification of the policy, then this instantiation may be used. In case there are no available agent instantiations, executables corresponding to the agents found in the search are used.

Mapping Mechanism

This component maps events and rules specified in the policies to a form that is understood by the management system.

Distribution Mechanism

This component starts an agent instance, sends a new condition to be monitored to an agent instance, sends a new request for values of a set of attributes that the agent instance can support, and sends event information and the set of condition-actions rules of the policy to the appropriate manager.

Manager

There must be at least one manager that is notified about the events received from the event-handler. Management systems often include one or more event handlers to deal with the collection of and distribution of events to other management components. The manager evaluates the rules associated with the events received.

Interactions

To illustrate the interactions among the components, the following example policy is used:

Example 4.1: **If** *the CPU load is greater than 95 for any of the UNIX system lab hosts* **then** *email the administrator*

Assume that this policy is specified and stored in the Policy-Repository. The specification of this policy requires the following:

- A policy named *cpu_load_policy* is created which has a primitive event named *cpuload_evt* and one rule named *cpu_load_rule.*
- A logical expression named *cpuload_exp* representing the CPU-Load restriction, i.e., *(cpu_load >= 95)*, is created.
- The event-target assigned is TEC (Tivoli Enterprise Console). TEC is the default event target. The policy domain is the set of all UNIX based hosts in *Syslab.*
- The *cpu_load_rule* rule consists of no conditions and one action.
- The action to be taken is *send-email.*

- The mapping policy elements (i.e., event and the set of condition-actions rules) will be discussed in Chapter 6.

The steps needed to find the set of agents that can be used to enforce the example policy is carried out by the Agent Matcher component. These steps include the following:

1. The Agent Matcher searches the repository to determine the event expression associated with the event identifier, *cpuload_evt*.
2. The attribute identifiers from the event expression and from the condition of the rule components are extracted, i.e., the attribute *cpu_load* is extracted. This attribute is used to search for UNIX based agents (as specified in the policy domain) that can monitor that attribute. The proposed model allows a single domain to include machines of different operating systems. For example, if the target machines in the policy domain include both UNIX and Windows machines, then the Agent Matcher would search in both the UNIX and Windows based agents to find agents for each OS platform that can monitor the attribute.
3. Assume that an agent, *cpu_agent*, capable of monitoring the attribute *cpu_load* is found.
4. The Agent Matcher then checks to determine whether there are instantiations of the *cpu_agent* in the policy domain hosts. There are two cases; a) no existing instantiation, then a new instance needs to be configured to support the *cpu_load_policy* policy (i.e., decides the attributes to monitor and the event expression to evaluate), b) there are instances of the *cpu_agent* in the policy domain hosts, these existing instances will reconfigured to support the *cpu_load_policy* (i.e., existing instances would update to monitor and fire an event if the *cpu_load* attribute is greater than or equal to 95). The Distribution Mechanism is used to place the executables appropriately and start a new instance. The Distribution Mechanism component is provided with enough information (e.g., the location of the agents executables and libraries used) to start an agent instance. In addition, the set of condition-actions rules of the policy is mapped to an executable

representation and then distributed to a manager that evaluates rules. If the target machines in the policy domain include UNIX and Windows machines then two agents instantiations would be required - one for each operating system platform.

4.2 A Roadmap for Automating PBM Systems

This Chapter provides a roadmap for the core work presented in this Thesis, specifically, the design, development and deployment of the components as illustrated in Figure 4.1. The details of the framework are described in Chapters 5-8 of the Thesis which are summarized below:

- **Chapter 5: Management Agents.** In this Chapter, the description of management agents with the depiction of the management agent components and design that underpins all management agents is developed and described. In particular, this Chapter describes a service to automate agent finding and agent configuration.
- **Chapter 6: Mapping Mechanisms.** A policy needs to be understood by a management system that enforces this policy. The principal focus of this Chapter is the design of approaches that uses a policy to determine configurations of management entities. This is referred to as *mapping*.
- **Chapter 7: Implementation and the Prototype.** This Chapter describes the implementation of the Policy-Management Agent Integrated Console (PMagic) prototype. PMagic represents the proof of concept of the approaches developed in this Thesis towards automating PBM systems.
- **Chapter 8: Evaluation.** This Chapter describes the experiments conducted to evaluate PMagic and presents the conclusions drawn from these experiments
- **Chapter 9: Conclusions and Future Work.** This Chapter outlines the Thesis contributions. Conclusions and future work will be highlighted in this Chapter.

Chapter 5

MANAGEMENT AGENTS

This Chapter begins with a discussion of the required management services for operations related to management agents to support policy enforcement. The Chapter then presents an information model that is a representation of the components of an agent and its instantiations. This is followed with a discussion of a management agent design and its interface. This is used as the basis for the Agent Matcher component.

5.1 Introduction

As defined in Chapter 1, a management agent is a logical entity that provides a single interface and performs management operations (i.e., monitor and collect data, analyze data collected, carry out control actions) on managed objects and emits notifications on behalf of managed objects. Existing management systems do not provide facilities to automate the deployment of management entities i.e., finding and configuring management agents that monitor, analyze and control the managed system to enforce policies. Currently these types of

activities are considered the responsibility of the system administrator. One of the goals of this work is to have the management system automatically carry out these activities in response to a change in the policies or in the system being managed. For example, assume that an administrator has just added the policy specified in Example 3.1. This requires that there are agents that can monitor the attributes used in the specification of the policy and carry out any actions specified in the policy. For Example 3.1, the agents needed should be able to monitor the attributes representing the total number of users logged in (*usersloginstotal*), the CPU load (*cpuload*) and the number of processes (*cpuprocesstotal*) attributes. A search of the existing set of agents based on the attributes to be monitored is needed to determine a set of agents that can used to enforce the policy.

Searches should not be limited to just the set of agents available. It is important to be able to determine any existing instantiations of those agents since it may be possible to use these instantiations. This is illustrated with the following example policies:

Example 5.1: if *su root is used to login into any of the UNIX hosts in the Computer Science Department* **then** *email the administrator* **if** *the user is "AAA"*

Example 5.2: if *the number of failed attempts to log into any of the UNIX hosts in the Computer Science Department under a specific login name exceeds 3* **then** *lock this account*

Both example 5.1 and example 5.2 policies require the monitoring of the *syslog* file. Assume that only the management agent *syslog_agent* monitors the *syslog* file for each UNIX host machine. Ideally, it should be possible to activate these policies at different times yet use one instance of *syslog_agent.* For the policy stated in Example 5.1, an instance of the management agent *syslog_agent* is created to monitor the event *su root is used to login.* Assume that the policy specified in Example 5.2 is added later. If the management system also maintained information about agent instances then the instance of the management agent *syslog_agent* could

also be used for the policy stated in Example 5.2. The management agent instance that is in use for the policy specified in Example 5.1 needs to be configured to evaluate the event used in the policy specified in Example 5.2.

Searches for agents and agent instances require an information model that provides information about agents, agent instances and their relationship. Section 5.2 describes an information model.

5.2 Management Agent Information Model

To facilitate the usage of management agents for supporting policies, a management system should have a service that finds agents. Therefore, for each agent the following information is needed:

1. The attributes that can be monitored.
2. Information characterizing an agent executable e.g., the path of an agent executable such as */PMagic_Manager/agents/cpu_agent.* In this work, the term agent refers to the agent executable.
3. Information characterizing an instantiation of an agent i.e., the agent that is in execution. This is called an *agent instantiation* or *agent instance* in this work.

The information model (graphically depicted in Figure 5.1 using UML) that represents this information is described in this Section. This information model is used as the basis for the agent repository in the prototype introduced in this Thesis (see Chapter 7 for more details). Information about the agent is represented by the *Agent* class and information about a specific instantiation is represented by the *AgentInstantiation* class. A description of these classes is provided below.

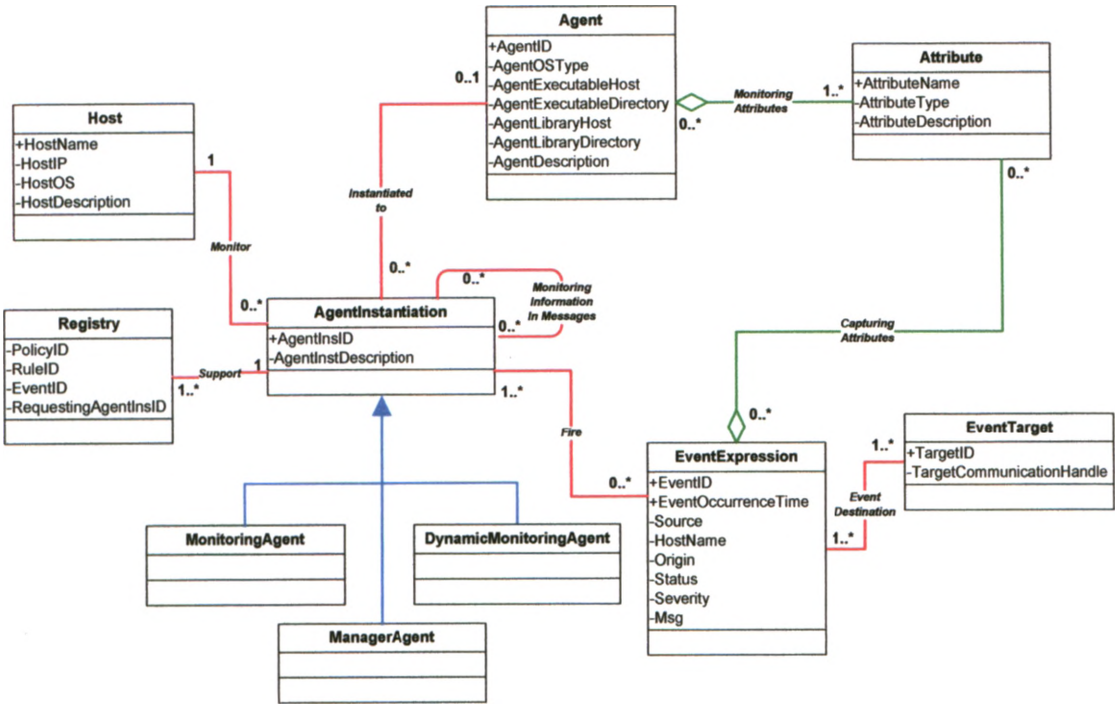


Figure 5.1: Agent Information Model

The *Agent* class represents information about an agent. The attributes characterizing this class include the following:

- AgentID: This is the agent identifier.
- AgentOSType: This specifies the operating system under which this agent can execute e.g., UNIX, Windows, OS2, Linux.
- AgentExecutableHost: This represents the IP address of the host that stores the agent executable.
- AgentExecutableDirectory: This represents the directory of the agent executable file. The AgentExecutableDirectory is the path to the directory of the agent executable and supporting files e.g., adapter configuration files. The combination of AgentID, AgentExecutableHost and AgentExecutableDirectory is used to determine the location of a specific agent executable.
- AgentLibraryHost: This represents the IP address of the host that has the libraries needed by the agent for execution.

- AgentLibraryDirectory: This represents the directory where the libraries are located.
- InformationServiceDirectory: This is the path to a directory with the executable of an information service. Not all agents need to assign a value for this.
- AgentDescription: This provides a textual description of the agent.

Table 5.1 represents an example of the values that the Agent class attributes may hold.

Attribute Name	Value
AgentID	cpu_agent_1
AgentOSType	UNIX
AgentExecutableHost	129.100.18.32
AgentExecutableDirectory	"/sl/wolfbiter/PMagic_Manager/agents/cpu_agent_1/"
AgentLibraryHost	129.100.18.32
AgentLibraryDirectory	"/sl/wolfbiter/PMagic_ Manager/libraries/"
AgentDescription	"an agent to monitor the cpu and processes on managed host"

Table 5.1: Example of the Values the Agent Class Attributes may Hold

Instances of agents monitor attributes characterizing the behaviour in a system (e.g., *cpu_load* attribute). Attribute information is represented by the *Attribute* class. Information about an attribute includes a unique attribute name, the type of the attribute and a description. As represented in the model, an attribute can be monitored by more than one agent and an agent can monitor more than one attribute. It may be the case that there are different agent executables with the same functionality that execute under different operating systems. In this model, these agents are represented by different *Agent* objects.

The *AgentInstantiationClass* class represents information about an instantiation of an agent. It is possible to have multiple agent instances of an agent. Thus there is a zero-to-many relationship between *Agent* and *AgentInstantiation*. An agent instance may be an initialization of an agent executable or a dynamically created agent or a manager agent. More details about

management agent types are described in Section 5.6. Each agent instance is associated with a host. An agent instantiation has a unique identifier, *AgentInsID*, e.g., the combination of the *AgentID*, *AgentInstanceCommunicationPort* (i.e., the communication port number that this instance uses to communicate with the host) and the *HostName*.

Agent instantiations are used to monitor attributes to determine if a condition that characterizes an event is satisfied. An event is modelled by the class *EventExpression*. An agent instantiation may generate more than one event and an event may be generated by more than one agent instance. Thus there is a many to many association between the *AgentInstantiation* class and the *EventExpression* class. Upon detecting an event, the agent instantiations send a notification of that event to the designated target (e.g., management application). The information (e.g., host name and port number) about the designated target is represented by the *EventTarget* class.

The work in [16,25] addresses the issue of canonical and common base event data. The management agents introduced in this Chapter are designed to construct event notification messages with important data, such as data that uniquely identifies the managed object, the source of the event (the component that is generating the event) and the timestamp of the event. The data necessary to synchronize and aggregate events with other events, in composite events situation, are also included in the generated event messages. More about the data maintained in event messages can be found in Chapter 7.

Information about what an agent instance supports is maintained and is represented by the *Registry* class. An agent instance can support one or more policies in that it can either generate events specified in the policy or monitor attributes specified in the policy (but not actually generate an event). The manager uses and updates the *Registry* class information in the agent

configuration task. More details about agent configuration tasks will be elaborated in Section 5.6, 5.7.2, and Chapter 6.

5.3 Management Agent Design

An agent monitors attributes that characterize the state of the system. There are many information services that provide monitoring mechanisms to get values of the attributes represent the state of the system. Provide. Examples of information services are; Simple Network Management Protocol (SNMP) agents [65,103,144], Application Response Measurement (ARM) [121] and Tivoli monitoring agents [151], Web-Based Enterprise Management with CIM (WBEM/CIM services) [58,116], Java Management Extensions (JMX) [149] , Windows Management Instrumentation (WMI) [105], Web Service Distributed Management (WSDM) [158], scripts, and logfile analyzers. There are two challenges in using these services:

(i) The information services directly extract information from the object being monitored. An information service typically involves some form of instrumentation. Each information service has an interface, but these interfaces are different for different information services.

(ii) Events may be specified using attributes from different managed objects. An event may be characterized by a condition that checks the status of a printer and the size of the document that needs to be printed. Such a condition would include attributes from different managed objects. This condition may need two information services to collect the attribute values needed to evaluate the condition characterizing the event.

To automate the identification of and search for the services that can monitor the system attributes requires that all management agents must have a common interface. Figure 5.2 shows how the proposed monitoring structure as outlined in this Thesis would communicate with existing monitoring mechanisms that have different interfaces. The arrows indicate a flow of

information. A management agent can request information from an information service. The information service responds to the management agent with the requested information. This requires that the management agent that interacts with the information service use the API of the information service.

A proposed management agent design is shown in Figure 5.3. In this design, management agents have an interface that is known to the managers and other agents, and through that interface the manager can configure management agents (upon creating an agent instance), reconfigure management agents based on the policy specification and terminate management agents. A management agent is designed to maintain information about what it asked to support, e.g., events to be monitored. More details about this information and about the interface are described in this Section.

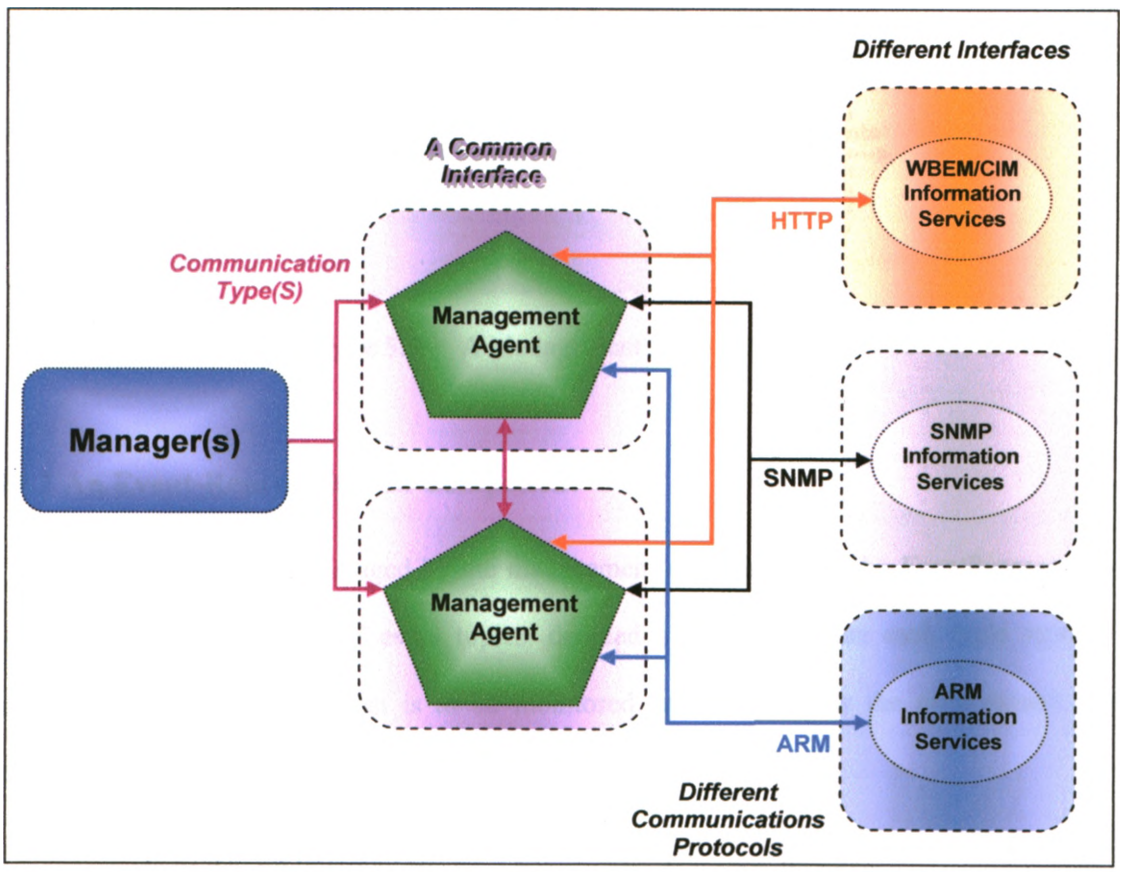


Figure 5.2: Management Services Interfaces

5.4 Management Agent Components

A management agent is designed to maintain the information needed to support policy deployment and to carry out the enforcement of the policy rules if asked to do so. The management agent has the *PolicyRepresentation*, *EventRepresentation*, *MessageRepresentation* and *ActionRepresentation* components.

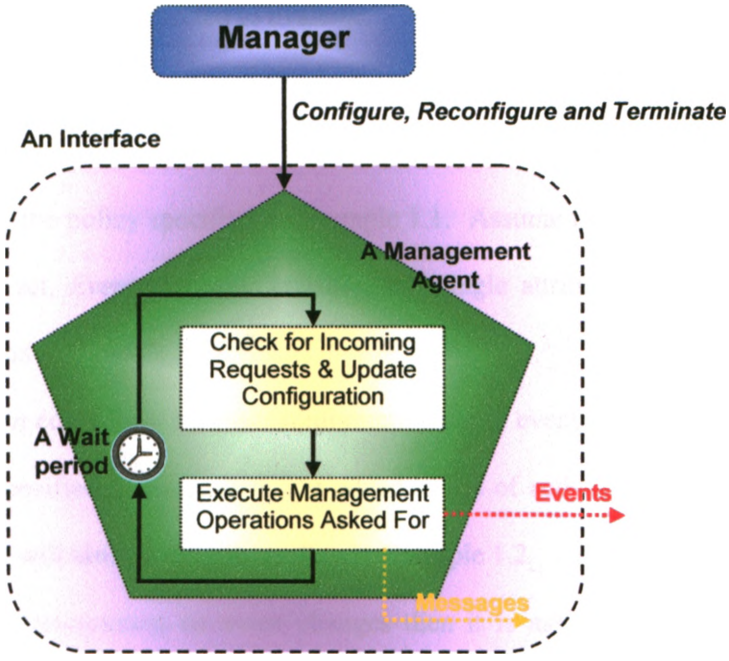


Figure 5.3: A Management Agent Structure

5.4.1 An Event-Representation Component

For each event to be managed by the management agent instance, the *EventRepresentation* component maintains a set of event tuples, denoted by *Events*, where each tuple represents information about an event that is to be monitored and triggered by the management agent instance. The event tuple is of the form:

<EventID, EventExpression, EventAttrtibutes>

The components of the event tuple are briefly described as follows:

- *EventID*: The event identifier.
- *EventExpression*: The event expression represents the event. In case the event is a primitive event, this is the logical expression (condition) that characterizes the event that the management agent is expected to monitor. In case the event is a composite event, this is the event expression that represents the event composition that the management agent is expected to evaluate.
- *EventAttributes*: This is a set of the attribute names used in the specification of the event.

As an example consider the policy specified in Example 1.1. Assume that the event identifier is *sessionidle_exp*. The set, *EventAttributes*, consists of a single attribute, *sessionidle*, and the event expression stored is *sessionidle >20*.

The *EventRepresentation* component maintains information about events that is independent of a policy. If the policy specified in Example 1.2 is added to the of active policies then the tuple created for Example 1.1 will also apply to the policy in Example 1.2.

If the expression characterizing an event changes then it is associated with a new event identifier.

5.4.2 A Policy-Representation Component

The *PolicyRepresentation* component maintains information about each policy associated with the management agent instance. There are two sets of tuples maintained by this component. Information about an event that is specific to a policy is stored as a tuple in the *Policy-Event* set. The tuples are in this form:

<PolicyID, EventID, EventTargets>

The event specified in the policy is referred to as a *policy event.* The following is a brief description of the tuple elements:

- *PolicyID*: The policy identifier.
- *EventID*: The event identifier.
- *EventTargets*: This is a set in which each element of the set represents information about a target (handler) that is interested in receiving and handling a notification of the event occurrence. The general form of an *EventTarget* element is the following:

<Target, {WaitPeriod|EventTimeWindows}>

- *Target:* This is a handle (a handle is the information needed to communicate with the process) representing the process to be notified of the event occurrence. Since an event may be common to more than one policy there is the potential for an event target to receive multiple event notifications. Our design assumes that the management agent which sends the event notifications also filters out multiple event notifications. The reason for doing this is that if a target is a manager then the manager would have to deal with multiple notifications. Our approach means that managers do not have this responsibility.
- *WaitPeriod*: The *WaitPeriod* is used only for primitive events. It is not applicable to composite events. The *WaitPeriod* is the time between evaluations of the condition specified in *EventExpression*. The management agent design takes into account that it may not be needed for a management agent instance to verify every condition at once. For example, consider a management agent instance that can evaluate the event expression (logical expression) *usersloginstotal >5* (used in the policy specified in Example 3.1) and the logical expression *sessionidle>20* (used in the policy specified in Example 1.1). The first logical expression may need to be verified every two minutes while the second logical expression needs to be verified every 10 minutes. This implies that a management agent instance should maintain how often an event expression (characterizing an event) should be evaluated. The reason for why the

time between condition evaluations may vary is that monitoring and evaluation of monitored data consumes resources. Providing a mechanism to control condition evaluations provides some control over resource consumption. Frequency is controlled by *WaitPeriod*. We assume that if this is not specified in the policy that a default frequency is assumed. This assumes that the policy specification language does allow for the specification frequency. The prototype in Chapter 7 supports this ability.

- *EventTimeWindow*: This is defined only for composite events and is not applicable to primitive events. The *EventTimeWindow* is the time period in which the event expression should be evaluated. More specifically, *EventTimeWindow* represents a time period that is delimited by two specific boundaries of time points, e.g., 10 minutes from current system time (now). This time period is used to evaluate the occurrence of a composite event.

Information that relates a policy, events and rules is represented as a tuple in PolicySet:

<PolicyID, EventID, EventTree, PolicyInterval, Rules>

The elements of the tuple are briefly described as follows:

- *PolicyID*: The policy identifier.
- *EventID*: The event identifier of the event which is either the policy event or a constituent event of the policy event of the policy identified by *PolicyID*. Example 3.1 has a policy event in the form of E_i E-SEQ E_j, where E_i and E_j are events that may be monitored by different management agents. The management agent monitoring for E_i would have a tuple with the policy identifier associated with the example, the event identifier for E_i and the event tree associated with E_i. The management agent monitoring for E_j would have a tuple with the policy identifier

associated with the example, the event identifier for E_j and the event tree associated with E_j.

- *EventTree*: This component maintains the event tree for the event identifier specified in *EventID*. We use an event tree to represent the relationship the event identified by *EventID* has with other events. Each node of the event tree corresponds to an event. The root node of the tree is the event associated with *EventID*. The leaf nodes of the event tree correspond to primitive events while all other nodes of the event tree represent composite events. An event tree with one node means that the event associated with *EventId* is primitive; otherwise the event associated with *EventId* is composite. The information for each node of the event tree is found in the *Policy-Event* set. We will show later in this Chapter how *EventTree* is used for composite event detection.
- *PolicyInterval*: This denotes the interval associated with the policy as discussed in Section 3.2.
- *Rules*: This is a set where each element of the set represents information about a policy rule. For example, consider a management agent instance that monitors the event used in the policy specified in Example 1.1, i.e., the agent instance evaluates the event that is characterized by the condition *sessionidle>20*. This agent instance may be configured to evaluate the policy rule *close the session*. Note that the policy rule associated with the policy of Example 1.1 has no condition that needs to be evaluated before enforcing the action. Thus a rule could end up being an action. The Rules set may be empty. For example 3.1 the management agent monitoring E_i would have this set be empty since it is not the policy event but rather a constituent event of the policy event. The general form of a rule is the following:

<Logical Expression, ActionInfo>

where *LogicalExpression* represents the condition and *ActionInfo* is information about the action. This is described in more detail later in this Section.

The use of the *PolicyRepresentation* component by agents is discussed in Section 5.6.

5.4.3 A Message-Representation Component

It may not be possible for one management agent to monitor all of the attributes used in the specification of a primitive event. An arrangement of agents may be required to monitor all attributes of a primitive event. More on this is discussed in Sections 5.6 and 5.7. Agents that collectively monitor the attributes periodically exchange information through messages. A message is a collection of attribute/value pairs. A management agent is designed to maintain a set, *Messages,* of message tuples related to the messages that need to be sent by an instance. This message tuple has the form:

<PolicyID, EventID, MessageAttributes, MessageTarget, PolicyInterval,WaitPeriod>

The components of the message tuple are briefly described as follows:

- *PolicyID*: The policy identifier.
- *EventID*: The event identifier of the event for which a message of attribute values is needed
- *MessageAttributes*: This is a set of the attribute names whose values are included in the message sent.
- *MessageTarget*: This indicates where to send the message. Typically, a message is sent to one or more other management agents.
- *PolicyInterval*: This denotes the associated interval with the policy as discussed in Section 3.2.
- *WaitPeriod*: This is the time to wait between sending of multiple instances of this message to its target.

A tuple is needed when an agent is only able to monitor part of a condition characterizing a primitive event and the attributes it is monitoring are sent to another agent that is collecting attribute values in order to evaluate the condition.

5.4.4 An Action-Representation Component

Management agents can be used to carry out actions, some of which may be on a periodic basis. For example, an agent may be used to execute the virus checking software every night at 2:00 AM. The management agent design maintains a list of action tuples where each tuple has the following form:

<ActionID, ActionExecutableHandle, ManagedResoursesList, {Schedule}>

where:

- *ActionID*: The action identifier.
- *ActionExecutableHandle*: Information about the action, e.g., the directory where the action executable resides, the library directories, how to call this action, etc.
- *ManagedResoursesList*: A list of the managed resources on which the agent carries out the action.
- *Schedule*: The time when the action should execute (optional). If the action is used in a rule then it is assumed that the *Schedule* is not used.

5.5 A Management Agent Interface

The interface has a method that enables the manager to configure the state of the management agent, such as asking the agent instance to sleep or shutdown if there are no active policies for the agent to support. The interface also provides methods that allow a manager to add, delete or modify policy and event tuples in a management agent instance. Similar methods are defined for message and action tuples.

5.6 Types of Management Agents

All agents have methods that permit for the manipulation of information described in Section 5.3.1. This Section describes how the interface is extended for three types of agents: monitoring agents, dynamic monitoring agents and manager agents. These agents have specific purposes.

Definition 5.1: A *monitoring agent* (*ma*) is a management agent that communicates with existing information services, is able to detect primitive events and is able to send messages of attributes/values pairs to dynamic monitoring agents.

Definition 5.2: A *dynamic monitoring agent* (*dma*) is a dynamically created management agent that collects data from monitoring agents. A *dynamic monitoring agent* is used to detect a primitive event that cannot be detected by single monitoring agent.

Definition 5.3: A *manager agent* (*manager_agent*) is a dynamically created management agent that is dedicated to detecting composite events.

The rest of this Section describes these agents.

5.6.1 Monitoring Agents

Primitive events require direct extraction of the information from the object being monitored. This requires some form of instrumentation which is specific to the managed object. There are many *information services* that provide some form of instrumentation. Each information service has an interface, but these interfaces are different for different information services. A *monitoring agent* (*ma*) is a management agent that communicates with existing information services, is able to detect primitive events and send notification messages of the detected events to the event targets. These targets could be manager agents and/or eventhandlers. The *ma* is able to send messages of attributes/values pairs to dynamic monitoring agents. Let us

consider the policy (denoted by *cpu_load_policy*) of Example 4.1. In this example, the attribute *cpu_load*, representing cpu load, is monitored. We assume that the monitoring is done by an information service e.g., SNMP agent. In the case of an SNMP agent the monitoring agent uses the SNMP GET command to get the current cpu load. The monitoring agent (denoted by *cpu_agent*) uses these values to determine if *cpu_load>95*. Thus the monitoring agent has a tuple in the *Events* set of this form <*cpu_load_evt, cpu_load>95, {cpu_load}*>. The monitoring agent has a tuple in the *Policy-Event* set of this form <*cpu_load_policy*, *cpu_load_evt*, {cpu_agent}, 1 minute>. Assume that the information service that monitors the cpu load can also monitor memory usage. It is then easy to add a policy that specifies an event based on the memory usage by asking the monitoring agent to add the event information to the *Events* set and information about the event and policy to the *Policy-Event* set.

The event target is the agent itself and the 1 minute indicates how often the monitoring agent is to get a cpu load value. The monitoring agent may either poll the information service or have the information service push the information. This is considered implementation dependent. If the information service is an SNMP agent and polling is used then the monitoring agent may issue a GET command to the SNMP agent every minute. There are different possibilities for the target of the event notification including the following: (i) If the association between an event and a rule is through an event-driven rule-based engine then a possible target of the event notification is an event handler; (ii) If the agent is to carry out the policy rules enforcement then the target may be itself; (iii) If the event is a constituent event in a policy event monitored by another management agent then the target is that management agent.

Attributes are associated with monitoring agents. When searching for an agent to use (more on this in Section 5.7), the search is based on the monitoring agents. The implementation of a monitoring agent uses an executable that has the code for maintaining event and policy information as described earlier in this Chapter. However, assume the monitoring agent needs to

interact with an information service such as SNMP. There are two cases. The first is that the service is already assumed to be started. This is often the case with SNMP which is started when the system boots. This is relatively easy to test by assuming that the monitoring agent has a method that tests to see if the information service is started. The second case is that the information service is not started. In this case it is assumed that the locations of the information service executables is known and can be found and used by the Distribution Mechanism (Chapter 4).

For the policy specified in Example 3.1, there is a monitoring agent for event E_i and another monitoring agent for E_j. These agents use UNIX scripts to collect the required monitoring information, i.e., the required attributes/values. If a new policy is added that requires that an event is to be generated if the CPU load is greater than 95, then this can be accomplished by requesting that the agent instance monitoring E_j also monitors this new event.

5.6.2 Dynamic Management Agents

A dynamic monitoring agent is dynamically instantiated and dedicated to detecting a primitive event when all the attributes in this primitive event cannot be collected by a single monitoring agent. The monitoring agents are dynamically configured to send the attributes/values pairs almost simultaneously to the dynamic monitoring with which they communicate. The code of the dynamic monitoring agent maintains the information discussed in Section 5.4. It provides the interface in 5.5 but it has additional methods that are used to configure the agent so that the relevant monitoring agents can communicate with it.

For Example 3.1, if the management system provides a monitoring agent that monitors users logins, another monitoring agent that monitors CPU load, and a third monitoring agent that monitors processes, then a dynamic monitoring agent is dynamically instantiated to receive the values of CPU load and the number of running processes attributes from the last two monitoring

agents, and then the dynamic monitoring agent is used to detect instances of the primitive event E_j. We note that the monitoring agents have a tuple in the set of tuples managed by the Message-Representation component.

5.6.3 Manager Agents

As indicated in Section 2.1, management systems provide an event handler to deal with the collection and distribution of events to other management components. One of these components may associate events with rules. An example is the Tivoli Enterprise Console (TEC). This type of component is sometimes referred to as an event-driven rule-based engine and typically has functionality that can be configured to implement the event operators introduced in Chapter 3 and thus allow for a complex analysis of the received events before determining an action. A useful mechanism to have is a management agent that can determine the action without having to send an event to the rule base engine. This allows for a distribution of event handling. Therefore, the manager agent is introduced. A manager agent is dynamically instantiated when the policy event is of type composite event. The manager-agent has an additional component, *EventMemory*, which maintains a list of the event notification messages received. The event notification messages to be received by a manager agent are those for the policy assigned to that agent. The *EventMemory* list consists of the following tuples:

<EventID, EventTimeStamp, EventAttributesValues, EventStatus>

where:

- *EventID*: The event identifier.
- *EventTimeStamp*: The time that the event occurred.
- *EventAttributesValues*: A list of the attribute name and value pairs.
- *EventStatus*: The status of the event, e.g. *Open* or *Closed* which indicates whether the event has already been processed or not.

The manager agent is dynamically configured to receive events from other management agents. These events are the constituent events that are used in the construction of the composite events this manager agent detects. The detection of composite events using manager agents is explained in Section 5.6.4.

