INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

An Enterprise Policy Specification Tool

Venkatachalam

Kanthimathinathan

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

February 2001

ABSTRACT

An Enterprise Policy Specification Tool

Venkatachalam Kanthimathinathan

As enterprises begin to increasingly conduct their businesses over multiple interconnected networks spanning many countries using large computer systems, there is a need to be able to specify, represent and manipulate enterprise policies in these systems so as to be able to efficiently monitor and regulate enterprise processes and business transactions. A large enterprise typically comprises of hundreds of smaller units that to a large extent operate independently but within the overall goals of the enterprise. To reduce any bottlenecks or delays in everyday operations of enterprise units, it should be possible for various enterprise unit heads to create, delete or modify policies relevant to them with minimum reliance on a centralized policy administrator. However, when multiple users begin to manage enterprise policies, there is a need to identify "who can create policies on what resources" and "who should follow these policies". Also, when multiple units compete for common resources, policy conflicts may arise. It is important to detect and resolve conflicts between policies before accepting them for enforcement.

An Enterprise Policy Specification Tool (PST) is introduced in this thesis that provides a solution to the above problems. The tool is unique since it provides a model-driven policy specification language that incorporates constructs to denote the enterprise model in the specified policies. We have defined policy conflicts that can arise and provide a mechanism to detect and resolve them at the time of specification. Although our proposed model is domain independent, the implementation and testing have been carried out in the context of a Message Notification System. A Java-based partial implementation of our proposed architecture has demonstrated the feasibility of the model-based approach to policy specification.

Acknowledgement

I thank my supervisor Dr.Thiruvengadam Radhakrishnan for his guidance and constant encouragement during the course of this work. I thank Dr.Clifford Grossner for his feedback and advice that paved way for this work. I am grateful to Nortel Networks for providing financial support that enabled this work. I thank Mark Beirel for providing initial momentum with the implementation aspects of this thesis. A special note of thanks to my friend, Ragavan, whose sense of humor and good company made things easier during my stay in Ottawa. I thank my family for their infinite belief in my abilities and constant support and encouragement. I thank my best friends, Sriram, Bhaskar, Masoud, Sid and Girma for all the good times they have given me both at work and home. I would like to make a special mention of Roopana for being a great source of encouragement and inspiration during the difficult times. Finally, I would like to thank the excellent support of Stan Swiercz. His presence in the system analyst team gave me immense confidence when working on the implementation aspects of this thesis.

Table of Contents

List of Figures

1. Introduction

1.1 Enterprise and Enterprise Policies

A large enterprise or business organization consists of thousands of human entities and resources operating in a constantly changing environment. The prominent aspects of an enterprise are its *structure, processes* and *product. Structure* is the identification of various entities in the enterprise and relationship between them. *Process* is the sequence of events or interactions between these enterprise entities and *product* is the final goal of an enterprise. *Enterprise policies* are sets of rules framed by an enterprise to control and manage its *structure* and *process* in order to achieve its *product* effectively. Since goals are usually divided into sub-goals, entities usually work in clusters to achieve a common sub-goal. Group of entities working to achieve a common sub-goal is called a *unit*. In an enterprise, some policies are applicable to all units in the enterprise and some only to specific units. Policies that are applicable to all units are called *global policies* and those applicable only to specific units are called *unit policies*. It is important to ensure that *global* and *unit* policies are consistent with each other. When entities locally compete and/or globally co-operate to achieve overall goals of the enterprise, ensuring this consistency is a complex task.

Enterprise policies monitor and control the activities of enterprise entities by applying various constraints or degrees of empowerment to initiate enterprise processes by including statement of permissions, prohibitions and obligations. As enterprises begin to increasingly conduct their businesses using large computer systems which span multiple interconnected networks in different countries, there is a need to be able to specify, represent and manipulate enterprise policies using *policy systems*. A *policy system* is essentially a computer software system that demonstrates [IETF] the following three abilities:

- ability to enable a user to define and update policies
- ability to store and retrieve policies
- ability to interpret, implement and enforce policies

Also, with the automation of many aspects of business to business transactions using *Software Agents* [Wies] or *softbots* there is a need to represent such policies in computer systems so that they can be easily interpreted by these agents to influence their activities. In order to achieve a policy-based enterprise management, an enterprise first represents its rules and regulations in the *policy system* and lets all processes governed by it. The *policy system* controls these processes by permitting, restricting or acting upon them based on enterprise rules and regulations stored in the policy database.