5.6.4 A Manager-Agent Procedure for Handling Events

The diagram in Figure 5.4 illustrates a procedure for processing an event as it is received by the manager-agent. When an event is received, the procedure determines if the received event occurs within the time frame specified in the policy interval. If this is not the case, the procedure terminates. Otherwise the event is added to the set of event tuples maintained by the *EventMemory* component. The procedure then determines if the policy event or its constituent events have occurred. For an event in the set maintained by the *EventTree* of the *PolicyRepresentation* component, if all the preceding events have occurred then the event being examined has occurred subject to any existing timing constraints. For example, the occurrence of an event expression using the *E_SEQ* operator needs the two constituent events, LHS and RHS, events to occur and the LHS event in the expression to occur before RHS event.

The occurrence of an event causes a notification message to be sent to the targets specified in *EventTargets* and the *EventMemory* component to be updated. When the policy event, which is the last event represented in the *EventTree* component, occurs, then the rules specified in the policy, and represented in *Rules* of the *PolicyRepresentation* component, are executed and the procedure terminates.

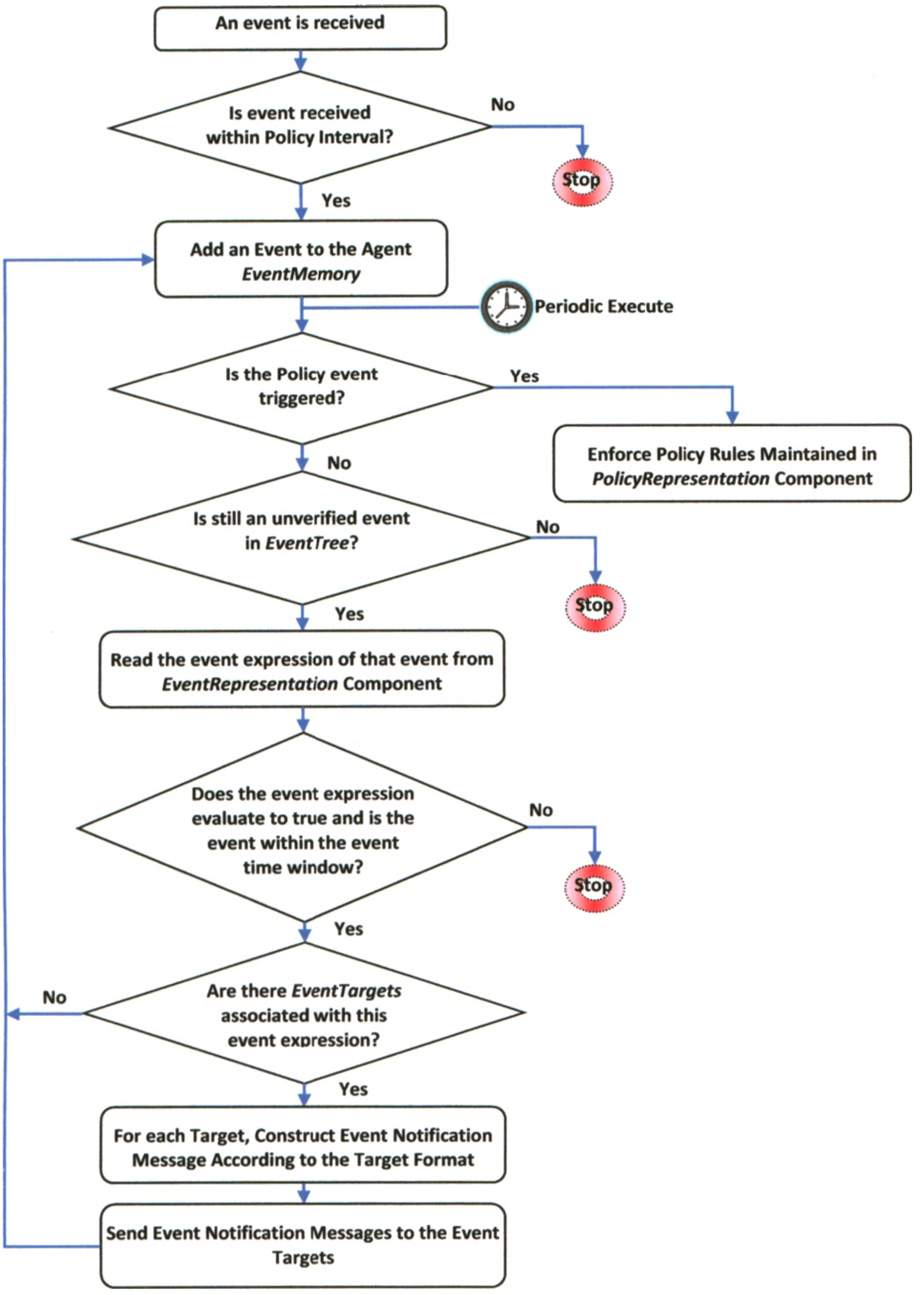


Figure 5.4: A Flow Diagram Illustrating a Manager-Agent Procedure for Handling Events

The procedure is designed to process the event expressions, represented in the *EventRepresentation* component, with events in *EventMemory* on a periodic basis, e.g., every 5 minutes, as long as no new event is received. The periodic execution of the procedure is necessary to detect events that are characterized by an event expression where the time constraint associated with the event represents an earlier time frame than the current time (e.g., the event should have happened 10 minutes ago).

5.6.5 Relationship between the Different Types of Agents

Figure 5.5 shows the communications between the three types of the management agents. For instance, a *ma* can detect primitive events and sends primitive event notification messages to *man-agent(s)*. A *ma* can send messages of attribute/value pairs only to *dma(s)*. A *dma* receives messages of attribute/value pairs from *mas*. A d*ma* detects a primitive event and can send a notification message of this primitive event to *man-agent(s)*. A *manager_agent* receives notification messages of primitive events from *ma(s)* and *dma(s)*. A *manager_agent* can receive notification messages of composite events from other *manager_agent(s)*. A *manager_agent* detects a composite event and can send a notification message of this composite event to other *man-agent(s)*.

For Example 3.1, if the management system relies on agents only to deploy the policy in this example (see Chapter 6 for another deployment approach), then a manager agent is dynamically created to receive the primitive events E_i and E_j from monitoring agents, and then the manager agent is used to detect instances of the composite event E. More details on how a manager-agent handles and detects a composite event are outlined in Section 5.6.4.

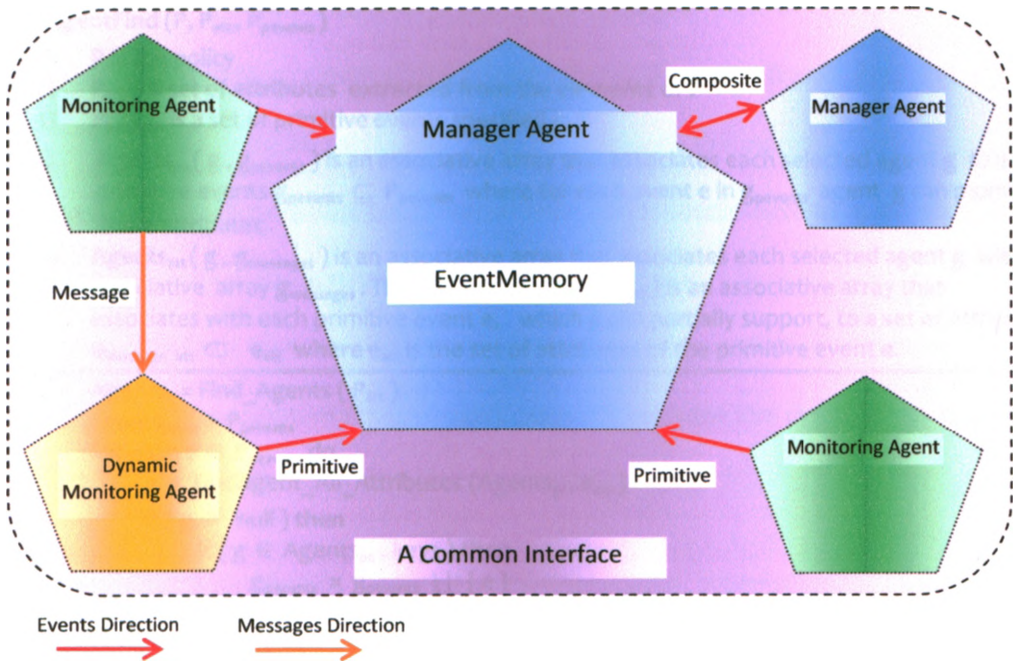


Figure 5.5: A Communication between Management Agents

5.7 Agent Matcher

The Agent Matcher component, introduced in Section 4.1, is used to find and configure the management agents that can enforce the policies. There are three tasks involved: finding potential agents, finding existing agent instances and developing an agent configuration. This Section describes these tasks and addresses the algorithms used to achieve these tasks.

5.7.1 Finding Agents

Upon adding a policy to the system, the Agent Matcher searches the agent repository to find a set of agents that can be used to monitor the system being managed so that the policy can be enforced. The matching process is based on the attributes extracted from the policy and the attributes that a management agent monitors. The agent finding algorithm is invoked when a policy is added (see Figure 5.6).

Algorithm AgentFind (P, P_{att}, $P_{pevents}$)

Input: 1) P is the policy
2) P_{att} is a set of attributes extracted from the elements of P
3) $P_{pevents}$ is a set of primitive events specified in P

Output: 1) $Agents_{pe}(g, g_{pevents})$ is an associative array that associates each selected agent g to a set of primitive events $g_{pevents} \subseteq P_{pevents}$ where for each event e in $g_{pevents}$ agent g can monitor all the e attributes.
2) $Agents_{att}(g, g_{messages})$ is an associative array that associates each selected agent g with an associative array $g_{messages}$. The $g_{messages}(e, e_{message\_att})$ is an associative array that associates with each primitive event e, which g can partially support, to a set of attributes $e_{message\_att} \subset e_{att}$ where e_{att} is the set of attributes of the primitive event e.

```
1.    Agents_p = Find_Agents ( P_att )
2.    Events_temp = P_pevents
3.    for each e ∈ P_pevents do
4.        g = FindAgent_All_Attributes (Agents_p , e_att )
5.        if ( g ≠ null ) then
6.            if ( g ∈ Agents_pe . keys ) then
7.                g_pevents = g_pevents ∪ { e }
8.            else
9.                Agents_pe ( g , g_pevents ) = Agents_pe ( g , { e } )
10.           end if
11.               Events_temp = Events_temp \ { e }
12.       end if
13.   end for
14.   if ( Events_temp == ∅ ) then
15.       return ( Agents_pe ( g , g_pevents ) , ∅ )
16.   end if
17.   for each e ∈ Events_temp do
18.       Agents_temp = Agents_p
19.       while (e_att ≠ ∅ ) and (Agents_temp ≠ ∅ ) do
20.           g = FindAgent_Any_Attributes ( Agents_temp , e_att )
21.           if ( g == null ) then
22.               FAIL
23.           else
24.               if ( g ∈ Agents_att . keys ) then
25.                   g_messages ( e , e_message_att ) = g_messages ( e , e_att ∩ g_att )
26.               else
27.                   Agents_att ( g , g_messages ) = Agents_att ( g , g_messages ( e , e_att ∩ g_att ))
28.               end if
29.           end if
30.           Agents_temp = Agents_temp \ { g }
31.           e_att = e_att \ g_att
32.       end while
33.       if ( e_att ≠ ∅ ) then
34.           FAIL
35.       else
36.           Agents_pe ( g , g_pevents ) = Agents_pe ( g_e-dynamic , { e } )
37.       end if
38.   end for
39.   return ( Agents_pe ( g , g_pevents ) , Agents_att ( g , g_messages ))
exit  algorithm
```

Figure 5.6: An Agent Finding Algorithm

The variable P_{att} represents the set of attributes extracted from the policy elements e.g., conditions. The variable $P_{pevents}$ represents the primitive events specified in the policy P. The algorithm in Figure 5.6 uses an associative array to represent the output of the algorithm. An associative array (or container), is a collection of keys and values, where each key is associated with one value. The key is an agent and a value is a set of events. The associative array, $Agent_{pe}$, represents agents that can monitor all of the attributes specified in at least one of the primitive events in $P_{pevents}$. The associative array, $Agent_{att}$, represents agents that can monitor some of the attributes specified in at least one of the primitive events in $P_{pevents}$.

The algorithm first determines the set of monitoring agents, represented by $Agents_p$, that can monitor any of the attributes found in P_{att} (Step 1). In Steps 3 to 13 the algorithm finds a subset of $Agents_p$ that can be used to monitor all the attributes associated with primitive events specified in $P_{pevents}$. However, it is not necessarily the case that all attributes in an event can be monitored by a single agent. Thus it is necessary to find a set of agents that collectively can monitor all the attributes used in the specification of a primitive event. This is done in Steps 17 to 36 of the algorithm.

For each event e in $P_{pevents}$ the algorithm searches to find a monitoring agent g that can monitor all the attributes specified in e (Step 4). If there is such an agent then in Step 6 the algorithm checks if the returned agent, g, can monitor other events in $P_{pevents}$. This is done by checking (Line 6) if g is a key in $Agents_{pe}$ (the set of keys is denoted by $Agents_{pe}$. keys). If g is a key, then the algorithm (Step 7) adds an event e to the set of events $g_{pevents}$ that g is able to monitor. If g is not a key then an entry (Step 9) is created in $Agents_{pe}$.

In Step 11 the algorithm reduces the set of primitive events by the event e since there is an agent that can be used to monitor all of the attributes used in the specification of e. After all events have been examined, the events in $Events_{temp}$ are events for which there is no single agent

that can monitor all of the attributes used in the specification of these events. If $Events_{temp}$ is empty then the algorithm finishes and returns $Agents_{pe}$ (Step 15). Otherwise, Steps 17 to 35 compute for each event e in $Events_{temp}$ a set of agents that can be used to monitor all of the attributes used in the specification of e. The approach is similar to that used in lines 3 to 13. The difference is in the purpose of the set of agents found and added to $Agents_{att}$. Agents in $Agents_{att}$ are selected such that each agent monitors a non-empty subset of the attributes in e_{att}. An event's attributes may be monitored by several monitoring agents. A monitoring agent may be used to monitor parts of multiple primitive events. The $g_{messages}$ (e , $e_{message\_att}$) is an associative array that associates each primitive event e, which g can partially support, to a set of attributes $e_{message\_att}$, where $e_{message\_att}$ is a subset of e_{att}. The $g_{messages}$ represents the values of $Agents_{att}$.

If the algorithm does not find an agent g in Step 20, then the algorithm terminates indicating a failure to find the agents needed to support a policy. The algorithm checks to see if g is already selected to partially monitor other events (Step 24). If so, then in Step 25 the algorithm associates the event e with the set of attributes that results from the intersection of e_{att} and the agent attributes g_{att} in $g_{messages}$; otherwise, i.e., g is not selected to partially monitor any other events yet (Steps 26 and 27). In this case, the algorithm in Step 27 associates g the $g_{messages}$, after adding an association in $g_{messages}$ of e with the result set of the intersection between e_{att} and g_{att}.

The algorithm fails if any of the attributes of any primitive event cannot be monitored (Step 34). The algorithm only succeeds when all the attributes of all primitive events of the policy can be monitored. In Step 36, an entry of the dynamic monitoring agent $g_{e\text{-}dynamic}$, which needs to be dynamically created to monitor the event e, is created in $Agents_{pe}$. As seen (Step 36) the algorithm not only finds the monitoring agents, but also determines the dynamic monitoring agents needed to support a policy. It is possible that an agent may be in both $Agents_{pe}$ and $Agents_{att}$ meaning that an agent can generate an event and monitor attributes' values for other agents. If the algorithm

succeeds, the outputs represented in Step 37 are the two associative arrays $Agents_{pe}$ and $Agents_{att}$.

In considering the time complexity of the algorithm, let us assume the following:

- n is the number of distinct attributes that are monitored by (i.e., associated with) any monitoring agent within the repository.
- m is the number of the primitive events in $P_{pevents}$.

The worst case is when each attribute can only be monitored by a single agent, i.e., the number of monitoring agents is also n. Considering the first and second *for* loops of the algorithm that iterate over the number of the primitive events m, these loops have a worst case running time of $O(n)$. Thus, the worst case the time complexity of the algorithm is $O(mn)$. More discussion can be found in Section 8.4.2.

5.7.2 Finding Agents Instances

Let X be the union of the set of keys of $Agents_{pe}$ and the set of keys of $Agents_{att}$. The next step is to determine for each agent in X if there is an instantiation of that agent i.e., to determine the set *instances(X)* of already defined instances of agent classes found in X. The algorithm to find *instances(X)* is a straightforward searching algorithm in the database of the existing agent instances. Therefore the set $X'=X \setminus instances(X)$ is the set of agent classes that need to be instantiated.

5.7.3 Configuration of Management Agents

Deploying and enforcing a policy by management agents (monitoring, dynamic monitoring and/or manager agents) requires the configuration of (i.e., instantiation of or the addition to or updating) the *PolicyRepresentation*, *EventRepresentation* *MessageRepresentation* and/or *ActionRepresentation* components of the management agents using the corresponding specified policy elements. This Section addresses these configurations. If the Finding Agents algorithm in

Section 5.7.1 successfully terminated, i.e., finds monitoring agents that can support the added policy P by monitoring all the attributes that are used in the specification of policy P's primitive events (denoted by $P_{pevents}$), then the algorithm will output the following two associate arrays:

1. $Agents_{pe}$ (g , $g_{pevents}$) is an associative array that associates each selected agent g to a set of the primitive events $g_{pevents}$, $g_{pevents} \subseteq P_{pevents}$ where for each event e in $g_{pevents}$ the agent g can monitor all the attributes of e.

2. $Agents_{att}$ (g , $g_{messages}$) is an associative array that associates each selected agent g with an associative array $g_{messages}$. The $g_{messages}$ (e , $e_{message\_att}$) is an associative array that associates with each primitive event e, which g can partially support, to a set of attributes $e_{message\_att} \subset e_{att}$ where e_{att} is the set of attributes of the primitive event e.

There are several cases to be considered in instantiating agents:

1. The policy event is a primitive event, e, and one monitoring agent, g, can monitor all the attributes in that event. In this case $Agents_{pe}$ (g , $g_{pevents}$) has one key and value pair where the key is g and the value is e. The array $Agents_{att}$ (g , $g_{messages}$) is NULL. The deployment of policy P requires the following tuples:

   - The tuple <P, e, e.tree, P.Interval, Rules> in the *PolicySet* set of g.
   - The tuple <P, e, {g, e.waitperiod}> in the *Policy-Event* set of g.
   - The tuple <e, e.expression, e.attributes> in the *Events* set of g.

   where *e.tree*, *e.waitperiod*, *e.expression*, *e.attributes* denote the event tree associated with e, the wait period associated with e, the event expression characterizing e, and the attributes used in the specification of e. The notation *P.Interval* is used to denote the interval associated with policy P.

2. The policy event is a primitive event, *e,* and multiple monitoring agents are needed to monitor all the attributes in event *e*. In this case $Agents_{pe}$ (g , $g_{pevents}$) has one key and value pair where the key is dynamic monitoring agent g and the value is *e*. Let G denote the set of keys in $Agents_{att}$. This corresponds to the set of agents that collectively can monitor the attributes of *e*. The deployment of the policy *P* requires the following tuples:
   - The tuple <P, e, e.tree, P.Interval, Rules> in the *PolicySet* set of g.
   - The tuple <P, e, {g, e.waitperiod}> in the *Policy-Event* set of g.
   - The tuple < e, e.expression, e.attributes> in the *Events* set of g.
   - for each g_i in G the following tuple needs to be added to the *Messages* set of g_i.

     <P, e, $e_{message\_att}$, g, P.Interval, e.waitperiod >

3. The policy event, *e*, is a composite event of the form e_i *e-op* e_j. Assume that g_i and g_j are agent instances configured to monitor e_i and e_j respectively. Assume that g is the manager agent used to detect the policy event *e* and carry out the policy rules. We assume that the agent instantiations needed to monitor all other events in the event tree have been determined and configured. Thus the focus in the discussion in this case is g, g_i and g_j. The deployment of policy *P* requires the following tuples:
   - The tuple <P, e, e.tree, P.Interval, Rules> in the *PolicySet* set of g.
   - The tuple <P, e, {e.targets, e.timewindow}> in the *Policy-Event* set of g.
   - The tuple < e, e.expression, e.attributes> in the *Events* set of g.
   - The tuple <P,e_i,{g, e_i.waitperiod |e_i.timewindow}> in the *Policy-Event* set of g_i.
   - The tuple <P, e_j,{g, e_j.waitperiod |e_j.timewindow}> in the *Policy-Event* set of g_j.
   - The tuple < e_i, e_i.expression, e_i.attributes> in the *Events* set of g_i.
   - The tuple < e_j, e_j.expression, e_j.attributes> in the *Events* set of g_j.

There is one note to be made about this case. First, g, g_i an g_j are not necessarily distinct. In other words, g could be used to detect e_i and e_j, if e_i and e_j are composite events. Currently in our prototype if e_i and e_j are composite events then g is configured to detect these events. This is done for efficiency reasons. The goal of this thesis is not to determine an optimal arrangement of agents but rather to ensure that the agent model is flexible enough to support a wide range of arrangements. The flexibility of the model comes from the features of the model which enable it to accommodate any arrangement of agents that might be adopted to deploy policies. Future work will carefully explore issues related to optimal arrangements of agents.

5.8 Discussion

Support for Legacy Code: The design of management agents needed to address several challenges. First, legacy information services such as SNMP and scripts need to be supported. This is done by having monitoring agents. The monitoring agents have the interface described in 5.5; however these agents hide their interaction with the legacy agents, i.e., information services. For example, a monitoring agent that interacts with an SNMP agent is implemented with the interface described in Section 5.5 but it is also implemented to be able to compose and receive messages from SNMP agents. In Section 5.6.1 it is assumed that the monitoring agent implementation is based on two executables, with one of those executables being an SNMP agent or some other legacy information service and the other being the executable of the monitoring agent itself. The instantiation of a monitoring agent requires that the legacy information service be instantiated and that the executable that has the code for event handling be retrieved. In Chapter 4, we described the need for a Distribution Mechanism component that is provided with enough information to start an agent instance. This information includes the location of executables needed. The model introduced in this chapter allows for implementations including:

- The use of mobile agents (e.g.,[48,123,160]). This approach typically assumes that the legacy information service already resides on the managed object. The executable that deals with event handling is similar to the concept of mobile agent.
- The two executables that constitute a monitoring agent are instantiated together.
- The executable with the legacy code is instantiated but not the other executable. It is often the case that SNMP agents already exist as a process.

Currently the implementation uses the second and third approaches. The use of mobile agents typically assumes the existence of a process that can take mobile code, which normally requires an infrastructure in place. We wanted to use an approach that is relatively easy to incorporate within existing management systems.

We would like to note that the development of monitoring agents where the information service is SNMP can be aided by tools such as those found in [126].

Distribution of Management functionality: Centralized management systems do not scale well (e.g., [134,135,149]). A good deal of work has been done to address this including (e.g.,[52,161]). The work in [161] first introduced the concept of distributed management functionality in the seminal work on the Management by Delegation (MbD) model. The IETF has integrated the MbD model into their management frameworks based on the work found in [135] which uses the IETF Script MIB to transfer management scripts to a manager that is able to receive these scripts and execute them. Examples of implementations are found in [7,101]. Our design allows for this functionality with its ability to allow actions to be associated with rules. The actions can be implemented using scripts. There have been attempts to develop peer-to-peer management systems (e.g., [30,123,160]). This requires flexibility in sending event notifications. Although our work does not assume peer-to-peer management systems, we believe that the agent design described here can be used in these types of management systems.

Support for Agent Finding and Agent Matching: Chapter 4 describes the need for several services to determine agents that can support the policy. This chapter presented an Agent Finding service based on associating attributes with monitoring agents. Chapter 8 provides an analysis of this algorithm.

Relationship to Existing Management Systems: Event notifications can be used by multiple management applications. The agent design requires that event targets be defined. These targets can be management entities that include an event-driven rule-based engine, an event handler or any other management application. Agent executables can be placed on machines using existing management system distribution mechanisms as seen in Chapter 7.

5.9 Chapter Summary

The Chapter presented an information model for agents and their instantiations. The management agent design, interface, components and types are addressed. The information model was used as the basis for the Agent Matcher component. This Chapter highlights how our proposed PBM system architecture can automatically identify and configure and reconfigure management agents in response to changes in policies. The proposed solution is based on the matching between the management operations that are carried out by the management agents and the policies. The matching process relies on the attributes that the agents can monitor and the extracted attributes from the components of the policies.

Chapter 6

MAPPING MECHANISMS

Chapter 3 defines an event as a message that notifies of a change in system state that is of interest. Different management systems have different formats of these messages. Events are associated with rules. Managers that interpret rules may use an *event-driven rule-based engine*. The rules interpreted by a specific event-driven rule-based engine require that rules be specified in a particular format. This Chapter discusses the generation of the event messages and the rules from the language used to specify the policy.

6.1 Introduction

A set of templates is used to guide mapping from the specification of a policy to the generation of messages associated with events and rules that can be understood by the management system's event-driven rule-based engine. A template is associated with a policy element. Each class in the policy information model defined in Figure 3.2 is a policy element. A

template has a set of variables. The template variables are instantiated based on a mapping that defines how a policy element should instantiate the template variables. More formally we have the following:

Definition 6.1: *A transformation rule* r = (p,α,t) is a triple where

p : This is the specified policy element

t : This is the template associated with the policy element, p.

α : This is a mapping that defines how the variables of t are assigned values.

Multiple transformations rules are applied to a policy. A transformation is associated with a policy element. A transformation rule may be applied more than once in transforming policy elements, typically for the elements of the same type. It should be noted that not all policy elements need to have transformation rules. For example, actions correspond to names of executables. It is not necessary to use a template for this.

6.2 Event Format Mapping

The class representing *EventExpression* in the UML diagram in Figure 3.2 is further expanded in Figure 3.4. There are two subclasses. One is labelled condition. This models a condition and is used to characterize a primitive event. The other subclass models a composite event. A transformation is associated with each of these classes. The rest of this Section discusses the transformations needed for mapping specified events to a form that can be understood by management systems.

6.2.1 Event Format Mapping for Primitive Events

Typically in management systems, event class definitions are used to define an event type. Instances of events of the same type have an identical message structure. The event class

definitions define a set of attribute names and the type of each of these attributes. Some of the attributes are found in each event type e.g., source identifier that generated the event, timestamp, event name and condition name. These are referred to as *common* attributes. The rest of the attributes are specific to the policy e.g., an attribute representing the length of an idle session. These are referred to as *policy-specific attributes*. In HP-Openview [61] and CA-Unicenter [23] event types are expressed using the C language structure type, while in Tivoli [151] event types are expressed using the *Basic Recorder of Objects in C (BAROC)* language where the attributes are called *slots*.

The following example shows how templates can be used to generate the definition of event types that can be understood by Tivoli.

Example 6.1:

This example is based on the example policy stated in Example 1.1. The condition characterizing the event is restated here: *login session is idle for more than 20 minutes*. The policy element, p, is the event expression. The template for primitive events is shown in Figure 6.1. The resulting BAROC file is shown in Figure 6.2. The mapping α determines the list of policy-specific attributes, which is used to instantiate a template variable that represents a list of policy-specific attributes. Other template variables that need values to be assigned include the event name and the condition name. Slots of the *common* attributes include the event name and the condition name. Slots of the *policy-specific attributes* include the length of the idle session (*sessionidlelong)*, the process identifier of the session *(processid)* and the user that initiated the session (*userid*). The *policy-specific attributes* are assigned values when the event is generated. The management agent that monitors the attributes and generates an instance of the event has a method that is able to create the event message in the correct format.

```
#####################################################################
#   Main Class Definition for Primitive Event 'Event-ID-Variable'
#   Automatically generated by the PMagic Model
#   On  System Date-Time at Template Instantiation
#####################################################################
TEC_CLASS:
    Event-ID-Variable ISA EVENT
    DEFINES {
              source: default= 'PMagic';
              sub_source: default= 'PMagic_Agents';
              sub_source_port:  INTEGER , default=0 ;
              operation_number:  INTEGER , default=0 ;
              severity: default = HARMLESS;
              eventname: STRING, default='Event-ID-Variable';
              condition_name: STRING, default='Condition-ID-Variable';
               [ Attribute-Name :  Attrubute-Type ; ]*
            };
END
#########################################
#   End Event 'Event-ID-Variable' Class
#########################################
```

Figure 6.1: Tivoli TEC BAROC Template File for Primitive Events

```
#####################################################################
#   Main Class Definition for Primitive Event 'session_Idle'
#   Automatically generated by the PMagic Model
#   On  Sun Nov 16 04:44:45 EST 2008
#####################################################################
TEC_CLASS:
    session_Idle ISA EVENT
    DEFINES {
              source: default= 'PMagic';
              sub_source: default= 'PMagic_Agents';
              sub_source_port:  INTEGER , default=0 ;
              operation_number:  INTEGER , default=0 ;
              severity: default = HARMLESS;
              eventname: STRING, default='session_Idle';
              condition_name: STRING, default='session_Idle_20_mins';
              processid: INTEGER;
              sessionidlelong: INTEGER;
              userid: STRING;
            };
END
#########################################
#   End Event 'session_Idle' Class
#########################################
```

Figure 6.2: Tivoli TEC BAROC File for the session_Idle Primitive Event

6.2.2 Event Format Mapping for Composite Events

The event specified in Example 6.1 is a primitive event. However, the event could be a composite event. As stated in Chapter 3, composite events are specified in one of these two forms:

- E_i *eop* E_j, where *eop* is either E-SEQ, E-OR or E-AND
- *eop* E_j [arg] , where *eop* is either E-COUNT or E-NOT.

Each composite event requires that the constituent events are determined. An event tree needs to be created where each node corresponds to an event. The leaf nodes of the event tree correspond to primitive events while all other nodes of the event tree represent composite events. A post-order traversal (the reason for this will become clear in Section 6.3.1) of this tree is carried out where the appropriate transformation rule is applied to each node. A transformation is needed for a composite event element. The specification of a composite event type typically consists of attributes for the event name, event operator and the operands of the events. The algorithm *MapEvent* is shown in Figure 6.4 and will be described later.

Example 6.2:

For Tivoli a template for composite events is seen in Figure 6.3. This template is used with either of the composite events forms that are used to specify the composite events. An example of the resulting BAROC file is shown in Figure 6.8. A more detailed discussion of this template is presented later in this Section.

```
#####################################################################
#    Main Class Definition for Composite Event 'Event-ID-Variable'
#    Automatically generated by the PMagic Model
#    On  System Date-Time at Template Instantiation
#####################################################################
TEC_CLASS:
    Event-ID-Variable ISA Event
    DEFINES {
              source: default= 'PMagic';
              sub_source: default= 'TEC';
              sub_source_port:  INTEGER, default=0;
              operation_number:  INTEGER, default=0;
              severity: default = HARMLESS;
              event_name: STRING, default='Event-ID-Variable';
              event_operator: STRING, default='Event-Operator-Variable';
              left_event_name: STRING, default='Left-Event-ID-Variable';
              left_date_reception: INT32, default=0;
              left_server_handle: INTEGER, default=0;
              left_event_handle: INTEGER, default=0;
              right_event_name:STRING,default='Right-Event-ID-Variable';
              right_date_reception: INT32, default=0;
              right_server_handle: INTEGER, default=0;
              right_event_handle: INTEGER, default=0;
              count_number: INTEGER, default=0;
            };
END
########################################
#    End Event 'Event-ID-Variable' Class
########################################
```

Figure 6.3: Tivoli TEC BAROC Template File for Composite Events

The MapEvent Algorithm

The algorithm *MapEvent* is depicted in Figure 6.4. In this algorithm the variable *E* represents an event tree node. The algorithm determines (line 1) if the event node *E* has any outgoing edges i.e., whether the node is a leaf node. A leaf node indicates that the node corresponds to a primitive event. In the case of a primitive event, the algorithm returns the result of instantiating a template with the event in node *E* (line 2) using the *Apply_Primitive_Template* function. *Apply_Primitive_Template* represents the transformation rule that applies to policy elements that

are primitive events, while *Apply_Composite_Template* represents the algorithm of the transformation rule that applies to policy elements that are composite events.

```
Algorithm MapEvent ( E )

Input:  1)  E is an event tree node

Output: 1)  E_mapped is a set of instantiated templates representing the specified event in the node E.

begin
1.      if ( OutgoingNodes(E) == 0 ) then
2.          return Apply_Primitive_Template (E )
3.      end if
4.      if (OutgoingNodes(E) == 1 )
5.          E_mapped = MapEvent ( leftnode(E) ) ∪ Apply_Composite_Template(E)
6.          return E_mapped
7.      else
8.          E_mapped = MapEvent ( leftnode(E) ) ∪ MapEvent ( rightnode(E) ) ∪
                                                  Apply_Composite_Template(E)
9.          return E_mapped
10.     end if
end
```

Figure 6.4: A MapEvent Algorithm

Composite events are the focus of lines 4 to 10 of the algorithm. There are two cases that are characterized by the number of outgoing edges from node *E*. If the event node *E* represents a composite event that uses a unary operator (line 4) then there is one outgoing edge from *E*. A template is initialized, using the function *Apply_Composite_Template*, for representing the composite event *E*. The algorithm is recursively called with the child of *E* to generate the templates corresponding to the events in the sub-tree rooted at the child of *E* (line 5). Lines 7 to 10 of the algorithm handle an event node representing a composite event that uses a binary operator.

The *Apply_Composite_Template* function represents the mapping α from Definition 6.1. For Tivoli, the *Apply_Composite_Template* function extracts from *E* the values of the slots *event_name, event_operator, left_event_name, right_event_name* and *count_number*. The *left_event_name* and *right_event_name* represent the names of the events that compose the composite event in *E*.

The slots *left_date_reception, left_server_handle, left_event_handle, right_date_reception, right_server_handle* and *right_event_handle* represent the BAROC implementation slots that uniquely identify the events instances (i.e., the left and right events in the event expression) that trigger the composite event. These slots, i.e., the slots that uniquely identify the event instances, are assigned values when the event is generated. The *date_reception* is the event timestamp slot; the value represents the number of seconds since the epoch (i.e., since midnight of January 1, 1970) to the occurrence of the event. If more than one instance of the event is received within the same second, the *event_handle* slot is used to distinguish between such instances. For the first event instance received, the *event_handle* will be given the value 1 and incremented for each subsequent event instance received within the second. The *server_handle* slot value represents the event server location, e.g., *event_server* slot value is 1 for the event server in the local Tivoli Management Region.

We note that although the *Apply_Composite_Template* function is specific for a management system the implementation of *MapEvent* is not. A Java abstract class can be defined with the *MapEvent* method. This class can be subclassed for each specific management system.

Example 6.3:

This example is based on the example policy stated in Example 3.1. The event expression is that *the total number of user logins is greater than 5* **followed by** *the CPU load is greater than 90 and the total number of processes running is greater than 3*.

This policy consists of a composite event, *users_cpu_process_High* of the form *users_Limit* E-SEQ *cpu_process_High,* with two primitive events where,

1) *users_Limit* is a primitive event that is characterized by the condition *total number of logins is greater than 5*. The logical expression that represents this condition is *usersloginstotal>5*.
2) *cpu_process_High* is a primitive event which is characterized by the condition *the CPU load is greater than 90 and the total number of processes running is greater than 35*. The logical expression which represents this condition is *cpuload>90 && cpuprocesstotal >35*, i.e., the *cpu_process_High* event condition has two attributes to be evaluated.

The *users_cpu_process_High* event is parsed into an event tree, as shown in Figure 6.4.

The application of the *MapEvent* algorithm, results in three BAROC classes: *users_Limit.baroc*, *cpu_process_High.baroc* and *users_cpu_process_High.baroc* (Figures 6.6 , 6.7 and 6.8 respectively). The *users_Limit.baroc* and *cpu_process_High.baroc* classes are constructed using the transformation rule for primitive events. The *users_cpu_process_High.baroc* class is constructed using the transformation rule for composite events that assumes that the event operator is binary. The variables of this template include the event_name *users_cpu_process_High*, the left_event_name *users_Limit*, the right_event_name *cpu_process_High* and the event_operator *E_SEQ*.

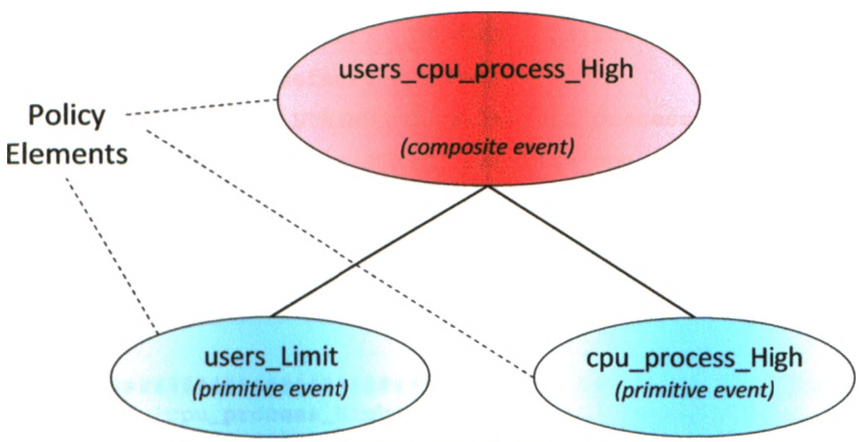


Figure 6.5: An Example of Composite Event Tree

```
#####################################################################
#   Main Class Definition for Primitive Event 'users_Limit'
#   Automatically generated by the PMagic Model
#   On  Sun Nov 16 15:44:11 EST 2008
#####################################################################
TEC_CLASS:
    users_Limit ISA EVENT
    DEFINES {
              source: default= 'PMagic';
              sub_source: default= 'PMagic_Agents';
              sub_source_port:  INTEGER , default=0 ;
              operation_number:  INTEGER , default=0 ;
              severity: default = HARMLESS;
              eventname: STRING, default='users_Limit';
              condition_name: STRING, default='users_More_than_5';
              hostname: STRING;
              userlogintotal: INTEGER;
            };
END
#########################################
#   End Event 'users_Limit' Class
#########################################
```

Figure 6.6: Tivoli TEC BAROC File for the users_Limit Primitive Event

```
#####################################################################
#   Main Class Definition for Primitive Event 'cpu_process_High'
#   Automatically generated by the PMagic Model
#   On  Sun Nov 16 15:44:11 EST 2008
#####################################################################
TEC_CLASS:
   cpu_process_High ISA EVENT
   DEFINES {
             source: default= 'PMagic';
             sub_source: default= 'PMagic_Agents';
             sub_source_port:  INTEGER , default=0 ;
             operation_number:  INTEGER , default=0 ;
             severity: default = HARMLESS;
             eventname: STRING, default='cpu_process_High';
             condition_name: STRING,default='cpu_over_90_and_process_Cond';
             cpuload: REAL;
             cpuprocesstoltal: INTEGER;
             hostname: STRING;
            };
END
#########################################
#   End Event 'cpu_process_High' Class
#########################################
```

Figure 6.7: Tivoli TEC BAROC File for the cpu_process_High Primitive Event

```
###################################################################
#   Main Class Definition for Composite Event 'users_cpu_process_High'
#   Automatically generated by the PMagic Model
#   On  Sun Nov 16 15:44:11 EST 2008
###################################################################
TEC_CLASS:
    users_cpu_process_High ISA Event

    DEFINES {
              source: default= 'PMagic';
              sub_source: default= 'TEC';
              sub_source_port:  INTEGER, default=0;
              operation_number:  INTEGER, default=0;
              severity: default = HARMLESS;
              event_name: STRING, default='users_cpu_process_High';
              event_operator: STRING, default='E_SEQ';
              left_event_name: STRING, default='users_Limit';
              left_date_reception: INT32, default=0;
              left_server_handle: INTEGER, default=0;
              left_event_handle: INTEGER, default=0;
              right_event_name:STRING,default='cpu_process_High';
              right_date_reception: INT32, default=0;
              right_server_handle: INTEGER, default=0;
              right_event_handle: INTEGER, default=0;
              count_number: INTEGER, default=0;
            };
END
#######################################
#   End Event 'users_cpu_process_High' Class
#######################################
```

Figure 6.8: Tivoli TEC BAROC File for the users_cpu_process_High Composite Event

6.3 Mapping Policies

Mapping policy events to an event definition in the underlying management system is just one part of the overall process. This Section addresses the subsequent problem of mapping policy to a form that can be understood by the management system. There are two issues related to mapping that must be considered. The first is that of composite event detection. Detecting primitive events is not sufficient for determining if a composite event has been used. The approach commonly used in management systems is to use a rule-base system. A second issue is that once an event is detected, the set of *if condition then actions* rules associated with the specific

event should be evaluated. An event-driven rule-based system can also be used for this evaluation. More details about the structure and the components of event-driven rule-based systems can be found in [71, 73, 90]. However, event-driven rule-based systems imply that event detection is centralized. Computation can be offloaded by allowing management agents to detect composite events and evaluating *if condition then actions* rules. This Chapter discusses mappings from policies to both of these approaches.

6.3.1 Mapping Policies to a Rule-Engine Platform

Management systems such as Tivoli [151], HP-Openview [61], CA-Unicenter[23], etc, are event-driven rule-based systems. An event-driven rule-based system has a set of rules, which is referred to as a rulebase, where a rule has the following form:

on *event* [event filter]

[**when** *condition*]

do *action*

When an event instance is received then a condition is evaluated to determine actions to be taken. Event-driven Rule-based systems maintain information about event instances that have been received. The term *event* is used to denote any event type of received event instance. The term *event filter* is used to denote the set or subset of event types found in specification of the policy event. When an event instance is received, if the type of the received event instance is found in *event filter*, then a condition is evaluated to determine if the specified action should be executed. The event-driven rule-based system evaluates the rules in the order that the rules are presented in the rulebase. For a received event instance there is a possibility that it can satisfy multiple event filters. There are two reasons for this: The first reason is that for an event there might be multiple actions possible depending on the values of the attributes of the event. The

second reason is that an event can be used in the specification of the event expression in multiple policies.

Example 6.4: This example shows how the policy of Example 1.1 is mapped to a rule typically found in a rule base system. In this example, the set of events specified in the policy of Example 1.1 contains only one event, the *session_Idle* event. It is assumed that E_r denotes the type of the received event instance.

on E_r in { *session_Idle* }

do *close the idle session*

Example 6.5: This example describes the rules for the event-driven rule-based system for the policy specified in Example 6.3; **if** *the total number of user logins is greater than 5* **followed by** *the CPU load is greater than 90 and the total number of processes running is greater than 35,* **then** *block any new user logins.* A policy interval associated with this policy is denoted by *normal_working_hours*. Example 6.3 specifies the policy event as a composite event *users_cpu_process_High* in this form:

users_cpu_process_High = *users_Limit* ***E-SEQ*** *cpu_process_High,*

where *users_Limit* and *cpu_process_High* are primitive events. The received event instance is denoted by e_r and $e_r.timestamp$ denotes the time that it was generated at. E_r denotes the type of the received event instance.

As discussed in Chapter 3, the policy interval is used to specify the interval in which the policy is valid or active. This suggests that the policy event as well as the constituent parts of the policy event should only be checked if these events occur within the policy interval. Rule 1 is used to determine if the event instance received occurs in the specified policy interval. The interval is checked only if the type of the received event instance is of the composite event or one of the constituent events.

Rule 1: **on** E_r in { *users_Limit, cpu_process_High, users_cpu_process_High* }

when checkInterval(*normal_working_hours* , e_r.*timestamp*)

do continue

This rule should be before any of the other rules generated for this example policy. If the received event does not occur within the specified interval then no further processing is needed. Rule 2 is used to generate a notification of event *users_cpu_process_High* if E_r is either of type *users_Limit* or *cpu_process_High*. It is assumed that *occ(users_Limit* E-SEQ *cpu_process_High)* is able to check if instances of type *users_Limit* and *cpu_process_High* have both occurred since rule base systems can keep track of event instances that have been received. Assume T is the time window for the event expression that characterizes the composite event.

Rule 2: **on** E_r in { *users_Limit* , *cpu_process_High* }

when occ (*users_Limit* E-SEQ *cpu_process_High*) T

do trigger a message notifying the occurrence of event *users_cpu_process_High*

The third rule, Rule 3, associates the action to be taken if the composite event, which is the policy event, has occurred.

Rule 3: **on** E_r in { *users_cpu_process_High* }

do execute action *block any new user logins*

Policy Mapping

As can be seen from Example 6.5, mapping a policy to rules in a rule–based system requires the following issues to be addressed: determining the policy interval, ensuring policy enforcement and handling composite events.

Interval Checking: A policy may specify an interval in which constituent events of the policy event must occur. A rule is needed to determine if a received event instance occurs within the desired interval. The template includes a variable x for the event filter. This variable denotes a set of event types and thus should be instantiated with the type of the events used in the specification of the policy event. There is another variable y for the specified interval that should be checked in the condition. The transformation rule instantiates these variables based on the corresponding policy specified elements. The following is an example of the Interval Checking rule:

on E_r in $\{x\}$

when checkInterval(y , $e_r.timestamp$)

do continue

We note that the above is an abstract representation of an actual template that would be used. An example template used for Tivoli is found in Appendix D.

Policy Enforcement: The template has variables representing the information of the routine to be invoked, e.g., the routine name, routine type (such as Java, UNIX shell script, etc.) and the parameters of the routine. The transformation rule instantiates these variables based on the policy rule e.g., the specified set of the condition-actions rules of the policy and the policy event attribute names. Other variables represent the policy event name and the event time window. An example template used for Tivoli is found in Appendix D.

Policy Composite Events: For each composite event in the policy event tree a detection rule is created. To facilitate the creation of this rule a template is needed for each event operator. One variable of the template denoted by x, is the event filter. The transformation rule initializes this variable with the set of event types that constitute the composite event which are denoted by x_i and x_j. Another variable of the template is the time window, denoted by y, which is definable

with the composite event expression. The action to be taken is to send a notification message, denoted by template variable *z*, of the detected composite event. The following is an example rulebase representation for a binary event operator that denoted by *EOP*:

on E_r in $\{ x \}$

when occ $(x_i$ EOP $x_j)$ y

do *trigger a message notifying occurrence of event z*

Unary event operators require another rulebase representation. More details on the construction of the detection rules are described in Section 6.3.2 and Appendix D.

6.3.2 Constructing Composite Event Detection Rules

The *MapEventDetectionsRules* algorithm (presented in Figure 6.9) is used to create a set of rules to be used for composite event detection. Leaf nodes in the event tree represent primitive events. All other nodes represent composite events. The rules generated for detecting events at the $i+1^{th}$ level of the event tree must precede the rules that detect events at the i^{th} level of the event tree. The reason for this has to do with the way event-driven rule-based systems are organized. Rules are evaluated in the same order that they are presented.

We note that the algorithms for generating instantiated templates representing messages notifying of an event (*MapEvent*) and the algorithm *MapEventDetectionsRules* for generating rules for complex event detection are presented separately. This is done for presentation purposes. Practically, only one traversal of the event tree is actually needed.

There is a template associated with each composite event operator. The event operator, *eop*, in Lines 6, 8, 13, 15 and 17 correspond to each of the event operators: E_COUNT, E_NOT, E_SEQ, E_AND, and E_OR respectively (see Appendix D for examples of Tivoli TEC Rule Templates).

Algorithm MapEventDetectionsRules (E)

Input: 1) **E** is an event tree node

Output: 1) E_{ruled} is a set of instantiated templates representing the rules for detecting the specified composite event in node **E** .

```
begin
1.    if ( OutgoingNodes(E) == 0 ) then
2.        return {}
3.    end if
4.    if (OutgoingNodes(E) == 1 )
5.        if (eop == "E_COUNT" )
6.            E_ruled = MapEvent DetectionsRules( leftnode(E)) U
                                                  Apply_E_COUNT_Template (E )
7.        else
8.            E_ruled = MapEvent DetectionsRules( leftnode(E)) U
                                                  Apply_E_Not_Template (E )
9.        end if
10.       return E_ruled
11.   else
12.       if (eop == "E_SEQ" )
13.           E_ruled = MapEvent DetectionsRules( leftnode(E)) U
                        MapEvent DetectionsRules( rightnode(E)) U Apply_E_SEQ_Template (E )
14.       else if (eop == "E_AND" )
15.           E_ruled = MapEvent DetectionsRules( leftnode(E)) U
                        MapEvent DetectionsRules( rightnode(E)) U Apply_E_AND_Template (E )
16.       else
17.           E_ruled = MapEvent DetectionsRules( leftnode(E)) U
                        MapEvent DetectionsRules( rightnode(E)) U Apply_E_OR_Template (E )
18.       end if
19.       return E_ruled
20.   end if

end
```

Figure 6.9: A MapEventDetectionsRules Algorithm

The mapping to an event-driven rule-based system is illustrated with the Tivoli TEC event-driven rule-based engine platform. The rule language provides a simplified interface to the

Prolog programming language, which is the language actually used internally by the TEC event-driven rule-based platform. A TEC event server can have only one active rulebase. A rulebase is a collection of definitions of event classes and rules that apply to those event classes. The following example shows how the use of templates and the transformation rules steer the automatic mapping of polices to Tivoli.

Example 6.6: This example shows the mapping of the policy described in Example 6.3 to TEC event-driven rule-based system. The generated rules are put into a text file called *load_Control.rls*.

- The first rule, which handles the verification of a policy interval, is appended to the TEC ruleset *load_Control.rls*. This rule will be created using the transformation rule that uses the template *IntervalChecking*. A Tivoli TEC rule template for *IntervalChecking* is given in Appendix D. The variables for this template include the policy_name, interval_name and a set of event names that constitute the policy event. The transformation rule instantiates these variables based on the policy specified elements, e.g., policy_name to *load_Control*, interval_name to *normal_working_hours* and a set of events names: *users_Limit, cpu_process_High and users_cpu_process_High.* The policy element *p* in the used transformation rule is the policy interval *normal_working_hours*. The result of the transformation rule is shown in Figure 6.10.

- The second rule appended to *load_Control.rls* will be the rule for detecting the policy composite event *users_cpu_process_High* which is the policy element *p*. The *E_SEQ* template is used which corresponds to the E_SEQ event operator. This template is another TEC rule that is constructed to evaluate any event expression that uses the E_SEQ event operator. A Tivoli TEC rule template for composite event detection of the *E_SEQ* operator is given in Appendix D. The template's variables are the

composite event name, the composite event constituent events and the event time windows that are specified in the event expression. The transformation rule automatically instantiates these variables based on the policy specified elements, e.g., the composite event name to *users_cpu_process_High*, the composite event constituent events to *users_Limit and cpu_process_High* , etc. The result of the transformation rule is shown in Figure 6.11.

- The last rule to be appended to *load_Control.rls* is the one that constructs the enforcement rule. The policy element *p* is the policy event. The *PolicyEnforce* template used is also a TEC rule that was developed to guide enforcement of policy rules by calling an enforcement routine (the template used is presented in Appendix D). The enforcement routine is the executable to be called to handle the policy rules. The template's variables are the policy event name, an enforcement routine name, enforcement routine type, and other information for the location of the enforcement routine executables and the executable libraries. The transformation rule instantiates these variables, e.g., policy event name to *users_cpu_process_High*, the routine name to *RuleEnforcementForTEC*, routine type to *Java class*, etc. This rule determines if the event represented by the policy has occurred and, if so, initiates a call to the manager routine that handles the conditions-actions part, e.g., on the occurrence of *users_cpu_process_High* event execute the *RuleEnforcementForTEC* Java class. The output of the transformation rule is shown in Figure 6.12.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This TEC rule is to validate the policy 'load_Control' which has the
% event 'users_cpu_process_High' and the interval 'normal_Working_Hours'
% Automatically generated by the PMagic Model
% On  Sun Nov 16 15:51:11 EST 2008
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   First Rule is to validate the policy interval 'normal_Working_Hours'

rule: 'load_Control_normal_Working_Hours':
(
  description: 'Verify the policy interval normal_Working_Hours',

% Following set represents events specified in policy 'load_Control'
  event: _ev_at_interval_check of_class within

                                ['cpu_process_High' ,
                                 'users_Limit',
                                 'users_cpu_process_High' ]
         where [ ]  ,

  reception_action:
  action_load_Control_normal_Working_Hours_check:
    (
      exec_program( _ev_at_interval_check,
      '/sl/wolfbiter/java/jdk.16.0_10/bin/java -cp
      /sl/wolfbiter/PMagic_Manager/classes/ PMagic.classes.IntervalChecking'

         ,'%s %s'
         , ['normal_Working_Hours',
            '/sl/wolfbiter/PMagic_Manager/TEC_Policies/load_Control']
         , 'YES' ) ,
      fopen(_fp
      , '/sl/wolfbiter/PMagic_Manager/TEC_Policies/load_Control/
         normal_Working_Hours_result.txt'
      , r) ,
         readln(_fp, _result),
      fclose(_fp) ,
        ( _result == true ,
           commit_action
           % exit this action and continue the reset of the rule
         ;   % else
           commit_rule
           % exit the whole rule at this point
         )
    )
).
% End of the rule: 'load_Control_normal_Working_Hours'
```

Figure 6.10: Tivoli TEC Rule for Checking the Policy Interval normal_working_hours

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This TEC rule is to generate the event 'users_cpu_process_High' which
% occurs when the event 'users_Limit'
%        and then event 'cpu_process_High' occurred in SEQUENCE.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rule: 'users_Limit_E_SEQ_cpu_process_High':
(
  description: 'Generate event users_cpu_process_High',

  event: _ev1_at_ESEQ of_class 'users_Limit'
         where [date_reception: _left_date_reception,
                server_handle:  _left_server_handle,
                event_handle:   _left_event_handle]  ,

% The E_SEQ rule generates the result as a new event 'users_cpu_process_High'
% by using the exec_program that calls an external program.

  reception_action:
  action_users_Limit_E_SEQ_cpu_process_High:
     (
         first_instance(event: _ev2_at_ESEQ
                               of_class 'cpu_process_High'
            where [date_reception: _right_date_reception
                                   greater_than _left_date_reception,
                   server_handle:  _right_server_handle,
                   event_handle:   _right_event_handle ] ,

              _ev1_at_ESEQ - 0 - 360 ) ,
% The time window for searching the _ev2_at_ESEQ is surrounding by the
% 360 seconds after the _ev1_at_ESEQ time

        exec_program(_ev2_at_ESEQ,
         '/sl/wolfbiter/java/jdk.16.0_10/bin/java -cp
         /sl/wolfbiter/PMagic_Manager/classes/ PMagic.classes.EventGeneration'

         ,'%s %s %s %s %ld %d %d %s %ld %d %d %d'
         , ['users_cpu_process_High', 'createESEQRule' , 'E_SEQ'
         , 'users_Limit', _left_date_reception
         , _left_server_handle  , _left_event_handle
         , 'cpu_process_High', _right_date_reception
         , _right_server_handle , _right_event_handle, 0 ], 'YES') ,

            commit_action % exit the action regarding the scanned events
     )
).
% End of the rule: 'users_Limit_E_SEQ_cpu_process_High'
```

Figure 6.11: Tivoli TEC Rule for Detecting the Composite Event users_cpu_process_High

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This TEC rule is to enforce the specified rule if the event
% 'users_cpu_process_High',   which considers the main event of the policy
% 'load_Control', triggered.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rule: 'load_Control_Rule_Enforcement':
(
  description: 'Fire the rule(s)of the policy load_Control' ,
  event: _ev_rule_main of_class 'users_cpu_process_High'
         where [date_reception:  _ev_date_reception ,
                server_handle:   _ev_server_handle  ,
                event_handle:    _ev_event_handle   ,
                hostname:        _ev_hostname       ,
                sub_source:      _ev_sub_source     ,
                sub_source_port: _ev_sub_source_port ] ,
  reception_action:
  action_load_Control_users_cpu_process_High_enforce_rule:
     (
       exec_program(_ev_rule_main ,
        '/sl/wolfbiter/java/jdk.16.0_10/bin/java -cp
         /sl/wolfbiter/PMagic_Manager/classes/
         PMagic.classes.RuleEnforcementForTEC'

           ,'%s %s %s %s %d %ld %d %d'
           , ['load_Control','users_cpu_process_High' , _ev_hostname
           , _ev_sub_source , _ev_sub_source_port , _ev_date_reception
           , _ev_server_handle  , _ev_event_handle ], 'YES') ,