1.2 The Problem

Policy systems in use today are primarily driven by research in the area of network management [Brites][Sloman], systems management [Koch] [Moffet] and active databases [Herbst]. As a result, policies today are modeled based on the Event-Condition-Action (ECA) mechanism that allow specification of policies with a simple ECA rules: if the event occurs in a situation where the condition is true then the specified set of action will be executed. Events that trigger policies are generated in any one of the three ways [Koch]: by means of system monitoring, policy action or human users. Existing policy systems are largely focused on events that are generated by means of system monitoring. However, in an enterprise, when a human user initiates an enterprise process (eg: initiates expense claim process) it needs to be processed based on the user's authority and role within the enterprise. Also, when entities and resources are grouped into hundreds of smaller departments, it is impractical for one person to specify policies related to all entities. It should be possible for each unit within an enterprise to define, update, store or retrieve their own set of policies. In which case, it becomes important to verify who can specify policies (eg: policy for sanctioning an expense claim) on other entities. Existing policy systems [Bala] [Kramer] [Sobieski] assume that policies are specified by a single user namely the system administrator and

do not recognize the need to verify the policy creator's authority to specify policies on other entities.

When multiple units are allowed to mange their own policies, two units may compete for a common resource, in which case policies from different units may conflict. Therefore, when new policies are entered into the policy system, before they are stored in the policy database conflict detection should be performed and feedback should be provided to the policy creator about the detection results. This feedback could be a "Valid/Invalid Policy" message or to indicate the erroneous rule conditions or actions and resolve them interactively with the user. In policy systems today, policy conflicts are resolved based on the priority value of a policy, which is a numerical value provided to it at the time of specification or by prioritizing negative authorizations over positive authorizations. However, in an enterprise, conflicts between two policies are usually resolved based on the relative authorities of entities that created the policies and hence the present conflict resolution methods [Lupu] are not sufficient.

The dynamic nature of the enterprise has a potential to introduce new inconsistencies in the policy database when entities leave the enterprise or some resources cease to exist. In this context, several questions arise: if the specifier of a policy leaves the organization, should that policy continue to exist or not? When domains or groups within the enterprise are deleted or merged with other groups how should the policies that target these domains be modified to reflect the new changes? It is important to realize that checking for consistency of policy database is not only required when a new policy is introduced into the policy system but also when there is a change in the enterprise structure. Policy systems today, do not recognize need for such incremental maintenance and hence their policy database may have policies that may never be executed or may throw exceptions at the time of execution. Marriot 's Policy Service [Marriot] has partially addressed this issue by allowing an optional exception clause in policy statements. If the execution of a policy throws exception then the exception clause is executed. However, this method is not very efficient since the inconsistent policy will continue to exist in the policy database. In summary, we note that the present policy systems:

- assume policies are specified by single users namely the system administrator
- do not verify the authority of a policy creator to define new policies

- do not provide a sufficient conflict resolution mechanism
- do not recognize the need for incremental maintenance of policy database

1.3 Our Contribution

In this thesis, we have proposed an enterprise authority (EA) model that denotes the structure of an enterprise and the relationship between various entities in the enterprise. We make use of this model to obtain the scope of a policy for its enforcement. A Policy Specification Language (PSL) is proposed in this thesis that enables a user to create new policies for enforcement. PSL policies obtain their scope automatically from the EA Model and this implicitly determines "who can specify policy on whom".

Another contribution of this thesis is a framework for detection of policy conflicts and a method to automatically resolve conflicts based on the enterprise authority (EA) model. We identify situations in an enterprise that may create new inconsistencies in the policy database and recommend incremental maintenance tasks.

We have designed and developed an Enterprise Policy Specification Tool (PST) that supports our EA Model, PSL, Conflict Detection and Resolution mechanisms. The underlying architecture of PST is flexible to support incremental maintenance tasks.