           commit_rule
       )
).
% End of the rule: 'load_Control_Rule_Enforcement'
```

Figure 6.12: Tivoli TEC Rule for Enforcing the Policy load_Control

6.3.3 Mapping a Policy to Management Agents

Mapping a specified policy to be deployed and enforced by management agents requires the configuration of the components of the management agents using the corresponding specified policy elements as described in Section 5.7.3. The use of template-based mapping approach applies as follows. There are templates for policy elements such as primitive events, composite events and rules. The templates needed here are of a different form than seen earlier. Here template refers to a method that uses the agent interface methods to add the appropriate tuples to the different sets. The template variables are two associative arrays (see Section 5.7.3). The transformation rule calls the Finding Agents algorithm described in Section 5.7.1 to determine the associative arrays to be assigned to the template variables.

6.4 Discussion

This section discusses issues related to policy mapping. This chapter assumes that policies are mapped to an event-driven rule-based system or to management agents. We focused on these two mappings since these are most commonly used. Figure 6.13 summarizes the processes needed to map a specified policy to an event-driven rule-based system. The only manual operations (see Figure 6.13) are those of the creation of the templates, which once defined by the administrator, can be used repeatedly by the mapping processes. A set of templates, as described in this Chapter and as seen Figure 6.13, is needed for each management system that used to deploy and enforce policies through it.

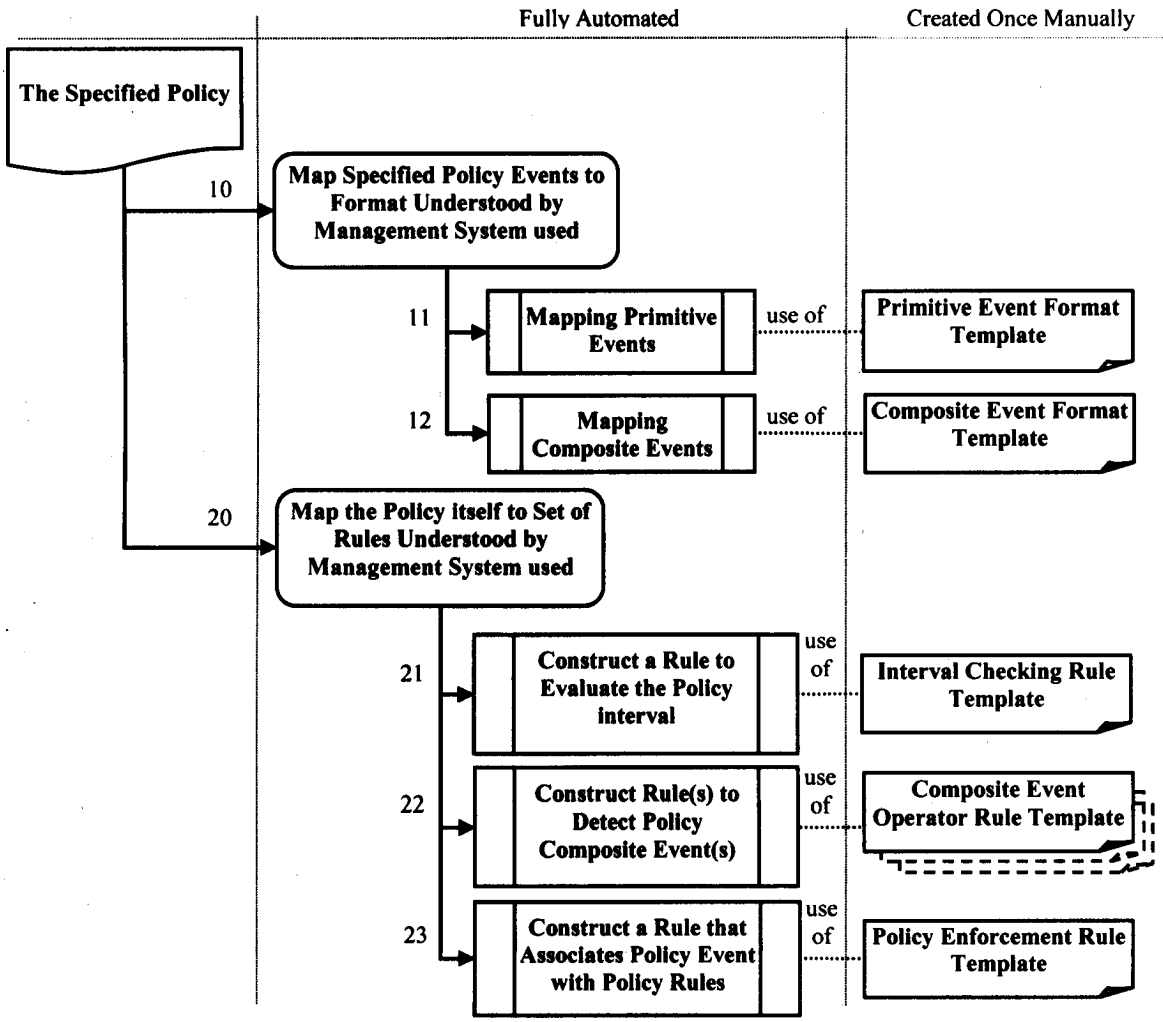


Figure 6.13: The Processes of Mapping a Policy to an Event-Driven Rule-Based Systems

The numbers shown in Figure 6.13 illustrate the order of the mapping processes. The mapping order shown should be followed for the following reasons:

1- Some processes are dependent on others. For example, composite events should be mapped after primitive events. The reason is that identifiers are assigned to primitive events and these are used in the definition of the composite event.
2- The first rule to be constructed is the rule that checks if the received event is within the policy interval. This means that if it is not, then there is no reason to evaluate other related rules.
3- The set of rules that evaluate the detection of the policy composite events should be generated second. Within this set of rules the order mentioned in Section 6.3.2 for composite event detection rules should be followed.
4- The rule that associates the policy event with the policy rules should be the last rule to be generated.

6.5 Chapter Summary

The research work presented in this Thesis has been motivated by the need to bridge the gap between specifying management policies and mapping these polices to manage distributed systems environments so that the policies can be realized. The Chapter has shown how to build the policy model and services on existing management services found in commercial management systems. Practically, we have shown how to map the high-level specified policies elements to events format files (as in BAROC) and to executable rules (as in TEC rules) by using the developed reusable templates that steer the automated mapping mechanism. The methodology shown represents a general approach that can be adapted not only by Tivoli, but also could be implemented for other management systems such as HP-OpenView, CA-Unicenter, etc. This Chapter also addressed the mapping of a policy to management agents.

Chapter 7

IMPLEMENTATION AND THE PROTOTYPE

The Policy Management Agent Integrated Consol (PMagic) prototype is the implementation of the policy-based model presented in this Thesis. The PMagic software is implemented using Java JDK version 1.6. The repository is implemented using IBM DB2 version 8.2. The management system used is the IBM Tivoli Management Framework version 4.1 (TMF) and Tivoli Enterprise Console (TEC) version 3.9. The prototype's software (TMF, TEC, gateway, Java and DB2 server) are installed on a Sun Blade 1000 Workstation with 1.5GB of memory that uses Solaris 5.8. There is 1 GB of memory dedicated to the Java JVM.

Figure 7.1 shows the design and implementation structure of PMagic. The main PMagic implemented components are the following: PMagic GUI User Interface, Agent Matcher, Mapping Mechanisms and PMagic Management Agents. The following Sections address the implementation of each of these components.

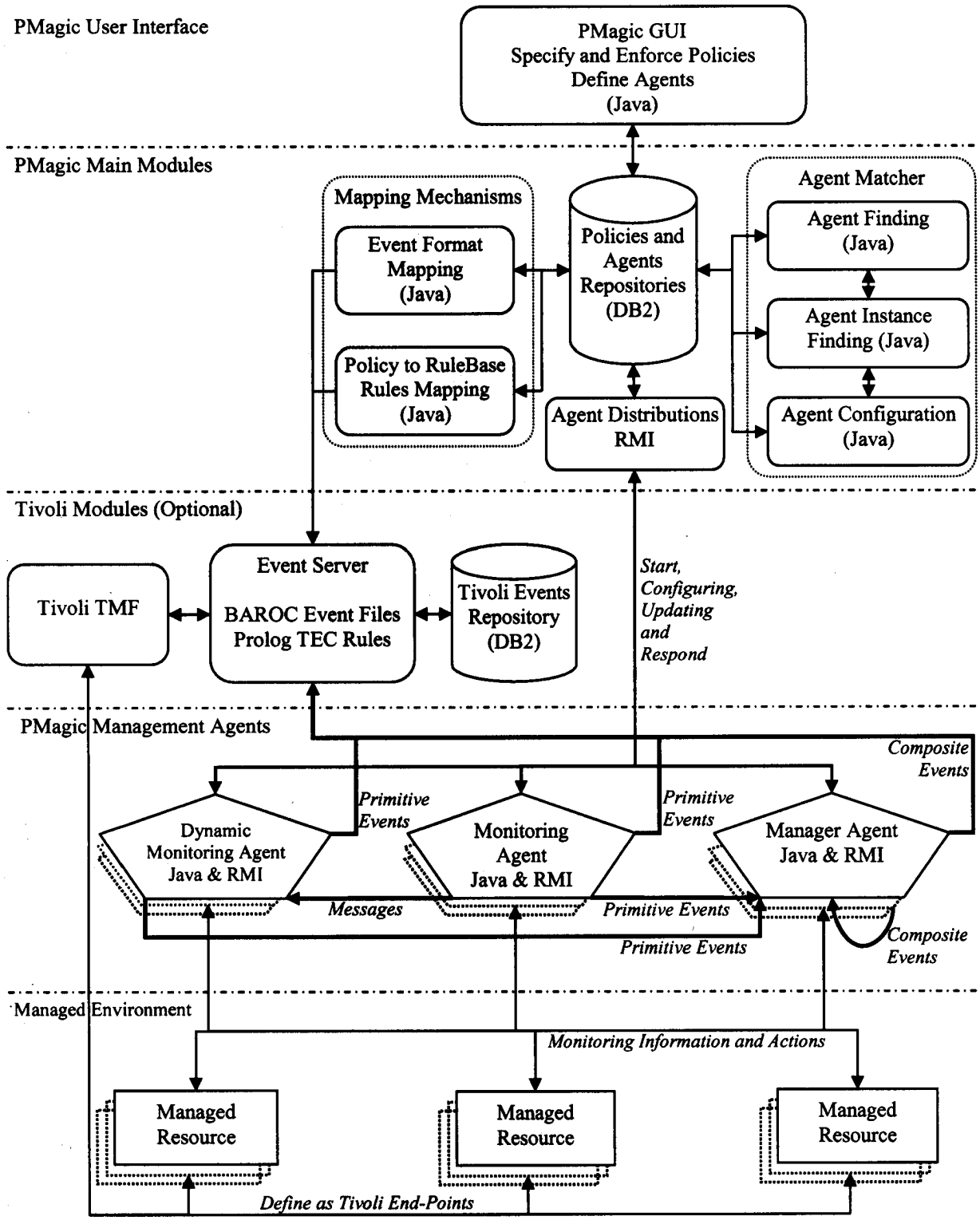


Figure 7.1: Policy-Management Agent Integrated Console Implementation Structure

7.1 PMagic Policy Specification and Agent Definitions

PMagic provides a GUI that allows a user to specify management policies. The start screen for PMagic is seen in Figure 7.2. The PMagic main menu organization is shown in Figure 7.3.

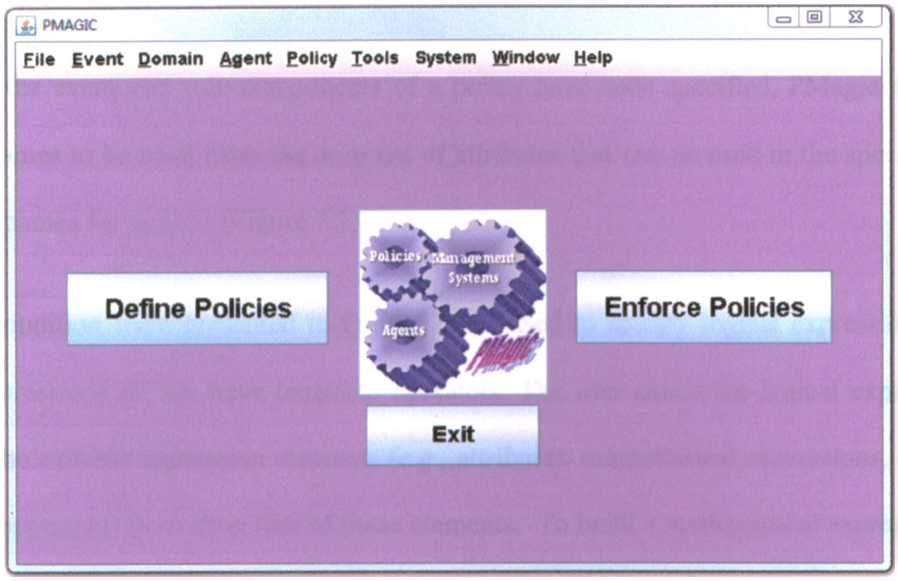


Figure 7.2: Policy-Management Agent Integrated Console-PMagic Main Form

The policy information model is described in Chapter 3. The information model allows for the specification of different types of policies. Figure 7.4 presents the screen that provides a *Policy Definition* form that consists of several tabs for specifying policies. A policy is specified by assembling its components from predefined definitions for events, rules, conditions, actions, intervals, and domains. For instance, Figure 7.5 shows the selection of the policy rules. The definitions for rules, events, conditions, actions, intervals and domains are reusable components. The tree shown in Figure 7.6, which is the Policy Tree tab of the *Policy Definition* form, represents the components (attributes, domains, events, rules, conditions and actions) of a policy. Users are able to click on a node to get more information. A policy needs to be fully specified before it can be enforced. The policy grammar is given in Appendix A. PMagic detects some conflicts in the policy specification. Specifically, a check is made to ensure that the rule interval and/or rule domain specification is within the specified policy interval and/or policy domain.

The *Rule Definition* form shown in Figure 7.8 is used to enter rules. The *Rule Definition* form consists of 4 tabs: Rule, Rule Conditions, Rule Actions and Rule Tree. The *Rule Definition* form can be called from the main menu using the Policy drop menu of Figure 7.3 or from within the *Policy Definition* form (Figure 7.5) by clicking the Rules button.

Once the event and rule components of a policy have been specified, PMagic extracts the attribute names to be used from the drop list of attributes that can be used in the specification of parameter names for actions (Figure 7.7).

The condition form presented in Figure 7.9 is used to specify logical expressions. These logical expressions do not have temporal operators. The user enters the logical expressions by selecting the suitable expression elements (e.g., attributes, mathematical expressions, logical and relational operators) from drop lists of these elements. To build a mathematical expression, users click the Mathematical Expression button shown in Figure 7.9. The mathematical expression form is shown in Figure 7.10. When the user clicks Exit in Figure 7.10, the constructed mathematical expression returns to the Value field of Figure 7.9. The forms ensure the user enters a syntactically correct condition.

The event definition form shown in Figure 7.11 is used to specify primitive and composite events. The bottom left hand side of Figure 7.11 shows the attributes extracted from an event specification. An event expression can also be entered by the user by using temporal operators on events already entered. There are drop lists for specified events and for the temporal operators.

Figure 7.12 shows the *Interval Definition* form that is used to specify intervals. As can be seen, intervals can be defined as a period between two dates. An interval can also be any selected date or time between two dates e.g., every Sunday in the next five years. For this reason, the interval entry form allows for the specification of a selection of dates or a period within an interval between two dates.

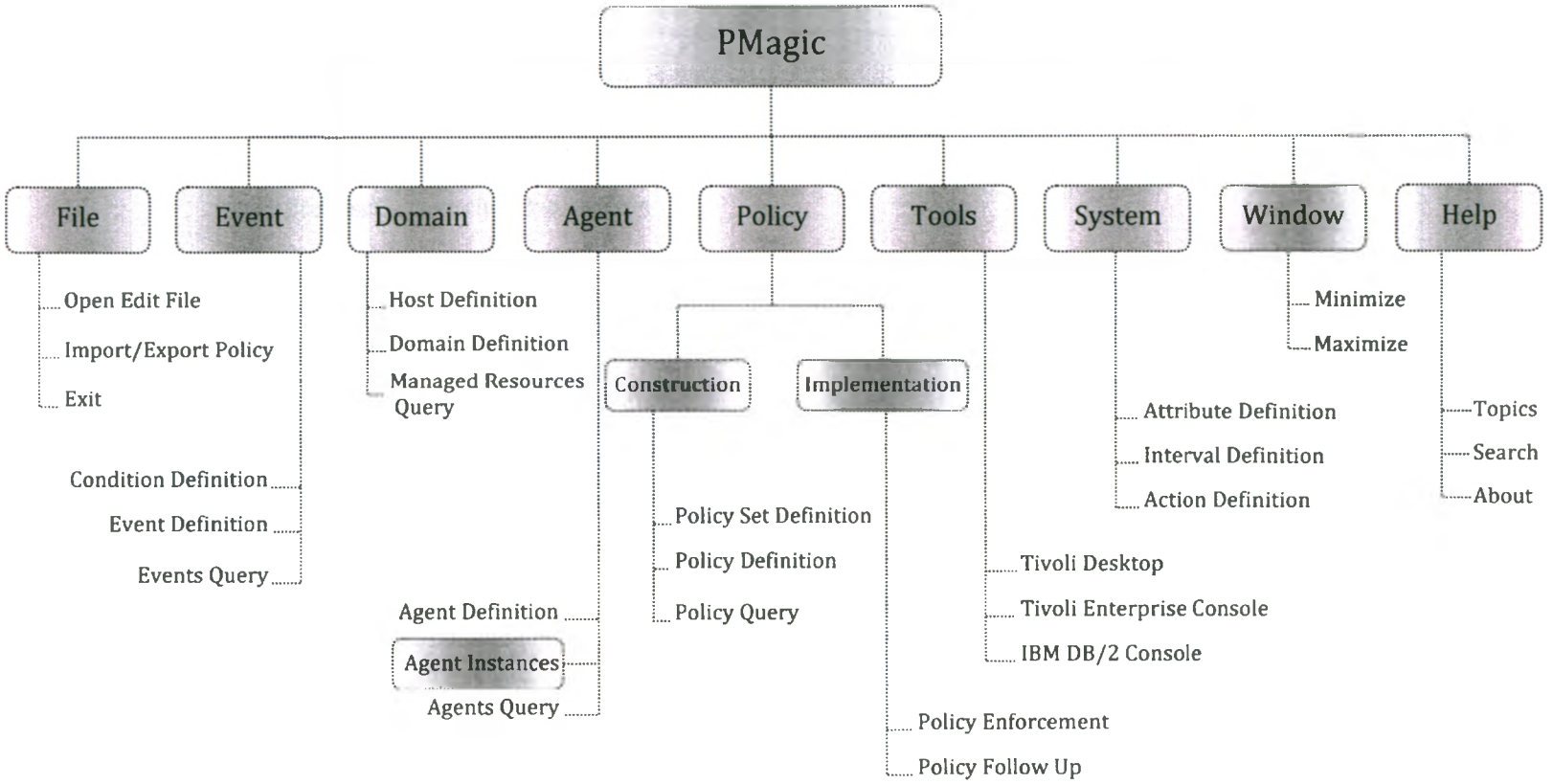


Figure 7.3: PMagic Menu Structure

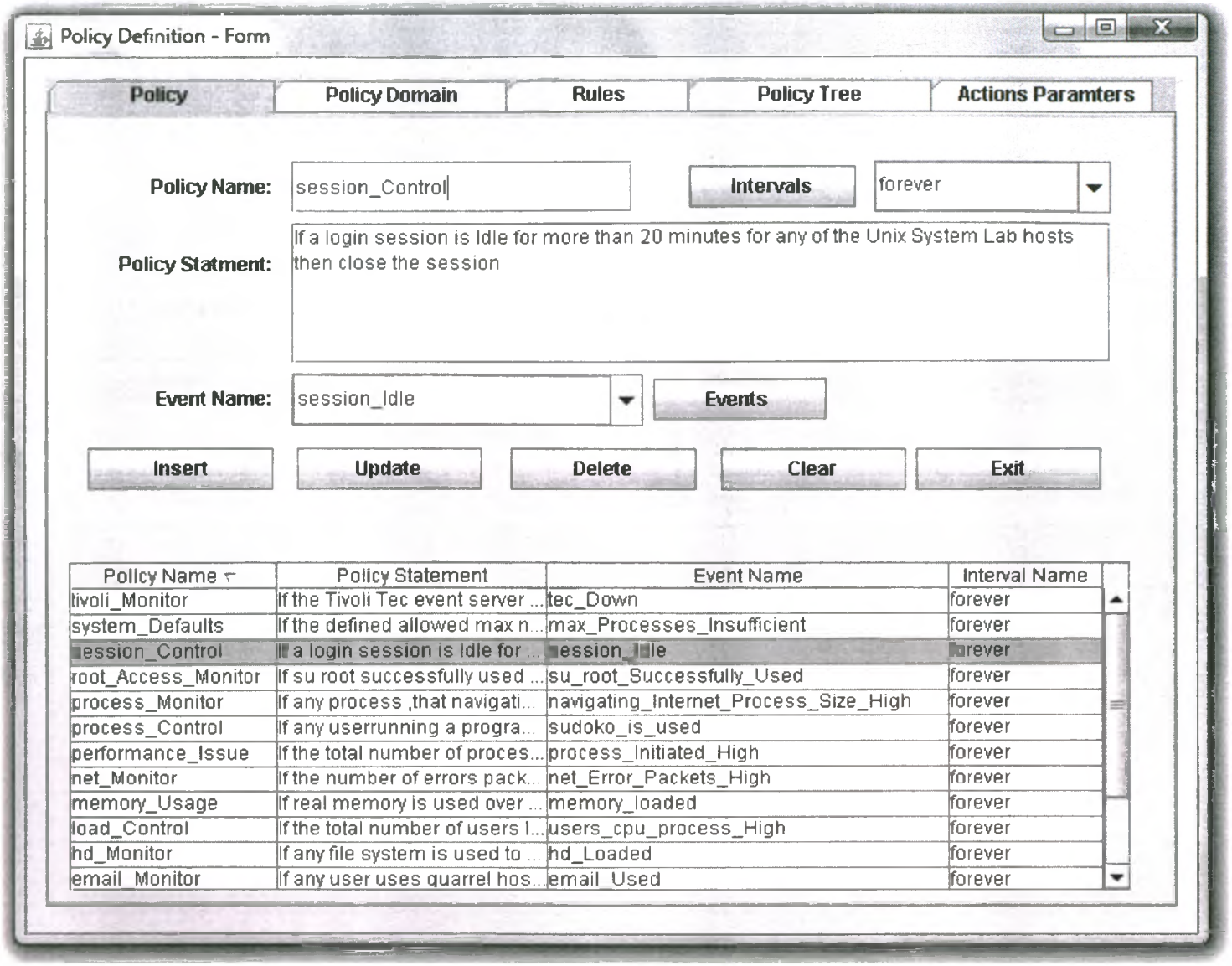


Figure 7.4: A Policy Definition Form

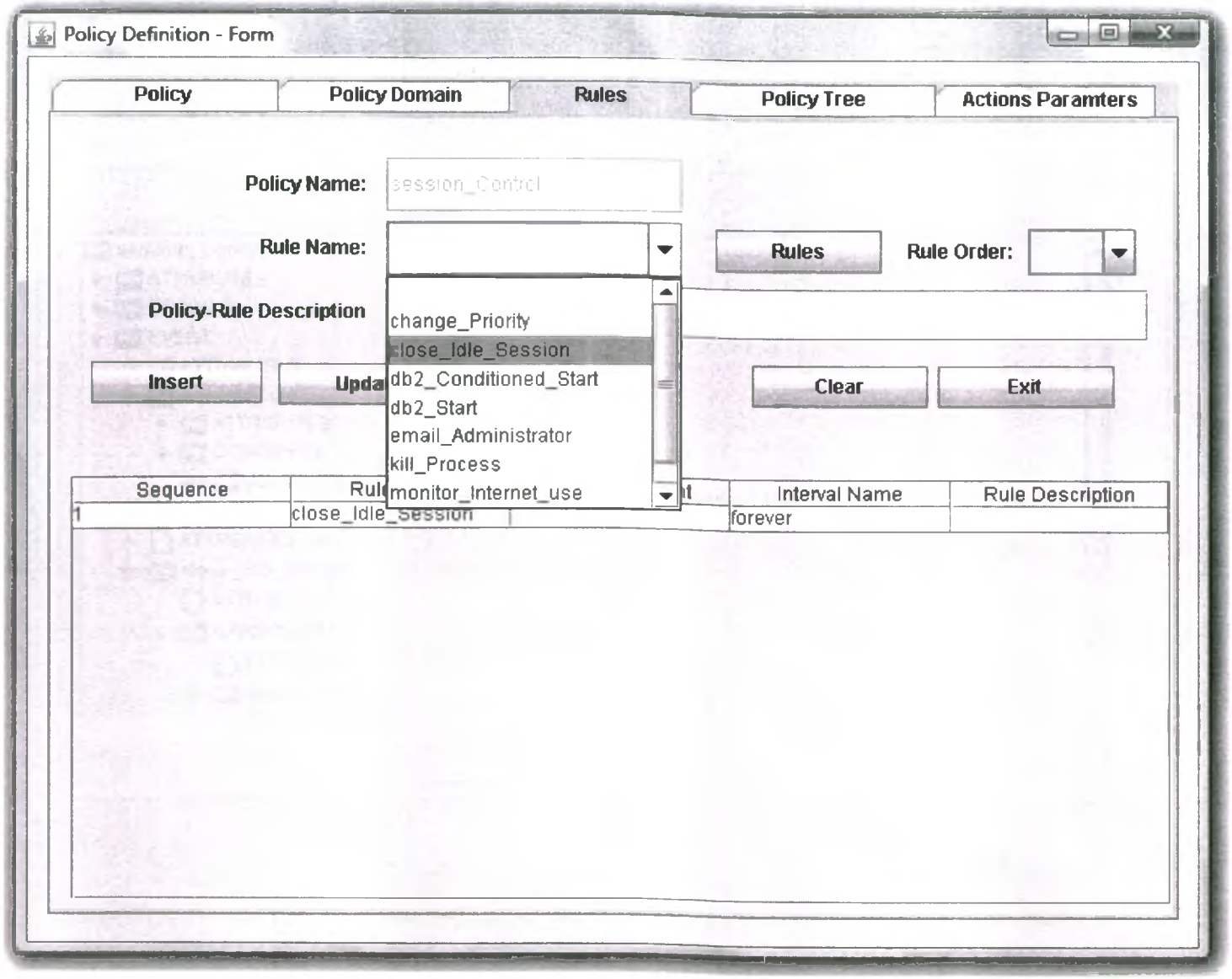


Figure 7.5: Reusable Building Blocks for Policies

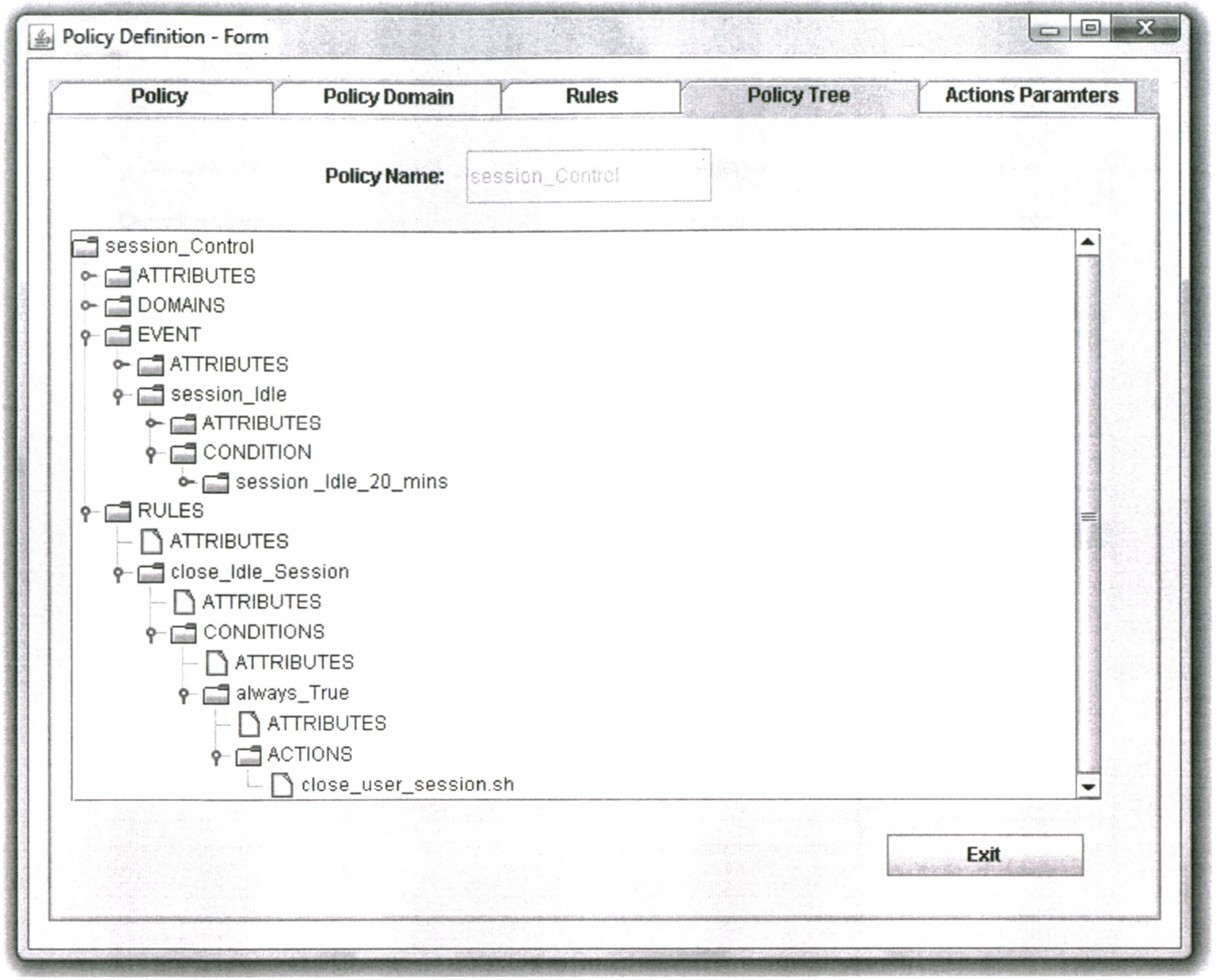


Figure 7.6: A Policy Tree

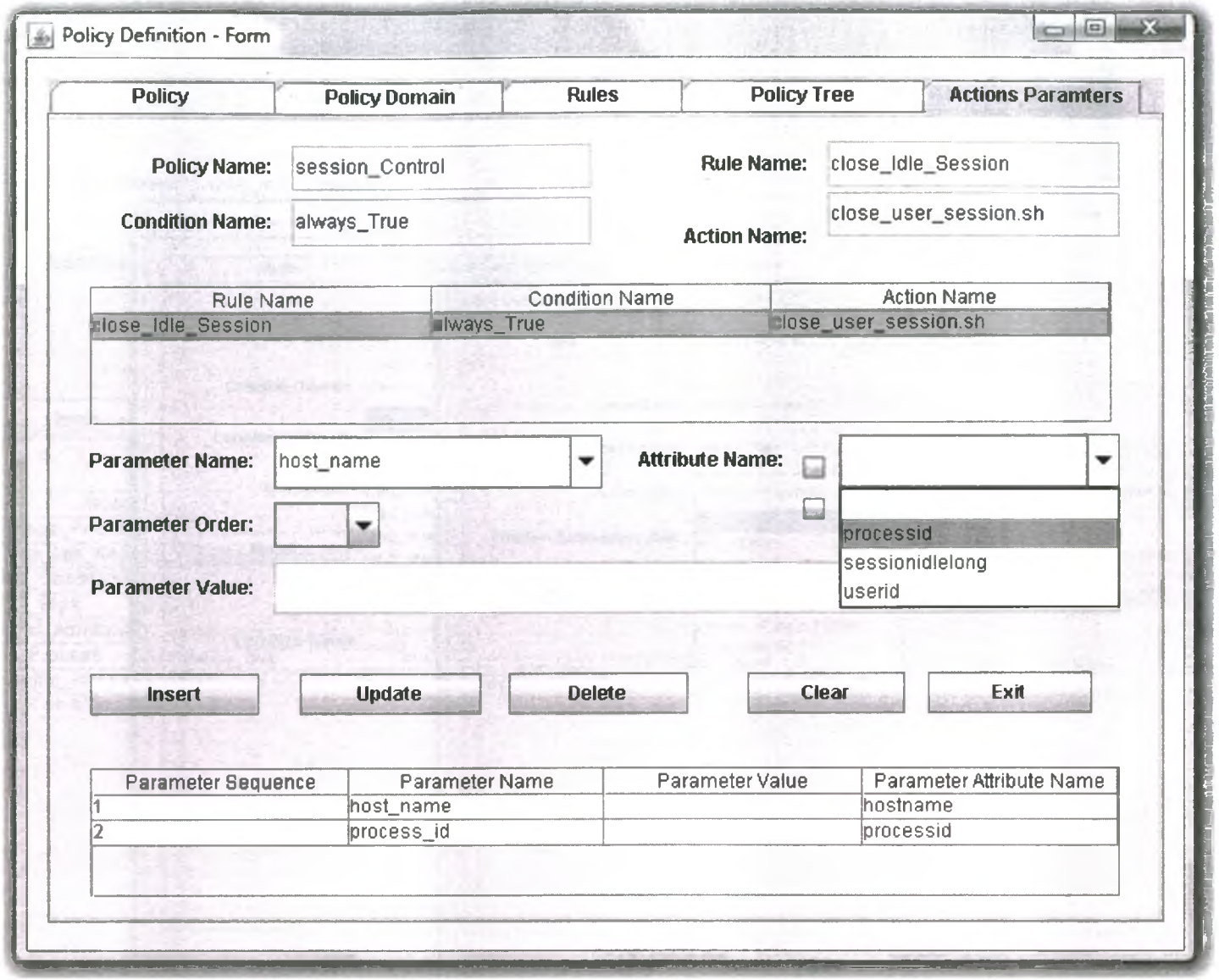


Figure 7.7: Mapping between Actions Parameters and Policy Extracted Attributes

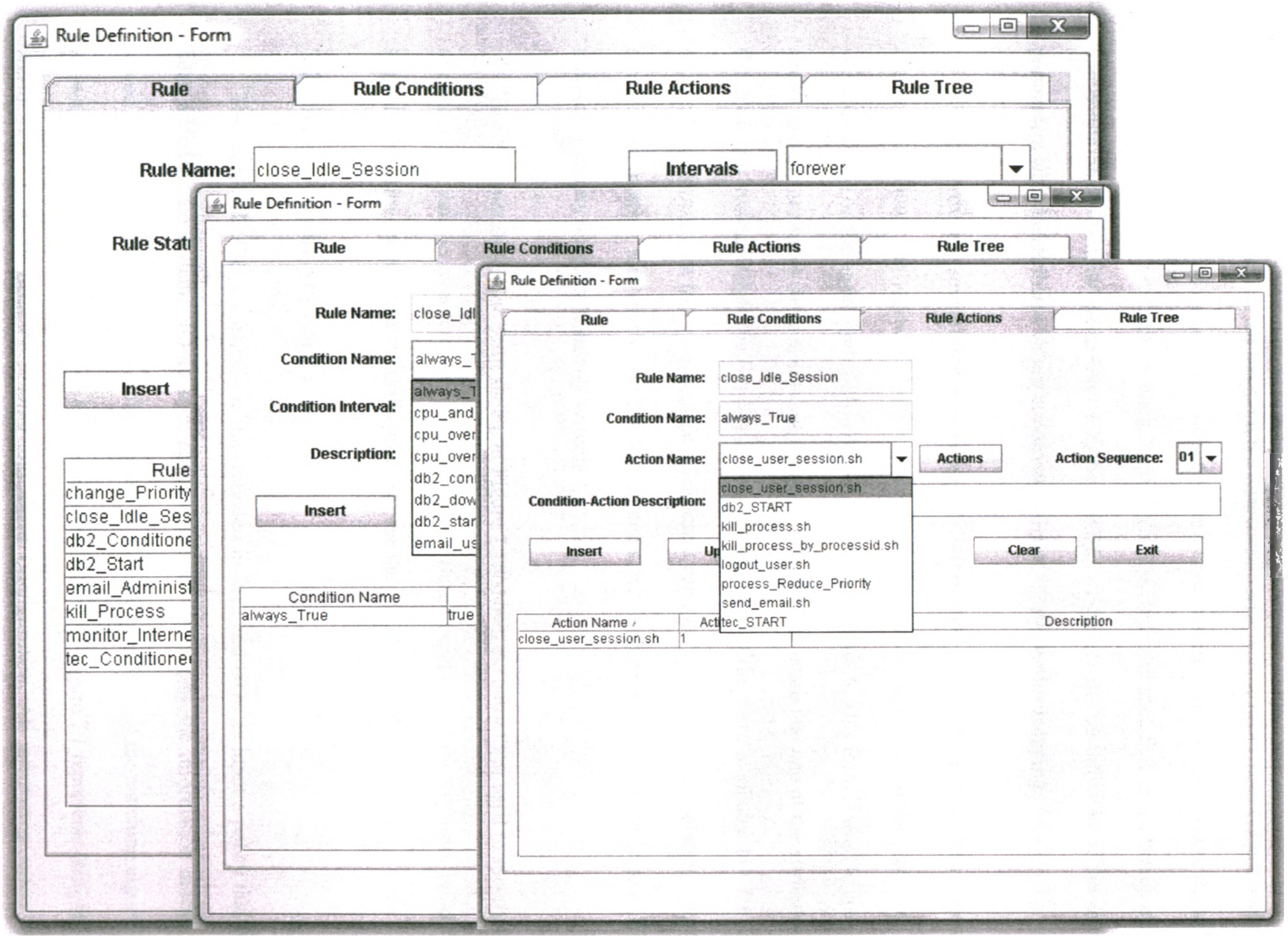


Figure 7.8: A Rule Definition Form

The *Action Definition* form shown in Figure 7.13 is used to specify information about actions. Actions are classified based on the executable code e.g., a Java Class, C Executable, a UNIX Shell Script. The names and order of the action parameters must be specified in the same way action arguments are built. A mapping between the attributes specified in the policy event and conditions to actions' parameters may be needed to facilitate the instantiation of the action parameters with the attributes values at runtime. Figure 7.7 shows this mapping.

PMagic provides several forms to maintain information about the management agents. Figure 7.14 shows the *Agent Definition* form that allows information about an agent to be specified based on the agent information model defined in Chapter 5. This form is used to specify agents and the attributes these agents can monitor. It should be noted that one of the attributes is the operating system which implies that if two agents monitor the same attributes but for two different platforms, then those are considered different agents. For example, the *memory_agent* is specified to monitor memory usage in UNIX systems, while another agent *memory_agent_WIN* monitors memory usage in Windows systems.

The *Attribute Definition* form shown in Figure 7.15 shows the entry form used to specify the attributes that can be monitored by PMagic agents. Based on these attributes, users build the logical expressions that characterize the primitive events.

7.2 PMagic Agent Matcher

The Agent Matcher component is implemented in Java and takes as input a policy that has been added to the system. Information about agents, as defined by the information model, is stored in DB/2. Currently, the agent finding task is an implementation of the matching algorithm shown in Figure 5.6. The *Agent Matcher* component depicted in Figure 7.1 represents three tasks: *Agent Finding*, *Agent Instance Finding* and *Agent Configuration*. All monitoring agents and manager agents are implemented using Java. Note that the implementation of the information

services that provide monitoring mechanisms (see Section 5.3 for more details) do not have to be implemented using Java.

7.3 PMagic Mapping Mechanisms

Policy deployment is the mapping of management policy elements (rules, events, conditions, actions) to the services needed to support execution and enforcement of the policy elements. The approaches and designs to carry out these mapping mechanisms were described in Chapter 6. The *Mapping Mechanisms* component depicted in Figure 7.1 represents the desired mapping. Chapter 6 presented two algorithms for mapping. The implementation of the algorithms is independent of the management system. The functions that implement the transformation rules are specific to the management system. However, an abstract class can be defined. The abstract class has the methods for the transformation rules. A subclass represents a specific implementation of these methods that is specific for the management system. The implementation was done using Java.

7.4 Distribution Mechanisms Used

To deploy a policy, the executables of the agents that support this policy need to be transferred as required e.g., a management agent that monitors login sessions needs to be located on the machine that it is to monitor. The executable may be on another machine. In most cases, distribution refers to starting agent executables on the machine to be monitored. In case of collecting attributes from different machines, the executable of dynamic and/or manager agents will be placed on one of the machines that is to be monitored.

To show the flexibility of this work two approaches were used for the start-up of the agent executables. Management systems often have an application for software distribution. Such an application distributes, configures or reconfigures and updates software applications, system patches and management agents. We used Tivoli Software Distribution version 4.2. To use the

software distribution from Tivoli to distribute agents, the agent executables must be added to the distribution profile. In a Tivoli environment, a profile is a container for application-specific information about a particular type of resource, e.g., agent executables. Every managed host, where the profile needs to be distributed, must be defined in TMF as an end-point. In a Tivoli environment, an end-point is the computer system that is the ultimate target for most Tivoli operations. Assigning the end points to the Tivoli distribution profile will depend on the domain associated with the policy. These tasks are carried out by PMagic. PMagic uses the Tivoli profile feature for the distribution of and starting of agents. This feature requires the specification of information needed (e.g., hostname and directory path of an agent executable and associated files). We can use the Tivoli profile feature to update agents when a change in policy occurs by sending a new agent executable configured to support the changes.

Another approach for distribution of executables is the UNIX remote shell *rsh* and *cp* copy commands for UNIX machines. To enable copies and/or executes of executables from UNIX to Windows machines, the remote shell software Winsock RSHD/NT [157] was installed in the Windows host machines. In this case, PMagic starts the management agents on the remote managed hosts using remote shell commands from within the PMagic Java classes. The two approaches were implemented to show the independence of the mapping process from any specific distribution mechanism.

Java's Remote Method Invocation (RMI) was chosen for implementing the communication between the agents and the PMagic manager processes, and for communication among the monitoring, dynamic monitoring and manager agents. RMI was chosen for its built-in simplicity and the fast prototype development that it enables. PMagic relies on RMI to communicate with managers.

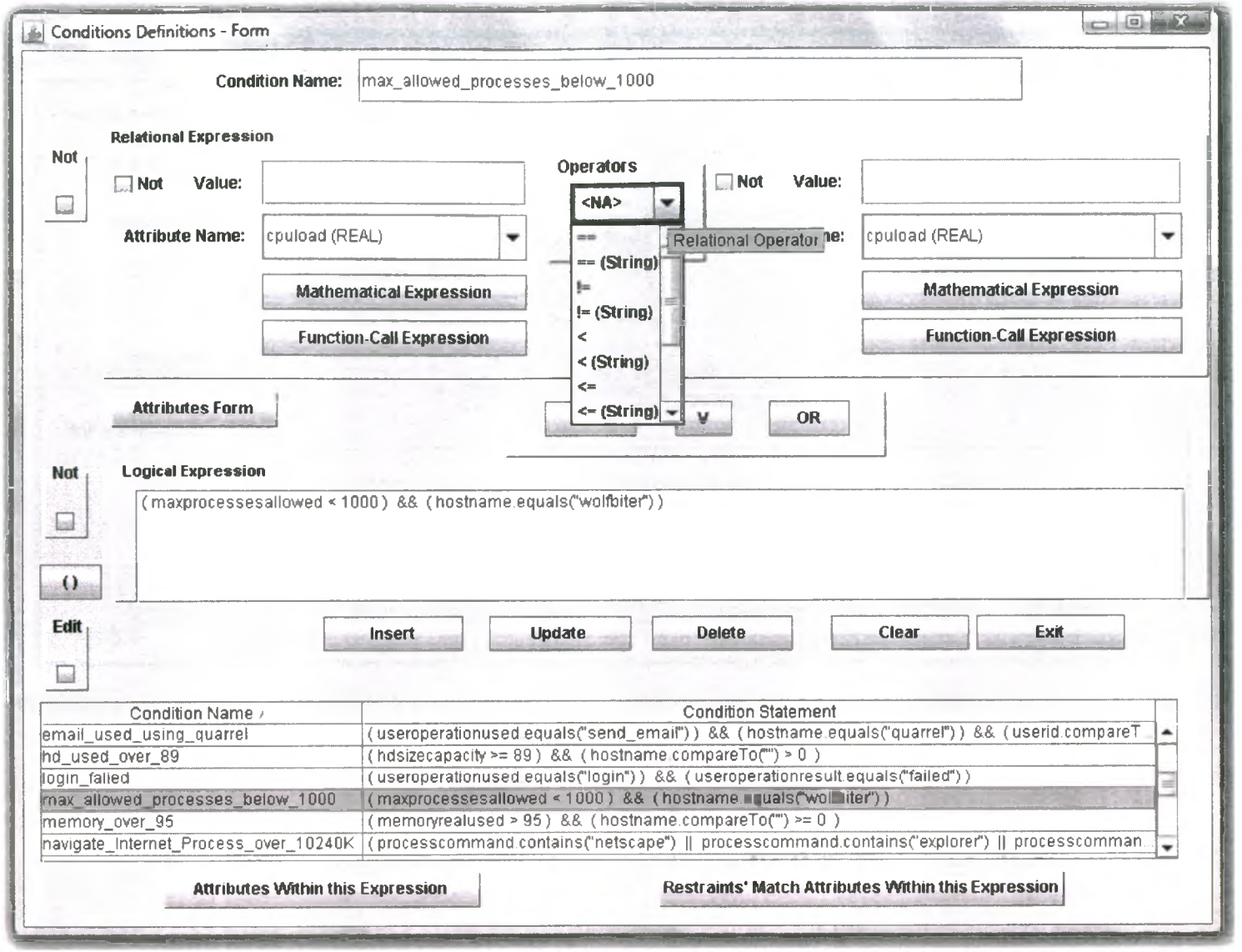


Figure 7.9: The Condition Definition Form

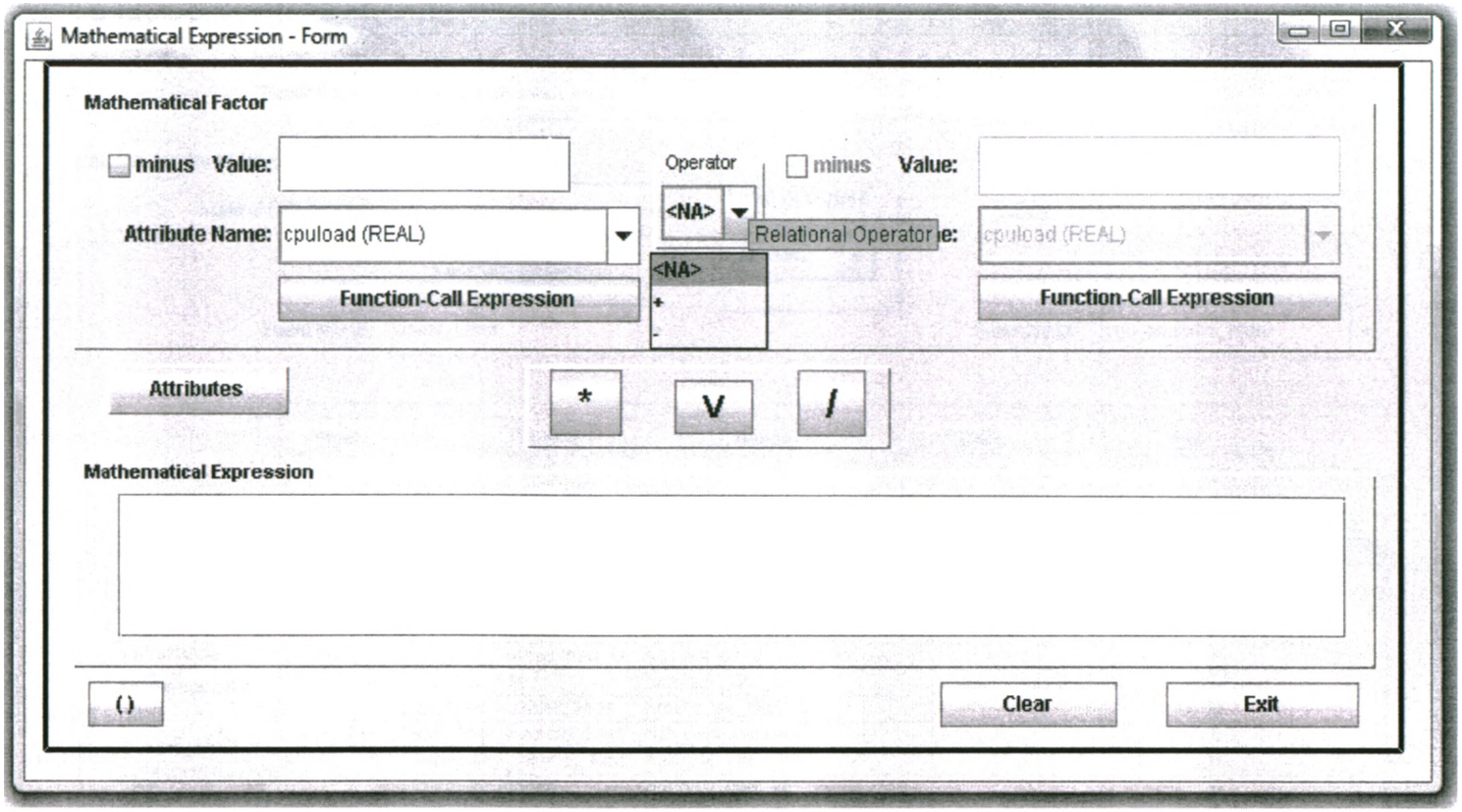


Figure 7.10: The Mathematical Expression Form

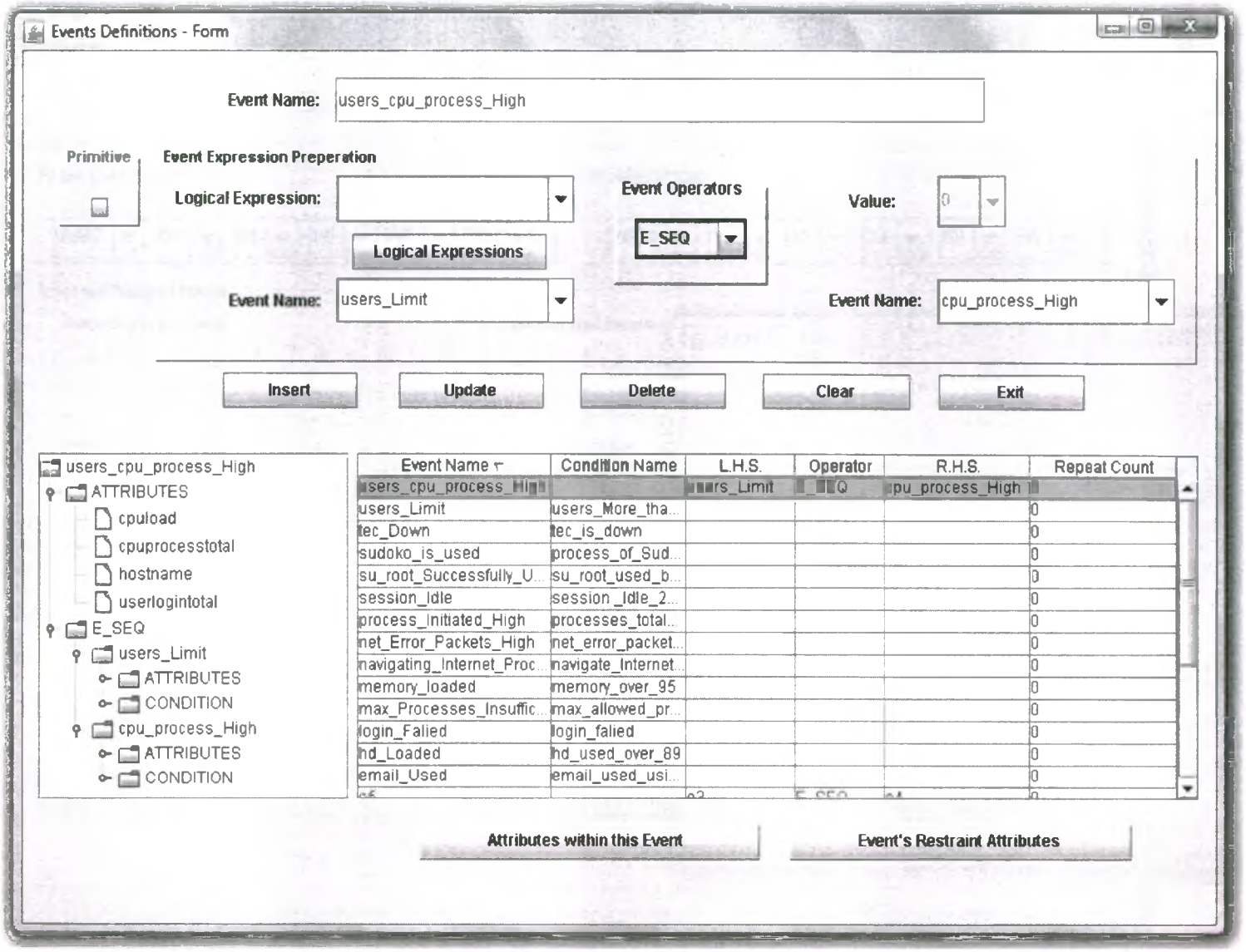


Figure 7.11: The Event Definition Form

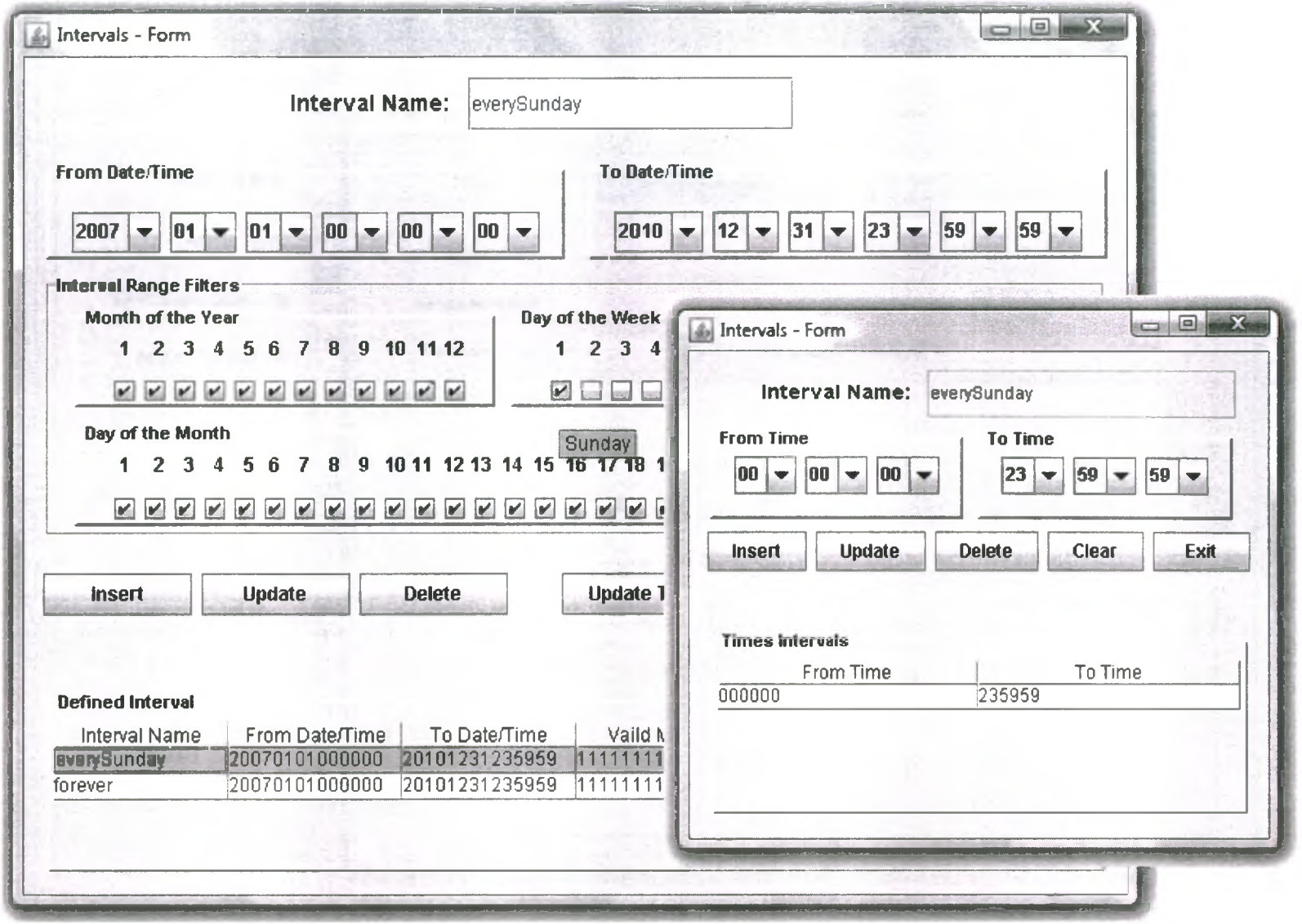


Figure 7.12: The Interval Definition Form

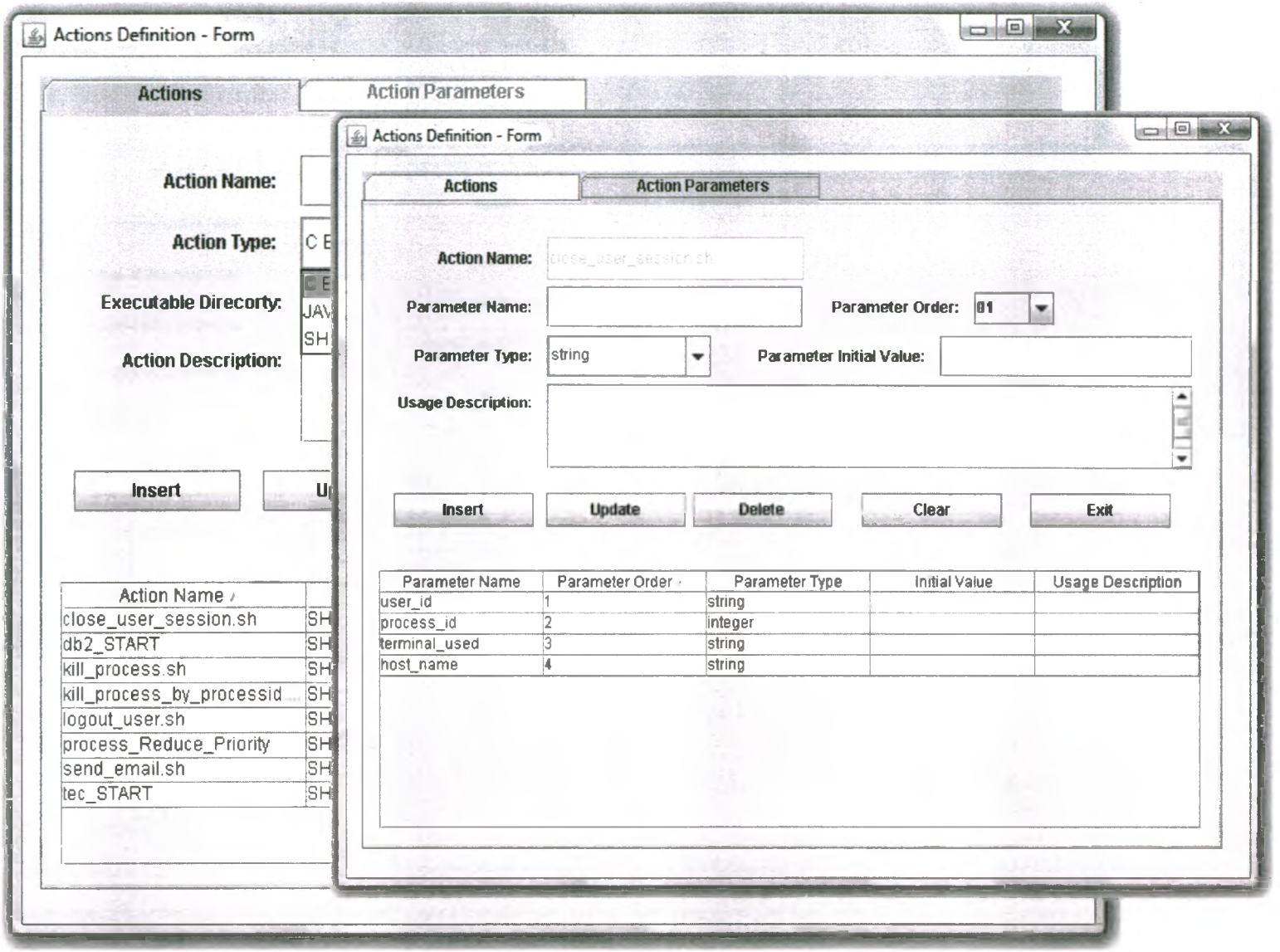


Figure 7.13: The Action Definition Form

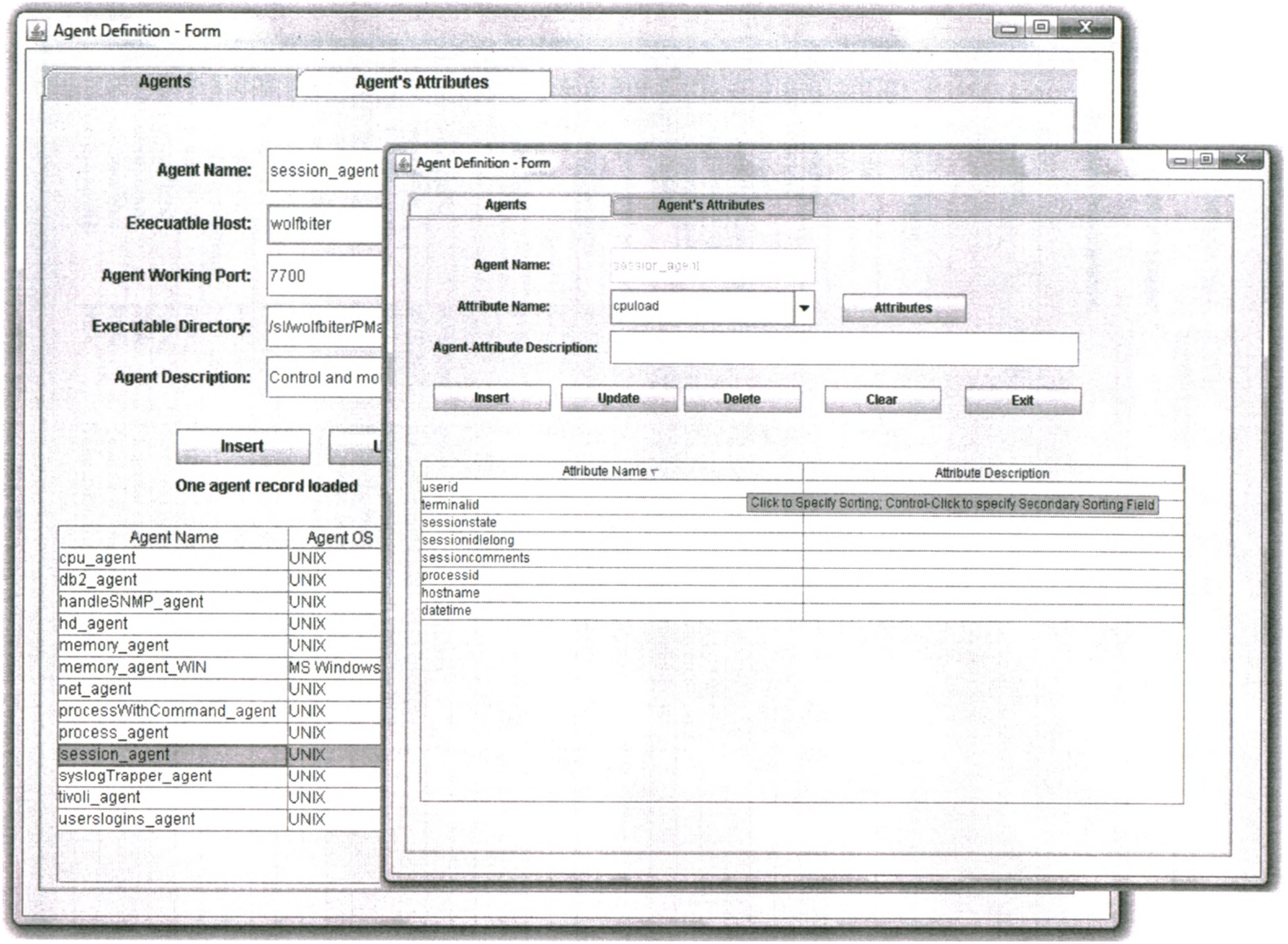
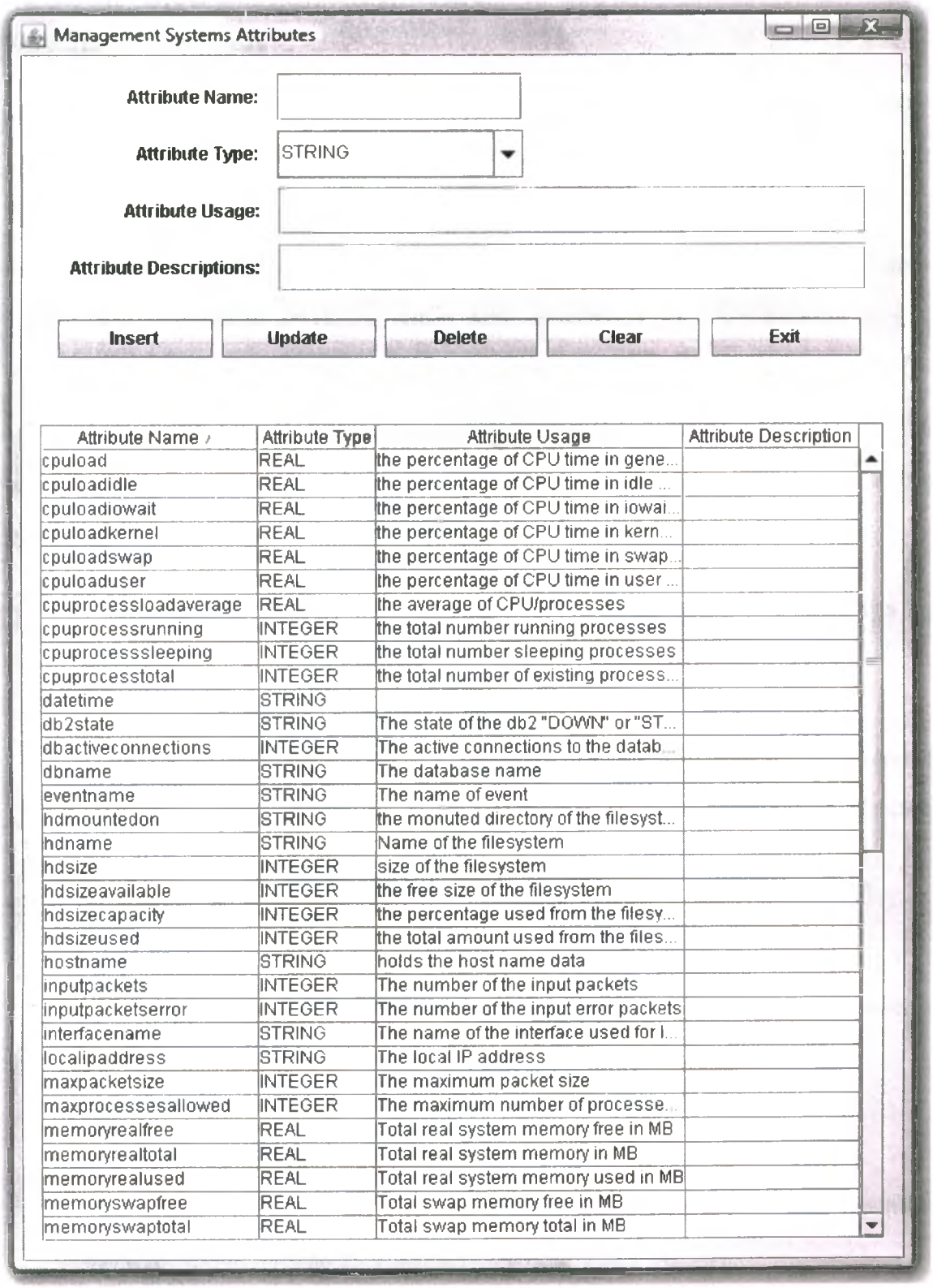


Figure 7.14: The Agent Definition Form



Attribute Name	Attribute Type	Attribute Usage	Attribute Description
cpuload	REAL	the percentage of CPU time in gene...	
cpuloadidle	REAL	the percentage of CPU time in idle ...	
cpuloadiowait	REAL	the percentage of CPU time in iowai...	
cpuloadkernel	REAL	the percentage of CPU time in kern...	
cpuloadswap	REAL	the percentage of CPU time in swap...	
cpuloaduser	REAL	the percentage of CPU time in user ...	
cpuprocessloadaverage	REAL	the average of CPU/processes	
cpuprocessrunning	INTEGER	the total number running processes	
cpuprocesssleeping	INTEGER	the total number sleeping processes	
cpuprocesstotal	INTEGER	the total number of existing process...	
datetime	STRING		
db2state	STRING	The state of the db2 "DOWN" or "ST...	
dbactiveconnections	INTEGER	The active connections to the datab...	
dbname	STRING	The database name	
eventname	STRING	The name of event	
hdmountedon	STRING	the monuted directory of the filesyst...	
hdname	STRING	Name of the filesystem	
hdsize	INTEGER	size of the filesystem	
hdsizeavailable	INTEGER	the free size of the filesystem	
hdsizecapacity	INTEGER	the percentage used from the filesy...	
hdsizeused	INTEGER	the total amount used from the files...	
hostname	STRING	holds the host name data	
inputpackets	INTEGER	The number of the input packets	
inputpacketserror	INTEGER	The number of the input error packets	
interfacename	STRING	The name of the interface used for I...	
localipaddress	STRING	The local IP address	
maxpacketsize	INTEGER	The maximum packet size	
maxprocessesallowed	INTEGER	The maximum number of processe...	
memoryrealfree	REAL	Total real system memory free in MB	
memoryrealtotal	REAL	Total real system memory in MB	
memoryrealused	REAL	Total real system memory used in MB	
memoryswapfree	REAL	Total swap memory free in MB	
memoryswaptotal	REAL	Total swap memory total in MB	

Figure 7.15: The Management System Attributes Definition Form

7.5 PMagic Managers

The work in Chapter 6 assumes that there is an event-driven rule-based engine. PMagic uses the Tivoli Enterprise Console (TEC) component from Tivoli. PMagic software creates BAROC files and TEC rules. This is implemented in Java, as is the implementation of the manager agents. Most methods implemented for finding agents, finding agent instances, agent configuration, distribution of agents, policy rules enforcement, etc., are designed to send information to one central log file that resides in the PMagic manager. This log file is useful for analyzing the results of the experiments described in Chapter 8. Information columns in a log line are separated by one '#' character. The information columns in the order they appear on a log file line, are as follows:

- *Timestamp:* The number of milliseconds since the epoch to the time when this log recorded.
- *Full-Date:* The date/time when this log recorded.
- *Operation-Number:* Each operation used to enforce a policy or a set of policies is assigned a number by PMagic, where numbers are incrementally increased. Numbers cannot be reused unless PMagic resets the database.
- *Host-Name:* The name of the host from which this log is sent.
- *Agent-Name:* The name of the agent instance that sends this log or GNA in the case of no agent instance, e.g., the method is *AgentFinding* that runs by the manager at *wolfbiter* host.
- *Agent-Port:* The port number of the agent instance, or TNA otherwise.
- *Method-Name:* The name of the method that sends this log.
- *Method-State:* There are three states; IN, OUT and EXP:*exception-name::exception_message.*
- *Policy-Name:* The name of the policy.
- *Event-Name:* The name of the event that the method handles or ENA otherwise.

- *Rule-Name:* The name of the rule that the method handles or RNA otherwise.
- *Condition-Name:* The name of the condition that the method handles or CNA otherwise.
- *Action-Name:* The name of the action that handles the method or ANA otherwise.
- *TEC-Used:* A Boolean value (true or false) that indicates whether this operation uses TEC or not.

7.6 PMagic Event Common Attributes

Table 7.1 represents the common attributes that each event notification message includes. Additional attributes are specified in an event according to the state that the event represents.

Attribute Name	Descriptions
eventname	Specifies the name of the event
timestamp	The number of milliseconds since the epoch to the time when this event occurred.
source	PMagic Prototype
sub_source	The name of the management agent that triggered the event
sub_source_port	The port number that the management agent which triggered the event used for communication
hostname	The name of host on which the event occured
origin	The name of the host where the management agent resides. Hostname and origin may be the same
severity	Specifies the severity of the event, e.g. UNKOWN, HARMLESS, WARNING, MINOR, CRITICAL, FATAL
status	Specifies the status of the event, e.g. CLOSED, OPEN.
operation_number	Each policy deployment has a unique operation_number; this number is used for the audit trail in the PMagic log file
msg	A description message of the event

Table 7.1: PMagic Event Common Attributes

7.7 Chapter Summary

This Chapter presented the Policy-Management Agent Integrated Console (PMagic) software.

Chapter 8

EVALUATION

This Chapter describes the experiments conducted to evaluate PMagic and presents several conclusions drawn from these experiments.

8.1 Experiments Environment

There are 18 policies used in the evaluation. These policies are shown in Appendix B. The policies were chosen to represent a variety of system, application and network management tasks. Several policies measure resource usage and take action if the usage exceeds a threshold. Several policies are used to configure the system. Two policies do event correlation and take an action in support of a fault management task.

The experiments required the use of twelve monitoring agents. The descriptions of these agents are in Appendix C. These agents cover almost 74 attributes and this number can

incrementally increase if we consider the attributes of SNMP agents. Management agents use the *postemsg* command from the Tivoli Management Framework (TMF). The *postemsg* command sends an event to Tivoli. This command does not require defining each managed host as an end-point, while the *wpostemsg* command does. There are eleven managed host machines. Ten of these machines use a UNIX platform and one uses Windows XP. In the experiments conducted in this Chapter, the remote shell command *rsh* is used to start agents at remote hosts (as described in Section 7.4). The experimental times shown in this Chapter were the best times; see Appendix F for more details about averages times and the standard deviations.

8.2 Basic Experiments Using an Existing Management System

The basic experiments make use of Tivoli Enterprise Console (TEC) as the event server and rule-engine. In deploying policies, the PMagic Management Agents will be configured to send all events to TEC. The basic experiments focus on the deployment time of policies. Deploying a policy requires the following tasks to be carried out:

1. Find the agents that can support the policy to be deployed.
2. Find any agent instances of agents that were found in Step 1.
3. Configure agent instances to support the added policy. Configuring an agent instance involves updating the Registry repository (see Chapter 5, the Agent Information Model).
4. Create the BAROC files and TEC rules as shown in Chapter 6.
5. Import, compile and load the BAROC files and TEC rules created in Step 5 to Tivoli TEC engine.
6. Start and/or update agent instances at remote hosts.

We are specifically interested in these times:

- *Overall Time –OT:* This is the time taken to finish Steps 1 to 6.
- *PMagic Time – PT:* is the time taken to complete Steps 1 to 4 and 6.
- *Remaining Time – RT:* This is the time taken to complete Step 5.

8.2.1 Deployment Policies of Primitive Events as Domain Size Increases

The purpose of this experiment is to study the impact on the time to deploy a policy as the number of host machines to which the policy applies increases. The results described in this Section are based on two policies. Each of these policies uses primitive events. These two policies differ in the number of management agents needed. The deployment of each policy assumes that there are no existing agent instances.

The first policy is the *cpu_Usage* (see Policy number 1 in Appendix B). The deployment of this policy requires one monitoring agent, one BAROC file and a TEC rule set of two rules. The first column in Table 8.1 represents the overall time in seconds to deploy the *cpu_Usage* policy.

The second policy is the *process_Monitor* (see Policy number 7 in Appendix B). The deployment of this policy requires two monitoring agents, one dynamic monitoring agent, one BAROC file and a TEC rule set of two rules. The second column in Table 8.1 represents the overall time in seconds to deploy the *process_Monitor* policy.

Policies	cpu_Usage	process_Monitor
1 Host	25	29
2 Hosts	27	31
3 Hosts	28	34
5 Hosts	30	38
7 Hosts	32	43
10 Hosts	36	46

Table 8.1: Deployment Time for Two Different Policies of Primitive Events

The results in Table 8.1 show that the time it takes to deploy each policy is approximately linear with respect to the number of hosts. Table 8.2 shows the time breakdown of the 25 seconds taken to deploy the *cpu_Usage* policy in one managed host, and Table 8.3 shows the breakdown of the 29 seconds spent to deploy the *process_Monitor* policy in one managed host.

As can be seen from Tables 8.2 and 8.3, the PT time is relatively small compared to the RT and the OT. We will not breakdown the other times in Table 8.1 to deploy either of the two policies to different number of hosts since the incremental time is approximately linear with respect to the number of hosts. In addition, the increment in the deployment time of both policies as the number of hosts increases is reasonable. Note that the only times affected as the number of hosts increase are the times for the Agent-Configuration and Agent-Startup tasks. The time needed for other tasks remains almost the same. This is because the number of agent instances that need to be configured and started up increases as the number of managed hosts increases.

Time Group	Task	Time in Seconds
PT	Agent-Finding	1.681
	Agent-Instance-Finding	0.205
	Agent-Configuration	.879
	Agent-Startup	.239
	Mapping To Tivoli	.122
	PT	**3.126**
RT	BAROC Import	6.010
	Rule Set Import	5.201
	Rule-Base Compile	6.122
	Rule-Base Load	5.022
	RT	**22.355**
	OT	**25.481**

Table 8.2: Deployment Time Breakdown for Deploying cpu_Usage policy

Time Group	Task	Time in Seconds
PT	Agent-Finding	2.412
	Agent-Instance-Finding	0.293
	Agent-Configuration	2.747
	Agent-Start-up	1.090
	Mapping To Tivoli	.135
	PT	**6.677**
RT	BAROC Import	5.771
	Rule Set Import	5.233
	Rule-Base Compile	6.431
	Rule-Base Load	5.192
	RT	**22.627**
	OT	**29.304**

Table 8.3: Deployment Time Breakdown for Deploying process_Monitor policy

8.2.2 Deployment of Policies of Composite Events as Domain Size Increases

The purpose of this experiment is similar to the experiment described in Section 8.2.1, i.e., to study the impact on the time to deploy a policy as the number of host machines to which the policy applies increases. However, the results described in this Section are based on two policies that use composite events. These two policies differ in the number of agents needed and the number of BAROC files and TEC rules. The deployment of each policy assumes that there are no existing agent instances.

The first policy is the *access_Monitor* (Policy number 17 in Appendix B). The deployment of this policy requires one monitoring agent, two BAROC files and a TEC rule set of two rules. The first column in Table 8.4 represents the overall time in seconds to deploy the *access_Monitor* policy.

The second policy is the *load_Control* policy (Policy number 15 in Appendix B). The deployment of this policy requires two monitoring agents, three BAROC files and a TEC rule set of 3 rules. The second column in Table 8.4 represents the overall time in seconds to deploy the *load_Control* policy.

Policies	access_Monitor	load_Control
1 Host	33	44
2 Hosts	34	46
3 Hosts	35	49
5 Hosts	37	52
7 Hosts	39	55
10 Hosts	43	59

Table 8.4: Deployment Time for Two Different Policies of Composite Events

The results in Table 8.4 show that the overall time it takes to deploy each policy is approximately linear with respect to the number of hosts. Table 8.5 shows the time breakdown of

the 33 seconds taken to deploy the *access_Monitor* policy in one managed host, while, Table 8.6 shows the breakdown of the 44 seconds spent to deploy the *load_Control* policy in one managed host.

As can be seen from Tables 8.5 and 8.6, the PT time is relatively small compared to the RT and the OT. We can observe from Table 8.4 that the incremental increase in the deployment overall time is approximately linear with respect to the number of hosts and the incremental amount of time is reasonable.

Time Group	Task	Time in Seconds
PT	Agent-Finding	1.151
	Agent-Instance-Finding	0.108
	Agent-Configuration	1.335
	Agent-Start-up	.454
	Mapping To Tivoli	.187
	PT	**3.235**
RT	BAROC Import	13.904
	Rule Set Import	5.793
	Rule-Base Compile	5.936
	Rule-Base Load	4.487
	RT	**30.120**
	OT	**33.355**

Table 8.5: Deployment Time Breakdown for access_Monitor policy

Time Group	Task	Time in Seconds
PT	Agent-Finding	2.667
	Agent-Instance-Finding	0.464
	Agent-Configuration	1.985
	Agent-Start-up	.691
	Mapping To Tivoli	.591
	PT	**6.398**
RT	BAROC Import	15.443
	Rule Set Import	7.201
	Rule-Base Compile	8.411
	Rule-Base Load	6.500
	Tivoli Tasks Total	**37.555**
	OT	**43.953**

Table 8.6: Deployment Time Breakdown for load_Control policy

8.2.3 Enforcement of Policy Rules

The purpose of this experiment is to study the impact on the time to enforce a policy rule using a centralized event handler. Particularly, this experiment determines the time of carrying out the action as the number of host machines to which the policy applies increases and as the number of triggered events increases. The results described in this Section are based on enforcement of the action *close the session* of the *session_Control* policy. The *session_Control* policy uses a primitive event (Policy number 3 in Appendix B). Deployment of this policy is done in the same way as described in the previous two experiments. The time shown in Table 8.7 represents the time from the detection of the first event (i.e., the first event detected in any of the managed hosts) and the action taken that corresponds to the last event detected (i.e., the last event detected in any of the managed hosts). This means that the time of 44 seconds shown in column 2 row 2 of Table 8.7 represents the time between the detection of the first event at one of the two hosts and the time the action is taken to close the last idle session found in one of the two hosts.

session_Control Policy **Time Between the first Detected Event and Last Action Taken (in Seconds)**				
	1 Event	**2 Events**	**5 Events**	**10 Events**
1 Host	12	31	69	104
2 Hosts	14	44	82	125
5 Hosts	17	63	101	153
10 Hosts	20	82	122	189

Table 8.7: The Enforcement of Policy Rule by using Tivoli

8.2.4 Discussion of Experiment Results

The experiments in Sections 8.2.1 and 8.2.2 addressed the time of deployment of specific policies using PMagic as the number of hosts grow. We found that the policy deployment time is acceptable with regards to increases in the number of hosts. We also found that the PT time is less than the RT and OT policy deployment times. Generally, most of the deployment time goes into importing, compiling and loading the TEC configurations that represent policies in TEC. This may suggest that configuring management agents, as addressed in Chapters 5 and 6, to carry out the policy rules enforcement could reduce the deployment overhead time.

The experiment in Section 8.2.3 studied the time to enforce a policy using TEC as a centralized event handler. The times showed that when several events came from the same host, the actions taken by TEC are delayed a bit. Generally, the times reported in this experiment show some delay in enforcement of the policy rules, especially when the number of hosts and event notification messages increase. This suggests that there may be some advantage to using a more decentralized policy enforcement approach instead of sending events to a central event processing engine. For instance, we may ask the management agents to operate as managers that process events and enforce policy rules (see Chapter 5 for more details). The next Section will explore this alternative.

8.3 Alternative Strategies for Optimization

This Section explores alternatives introduced in the Thesis, such as the reuse of existing agent instances to support more policies and the updating of the agent instances to adopt changes in policies. This Section also explores the use of management agents as managers to process events and enforce policy rules.

8.3.1 Experiments on Agent Reuse

In the experiments in Section 8.2 experiments we assumed that no agent instances are already instantiated. The experiments conducted in this Section compare the PT time of deploying policies when agent instances do not exist and when they do exist. The reuse of an existing agent instance to support an added policy means that existing management agent instances should be reconfigured to monitor and trigger the events of the added policy. Section 8.3.2 addresses how to facilitate the configuration of existing agent instances.

For the experiment of this Section, we selected the three policies: *cpu_Usage*, *process_Monitor* and *load_Control* (see Section 8.2 for more details about these policies). Table 8.10 shows comparisons between the PT times to deploy the policy when there are no existing agent instances and the time to configure existing agent instances which are already instantiated in the remote hosts that constitute the domain of the added policy.

We can see from the results shown in Table 8.8 that the reuse of existing agent instances will save almost half of the PT deployment time. Though it is not tested, we infer that the reuse of existing agent instances to support more policies could require less computational overhead in managed hosts than creating more instances of management agents.

Policies	cpu_Usage		process_Monitor		load_Control	
	No Instance	Instance Exists	No Instances	Instances Exist	No Instances	Instances Exist
1 Host	3	1.7	6.3	2.3	6.4	2.3
2 Hosts	4	2.4	7	3	7.5	3
3 Hosts	5	3	7.6	3.3	8	3.4
5 Hosts	7	3.5	8.7	4.2	9	4.2
7 Hosts	8	4	10	5.1	10	5.2
10 Hosts	10	5.2	12	6.3	13	6.3

Table 8.8: The Reuse of Existing Agents' Instances in Policy Deployment

8.3.2 Experiments on Policy Re-Enforcement

The proposed model addressed in Chapter 4 was structured in order to be able to provide dynamic adaptation to changes in polices. A change in an existing policy means that one or more thresholds of the conditions characterizing the primitive events are changed. These changes need to be reflected in the management agent instances used to generate the events. Section 8.3.1 has addressed the reuse of the management agent instances. To enable such reuse, the management agent instances need to be configured at runtime to support more policies. The reuse and updating of management agent instances suggests that direct communication between the manager and the management agent instances is needed. Updating the executing management agent instances with new changes and/or new configurations is typically done on the fly using direct communications (see Chapter 5 for more details). Management agents need to be built to allow such direct communications. RMI and Web-Services are examples of communications mechanisms that could be used in management agents. In our particular case, we chose Java RMI, but other communication mechanisms are also possible. The experiments in this Section measure the time in seconds it takes to reconfigure the agent instances to adopt the changes in policies.

Policies	cpu_Usage		process_Monitor		load_Control	
	PT	Update	PT	Update	PT	Update
1 Host	3	1	6.3	1.5	6.4	1.6
2 Hosts	4	1.2	7	2	7.5	2
3 Hosts	5	1.5	7.6	2.1	8	2.1
5 Hosts	7	2	8.7	2.4	9	2.4
7 Hosts	8	2.3	10	3	10	3
10 Hosts	10	3.4	12	4	13	4

Table 8.9: The Re-Enforcement of Three Different Policies using PMagic

The experiments in this Section make use of the policies that were used in the deployment experiments in Section 8.3.1. The goal of this experiment is to compare the time for updating agents with a new configuration (i.e., an existing policy has been changed such that one or more thresholds of the conditions characterizing the primitive events are changed) to the time required if these management agents were first found and then deployed (i.e., PMagic deploy time PT). An example of a possible change is a change of the *cpu_Usage* policy to alter the condition *cpuload>90* to *cpuload>85*. Such a change in a policy should be adopted by the agent instance that supports this policy, i.e., the agent instance needs to be asked to handle the new condition instead of the old one. Table 8.9 presents comparisons between the time for deploying and configuring management agents from scratch and the time to update to the configuration of management agent instances. As we can see from the results of Table 8.9, PMagic can update management agent instances to adopt the changes in the policies in a reasonable time even as the number of hosts scale.

Note that there are no Agent-Finding and Agent-Startup tasks performed in this re-enforcement experiment, since we assumed that the changes were only in the thresholds of the conditions in existing policies. However, if the changes involve a change in the attributes which must be monitored, for example in the construction of the condition of a primitive event, we may still need to do a full PMagic deployment, i.e., we still need to find and start the agents that can monitor the attributes specified in the condition. In such a case it may be necessary to keep track of what changes were done. This is considered to be future work.

8.3.3 Use of Management Agents as Managers

Experiments conducted in Section 8.2 assumed the use of TEC as the event-driven rule-based engine for handling and processing events and directing the enforcement of the policy rules. Chapter 6 also proposed that management agents could detect events and evaluate policy rules. By detecting events we mean the detection of both primitive and composite events. Chapter 5

described how primitive events can be detected by using monitoring and dynamic monitoring agents, and composite events can be detected using manager agents. We conducted the experiments of Section 8.2 such that the management agent be configured to carry out the policy rules, i.e., perform the enforcement of the policies the management agent supports. This means that events do not have to be sent to TEC for further processing and also that policies do not need to be mapped to Tivoli configurations. Specifically, we make use of management agents to work as managers for enforcing the policy rules that are associated with the triggered events these agents detect. In this case, we found that the policy deployment OT times were less than or almost equal to the PT times shown in Section 8.2 experiments.

We also conducted the same experiments as in Section 8.2.3, but with management agents configured to carry out both event detection and policy enforcement. The results are shown in Table 8.10. These results represent times that are substantially shorter compared to the times presented in Table 8.7.

session_Control Policy **Time Between the first Detected Event and Last Action Taken (in Seconds)**				
	1 Event	2 Events	5 Events	10 Events
1 Host	.028	.097	.621	1.603
2 Hosts	.035	.120	.887	1.917
5 Hosts	.041	.174	1.011	2.389
10 Hosts	.066	.251	1.633	2.907

Table 8.10: The Enforcement of Policy Rule by using Management Agents

8.3.4 Discussion of Alternative Strategies

The results of the experiments presented in the previous Sections show that the alternative strategies of utilizing management agents to support policies offer promising results.

Specifically, we found that: 1) The reuse of existing instances of management agents is better than creating new management agent instances to support the added policies; 2) Updating management agent instances to adopt the changes in policies is better than starting the management agents from scratch with the new changes, 3) Using management agents as managers for policy enforcements beside the event detections tasks, could be a promising direction.

8.4 Final Discussion and Conclusions Drawn

This Section discusses several key points that were addressed in the Thesis based on the experiments in this Chapter.

8.4.1 Mapping Policies to Tivoli

In the early Chapters of this Thesis we highlighted how the use of policies can facilitate the management system to be adaptable to changes in management strategies without requiring the recoding of the management system. The policies can be represented and specified by high-level policy languages. The principle foci of this Thesis are the design of approaches to map the specified policies to be realized by management systems. Chapter 6, together with the experiments of Section 8.2, show and validate how the proposed PMagic model does map different specified policies to Tivoli. Practically, we have shown and validated the mapping of the high-level specified policies elements to event format files (as in BAROC files) and to executable rules (as in TEC rules) by using the developed reusable templates that steer the automated mapping mechanism.

8.4.2 Identifying Management Agents to Support Policies

Existing management systems do not provide facilities to automate the efficient deployment of management entities i.e., finding, initiating and deploying management agents that monitor, analyze and control the managed system to support policies. More explicitly, existing

management systems are lacking in the definition of the link between their management rules and the management monitoring services (agents) that execute in order to collect events of interest to these rules from managed objects. This relationship between rules and agents is typically defined and configured by the system administrator. A key element of this Thesis work is policy deployment. All experiments conducted in this Chapter have validated that PMagic is bridging and automating the gap between expressing policies in a high-level specification language and the deployment of management agents to support policies.

One kind of question that may be raised is how efficient the agent finding algorithm is when there are many agents, i.e., when the number of agents scale. Let us briefly analyze this situation. The Agent-Finding algorithm tries to find agents that can monitor the policy primitive events. Normally, the number of specified primitive events in a policy is small. The search is done by matching the attribute names specified in these events to the attribute names that are associated with management agents. Specifying management agents and the associated attributes is done once and used many times, i.e., the Insert and Update operations to the database tables represent a negligible number of operations compared to the search (Select) operations in these tables. With respect to this fact, we construct the database agent attributes table to be indexed by the attribute names. The time to find an attribute name in a table indexed by attribute names is determined by the number of reads in a binary tree needed to find that attribute name. If n is the number of attribute rows in the agent attributes table, the number of reads is bounded by $log_2(n)$. For example, for 1,024 rows 10 reads are required and for 1,048,576 rows 20 reads, etc. More details on the power of indexing are in [55, 63]. These facts indicate that the increment in the number of management agent would have minimal effect on the performance of the agent finding algorithm.

However, this centralized rule-engine could potentially be a bottleneck. Thus, Chapter 6 proposed that management agents detect events and evaluate policy rules. Specifically, we used management agents as managers in enforcing the policy rules that were associated with the events these agents were detecting. To facilitate the decentralized event processing, the Thesis introduced manager agents to detect composite events. The algorithms used by the manager agents to evaluate the five composite event operators that were introduced in the Thesis are given in Appendix E. The initial results of experiments in Section 8.3.3 for using management agents as managers show promising results. Practically, more work is needed to decide when manager agents should be used versus just deploying an agent to detect primitive events.

8.4.6 Limitations of Experimental Environment

As introduced in Chapter 7, the prototype's software (TMF, TEC, gateway, Java and DB2 server) are installed on an old Sun Blade 100 Workstation with one 32 bit CPU of 0.49 GHz with 1.5GB of memory that uses Solaris 5.8. All communications in PMagic use Java RMI. This configuration is likely the cause of some slowness in deployment and enforcement times reported in the experiments discussed in this Chapter.

8.5 Chapter Summary

This Chapter described the experiments conducted to validate and evaluate PMagic and presented the conclusions drawn from these experiments.

Chapter 9

CONCLUSIONS AND FUTURE WORK

In this Chapter we review the contributions of the Thesis and discuss open issues and directions for future work.

9.1 Conclusions

The Thesis has reviewed and described several policy-based management systems, focusing primarily on policy specification languages and policy deployment systems. While there has been some work on automation of some aspects of policy-based management systems, there is clearly a need for more work on the automation of the mapping of policies to management elements (e.g. agents, rules), configuration of those management elements, the efficient runtime use and reuse of those elements, and the efficient reconfiguration of those elements in response to changes in the system being managed or in policies. This research focused on the means for a management system to automatically identify and deploy management operations, and management system

configurations for deploying policies. A central part of this research is the agent matcher concept which opens the door for more self-configuring management systems. The contributions of the Thesis can be considered as first steps towards the goal of automating policy-based management systems. The main contributions can be reviewed and summarized as follows:

- ***A Model for PBM System:*** A general PBM system model was proposed based upon features of the problem. The model is practical and also represents an abstract object model. The model provided a high-level language for representing policies and a means to abstractly characterize management agents. The model's primary characteristics are its ability to identify and deploy management entities and its ability to respond automatically to both changes to the system itself and to changes in the way the system is to be managed, i.e., changes to the set of management policies or sets of management agents. The model can be applied to any management system.

- ***The use of an existing management system:*** The Thesis has shown how to build the policy model and services on existing management services found in commercial management systems. We have shown, through Chapter 6 and with the experiments of Chapter 8, how to map different policies to Tivoli, and in particular, to event format files (BAROC files) and to executable rules (TEC rules) by using reusable templates to steer the automated mapping mechanism. The template-based approach introduced in the mapping of policies to any event-driven rule-based management system, enables the construction of more dynamic self-configuring PBM systems.

- ***Policy deployment algorithms:*** The Thesis has shown the needed services and sketched the necessary algorithms to identify, deploy and utilize management entities for policy deployment.

- ***A PMagic prototype implementation:*** The implementation demonstrated that a modular policy language could be implemented and used to specify policies. The prototype implementation successfully incorporated the agent finding algorithm and successfully demonstrated steps towards automating PBM systems with minimal administrator interaction. Construction of the implementation produced a number of insights into the challenges of automating policy-based management systems. These challenges are summarized in the following:

    1. It may be the case where one monitoring agent (*ma*) can not monitor all attributes of a primitive event. In this case a dynamic monitoring agent (*dma*) is instantiated to evaluate the condition that characterized the primitive event. For instance, in Example 6.3 (see also Appendix B Policy 16), if the management system provides a *ma* that monitors the *usersloginstotal* attribute, another *ma* that monitors the *cpuload* attribute, and a third *ma* that monitors the *cpuprocesstolal* attribute, then a *dma* is dynamically instantiated to receive the values of *cpuload* and *cpuprocesstolal* attributes from the last two *ma*s. The *dma* is used to detect instances of the primitive event characterized by the condition "*cpuload>90 && cpuprocesstotal>35*". The *dma* itself may be executed in one managed resource and receive messages of the required information from *ma*s in other managed resources. The problem here is that the monitoring information can be mixed when there are several resources (e.g., the value of *cpuload* of a host and the value *cpuprocesstotal* of another host), while the expression might need to be evaluated on information collected from the same managed resource. Thus, the values of *cpuload* and *cpuprocesstotal* need to be checked to confirm that they came from the same managed resource (i.e., host in our example case). We call this problem the *Information Collecting Challenge*. In PMagic, a logical expression can be associated with a set of attributes called the *Restraints Match*

Attributes for this Expression that defines the set of the attributes that need to be matched first by the *dma* among the received messages. *Restraints Match Attributes for this Expression* set can be entered using the PMagic interface and stored in the policy repository. If the received messages have a matching source, the *dma* then proceed to evaluate the condition.

2. A second issue, similar to the one addressed in the first point, deals with the event notifications messages, i.e., correlating the notification messages representing event instances can be mixed up from several resources. More practically, correlation of the received event instances to detect a composite event, which is constituted from these event instances, might be needed to evalutae only the received event instances that came from the same managed resource. Thus, correlation may rely on the values of some attributes in the event notification messages that need to be matched first. We call this problem *Event Correlating Challenge*. In PMagic a composite event can be associated with a set of attributes called *Event's Restraint Attributes* that defines the set of the attributes to be match among the events instances that constitute the composite event, e.g., *hotsname* could be one of the *Event's Restraint Attributes* that need to be matched in all the received events instances before any correlation.
3. The release of enforced actions when the system state represented in the triggered event, which causes the enforcement of the action, changes. In PMagic, the administrator might specify another new policy in which he/she specifies the policy event to be the composite event resulted from the application of the E-Not event operator to the event in our addressed case. The policy rule of the new policy in this case has a condition to check if the enforeced action that need to be released stills active and if so, the policy rule action of the new policy is to stop

or kill the enforcement of the action that needs to be released. More details on this are discussed in Section 9.2 on Future Work.

- ***Experiments:*** The experiments demonstrated the successful application of the model, the prototype and deployment of different policies into domains of differing numbers of hosts. The results of the basic experiments that evaluated the policy deployment using services of existing management systems motivated us to look for an alternative deployment approach for optimization, namely, one that relied on utilizing management agents for policy deployment.

- ***Reuse of management entities:*** Experimental results have demonstrated that reuse is a good strategy for management systems. Results show that it is possible to have a management system adapts to changes in the policies within a reasonable time as the number of hosts scale by reusing existing management agent instances. This strategy can be adopted and incorporated in any existing management framework, providing that the communications between the manager and the management agents instances to facilitate agent reuse and update exists.

- ***Decentralized event-handling mechanism:*** The Thesis introduced a further policy deployment approach in which policies are mapped to a configuration of management agents. Typically this approach requires management agents to work as managers for enforcing the policy rules that are associated with the events. This led to the introduction, design and implementation of the manager agents together with the algorithms needed to detect composite events.

9.2 Future Work

Although the work in this Thesis has achieved encouraging results, the research towards an optimal or a semi-optimal automated policy-based management system is still developing. There

a number of issues based upon the work in this Thesis and related research that have not been addressed that form the basis for future research:

- ***Searching and Agent Configuration:*** The algorithm described for agent finding can also be applied to agent instantiations, thus meaning that agent instantiations are first searched. A policy that is activated after an initial set of policies has been activated can minimize the number of instances of agents since information about agent instances is maintained. This in itself is not sufficient. Future algorithms should consider resource constraints and restrictions on the location and number of agent instantiations. A representation of this information, and mapped to an optimization model is needed. Work in [1] describes possible optimization models. These could be incorporated into the Agent Matcher component. Another possibility related to the management agents that were introduced in Chapter 5 would be to consider a single implementation of the three management agent types which could perform any of the tasks of *ma*, *dma* and/or *manager_agents*. This needs to be reviewed, and the performance and overhead of such a general agent would need to be carefully evaluated.

- ***Adding Consistency and Policy Conflicts Checking:*** Though the prototype provides consistency checking between policy intervals and policy rule intervals (if the latter are associated with a rule), there are still some areas that need consistency checks such as domains associated at the policy and policy rules levels. Also, as introduced in Chapter 2, the background and related work Chapter, the detection of policy conflicts is a challenging research problem that should be studied and incorporated in any production of PBM systems.

- ***Reuse of Mapping:*** The Thesis describes a template-based approach for generating event formats and rules from the policy specification. The actual template is management system specific. The mapping consists of parsing the policy specification to its

constituent components, e.g., events, where a mapping is defined from the constituent event to a template. The parsing is independent of the management system. For each management system a class could be defined where each method is associated with a policy element. This class is sub-classed for each management system. This approach makes it easier to use this work for different management systems.

- ***Communications Performance:*** The Thesis presented the design of management agents that can make use of existing (or legacy) agents. This adds an extra level of indirection which did result in additional overhead. However, recent work shows that the use of web services can be made feasible [112]. Our future work includes studying the specifications OASIS Management Using Web Services (MUWS) and DMTF Web Services for Management (WSManagement). Future work would look at using the work from the DMTF WBEM initiative.

- ***Actions Need Release:*** As a result of event triggering within the policy, actions of the policy rule will be enforced if some specified conditions are true. We consider the action of the policy of Example 6.3 (see also Policy number 16 in Appendix B) that arose from the deployment and enforcement of the policy *load_Control*. The action *block any new user logins* used in this example policy is executed remotely by a Java agent that continuously logs out any new user who tries to login to the host that triggered an instance of the event *users_cpu_process_High* once, i.e., violated the policy (*users_cpu_process_High* described before in our example policy *load_Control*). The action *block any new user logins* will be executed and will continue its job of logging out new users from the violated hosts without checking if *users_cpu_process_High* is triggered again or not. Such action needs to be released at the "not" occurrence of *users_cpu_process_High*. Although, PMagic has its vision on how to solve this challenge (see Section 9.1); future research will address this issue.

- ***Verify the Duplication in Logical-Expressions:*** One way to reduce the number of primitive events is to determine if the logical-expressions in primitive events are equivalent. The use of Reduced Ordered Binary Decision Diagram (ROBDD [133]), is a canonical form for the same logical expressions if the BDD is built using the same order of the labels of propositional formulas. The propositional formula corresponds to the Logical Factor used in the PMagic grammar (see Appendix A). This technique could also help to verify that the changes in the logical-expression that include attribute names are properly adopted by the agent instance (see Section 8.3.2 for more discussion about this problem).

- ***Adding Security:*** The PBM system model that has been presented in the Thesis does not explicitly address security, assuming an existing trust relationship between the manager and the managed resources. In addition, management agents are free to monitor the states of the managed resources. In many real-world environments, the authority to monitor, determine and change the configuration of a machine may be restricted and/or may have different security rights that assign to different groups or users. For the production of the proposed model this issue needs to be addressed.

- ***Evaluation for other Management Systems:*** Other areas of future work include evaluating the entire process within the scope of a different management system, such as CA Unicenter or HP Openview. Our inspection of these products suggests that our approach would work well, but this is clearly an area for more study.

- ***Larger Experimentation Environment:*** PMagic has only been evaluated within a limited environment. Future work is needed to test the algorithms and their scalability in a larger experimental environment.

References

1. H. Abdu, H. Lutfiyya and M. Bauer, "A Framework for Determining Efficient Management Configurations", Journal of Computer Networks, Volume 46, Issue 4, November 2005, pp 437-463

2. I. Adhicandra, C. Pattinson and E. Shaghouei, "Using Mobile Agents to Improve Performance of Network Management Operations", Postgraduate Networking Conference (PGNET 2003), Liverpool, UK, 2003, Available Online, http://www.cms.livjm.ac.uk/pgnet2003/submissions/Paper-12.pdf, Last accessed date June 15, 2009.

3. K. Al-Agha, M. Gerla and G. Pujolle, "Adaptive QoS Management for IEEE 802.11 Future Wireless ISPs", Journal of Ad-Hoc Networking ACM Wireless Networks Journal, Kluwer, Volume 10, Issue 4, July 2004, pp 413-421.

4. D. Agrawal, W. Lee and J. Lobo, "Policy-Based Management of Networked Computing Systems", IBM T. J. Watson Research Center, IEEE Communications Magazine, October 2005, pp 69-75.

5. D. Agrawal, C. Seraphin, W. Lee and J. Lobo, "Issues in Designing a Policy Language for Distributed Management of IT Infrastructures", In Proceedings of the 10th IEEE/IFIP International Symposium on Integrated Network Management (IM 2007), Munich, Germany, May 2007, pp 30-39.

6. D. Agrawal, J. Giles, W. Lee and J. Lobo, "Policy Ratification", In Proceedings of the 6th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy2005), Stockholm, Sweden, June 2005, pp 223-232.

7. T. Ahmed, A. Mehaoua and R. Boutaba, "Dynamic QoS Adaptation using COPS and Network Monitoring Feedback", In Proceedings of the IFIP/IEEE International Conference on Management of Multimedia Networks and Services, Santa Barbara, CA, October 2002, pp 250-262.

8. O. D. Alcántara and D. McCluskey, "Towards Policy-Based Management QoS in Multicommunicative Education", Lecture Notes in Computer Science (LNCS), Springer, Volume 2105, 2001, pp 237-248.

9. I. Aib, N. Agoulmine and G. Pujolle, "A Multi-Party Approach to SLA Modeling, Application to WLANs", In Proceedings of the 2nd IEEE Consumer Communications and Networking Conference (CCNC'05), Las Vegas, USA, January 2005, pp 451-455.

10. M. Baldi and G. P. Picco, "Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications", In Proceedings of the 20th IEEE International Conference on Software Engineering, Kyoto, Japan, April 1998, pp 146-155.

11. A. Bandara, E. Lupu, J. Moffett and A. Russo, "A Goal-Based Approach to Policy Refinement", In Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy2004), New York, USA, April 2004, pp 223-232.

12. A. Bandara, E. Lupu and A. Russo, "Using Event Calculus to Formalize Policy Specification and Analysis", In Proceedings of the 4th IEEE Workshop on Policies for Distributed Systems and Networks (Policy2003), Como, Italy, June 2003, pp 26-39.

13. C. Baral, M. Gelfond, and A. Provetti, "Representiing Actions: Laws, Observations and Hypothesis", Journal of Logic Programming, Volume 31, Issue 3, October 1997, pp 201-244.

14. M. Bauer and H. Akhand, "Managing Quality of Service in Internet Applications using Differentiated Services", Journal of Network and Systems Management, Volume 10, Issue 1, March 2002, pp 39-62.

15. M. Bearden, S. Garg, and W. Lee, "Integrating Goal Specification in Policy-Based Management", In Proceedings of the 2nd IEEE International Workshop on Policies for Distributed Systems and Networks (Policy2001), Bristol, UK, January 2001, pp 29-31.

16. BMC Software Common Event Format, Version 2.1.0, http://documents.bmc.com/products/documents/37/30/53730/53730.pdf, Last accessed date June 15, 2009.

17. J. Bradshaw and P. Beautement, A. Raj, M. Johnson, S. Kulkarni and N. Suri, "Chapter12: Making agents acceptable to people", In N. Zhong and J. Liu (Eds.) 2002, Handbook of Intelligent Information Technology, Amsterdam, The Netherlands, Available Online, http://www.ihmc.us/research/projects/KAoS/biit-jeff.pdf, Last accessed date June 15, 2009.

18. L. Brownston, R. Farrell and E. Kant, "*Programming Expert Systems in OPS5 Reading*", Addison-Wesley. 1995.

19. F. Bry, M. Eckert and P. Patranjan, "Reactivity on the Web: Paradigms and applications of the language XChange", Journal of Web Engineering, Volume 5, Issue 1, May 2006, pp 3-24.

20. M. Brunner and J. Quittek, "MPLS Management Using Policies", In Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management (IM2001), Seattle, WA, USA, May. 2001, pp 515-528.

21. M. Brunner and A. Prieto, "SLS to DiffServ Configuration Mappings", In Proceedings of the 12th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM2001), Nancy, France, October 2001, pp 15-17.

22. M. Burgess, "Cfengine: a system configuration engine", Technical report number 1993-9, University of Oslo, October 1993, Available Online, http://www.iu.hio.no/~mark/papers/cfengine_history.pdf, Last accessed date June 15, 2009.

23. CA-Unicenter, http://www.ca.com, Last accessed date June 15, 2009.

24. S. Calo and M. Sloman, "Policy-Based Management of Networks and Services", Journal of Network and Systems Management, Springer Netherlands, Volume 11, Issue 3, September 2003 , pp 249-377.

25. Canonical Situation Data Format: The Common Base Event, http://www.eclipse.org/tptp/platform/documents/resources/cbe101spec/CommonBaseEvent_SituationData_V1.0.1.pdf, Last accessed date June 15, 2009.

26. N. Carver, "A Revisionist View of Blackboard Systems", In Proceedings of the 8th Midwest Artificial Intelligence and Cognitive Science Society Conference (MAICS '97), Dayton, Ohio, USA, May 1997, pp 15-22.

27. H. Chaouchi and A. Munaretto, "Adaptive QoS Management for IEEE 802.11 Future Wireless ISPs, Center for Telecommunications Research", In Wireless Networks, Volume 10, Issue 4, July 2004, pp 413-421.

28. J. Chomicki and J. Lobo, "A Logic Programming Approach to Conflict Resolution in Policy Management", In Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR2000), Breckenridge, Colorado, USA, April 2000, pp 121-132.

29. J. Chomicki and J. Lobo, "Monitors for history-based policies", In Proceedings of the 2nd IEEE Workshop on Policies for Distributed Systems and Networks (Policy2001), Bristol, UK, January 2001, pp 57-72.

30. L. Choonhwa, A. Helal, N. Desai and V. Verma, B. Arslan, "Konark: A system and protocols for device independent, peer-to-peer discovery and delivery of mobile services", In the Proceedings of the IEEE Transactions on Systems and Humans, Volume 33, Issue 6, November 2003, pp 682-696.

31. QPM-Cisco's QoS Policy Manager, http://www.cisco.com/en/US/products/sw/cscowork/ps2064/index.html, Last accessed date June 15, 2009.

32. CiscoAssure, http://newsroom.cisco.com/dlls/prod_031098.html, Last accessed date June 15, 2009.

33. N. Damianou, "A Policy Framework for Management of Distributed Systems", PhD Thesis, Department of Computing, Imperial College, London, UK, March 2002.

34. N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "Ponder: A Language for Specifying Security and Management Policies for Distributed Systems: The Language Specification (version 2.2)", Technical report number 2000-01, Deptartment of Computing, Imperial College, London, UK, April 2000.

35. V. Danciu and B. Kempter, "From Processes to Policies -Concepts for Large Scale Policy Generation", In Proceedings of the 9th IEEE/IFIP Network Operations and Management Symposium (NOMS2004), Seoul, Korea, April 2004, pp 17-30.

36. R. Darimont and A. Lamsweerde, "GRAIL/KAOS: An environment for goal-driven requirements engineering", In Proceedings of the 19th IEEE International Conference on Software Engineering (ICSE1997), Kyoto, Japan, April 1997, pp 612-613.

37. R. Darimont and A. Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", In Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering (FSE4), November 1996, pp 179-190.

38. M. Debusmann and A. Keller, "SLA-driven Management of Distributed Systems using the Common Information Model", In Proceedings of the 8th IEEE/IFIP International Symposium on Integrated Network Management (IM2003), Colorado, USA, March 2003, pp 563-576.

39. DMTF, CIM Core Model White Paper (CIM Version 2.4), 2002, www.dmtf.org, Last accessed date June 15, 2009.

40. DMTF, CIM Policy Model White Paper (CIM Version 2.7). 2003, www.dmtf.org, Last accessed date June 15, 2009.

41. N. Dulay, E. Lupu, M. Sloman and N. Damianou, "A Policy Deployment Model for the Ponder Language", In Proceedings of the 7th IEEE/IFIP International Symposium on Integrated Network Management (IM2001), Seattle, WA, USA, May 2001, pp 529-543.

42. N. Dunlop, J. Indulska and K. Raymond, "Dynamic Policy Model for Large Evolving Enterprises", In Proceedings of the 5th IEEE Enterprise Distributed Object Computing Conference, Seattle, WA, USA, September 2001, pp 193-197.

43. N. Dunlop, J. Indulska and K. Raymond, "Dynamic Conflict Detection in Policy-Based Management Systems", In Proceedings of the 6th IEEE Enterprise Distributed Object Computing Conference EDOC'02), Lausanne, Switzerland ,September 2002, pp 15-26.

44. N. Dunlop, J. Indulska and K. Raymond, "Methods for Conflict Resolution in Policy-Based Management Systems", In Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference, Brisbane, Australia, September 2003, pp 98-109.

45. D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry, "The COPS (Common Open Policy Service) Protocol", Standards Track RFC 2748, IETF, Network Working Group, January 2000.

46. T. Dursun, T. Uekae and P. Yolum., "A Generic Policy-Conflict Handling Model", Lecture Notes in Computer Science (LNCS), Springer, May 2005, Volume 3733, pp. 193-204,

47. L. Fallon, D. Parker, M. Zach, M. Leitner and S. Collins, "Self-Forming Network Management Topologies in the Madeira Management System", Lecture Notes in Computer Science (LNCS), Springer, July 2007, Volume 4543, pp 61-72.

48. M. Ferudin, W. Kasteleign and W. Krause, "Distributed Management with Mobile Components", In Proceedings of the 6th IFIP/IEEE International Symposium on Integrated Network Management (IM1999), Boston, MA, USA, May 1999, pp 515-528.

49. R. Helm, R. Johnson, and J. Vlissides, "*Design Patterns- Elements of Reusable Object*", Addison Wesley, November 1995, pp. 163-195.

50. A. Gilbert and C. Schaubach, "What is PMAC (Policy Management for Autonomic Computing?)", IBM alphworks presentation, 2005.

51. C. Goh, "A Generic Approach to Policy Description in System Management", In Proceedings of the 8th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM1997), Sydney, Australia, October 1997, pp. 1-12.

52. G. Goldszmidt and Y. Yemini, "Delegated agents for network management" IEEE Communications Magazine, Volume 36, Issue 3, March 1998, pp. 66-70.

53. Gorgias: Argumentation and Abduction, http://www2.cs.ucy.ac.cy/~nkd/gorgias/, Last accessed date June 15, 2009.

54. L. Z. Granville, R. S. Alves, M. J. Almeida and L. M. Tarouco, "Proposal, Implementation, and Analysis of an Atomic Policy Deployment Protocol for QoS-Enabled Networks", Lecture Notes in Computer Science (LNCS), Springer, October 2006, Volume 4268, pp 132-143.

55. P. Gulutzan and T. Pelzer, "*SQL Performance Tuning*", Addison Wesley, September 2002.

56. M. Hasan, "The Management of Data, Events, and Information Presentation for Network Management", PhD Thesis, Computer Science Departement, University of Waterloo, May 1996.

57. W. J. Heaven and A. Finkelstein, "A UML Profile to Support Requirements Engineering with KAOS", IEEE Software, Volume 151, Issue 1, September 2004, pp.10-27.

58. H. G. Hegering, S. Abeck and B. Neumair, "*Integrated Management of Networked Systems: Concepts, Architectures and Their Operational Application*", Morgan Kaufmann, November 1999.

59. Hitachi JP1/Integrated Management, http://secunia.com/advisories/product/20778/, Last accessed date June 15, 2009.

60. M. Hitchens and V. Varadharajan, "Tower: A Language for Role Based Access Control", In Proceedings of the 2nd IEEE Workshop on Policies for Distributed Systems and Networks (Policy2001), Bristol, UK, January 2001, pp 88-106.

61. HP Openview, http://www.hp.com, Last accessed date June 15, 2009.

62. IBM, Autonomic Computing Policy Language, 2005, http://www.research.ibm.com/people/k/kangwon/publications/policy_comm_mag.pdf, Last accessed date June 15, 2009.

63. IBM DB2 UDB Version 8 Product ManualsAdministration Guide: Performance, ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/en_US/db2d3e81.pdf, Last accessed date June 15, 2009.

64. IBM, Policy Management for Autonomic Computing, http://www.alphaworks.ibm.com/tech/pmac, Last accessed date June 15, 2009.

65. IETF, http://www.ietf.org, Last accessed date June 15, 2009.

66. IETF, RFC 2748, The COPS (Common Open Policy Service) Protocol, http://www.ietf.org/rfc/rfc2748.txt, Last accessed date June 15, 2009.

67. IETF, RFC 2790, Host Resources Mib, http://www.ietf.org/rfc/rfc2790.txt, Last accessed date June 15, 2009.

68. IETF, RFC 3084, COPS Usage for Policy Provisioning (COPS-PR), http://www.ietf.org/rfc/rfc3084.txt, Last accessed date June 15, 2009.

69. IETF, RFC 3198 - Terminology for Policy-Based Management, 2001, http://www.ietf.org/rfc/rfc3198.txt, Last accessed date June 15, 2009.

70. International Organization for Standardization (ISO), http://www.iso.ch, Last accessed date June 15, 2009.

71. G. Jakobson and M. D. Weissman, "Alarm Correlation", IEEE Network, Volume 7, Issue 6, November 1993, pp 52-59.

72. G. Jakobson, J. Buford, and L. Lewis, "Towards an Architecture for Reasoning about Complex Event-Based Dynamic Situations", In Proceedings of the 3rd International Workshop on Distributed Event Based Systems (DEBS 2004), Edinburgh, UK, May 2004, pp 62-67.

73. P. Jackson, "*Introduction to Expert Systems*", Addison-Wesley, 1999.

74. JESS- The Rule Engine for the Java Platform, http://www.jessrules.com/, Last accessed date June 15, 2009.

75. T. Jonatan, "SLA Enforced by Policy", A M.Sc. Thesis, Computer Science Department, University of Twente, June 2001.

76. L. Kagal, "Rei: A Policy Language for the Me-Centric Project. Enterprise Systems Data Management Laboratory", Technical report number HPL-2002-270, HP Laboratories, September 30, 2002, Available Online, http://ebiquity.umbc.edu/_file_directory_/papers/57.pdf, Last accessed date June 15, 2009.

77. A. Kakas, A. Bandara, A. Russo, E. Lupu, M. Sloman and N. Dulay, "Reasoning Techniques for Analysis and Refinement of Policies for Service Management", Technical report number 2005-7, Department of Computing, Imperial College London, UK, June 2005.

78. Y. Kanada, "A Representation of Network Node QoS Control Policies Using Rule-Based Building Blocks", In Proceedings of the 8th IEEE International Workshop on Quality of Service (IWQOS2000), Pittsburgh, PA, USA, June 2000, pp 161-163.

79. A. Keller and H. Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services", IBM Research Division, JNSM, Volume 11, Issue 1, March 2003.

80. M. Kona and C. Z. XU, "An Integrated Mobile Agent Framework for Distributed Network Management", The International Journal of Parallel, Emergent and Distributed Systems, Volume 20, Issue 1, March 2005, pp 39-55.

81. D. Lamanna, J. Skene and W. Emmerich, "SLAng: A Language for Defining Service Level Agreements", In Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS2003), San Juan, Puerto Rico, USA, May 2003, pp 100-106.

82. A. Liotta, G. Pavlou, and G. Knight, "Exploiting Agent Mobility for Large-Scale Network Monitoring", In IEEE Network, Volume 16, Issue 3, June 2002, pp 7-15.

83. A. Liotta, G. Pavlou, and G. Knight, "On the Efficiency and Scalability of Decentralized Monitoring using Mobile Agents", In Proceedings of the 6th On-Line HP Openview University Association Plenary Workshop (Hp-OVUA'99), Bologna, Italy, June 1999.

84. A. Liotta, G. Pavlou, and G. Knight, "Decomposition Patterns for Mobile-Code-based Management", In Proceedings of the 5th On-Line HP Openview University Association Plenary Workshop (Hp-OVUA'98), ENST de Bretagne, Rennes, France. April 1998.

85. A. Liotta, G. Pavlou, and G. Knight, "Modelling Network and System Monitoring over the Internet with Mobile Agents". In Proceedings of the 6th IEEE/IFIP Network Operations and Management Symposium (NOMS1998), Louisiana, USA, April 1998, pp 303-312.

86. J. Lobo, R. Bhatia and S. Naqvi, "A Policy Description Language", In Proceedings of the 16th IEEE National Conference on Artificial Intelligence (AAAI1999), Orlando, Florida, USA, June 1999, pp 291-298.

87. Lucent Technologies RealNet Policy Rules, http://www.lucent.com/wps/portal/Solutions/detail?LMSG_CABINET=Solution_Product_Catalog&LMSG_CONTENT_FILE=Solutions/Solution_Detail_000035.xml, Last accessed date June 15, 2009.

88. H. Lutfiyya, G. Molenkamp, M. Katchabaw and M.Bauer, "Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework", In Proceedings of the 2nd IEEE Workshop on Policies for Distributed Systems and Networks (Policy2001), Bristol, UK, January 2001, pp 185-201.

89. L. Lewis, "On the Integration of Service Level Management and Policy-Based Control", In Proceedings of the 1st IEEE Workshop on Policies for Distributed Systems and Networks (Policy1999), Bristol, UK, November 1999.

90. L. Lewis, "*Service Level Management for Enterprise Networks*", Artech House, October 1999.

91. E. Lupu, M. Sloman, "Conflict Analysis for management Policies", In Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM1997), San Diego, California, USA, May 1997, pp 430-443.

92. L. Lymberopoulos, E. Lupu and M. Sloman, "Using CIM to Realize Policy Validation within the Ponder Framework", In Proceedings of the DMTF Global Management Conference (DMTF2003), San Jose, CA, USA, June 2003, pp 16-19.

93. L. Lymberopoulos, E. Lupu and M. Sloman, "Ponder Policy Implementation and Validation in a CIM and Differentiated Services Framework", In Proceedings of the 9th IFIP/IEEE Network Operations and Management Symposium (NOMS2004), Seoul, Korea, April 2004, pp 31-44.

94. L. Lymberopoulos, E. Lupu, and M. Sloman, "An Adaptive Policy Based Management Framework for Differentiated Services Networks", In Proceedings of the 3rd IEEE Workshop on Policies for Distributed Systems and Networks (Policy2002), Monterey, CA, USA, June 2002, pp 147-158.

95. V. Machiraju, A. Sahai and A. Moorsel, "Web Services Management Network: An overlay network for federated service management", In Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management (IM2003), Colorado, USA, March 2003, pp 351-364.

96. S. Mandis, B. Seraphin and D. Verma, "Policy Transformation Techniques in Policy-based Systems Management", In Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy2004), New York, USA, April 2004, pp 13-22.

97. M. Mansouri, "Monitoring of Distributed Systems", PhD Thesis, Department of Computing, Imperial College, London, UK, December 1995.

98. M. Mansouri and M. Sloman, "GEM: A generalized event monitoring language for distributed systems", Journal of Distributed Systems Engineering, Volume 4, Issue 2, July 1997, pp 96–108.

99. E. Marilly, O. Martinot, S. Betge-Brezetz and G. Delegue, "Requirements for service level agreement management", In Proceedings of the IEEE Workshop on IP Operations and Management (IPOM2002), Dallas, Texas USA, October 2002, pp 57-62.

100. D. Marriott and M. Sloman, "Management Policy Service for Distributed Systems", In Proceedings of the 3rd International Workshop on Services in Distributed and Networked Environments (SDNE1996), Macau, China, June 1996, pp 2-9.

101. M. Martinez,M. Brunner, J. Quittek, F. Straub, J. Schönwälder, S. Mertens and T. Klie, "Using the Script MIB for Policy-based Configuration Management", In Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS2002), Genova, Italy, April 2002, pp 187-202.

102. M. Masullo and S. Calo, "Policy Management: An architecture and approach", In Proceedings of the 1st IEEE International Workshop On System Management Techniques, Processes, and Services (SMTPS1993) , Los Angeles, CA, USA, April 1993, pp 13-26.

103. S. Mazumdar and K. Swanson, "WEB Based Management CORBA/SNMP Gateway Approach", In Proceedings of the 7th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM1996), L'Aquila, Italy, October 1996, pp 110-121.

104. S. Mazumdar, "Mapping of Common Management Information Service (CMIS) to CORBA Object Services", Technical report number BL0112540-96.09.30-02, Bell Laboratories, September 1996.

105. Microsoft SMS and MOM, http://www.microsoft.com, Last accessed date June 15, 2009.

106. J. Moffett and M. Sloman, "Policy Hierarchies for Distributed Systems Management", In Proceedings of the IEEE Special Issue on Network Managemnet (JSAC), Volume 11, Issue 9, December 1994, pp 1404-1414.

107. J. Moffett and M. Sloman, "Policy Conflict Analysis in Distributed System Management", Journal of Organizational Computing, Volume 4, Issue 1, April 1994, pp 1-22.

108. G. Molenkamp, H. Lutfiyya, M. Katchabaw and M. Bauer, "Diagnosing Quality of Service Faults in Distributed Applications", In Proceedings of the 21st IEEE International Conference on Performance, Computing, and Communications, Phoenix, AZ, USA, April 2002, pp 375-382.

109. M. C. Mont, A. Baldwin, and C. Goh, "POWER Prototype: Towards Integrated Policy-Based Management", In Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS2000), Honolulu, HI, USA, April 2000, pp 789-802.

110. B. Moore, E. Ellesson, J. Strassner and A. Westerinen, "Policy Core Information Model-PCIM Version 1 Specification", Standards Track RFC 3060, IETF, Network Working Group, February 2001.

111. B. Moore, "Policy core information model extensions", Internet Engineering Task Force RFC 3460, January 2003.

112. G. Moura, G. Silvestrin, L. Gaspary, L. Zambenedetti, "On the Performance of Web Services Management Standards – An Evaluation of MUWS and WS-Management for Network Management", In Proceedings of the 10th IEEE/IFIP International Symposium on Integrated Network Management (IM2007), Munich, Germany, May 2007, pp 459-468.

113. N. Muruganantha and H. Lutfiyya, "Policy Specification and Architecture for Quality of Service Management", In Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management (IM2003), Colorado, USA, March 2003, pp 535-548.

114. NerveCenter - Network Management and Event Correlation System, http://sun.systemnews.com/articles/58/1/marketplace/8418, Last accessed date June 15, 2009.

115. Nortel's Optivity, http://products.nortel.com/go/product_index.jsp, Last accessed date June 15, 2009.

116. Joint XOpen/NM Forum Inter-Domain Management Taskforce, "Comparison of the OSI Management, OMG and Internet Management Object Models", OMG Document Number 94.3.7, March 1994.

117. A. N. Ouda, H. Lutfiyya, and M. Bauer, "Mapping Policies to Management Systems", In Proceedings of the 10th IEEE Workshop on Policies for Distributed Systems and Networks (Policy2009), London, UK, July 2009, pp under publishing.

118. A. N. Ouda, H. Lutfiyya, and M. Bauer, "Towards Self-Configuring Policy-Based Management Systems", In Proceedings of the 9th IEEE Workshop on Policies for Distributed Systems and Networks (Policy2008), Palisades, New York, USA, June 2008, pp 215-218

119. A. N. Ouda, H. Lutfiyya, and M. Bauer, "Towards Automating the Adaptation of Management Systems to Changes in Policies", In Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS2006), Vancouver, Canada, April 2006, pp 1-4.

120. A. N. Ouda, H. Lutfiyya, and M. Bauer, "Understanding the Relationship Between High-Level Specification of Policies and Management Processes", In Proceedings of the 10th On-Line of HP Openview University Association, Plenary Workshop (Hp-OVUA'03), Geneva, Switzerland, July 2003.

121. Application Response Measurement (ARM), http://regions.cmg.org/regions/cmgarmw/marcarm.html, Last accessed date June 15, 2009.

122. RFC-4498, http://www.rfc-archive.org/getrfc.php?rfc=4498, Last accessed date June 15, 2009.

123. P. D. Rosa, C. Melchiors and L.Granville, "Designing the Architecture of P2P-Based Network Management Systems", In Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC2006), New York, USA, June 2006, pp 69-75.

124. B. Pagurek, Y. Wang and T. White, "Integration of Mobile Agents with SNMP: Why and How", In Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS2000), Honolulu, HI, USA, April 2000, pp 609-622.

125. P. Pereira1, D. Sadok and P. Pinto, "Service Level Management of Differentiated Services Networks with Active Policies", In Proceedings of the 3rd Conference on Telecommunications (ConfTele2001), Figueira da Foz, Portugal, April 2001, pp 542-546.

126. G. Perrow, J. Hong, H. Lutfiyya and M. Bauer, "The Abstraction and Modelling of Management Agents", In Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network Management(IM1995), Santa Barbara CA, USA, May 1995, pp 466-478.

127. P. R. Pietzuch, B. Shand and J. Bacon, "A Framework for Event Composition in Distributed Systems", In Proceedings of the IEEE International Middleware Conference, Rio de Janeiro, Brazil, June 2003, pp 62-82.

128. R. Pinheiro,A. Poylisher and H. Caldwell, "Mobile Agents for Aggregation of Network Management Data", In Proceedings of the 3th IEEE International Symposium on Mobile Agents, Palm Springs, CA, USA, March 1999, pp 130-140.

129. A. Ramnath and L. Ratan, "The Lucent Technologies Softswitch - Realizing the promise of convergence", Bell Labs Technical Online Journal, August 2002, pp 174-195.

130. G. Rodosek, "A Generic Model for IT Services and Service Management", In Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management (IM2003), Colorado, USA, March 2003, pp 171-184.

131. A. Sahai, V. Machiraju, M. Sayal, A. Moorsel and F. Casati, "Automated SLA Monitoring for Web Services", In Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM2002), Montreal, Canada, October 2002, pp 28-41.

132. M. Sallé and C. Bartolini, "Management by Contract", In Proceedings of the 9th IEEE/IFIP Network Operations and Management Symposium (NOMS2004), Seoul, Korea, April 2004, pp 787-800.

133. K. Schneider, "*Verification of reactive Systems – Formal Methods and Algorithms*", Springer, 2004.

134. J. Schonwalder, "Method and System for Network Management with Backup Status Gathering", United States Patent 7305461, Issued on December 4, 2007.

135. J. Schonwalder and J. Quittek,. "Secure Internet Management By Delegation", IEEE Computer Networks, Volume 35, Issue 1, January 2001, pp 39-56.

136. S. Schwiderski, "Monitoring the Behaviour of Distributed Systems",PhD Thesis, Computer Laboratory, University of Cambridge, April 1996.

137. E. Al-Shaer and H. Hamed, "Firewall policy advisor for anomaly detection and rule editing", In Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management (IM2003), Colorado, USA, March 2003, pp 17-30.

138. E. Al-Shaer and B. Zhang, "HiFi+: A Monitoring Virtual Machine for Autonomic Distributed Management", In Proceedings of the 15th IFIP/IEEE Distributed Systems: Operations and Management (DSOM2004), November 2004, pp 28-39.

139. M. Sloman, "Network and Distributed Systems Management", Addison Wesley, 1994.

140. M. Sloman, "Policy Driven Management for Distributed Systems", Journal of Network and Systems Management, Volume 2, Issue 4, December 1994, pp 333-360.

141. M. Sloman, "Specifying Policy for Management of Distributed Systems", In Proceedings of the 4th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM1993), Long Branch, USA, October 1993, pp. 52-67.

142. M. Sloman and E. Lupu, "Security and Management Policy Specification", IEEE Network, Volume 16, Issue 2, April 2002, pp. 10-19.

143. N. Smith, J.Leaney and T. Hunter, "A Policy-Driven Autonomous System for Evaluative and Adaptive Management of Complex Services and Networks", In Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05), Greenbelt, Maryland, USA, April 2005, pp 389-397.

144. SNMP-Simple Network Management Protocol, http://www.faqs.org/rfcs/rfc1157.html, Last accessed date June 15, 2009.

145. T. C. Son and J. Lobo, "Reasoning about Policies using Logic Programs", In Proceedings of the 1st International Workshop on Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning (ASP2001), Stanford, CA, USA, March 2001, pp 210-216.

146. G. Stone, B. Lundy and G. Xie, "Network Policy Languages: A survey and a new approach", IEEE Network, Volume 15, Issue 1, February 2001, pp 10-21.

147. J. Strassner, "*Policy-Based Network Management, Solutions for the Next Generation*", Morgan-Kaufmann, August 2003.

148. J. Strassner, "DEN-ng: Achieving Business-Driven, Network Management", In Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS2002), Genova, Italy, April 2002, pp 753-766.

149. R. Subramanyan, J. Miguel-Alonso, J. Fortes, "A Reconfigurable Monitoring System for Large-Scale Network Computing", Euro-Par 2003 Parallel Processing, 2003, pp. 98-108.

150. Sun Microsystems, http://java.com/en/, Last accessed date June 15, 2009.

151. Tivoli, http://www.tivoli.com, Last accessed date June 15, 2009.

152. Y. Udupi, A. Sahai and S. Singhal, "A Classification-Based Approach to Policy Refinement", In Proceedings of the 10th IEEE/IFIP International Symposium on Integrated Network Management (IM2007), Munich, Germany, May 2007, pp 785-788.

153. D. Verma, M. Beigi and R. Jennings, "Policy Based SLA Management in Enterprise Networks", IBM Thomas J Watson Research Center, Yorktown Heights, NY USA, 2007, Available Online, http://www.research.ibm.com/people/d/dverma/papers/PolicyWkShop2001.pdf, Last accessed date June 15, 2009.

154. D. Verma, "*Policy-Based Networking: Architecture and Algorithms*", New Riders, November 2000.

155. D. Verma, "Simplifying Network Administration Using Policy-Based Management", IEEE Network, Volume 16, Issue 2, April 2002, pp 20-26.

156. R. Wies, "Using a Classification of Management Policies for Policy Specification and Policy Transformation", In Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network Management(IM1995), Santa Barbara CA, USA, May 1995, pp 44-56.

157. Winsock RSHD/NT, http://www.denicomp.com/rshdnt.htm, Last accessed date June 15, 2009.

158. WSDM Management white paper, May 2007, http://www.ibm.com/developerworks/library/specification/ws-wsdmmgmt/, Last accessed date June 15, 2009.

159. Understanding X.500 - The Directory, http://sec.cs.kent.ac.uk/x500book/, Last accessed date June 15, 2009.

160. H. Xu, and D. Xiao, "Towards P2P-based Computer Network Management", International Journal of Future Generation Communication and Networking, Volume 2, Issue 1, March 2009, pp 25-32.

161. Y. Yemini, G. Goldzmidt, and S. Yemini, "Network management by delegation", In Proceedings of the International Symposium on Integrated Network Management, April 1991, pp 95-107.

162. M. Zapf, K. Herrmann, K. Geihs "Decentralized SNMP Management Agents", In Proceedings of the 6th IFIP/IEEE International Symposium on Integrated Network Management (IM1999), Boston, MA, USA, May 1999, pp. 623-635.

163. D. Zhu and A. S. Sethi., "SEL, a New Event Pattern Specification Language for Event Correlation", In Proceedings of the 10th IEEE International Conference on Computer Communications and Networks (ICCCN2001), Phoenix, AZ, USA, September 2001, pp 586–589.

Appendix A: The Policy Grammar

```
<Policy> ::= <Policy-Identification> <Domain-Identification> <Event-Expression>{<Rule>} +
             ("in" <Interval>)

<Rule> ::= <Logical-Expression>? {<Action>}+ ("in" <Interval>) ?

<Action> ::= <Function-Call-Expression>

<Event-Expression> ::= <Logical-Expression>
                     | <Event-Expression> "E-AND" <Event-Expression> ("in" < Event-Time-Window>)?
                     | <Event-Expression> "E-OR"  <Event-Expression> ("in" < Event-Time-Window>)?
                     | <Event-Expression> "E-SEQ" <Event-Expression> ("in" < Event-Time-Window>)?
                     | "E-COUNT" <Event-Expression> <Number> ("in" < Event-Time-Window>)?
                     | "E-NOT" <Event-Expression> ("in" < Event-Time-Window>)?
                     | "(" <Event-Expression> ")"

<Logical-Expression> ::= <Logical-Term> { "||" <Logical-Term> }

<Logical-Term> ::= <Logical-Factor> { "&&" <Logical-Factor> }

<Logical-Factor> ::= "!" <Logical-Factor>
                   | <Equality-Expression>
                   | <Simple-Reference>
                   | <Boolean-Constant>          // "TRUE" | "FALSE"
                   | <Date-Time-Instance>        // e.g.,"2006/10/10 10:10:10"
                   | "(" <Logical-Expression> ")"

<Equality-Expression> ::= <Relational-Expression>{ ("==" | "!=") <Relational-Expression> }
                        | <Mathematical-Expression> ("==" | "!=") <Mathematical-Expression>

<Relational-Expression> ::= <Mathematical-Expression>("<"|"<="|">"|">=")<Mathematical-Expression>

<Mathematical-Expression> ::= <Mathematical-Term> { ( "+" | "-" ) <Mathematical-Term> }

<Mathematical-Term> ::= <Mathematical-Factor> { ( "*" | "/" ) <Mathematical-Factor> }

<Mathematical-Factor> ::= "-" <Mathematical-Factor>
                        | <Simple-Reference>
                        | <Simple-Constant>
                        | "(" <Mathematical-Expression> ")"

<Simple-Reference> ::= <Attribute-Identification>
                     | <Function-Call-Expression>

<Function-Call-Expression> ::= <Function-Identification> "(" [<Parameter-List>] ")"

<Parameter-List> ::= <Parameter-Expression> {"," <Parameter-Expression>}

<Interval> ::= <Start-Date-Time> "to" <End-Date-Time> // e.g.,"2006/10/10 00:00:00
               <Valid-Month-Number-List>?
               <Valid-Day-Number-List>?
               <Valid-Day-Name-List>?
               {<Valid-Time-Interval>}

<Valid-Time-Interval> ::= <From-Time> "to" <To-Time>   // Time instance, e.g., 10:10:00am

< Event-Time-Window> ::= Within n seconds from a specific <Point-of-Time>
                         // n could be +ve or –ve. Specific left to determines by implementation algorithms
```

Appendix B: The Example Policies

Policy 1: ***cpu_Usage***

if *cpu load is over 90% for any of the UNIX machines in the Systems lab*

then *email administrator*

Policy Elements		**Description**
Event	Statement	cpu load is over 90%
	Type	Primitive
	Logical Expression	cpuload>90
Rule	Condition	Always true
	Action Description	email administrator
Attributes	cpuload	This represents the cpu load used
Domain		This policy applies to any of the UNIX machines in the Systems Lab. This will be denoted by *Syslab UNIX Hosts*
Possible Monitoring Agent(s)		cpu_agent
Validation Achieved		The deployment of this policy validates that a single management agent, i.e., a monitoring agent, can enforce such policy.

Policy 2: ***memory_Usage***

if *memory usage is over 95% for any of the Systems lab machines*

then *email administrator*

Policy Elements		**Description**
Event	Statement	memory usage is over 95%
	Type	Primitive
	Logical Expression	memoryused >95
Rule	Condition	Always true
	Action Description	email administrator
Attributes	memoryused	This represents memory usage
Domain		All Syslab Hosts
Possible Monitoring Agent(s)		memory_agent
Validation Achieved		• The deployment of this policy validates that a single management agent can enforce such policy. • The domain would include Unix hosts systems and Windows systems. • The deployment of this policy validates that our model can be implemented and deployed in a heterogeneous distributed systems.

Policy 3: ***session_Control***

if *a login session is idle for more than 20 minutes for any of the UNIX System lab hosts*

then *close the session*

Policy Elements		**Description**
Event	Statement	a login session is idle for more than 20 minutes
	Type	Primitive
	Logical Expression	sessionidlelong>20
Rule	Condition	Always true
	Action Description	close the session
Attributes	sessionidlelong	This represents time elapsed (idle) since the user's last activity
	processid	This represents the process identifier
Domain		Syslab UNIX Hosts
Possible Monitoring Agent(s)		session_agent
Validation Achieved		• The extracted attributes represent; one attribute sessionidlelong which specified in the logical expression and another attribute processid that extracted from attributes defined in the action parameter. • The deployment of this policy validates that the event notification message should include attributes that is used to carry actions. • The action executable objects can be distributed with agents. The action can also be called from remote managed resources and executed locally in these managed resources.

Policy 4: ***root_Access_Monitor***

if *su root successfully used by user Nasser to any of the UNIX System lab machines*

then *email administrator*

Policy Elements		**Description**
Event	Statement	su root successfully used by user Nasser
	Type	Primitive
	Logical Expression	suroot.equals("true") && userid.equals("nasser")
Rule	Condition	Always true
	Action Description	email administrator
Attributes	userid	This represents the user identifier that issued the su root
	suroot	This is the boolean attribute that indicates that a su root occurred and suroot is true when successful or false on fail of su root.
Domain		Syslab UNIX Hosts
Possible Monitoring Agent(s)		syslogTrapper_agent
Validation Achieved		• The deployment of this policy validates that a single management agent can enforce such policy. • The management agent used shows that our model does make use of the monitoring information that already collected by other monitoring information services, i.e., the UNIX *syslog* in this policy.

Policy 5: ***email_Monitor***

if *a user uses quarrel.syslab.csd.uwo.ca to send email*

then *email administrator*

Policy Elements		**Description**
Event	Statement	a user uses quarrel.syslab.csd.uwo.ca to send email
	Type	Primitive
	Logical Expression	sendemail.equals("true")
Rule	Condition	Always true
	Action Description	email administrator
Attributes	fromuserid	This is the user identifier of the user that sent the email
	sendemail	This is a boolean attribute that indicates an email sent, *sendemail* is true when message accepted for delivery or false otherwise
Domain		Host quarrel
Possible Monitoring Agent(s)		syslogTrapper_agent
Validation Achieved		• The deployment of this policy validates the reuse of already existing management agent instance that used in Policy example 4. • The domain consists of only one host. • The reconfiguration of the agent instance would be hold only to the instance that serves the host quarrel.

Policy 6: ***process_Control***

if *a user is running a Sudoku program Sudoku from any of the UNIX System lab hosts*

then *stop this program and email administrator*

Policy Elements		**Description**
Event	Statement	a user running a sudoku program (which is a game).
	Type	Primitive
	Logical Expression	processcommand.equals("sudoku")
Rule	Condition	Always true
	Actions Descriptions	1. Stop this program
		2. email administrator
Attributes	processcommand	The command used
	processid	The process identification
	userid	The process own user identification
Domain		Syslab UNIX Hosts
Possible Monitoring Agent(s)		process_agent
Validation Achieved		The deployment of this policy validates the enforcement of more than one action within one rule, i.e., several actions to enforce when a specific event triggered.

Policy 7: ***process_Monitor***

if *any process navigating Internet has a size over 10240K from any of the UNIX System lab hosts*

then *email administrator*

and *if* the process owner is *Nasser then* kill this process

Policy Elements		**Description**
Event	Statement	any process navigating Internet has a size over 10240K
	Type	Primitive
	Logical Expression	(processcommand.contains("netscape") || processcommand.contains("explorer") || processcommand.contains("foxpror")) && (processsize >= 10240)
Rule1	Condition	Always true
	Actions Descriptions	email administrator
Rule2	Condition	userid.equals("Nasser")
	Actions Descriptions	kill this process
Attributes	processcommand	The command used
	processid	The process identification
	processsize	The process size in KB
	userid	The process own user identification
Domain		Syslab UNIX Hosts
Possible Monitoring Agent(s)		process_agent and processWithCommand_agent
Validation Achieved		• The deployment of this policy validates the reuse of already existing management agent instance that used in Policy example 6. • The use of dynamic monitoring agent that communicates with two montoring agents. • A policy that has more than one policy rule. • A bit complicated logical expression.

Policy 8: ***hd_monitor***

if *any file system is filled to over 89% for any of the UNIX System lab hosts*

then *email administrator*

Policy Elements		**Description**
Event	Statement	any file system is at 89% capacity
	Type	Primitive
	Logical Expression	hdcapacity >89
Rule	Condition	Always true
	Action Description	email administrator
Attributes	hdname	The file system name
	hdsizecapacity	The percentage used from the file system
Domain		Syslab UNIX Hosts
Possible Monitoring Agent(s)		hd_agent
Validation Achieved		The use of different management agent that can monitor more other attributes of the system state.

Policy 9: ***db2_Control***

if *the DB2 database engine is down on wolfbiter*

then *start the DB2 database*

Policy Elements		**Description**
Event	Statement	The DB2 database engine is down
	Type	Primitive
	Logical Expression	db2state.equals("stop")
Rule	Condition	Always true
	Action Description	Start the DB2 database
Attributes	db2state	This represents the state of DB2. The state of DB2 is either that it has started or is down.
Domain		Host wolfbiter
Possible Monitoring Agent(s)		db2_agent
Validation Achieved		The use of different management agent that can monitor and control software application.

Policy 10: ***db2_Monitor***

if *the number of active connections to DB2 database exceeds 5*

then *email administrator*

Policy Elements		**Description**
Event	Statement	the current application's connections to DB2 database exceeds 5
	Type	Primitive
	Logical Expression	db2activeconnections>5
Rule	Condition	Always true
	Action Description	email administrator
Attributes	db2activeconnections	An attribute indicates the number of active connections to the DB2.
Domain		Host wolfbiter
Possible Monitoring Agent(s)		db2_agent
Validation Achieved		The reuse of management agent that can monitor and control software application.

Policy 11: ***system_Defaults***

if *the maximum number of processes allowed in wolfbiter is less 1000*

then *email administrator*

Policy Elements		**Description**
Event	Statement	The maximumnumber of processes allowed is less 1000
	Type	Primitive
	Logical Expression	maxprocessesdefined<1000
Rule	Condition	Always true
	Action Description	email administrator
Attributes	maxprocessesdefined	The defined maximum number of processes that can initiated in a system
Domain		Host wolfbiter
Possible Monitoring Agent(s)		handleSNMP_agent
Validation Achieved		The use of management agent that can get the management information via the SNMP information services.

Policy 12: ***printer_Monitor***

if *the spool queue has more than 30 jobs to print*

then *email administrator*

Policy Elements		**Description**
Event	Statement	the spool queue has more than 30 jobs to Print
	Type	Primitive
	Logical Expression	spooljobstotal>30
Rule	Condition	Always true
	Action Description	email administrator
Attributes	printername	The name of the printer
	spooljobstotal	The number of jobs in the spool queue
Domain		Syslab UNIX Hosts
Possible Monitoring Agent(s)		printer_agent
Validation Achieved		The use of different management agent that can monitor more other attributes of the system state, i.e., printer in this policy.

Policy 13: ***printer_Control***

if *the mandas printed failed*

then *email administrator*

Policy Elements		**Description**
Event	Statement	the mandas printer failed
	Type	Primitive
	Logical Expression	printerstat.equals("failed") && printername.equals("mandas")
Rule	Condition	Always true
	Action Description	email administrator
Attributes	printername	The name of the printer
	printerstate	The printer status
Domain		Syslab UNIX Hosts
Possible Monitoring Agent(s)		printer_agent
Validation Achieved		The reuse of management agent that can monitor and control system state.

Policy 14: ***net_Monitor***

if *the number of packet errors is greater than 10 packets for wolfbiter*

then *email administrator*

Policy Elements		**Description**
Event	Statement	the number of packet errors greater than 10 packet
	Type	Primitive
	Logical Expression	inputpacketerror>10 || outputpacketerror>10
Rule	Condition	Always true
	Action Description	email administrator
Attributes	inputpacketerror	The number of the input error packets
	outputpacketerror	The number of the output error packets
	ipaddress	The IP address
Domain		Host wolfbiter
Possible Monitoring Agent(s)		net_agent
Validation Achieved		The use of different management agent that can monitor more other attributes of the network state.

Policy 15: ***performac_Issue***

if *the total number of processes on wolfbiter is greater than 100*

then *change the processes priority owned by the user Nasser one degree*

Policy Elements		**Description**
Event	Statement	the total number of processes on wolfbiter is greater than 100
	Type	Primitive
	Logical Expression	cpuprocesstotal>100
Rule	Condition	userid.equals("Nasser")
	Action Description	change the processes priority
Attributes	cpuprocesstotal	The total number of initiating processes
	userid	The process own user identification
	processid	The process identifier
	pocessspriority	The process priority
Domain		Host wolfbiter
Possible Monitoring Agent(s)		cpu_agent and process_agent
Validation Achieved		The use of different management agent that can monitor and carry actions.

Policy 16: ***load_Control***

if *the total number of user logins is greater than 5 for any of the UNIX System lab hosts*
followed by *the CPU load is greater than 90 and the total number of processes running is greater than 35*
then *block any new user logins*

Policy Elements			**Description**
Event	Statement		the total number of user logins is greater than 5 followed by the CPU load is greater than 90 and the total number of processes running is greater than 35
	Type		Composite
	Logical Expression	1	usersloginstotal>5
		2	cpuload>90 && cpuprocesstotal>35
	Event Expression		Event characterized by logical expression in 1 E-SEQ Event characterized by logical expression in 2
Rule	Condition		Always true
	Action Description		block any new user logins
Attributes	usersloginstotal		The total number of current logins users sessions
	cpuload		The cpu load
	cpuprocesstotal		The total number of initiating processes
Domain			Syslab hosts
Possible Monitoring Agent(s)			session_agent and cpu_agent
Validation Achieved			• The configuration of the event format files. The format file is understood by a management system event handler TEC. • The ability to automatically map a policy to TEC rules, i.e., to a configuration of a management system to enforce the policy. • The creation and use of mapping templates.

Policy 17: ***tivoli_Monitor***

if *the Tivoli TEC event server is down on wolfbiter*

then

if the DB2 is down then start DB2

if DB2 is started then start the TEC event server

Policy Elements		**Description**
Event	Statement	The Tivoli TEC event server is down on wolfbiter
	Type	Primitive
	Logical Expression	teceventserverstat.equals("stop")
Rule 1	Condition	db2state.equals("stop")
	Action Description	Start the DB2 database
Rule 2	Condition	db2state.equals("start")
	Action Description	Start the TEC event server
Attributes	teceventserverstat	An attribute indicates that the state of the TEC event server. It has the value of i.e., stop when the TEC event server is down or not started and has the value start when the event server is already started
	db2state	An attribute indicates that the state of the data base engine. It has the value of i.e., stop when the DB engine is down or not started and has the value start when the DB engine is already started or active
Domain		Host wolfbiter
Possible Monitoring Agent(s)		tivoli_agent , db2_agent
Validation Achieved		• The deployment of this policy validates the reuse of already existing management agent instance. • The policy has more than one rule. • Each rule includes a condition that need to be verified before enforcing the action. • The distribution of agents that monitor the events as well as the agent that monitor the conditions.

Policy 18: ***access_Monitor***

if *the number of failed login attempts under a specific login name exceeds 3 for any of the UNIX System lab hosts*

then *email administrator*

Policy Elements		**Description**
Event	Statement	the number of failed login attempts under a specific login name exceeds 3
	Pattern	Composite
	Logical Expression	userlogin.equals("fail")
	Event Expression	E-COUNT Event characterized by logical expression 3 times
Rule	Condition	Always true
	Action Description	Email administrator
Attributes	usersid	The user identification
	userlogin	An attribute indicates the state of the user login, i.e., 'success' when the user successfully logins or 'fail' otherwise.
Domain		Syslab UNIX Hosts
Possible Monitoring Agent(s)		syslogTrapper_agent
Validation Achieved		• The ability to automatically create a manager agent that can detect the composite event .The manager agent represents also the mapping of the policy. • The use of other event operator, E-COUNT.

Appendix C: The Implemented Monitoring Agents

Agent	Attribute Name	Attribute Description
session_agent This agent monitors information about open user sessions. ***Can Support Policies:*** 3 and 17	userid	User's login name
	sessionstate	The capability of writing to the terminal
	terminalid	The name of the line found in /dev
	processid	The user's process id
	sessiontime	The time since user's login
	sessionidlelong	The time elapsed (idle) since the user's last activity
	sessioncomments	The session comments
cpu_agent This agent monitorwsand collects the information about UNIX CPU usage information. ***Can Support Policies:*** 1, 15 and 17	cpuloaduser	The percentage of CPU time in user mode
	cpuidle	The percentage of CPU time in idle mode
	cpuloadkernel	The percentage of CPU time in kernel mode
	cpuiowait	The percentage of CPU time in iowait mode
	cpuswap	The percentage of CPU time in swap mode
	cpuprocessloadaverages	The average of CPU/processes,
	cputotalprocesses	The total number processes,
	cpuprocesssleeping	The total number sleeping processes,
	cpuprocessrunning	The total number running processes,
hd_agent This agent monitors information about UNIX filesystems. ***Can Support Policies:*** 8	hdname	The filesystem name
	hdmountedon	The filesystem mounted on drive
	hdsize	The filesystem total size
	hdsizeused	The filesystem size used in KB
	hdsizeavaliable	The filesystem size available in KB
	hdsizecapacity	The filesystem size used percentage
memory_agent This agent monitors host memory . (There is another agent, ***memory_agent_WIN*** for Windows) ***Can Support Policies:*** 2	memoryrealtotal	The total real system memory
	memoryused	The total real system memory used.
	memoryfree	The total real system memory free
	memoryswapused	The total swap memory used.
	memoryswapfree	The total swap memory free.

Agent	Attribute Name	Attribute Description
process_agent This agent monitors information about UNIX processes . ***Can Support Policies:*** 6, 7 and 15	processid	The process task id
	userid	The process task user name
	processthreadso	The number of execution threads in the process
	pocessspriority	The process task priority
	processnice	If it be run with a different system scheduling priority, -ve nice values are higher priority
	processsize	The size of the task's code plus data plus stack space, in KB
	pocessresident	The total amount of physical memory used by the task, in KB
	processtime	Total CPU time the task has used since it started
	processcpuusage	Total CPU percentage used
	processstate	The state of the task is either sleep or cpu running.
	processcommandclass	The class of the used command
processWithCommand_agent This agent monitors information about UNIX processes with detail information about the command used in the process . ***Can Support Policies:*** 6, 7 and 15	processid	The process task id
	userid	The process task user name
	processthreadso	The number of execution threads in the process
	processsize	The size of the task's code plus data plus stack space, in KB
	terminalid	The terminal id from where this process was issued.
	processstate	The state of the task is either S or R or O or T.
	processcpupercentage	The % percentage this process uses from the CPU
	processmemprecentage	The % percentage this process uses from memory.
	processstarttime	The time when this process was started
	processcommand	The full command that used in the process

Agent	Attribute Name	Attribute Description
Db2_agent This agent monitors DB2 . ***Can Support Policies:*** 9, 10 and 16	db2state	An attribute indicates the state of the DB, i.e., stop when the DB2 down or not started and start when the DB2 is already active
	db2activeconnections	An attribute indicates the number of active connections to the DB2.
syslogTrapper_agent This agent parses newly added log lines to the syslog file. ***Can Support Policies:*** 4, 5 and 18	suroot	A Boolean attribute indicates a su root occurred and suroot is true when successful or false on fail of su root.
	userid	The user identification
	fromuserid	The user identification that sent the email
	sendemail	A Boolean attribute indicates an email sent and sendemail is true when message accepted for delivery or false otherwise
	userlogin	An attribute indicates the state of the user's login, i.e., 'success' when the user successfully logins or 'fail' otherwise.
tivoli_agent This agent monitors Tivoli ***Can Support Policies:*** 16	teceventserverstat	An attribute indicates the state of the TEC event server, i.e., stop when it is down or not started and start when the event server is already started
net_agent This agent monitors and collects information about the system network communications. ***Can Support Policies:*** 14	interfacename	Name of the interface used for IP traffic, e.g. interface, host, network and default routers.
	ipaddress	The local IP address
	netipaddress	The net/distinction IP address
	inputpacket	The number of the input packets
	inputpacketerror	The number of the input error packets
	outputpacket	The number of the output packets
	outputpacketerror	The number of the output error packets

Agent	Attribute Name	Attribute Description
printer_agent This agent monitors and collects the information about the printers. ***Can Support Policies:*** 12 and 13	printername	The name of the printer
	spooljobstotal	The number of jobs in the spool queue
	printerstate	The printer status
handleSNMP_agent This agent monitors and collects the information about the system by querying the SNMP information services to get the required attributes, principally, the Host-Resources-MIB. ***Can Support Policies:*** 11	maxprocessesdefined	The defined maximum number of processes that can initiated in a system

Appendix D: Tivoli TEC Rule Templates

A Tivoli TEC Rule Template for Interval Checking

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This TEC rule is to validate the policy 'Policy-Name-Variable' which has
% the event 'Event-ID-Variable' and the interval 'Interval-Name-Variable'
%   Automatically generated by the PMagic Model
%   On  System Date-Time at Template Instantiation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   First Rule is to validate the policy interval 'Interval-Name-Variable'

rule: 'Policy-Name-Variable_Interval-Name-Variable':
(
  description: 'Verify the policy interval Interval-Name-Variable',

% Following set represents events specified in policy 'Policy-Name-Variable'
  event: _ev_at_interval_check of_class within

                                  [ [ Event-ID-Variable ,]* ]
         where [ ]  ,

  reception_action:
  action_Policy-Name-Variable_Interval-Name-Variable_check:
       (
         exec_program(_ev_at_interval_check,'Program-Call-Method-Variable'

                   ,'%s %s'
                   , ['Interval-Name-Variable','Result-File-Name-Variable']
                   , 'YES') ,
         fopen(_fp
         ,'Result-File-Name-Variable/Interval-Name-Variable_result.txt'
         , r) ,
              readln(_fp, _result),
         fclose(_fp) ,
         ( _result == true ,
             commit_action
               % exit this action and continue the reset of the rule
           ;   % else
             commit_rule
               % exit the whole rule at this point
         )
       )
).
% End of the rule: 'Policy-Name-Variable_Interval-Name-Variable'
```

A Tivoli TEC Rule Template for Rule Enforcement

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This TEC rule is to enforce the specified rule if the event
% 'Event-ID-Variable',   which considers the main event of the policy
% 'Policy-Name-Variable', triggered.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rule: 'Policy-Name-Variable_Rule_Enforcement':
(
  description: 'Fire the rule(s)of the policy Policy-Name-Variable' ,

  event: _ev_rule_main of_class 'Event-ID-Variable'
         where [date_reception:  _ev_date_reception ,
                server_handle:   _ev_server_handle  ,
                event_handle:    _ev_event_handle   ,
                hostname:        _ev_hostname       ,
                sub_source:      _ev_sub_source     ,
                sub_source_port: _ev_sub_source_port ] ,

  reception_action:
  action_Policy-Name-Variable_Event-ID-Variable_enforce_rule:
       (
           exec_program(_ev_rule_main , ' Program-Call-Method-Variable'

           ,'%s %s %s %s %d %ld %d %d'
           , ['Policy-Name-Variable','Event-ID-Variable' , _ev_hostname
           , _ev_sub_source , _ev_sub_source_port , _ev_date_reception
           , _ev_server_handle  , _ev_event_handle ], 'YES') ,

           commit_rule

       )
).
% End of the rule: 'Policy-Name-Variable_Rule_Enforcement'
```

A Tivoli TEC Rule Template for Composite Event Detection of *E_SEQ* Operator

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This TEC rule is to generate the event 'Event-ID-Variable' which
% occurs when the event 'Left-Event-ID-Variable'
%         and then event 'Right-Event-ID-Variable' occurred in SEQUENCE.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rule: 'Left-Event-ID-Variable_E_SEQ_Right-Event-ID-Variable':
(
  description: 'Generate event Event-ID-Variable',

  event: _ev1_at_ESEQ of_class 'Left-Event-ID-Variable'
         where [date_reception: _left_date_reception,
                server_handle:  _left_server_handle,
                event_handle:   _left_event_handle]  ,

% The E_SEQ rule generates the result as a new event 'Event-ID-Variable'
% by using the exec_program that calls an external program.

  reception_action:
  action_Left-Event-ID-Variable_E_SEQ_Right-Event-ID-Variable:
       (
           first_instance(event: _ev2_at_ESEQ
                                 of_class 'Right-Event-ID-Variable'
              where [date_reception: _right_date_reception
                                     greater_than _left_date_reception,
                     server_handle:  _right_server_handle,
                     event_handle:   _right_event_handle ] ,

                _ev1_at_ESEQ - 0 - Max-Seconds-From-LHS-Event-Variable ) ,
% The time window for searching the _ev2_at_ESEQ is surrounding by the
% Max-Seconds-From-LHS-Event-Variable seconds after the _ev1_at_ESEQ time

           exec_program(_ev2_at_ESEQ, 'EventGeneration-Program-Variable'

           ,'%s %s %s %s %ld %d %d %s %ld %d %d %d'
           , ['Event-ID-Variable', 'createESEQRule' , 'E_SEQ'
           , 'Left-Event-ID-Variable', _left_date_reception
           , _left_server_handle  , _left_event_handle
           , 'Right-Event-ID-Variable', _right_date_reception
           , _right_server_handle , _right_event_handle, 0 ], 'YES') ,

           commit_action % exit the action regarding the scanned events
       )
).
% End of the rule: 'Left-Event-ID-Variable_E_SEQ_Right-Event-ID-Variable'
```

A Tivoli TEC Rule Template for Composite Event Detection of *E_AND* Operator

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This TEC rule is to generate the event 'Event-ID-Variable' which occurs when
% both events 'Left-Event-ID-Variable' and event'Right-Event-ID-Variable'occurred.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
rule: 'Left-Event-ID-Variable_E_AND_Right-Event-ID-Variable':
(
  description: 'Generate event 'Event-ID-Variable',
  event: _ev1_at_EAND of_class within ['Left-Event-ID-Variable'
                                      , 'Right-Event-ID-Variable' ]
         where [date_reception: _ev1_date_reception ,
                server_handle:  _ev1_server_handle  ,
                event_handle:   _ev1_event_handle]  ,
% The E_AND rule generates the result as a new event 'Event-ID-Variable'
% by using the exec_program that calls an external program.
  reception_action:
  action_Left-Event-ID-Variable_E_AND_Right-Event-ID-Variable':
       (    bo_get_class_of(_ev1_at_EAND, _ev1_name) ,
            ( _ev1_name == 'Left-Event-ID-Variable',  % if
                _ev2_name = 'Right-Event-ID-Variable'
              ; _ev2_name = 'Left-Event-ID-Variable'     % else
            ) ,
           first_instance(event: _ev2_at_EAND of_class _ev2_name
            where [date_reception: _ev2_date_reception ,
                   server_handle:  _ev2_server_handle  ,
                   event_handle:   _ev2_event_handle]  ,
           _ev1_at_EAND -
           Min-Seconds-From-Event-Variable - Min-Seconds-From-Event-Variable ),
% The time window for searching the _ev2_at_EAND is surrounding by the
% Min-Seconds-From-Event-Variable seconds before _ev1_at_EAND time and by the
% Max-Seconds-From-Event-Variable seconds after ev1_at_EAND time
            ( _ev1_name == 'Event-ID-Variable', % if
                _left_date_reception  = _ev1_date_reception ,
                _left_server_handle   = _ev1_server_handle ,
                _left_event_handle    = _ev1_event_handle ,
                _right_date_reception = _ev2_date_reception ,
                _right_server_handle  = _ev2_server_handle ,
                _right_event_handle   = _ev2_event_handle
              ;  % else
                _left_date_reception  = _ev2_date_reception ,
                _left_server_handle   = _ev2_server_handle ,
                _left_event_handle    = _ev2_event_handle ,
                _right_date_reception = _ev1_date_reception ,
                _right_server_handle  = _ev1_server_handle ,
                _right_event_handle   = _ev1_event_handle
            ) ,
           exec_program(_ev2_at_EAND, 'EventGeneration-Program-Variable'
           ,'%s %s %s %s %ld %d %d %s %ld %d %d %d %d'
           , ['Event-ID-Variable', 'createEANDRule' , 'E_AND'
           , 'Left-Event-ID-Variable', _left_date_reception
           , _left_server_handle  , _left_event_handle
           , 'Right-Event-ID-Variable', _right_date_reception
           , _right_server_handle , _right_event_handle, 0 , 0], 'YES') ,

           commit_action % exit the action regarding the scanned events
       )
).
% End of the rule: 'Left-Event-ID-Variable_E_AND_Right-Event-ID-Variable'
```

A Tivoli TEC Rule Template for Composite Event Detection of *E_OR* Operator

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This TEC rule is to generate the event 'Event-ID-Variable' which
% occurs when either event 'Left-Event-ID-Variable'
%                  or event 'Right-Event-ID-Variable' occurred.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rule: 'Left-Event-ID-Variable_E_OR_Right-Event-ID-Variable':
(
  description: 'Generate event Event-ID-Variable',

  event: _ev1_at_EOR of_class within ['Left-Event-ID-Variable'
                                    , 'Right-Event-ID-Variable' ]
         where [date_reception: _ev1_date_reception ,
                server_handle:  _ev1_server_handle  ,
                event_handle:   _ev1_event_handle]  ,

% The E_OR rule generates the result as a new event 'Event-ID-Variable'
% by using the exec_program that calls an external program.

  reception_action:
  action_Left-Event-ID-Variable_E_OR_Right-Event-ID-Variable':
       (
            bo_get_class_of(_ev1_at_EOR, _ev1_name) ,
           (  _ev1_name == 'Left Event-ID Variable' , % if
                 _left_date_reception  = _ev1_date_reception ,
                 _left_server_handle   = _ev1_server_handle  ,
                 _left_event_handle    = _ev1_event_handle   ,
                 _right_date_reception = 0x0 ,
                 _right_server_handle  = 0   ,
                 _right_event_handle   = 0   ,
                 _count_number         = 0
             ;  % else
                 _left_date_reception  = 0x0 ,
                 _left_server_handle   = 0   ,
                 _left_event_handle    = 0   ,
                 _right_date_reception = _ev1_date_reception ,
                 _right_server_handle  = _ev1_server_handle  ,
                 _right_event_handle   = _ev1_event_handle   ,
                 _count_number         = 0
           ) , % end if

           exec_program(_ev1_at_EOR, 'EventGeneration-Program-Variable'

           ,'%s %s %s %s %ld %d %d %s %ld %d %d %d %d'
           , ['Event-ID-Variable', 'createEORRule' , 'E_OR'
           , 'Left-Event-ID-Variable', _left_date_reception
           , _left_server_handle  , _left_event_handle
           , 'Right-Event-ID-Variable', _right_date_reception
           , _right_server_handle , _right_event_handle, 0, 0], 'YES') ,

          commit_action % exit the action regarding the scanned event

       )
).
% End of the rule: 'Left-Event-ID-Variable_E_OR_Right-Event-ID-Variable'
```

A Tivoli TEC Rule Template for Composite Event Detection of *E_COUNT* Operator

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This TEC rule is to generate the event 'Event-ID-Variable' which occurs
% when the event 'Right-Event-ID-Variable' is repeated n-Variable times.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
rule: 'E_COUNT_Right-Event-ID-Variable_ n-Variable_check':
(
  description: 'Check for repeated event Event-ID Variable' ,
  event: _ev_at_ECOUNT of_class 'Right-Event-ID-Variable'
         where [date_reception: _left_date_reception,
                server_handle:  _left_server_handle,
                event_handle:   _left_event_handle ,
                    [Event-Variable-To-Match,]* ],
  reception_action:
  action_E_COUNT_Right-Event-ID-Variable_n-Variable_add:
       (
           first_duplicate(_ev_at_ECOUNT, event: _ev_d_at_ECOUNT
            where [status: outside ['CLOSED'] ,
                   [Event-Variable-To-Match-Comparison,]* ],
            _ev_at_ECOUNT -
           Min-Seconds-From-Event-Variable - Max-Seconds-From-Event-Variable ),
% The time window for searching instances of _ev_at_ECOUNT is surrounding by
% Min-Seconds-From-Event-Variable seconds before last instance _ev_at_ECOUNT
% time and by Max-Seconds-From-Event-Variable seconds after this instance time

           add_to_repeat_count(_ev_d_at_ECOUNT , 1) ,
           drop_received_event ,
           commit_action
       )
).
% End of the rule: 'E_COUNT_Right-Event-ID-Variable_n-Variable'
% The E_COUNT rule generates the result as new event 'Event-ID-Variable'
% by using the exec_program that calls an external program.
rule: 'E_COUNT_Right-Event-ID-Variable_n-Variable':
(
  description: 'Generate event Event-ID-Variable',
  event: _ev_at_ECOUNT of_class 'Right-Event-ID-Variable'
          where [ date_reception: _right_date_reception ,
                  server_handle:  _right_server_handle ,
                  event_handle:   _right_event_handle ] ,
  reception_action:
  action_E_COUNT_ Right-Event-ID-Variable_n-Variable_limit_held:
       (   _count is n-Variable - 1 ,
           first_duplicate(_ev_at_ECOUNT, event: _ev_d_at_ECOUNT
              where [repeat_count: greater_or_equals _count ] ) ,
           exec_program(_ev_d_at_ECOUNT, 'EventGeneration-Program-Variable'
           ,'%s %s %s %s %ld %d %d %s %ld %d %d %d %d'
           , ['Event-ID-Variable', 'createECOUNTRule' , 'E_COUNT'
           , 'Right-Event-ID-Variable', _right_date_reception
           , _right_server_handle  , _right_event_handle
           , 'none' , 0x0 , 0 , 0 , n-Variable , 0], 'YES') ,
           drop_received_event ,
           commit_action
       )
).
% End of the rule: 'E_COUNT_ Right-Event-ID-Variable_n-Variable'
```

A Tivoli TEC Rule Template for Composite Event Detection of *E_NOT* Operator

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This TEC rule is to generate the event 'Event-ID-Variable' which occurs
% when the event 'Right-Event-ID-Variable'
% is NOT occurred between From-Time-Variable and To-Time-Variable period
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rule: 'E_NOT_Right-Event-ID-Variable_check':
(
  description: 'Check for occurrence of event Event-ID-Variable',

  event: _ev_any of_class _any_class
         where [date_reception: _right_date_reception ,
                server_handle:  _right_server_handle,
                event_handle:   _right_event_handle]  ,

  reception_action:
  action_E_NOT_Right-Event-ID-Variable_search:
       (
         pointertoatom(_fromtime, From-Time-Variable) ,
         pointertoatom(_totime, To-Time-Variable) ,
         _right_date_reception >= _totime ,
         (
            first_instance(event: _ev_at_ENOT
                                  of_class 'Right-Event-ID-Variable'
              where [date_reception: _ev2_date_reception
                                     greater_or_equals _fromtime ,
                     date_reception: _ev2_date_reception
                                     smaller_or_equals _totime ] ) ,
              commit_action
            ; % else
% The E_NOT rule generates the result as a new event 'Event-ID-Variable'
% by using the exec_program that calls an external program.

            exec_program(_ev_any, 'EventGeneration-Program-Variable'

            ,'%s %s %s %s %ld %d %d %s %ld %d %d %d %d'
            , ['Event-ID-Variable', 'createENOTRule' , 'E_NOT'
            , '' , 0 , 0 , 0
            , 'Right-Event-ID-Variable', _totime , _right_server_handle
            , _right_event_handle, 0, 0], 'YES') ,

            commit_action
         )
       )
).
% End of the rule: 'E_NOT_Right-Event-ID-Variable'
```

Appendix E: Algorithms Used to Implement Event Operators

The concepts of the algorithms in this Appendix are built using the semantics of the event operators that introduced in Chapter 3. More in composite event detections can be found in [56].

Algorithm E_SEQ is the algorithm used to implement the detection of an event that characterized by an event expression uses E_SEQ event operator. []

Algorithm E_SEQ (E , E_{LHS}_Name , E_{RHS}_Name , T_{min} , T_{max} , CloseUsedInstance)

Input:

1) **E** is an event that needs to detect an instance of.
2) E_{LHS}_Name is the event name of the LHS of the event expression
3) E_{RHS}_Name is the event name of the RHS of the event expression
4) T_{min} is the min time in seconds between the occurrence of events instances of E_{LHS} and E_{RHS}
5) T_{max} is the max time in seconds between the occurrence of events instances of E_{LHS} and E_{RHS}
6) **CloseUsedInstance** is a Boolean value to specify to close the events instances after process or not

Output: 1) **e** is an instance E, which is the correlation of event instances e_{LHS} E_SEQ e_{RHS}

```
-- EventMemory is a list of the event notification messages received, the list is assumed to be ordered by the newest received messages.

1.  Events_LHS = SubList (EventMemory , E_LHS_Name )
        -- SubList is a function that returns a sub-list from the EventMemory events list. In this case, the returned sub-list events are of event type E_LHS.
2.  if ( Events_LHS ≠ ∅ ) then
3.      for each event instance e_LHS ∈ Events_LHS do
4.          e_RHS = FindEventInstance (EventMemory , E_RHS_Name , e_LHS.timeStamp , T_min , T_max )
                    -- FindEventInstance is a function that returns a specific event instance from the EventMemory list. In the above initialization of FindEventInstance, the returned event instance must be of type E_RHS and satisfies the time constrains that delimited by e_LHS.timeStamp , T_min , T_max .
5.          if (e_RHS ≠ null ) then
6.              e = GenerateEventInstance( E , e_LHS , e_RHS , e_RHS.timeStamp)
                            -- GenerateEventInstace is a function that creates an event notification message (event instance) of a given type, which is in this case of type E. The other parameters are to help constructing the message information, i.e., the event instance attributes name/value pairs.
7.              if (CloseUsedInstance ) then
8.                  MarkClose(e_LHS)
9.                  MarkClose(e_RHS)
10.             end if
11.             return ( e )
12.         end if
13.     end for
14. end if
15. return ( null )

exit algorithm
```

Algorithm E_AND is the algorithm used to implement the detection of an event that characterized by an event expression uses E_AND event operator.

Algorithm E_AND (E , E_{LHS}_Name , E_{RHS}_Name , T_{min} , T_{max} , CloseUsedInstance)
Input:

1) **E** is an event that needs to detect an instance of.
2) E_{LHS}_Name is the event name of the LHS of the event expression
3) E_{RHS}_Name is the event name of the RHS of the event expression
4) T_{min} is the min time in seconds between the occurrence of events instances of E_{LHS} and E_{RHS}
5) T_{max} is the max time in seconds between the occurrence of events instances of E_{LHS} and E_{RHS}
6) **CloseUsedInstance** is a Boolean value to specify to close the events instances after process

Output: 1) **e** is an instance E, which is the correlation of event instances e_{LHS} E_AND e_{RHS}

---

1. $Events_{LHS}$ = SubList (EventMemory , E_{LHS}_Name)
   -- SubList is a function that returns a sub-list from the EventMemory events list. In this case, the returned sub-list events are of event type E_{LHS}.
2. **if** ($Events_{LHS} \neq \varnothing$) **then**
3. $Events_{RHS}$ = SubList (EventMemory , E_{RHS}_Name)
4. **if** ($Events_{RHS} \neq \varnothing$) **then**
5. **for** each event instance $e_{LHS} \in Events_{LHS}$ **do**
6. e_{RHS} = FindEventInstance ($Events_{RHS}$, E_{RHS}_Name, e_{LHS}.timeStamp , T_{min} , T_{max})
   -- FindEventInstance is a function that returns a specific event instance from the $Events_{RHS}$ sub-list. In the above initialization of FindEventInstance, the returned event instance must satisfy the time constrains that delimited by e_{LHS}.timeStamp , T_{min} , T_{max} .
7. **if** ($e_{RHS} \neq$ null) **then**
8. e=GenerateEventInstance(**E**,e_{LHS},e_{RHS},**max**(e_{RHS}.timeStamp,e_{RHS}.timeStamp))
   -- GenerateEventInstace is a function that creates an event notification message (event instance) of a given type, which is in this case of type E. The other parameters are to help constructing the message information, i.e., the event instance attributes name/value pairs.
9. **if** (CloseUsedInstance) **then**
10. MarkClose(e_{LHS})
11. MarkClose(e_{RHS})
12. **end if**
13. **return** (e)
14. **end if**
15. **end for**
16. **end if**
17. **end if**
18. **return** (null)

exit algorithm

Algorithm E_OR is the algorithm used to implement the detection of an event that characterized by an event expression uses E_OR event operator.

Algorithm E_OR (E , E_{LHS}_Name , E_{RHS}_Name , T_{min} , T_{max} , CloseUsedInstance)

Input:

1) **E** is an event that needs to detect an instance of.
2) E_{LHS}_Name is the event name of the LHS of the event expression
3) E_{RHS}_Name is the event name of the RHS of the event expression
4) T_{min} is the min time in seconds needed between the current time and either instances of E_{LHS} or E_{RHS}
5) T_{max} is the max time in seconds needed between the current time and either instances of E_{LHS} or E_{RHS}
6) **CloseUsedInstance** is a Boolean value to specify to close the events instances after process or not

Output: 1) **e** is an instance E, which is the correlation of event instances e_{LHS} E_OR e_{RHS}

```
1.  e_LHS = FindEventInstance (EventMemory , E_LHS_Name , , T_min , T_max )
2.  if (e_LHS ≠ null ) then
3.      e=GenerateEventInstance(E , e_LHS , , e_LHS.timeStamp)
            -- GenerateEventInstace is a function that creates an event notification message (event
            instance) of a given type, which is in this case of type E. The other parameters are to help
            constructing the message information, i.e., the event instance attributes name/value pairs.
4.      if (CloseUsedInstance ) then
5.          MarkClose(e_LHS)
6.      end if
7.      return ( e )
8.  else
9.      e_RHS = FindEventInstance (EventMemory , E_RHS_Name , , T_min , T_max )
10.     if (e_RHS ≠ null ) then
11.         e=GenerateEventInstance(E , e_RHS , , e_RHS.timeStamp)
12.         if (CloseUsedInstance ) then
13.             MarkClose(e_RHS)
14.         end if
15.         return ( e )
16.     end if
17. end if
18. return ( null )

exit algorithm
```

Algorithm E_NOT is the algorithm used to implement the detection of an event that characterized by an event expression uses E_NOT event operator.

Algorithm E_NOT (E , E_{RHS}_Name , T_{min} , T_{max} , CloseUsedInstance)

Input:

1) **E** is an event that needs to detect an instance of.
2) E_{RHS}_Name is the event name of the RHS of the event expression
3) T_{min} is the lower single point boundary timestamp to check events instances against
4) T_{max} is the upper single point boundary timestamp to check events instances against
5) **CloseUsedInstance** is a Boolean value to specify to close the events instances after process or not

Output: 1) **e** is an instance E, which is the correlation of event instances E_NOT e_{RHS}

```
1.  do Loop
2.      if (CurrentSystemTime.TimeStamp >= Tmim) then
3.          eRHS = FindEventInstance (EventMemory , ERHS_Name , , Tmin , Tmax )
4.          if ( eRHS == null && CurrentSystemTime.TimeStamp >= Tmax ) then
5.              e=GenerateEventInstance(E , eRHS , , Tmax)
6.              return ( e )
7.          else
8.              if (CurrentSystemTime.TimeStamp < Tmax) then
9.                  wait (WaitPeriod)
                        -- This is the WaidPeriod time defines for the manage agent, that is the time
                        between periodic evaluations of the expression(s) represented in the
                        EventRepresentation component.
10.             else
11.                 if (CloseUsedInstance ) then
12.                     MarkClose(eRHS)
13.                 end if
14.                 return ( null )
15.             end if
16.         end if
17.     else
18.         wait (WaitPeriod)
19.     end if
20. repeat Loop

exit algorithm
```

Algorithm E_COUNT is the algorithm used to implement the detection of an event that characterized by an event expression uses E_COUNT event operator.

Algorithm E_COUNT (E , E_{RHS}_Name , $Count_{number}$, T_{min} , T_{max} , CloseUsedInstance)

Input:

1) **E** is an event that needs to detect an instance of.
2) E_{RHS}_Name is the event name of the RHS of the event expression
3) T_{min} is the lower single point boundary timestamp to check events instances against
4) T_{max} is the upper single point boundary timestamp to check events instances against
5) **CloseUsedInstance** is a Boolean value to specify to close the events instances after process or not

Output: 1) **e** is an instance E, which is the correlation of event instances E_COUNT e_{RHS} $Count_{number}$

```
1.  do Loop
2.      if (CurrentSystemTime.TimeStamp >= Tmim) then
3.          EventsRHS = SubList (EventMemory , ELHS_Name , Tmin , Tmax )
4.          if (EventsRHS . Length >= Countnumber ) then
5.              e=GenerateEventInstance(E , eRHS , , CurrentSystemTime.TimeStamp)
6.              return ( e )
7.          else
8.              if (CurrentSystemTime.TimeStamp < Tmax) then
9.                  wait (WaitPeriod)
10.             else
11.                 if (CloseUsedInstance ) then
12.                     for each event instance eRHS ∈ EventsRHS do
13.                         MarkClose(eRHS)
14.                     end for
13.                 end if
14.                 return ( null )
15.             end if
16.         end if
17.     else
18.         wait (WaitPeriod)
19.     end if
20. repeat Loop

exit algorithm
```

Appendix F: Experimental Times

The Best and Average times (for 3 tests) , Standard Deviations (S.D.) and Population Standard deviation (P.S.D) for Section 8.2.1 experiment:

Policies	cpu_Usage				process_Monitor			
	Best	Average	S. D.	P.S.D.	Best	Average	S. D.	P.S.D.
1 Host	25.48	26.52	0.99	0.81	29.30	30.42	1.01	0.82
2 Hosts	27.21	28.99	1.78	1.45	31.41	32.84	1.33	1.08
3 Hosts	28.43	30.22	1.79	1.46	34.43	35.64	1.06	0.86
5 Hosts	30.39	33.13	2.76	2.25	38.39	39.60	1.50	1.23
7 Hosts	32.16	33.25	1.25	1.02	43.35	45.09	1.78	1.45
10 Hosts	36.44	37.90	1.42	1.16	46.49	48.47	2.26	1.84

Deployment Time for Two Different Policies of Primitive Events

Time Group	Task	Best	Average	S. D.	P.S.D.
PT	Agent-Finding	1.681	1.711	0.027	0.022
	Agent-Instance-Finding	0.205	0.218	0.013	0.011
	Agent-Configuration	0.879	0.888	0.009	0.007
	Agent-Startup	0.239	0.260	0.020	0.016
	Mapping To Tivoli	0.122	0.135	0.012	0.010
	PT	**3.126**	**3.212**		
RT	BAROC Import	6.010	6.083	0.064	0.052
	Rule Set Import	5.201	5.275	0.068	0.056
	Rule-Base Compile	6.122	6.439	0.354	0.289
	Rule-Base Load	5.022	5.514	0.452	0.369
	RT	**22.355**	**23.311**		
	OT	**25.481**	**26.523**		

Deployment Time Breakdown for Deploying cpu_Usage policy (First Table Time 25.48)

Time Group	Task	Best	Average	S. D.	P.S.D.
PT	Agent-Finding	2.412	2.463	0.047	0.039
	Agent-Instance-Finding	0.293	0.302	0.010	0.008
	Agent-Configuration	2.747	2.772	0.027	0.022
	Agent-Startup	1.090	1.222	0.130	0.110
	Mapping To Tivoli	.135	.146	0.010	0.008
	PT	**6.677**	**6.905**		
RT	BAROC Import	5.771	5.836	0.070	0.057
	Rule Set Import	5.233	5.330	0.087	0.071
	Rule-Base Compile	6.431	6.708	0. 283	0.231
	Rule-Base Load	5.192	5.636	0.458	0.374
	RT	**22.627**	**23.510**		
	OT	**29.304**	**30.415**		

Deployment Time Breakdown for Deploying process_Monitor policy (First Table Time 29.30)

Section 8.2.2 experiment times have almost the same averages and standard deviation ratios as those for experiment times of Section 8.2.1 experiment. Thus, there is no need to outline a detail times tables.

Sections 8.3.1 and 8.3.2 experiments times are all very close (i.e., a maximum of 0.35 second in difference) to the best time, that shown in Table 8.8: The Reuse of Existing Agents' Instances in Policy Deployment) and Table 8.9: The Re-Enforcement of Three Different Policies using PMagic). Therefore we found that no need to go further in analyzing these experiments times.

The Best and Average times (Avg.) (for 2 tests) and Population Standard deviation (P.S.D) for Section 8.2.3 experiment:

session_Control Policy Time Between the first Detected Event and Last Action Taken (in Seconds)												
	1 Event			2 Events			5 Events			10 Events		
	Best	Avg.	P.S.D.	Best	Avg.	P.S.D.	Best	Avg.	P.S.D.	Best	Avg.	P.S.D.
1 Host	12.19	12.76	0.57	31.03	32.27	1.24	69.33	73.17	3.85	104.31	123.11	18.79
2 Hosts	14.42	14.93	0.51	44.23	45.43	1.20	82.29	86.71	4.42	125.33	142.33	17.00
5 Hosts	17.11	17.95	0.84	63.17	64.64	1.47	101.07	111.13	10.06	153.29	174.41	21.12
10 Hosts	20.26	21.13	0.87	82.11	89.57	7.45	122.29	139.15	16.86	189.41	219.40	29.99

The Enforcement of Policy Rule by using Tivoli

The Best and Average times (Avg.) (for 2 tests) and Population Standard deviation (P.S.D) for Section 8.3.3 experiment:

session_Control Policy Time Between the first Detected Event and Last Action Taken (in Seconds)												
	1 Event			2 Events			5 Events			10 Events		
	Best	Avg.	P.S.D.	Best	Avg.	P.S.D.	Best	Avg.	P.S.D.	Best	Avg.	P.S.D.
1 Host	0.028	0.032	0.004	0.097	0.099	0.002	0.621	0.63	0.009	1.603	1.707	0.104
2 Hosts	0.035	0.036	0.001	0.120	0.129	0.009	0.887	0.925	0.038	1.917	2.172	0.255
5 Hosts	0.041	0.047	0.006	0.174	0.186	0.015	1.011	1.400	0.389	2.389	2.752	0.363
10 Hosts	0.066	0.074	0.008	0.251	0.281	0.030	1.633	2.094	0.461	2.907	3.342	0.438

The Enforcement of Policy Rule by using Management Agents