1.4 Organization of the Thesis

In Chapter 2 of the thesis, we provide an overview of several policy systems that use different specification languages and conflict detection and resolution mechanisms. In Chapter 3, we introduce the EA model based architecture that addresses the limitations of existing systems to operate in an enterprise environment. We describe the syntax and semantics for our proposed policy specification language and describe a method to detect and resolve policy conflicts. In Chapter 4, we present a Java based implementation of the system prototype and provide implementation details of various system components. Finally, Chapter 5 provides the conclusion and future work.

2. Survey of Policy Systems

2.1 Introduction

In this chapter we will review four popular policy systems that are presently used to provide some form of policy-based management in enterprises. Before reviewing these systems we will examine the various stages of a policy from the time of specification to its enforcement.

Policies can be specified at different levels of abstraction. At the highest level we can find policies specified using natural language. At low levels they are used to describe policies that deal with when and how to configure a device or how to manipulate different network elements under different conditions in order to achieve management goals. A high level policy when translated will yield many lower level policies. This derivation is obtained by refining the goals, partitioning the targets or delegating the responsibilities to other managers. Consider an example of a high level policy on availability of a printer in a department:

Example: "Atleast one printer should be available for use by graduate students".
This policy when translated will yield many other policies on printer maintenance, redundancy and error recovery. The ultimate aim is to be able to specify high-level policies and automatically generate the lower level ones. High-level goals and objectives, which are typically expressed in plain-text can be transformed into policies that are less abstract and at a level that can be implemented in a software system.

Moffet [Moffet] and Marriot [Marriot] identify the need to formally describe the relationships between high level and low level policies and their translations. It would be useful for a system to be able to record both the translation process and its inverse mapping. If a high level policy is changed or defined, it should be possible to decide what lower level policies must be newly created or modified. Weis [Weis] states that

degree of detail and technological aspects increase as one moves down the hierarchy while the business aspects decrease.

A new or updated policy that is specified to policy system has to be stored in the policy database for persistence. However, before the policy is stored in the repository, it is important to check its correctness. Such verification could result in a "valid" or "invalid" condition. Since policy rules can be syntactically correct yet make no sense, validation of semantics of a policy rule is necessary [IETF] to ensure that the policy is meaningful in a given context.

Conflicts between policies can occur when two policies that monitor the same set of events trigger opposite actions. Or conflicts can occur when two policies, one with negative authorization and other with positive authorization exist simultaneously [Sloman] in a policy set. Such conflict detection maybe performed either at the time of specification (i.e., before the policy is accepted) or during policy execution. The detection performed at the time of specification is also called off-line detection, meaning that it is not performed at the same time as the execution of the policy whereas on-line detection occurs at the time of policy execution. Conflict detection at the time of specification, check for static conflicts derived from set of policies whose conditions are simultaneously satisfied, but whose actions conflict with those of currently existing rules. However, not all policy conflicts can be detected at the specification level. Some policies rules may be based on time (specifying an effective validity period in the future) or based on dynamic state information. Conflicts between such policies may only be detected at the time a policy becomes valid and enforcement action is attempted. When a conflict is detected, conflict resolution mechanism is initiated. In order to resolve conflicts between two policies, we need to establish precedence of one policy over the other. Once the precedence is established, the policy with a higher precedence is accepted and the other is retracted or discarded. Policy systems today are experimenting with various [Lupu] precedence mechanisms such as modality [Marriot] [Sloman] based precedence, numerical value based precedence and subject (*subjects* are entities that are required to follow a policy) specificity based precedence to establish precedence between conflicting policies.

Policies are inherently dynamic in nature and they constantly undergo changes with changing business environment. Implementing policies in a procedural language

(eg: C/C++ or Java) makes maintenance difficult. Since, when a policy changes, policy administrators have to recompile the code to reflect new changes. This is inefficient and time consuming. Rule-Based languages (e.g., Prolog, Clips) support interpretive use of policies where rules are interpreted at run time that makes maintenance of policies easier. Also, it enables new policies to be dynamically added on the fly.

2.2 Case Study 1: Yeast

2.2.1 Overview

Specification of policies in Yeast [Bala] is based on event-action pairs. Events are monitored and when they occur actions are triggered. Each Yeast event-action specification defines a pattern of events and the appropriate action to be executed in response to the occurrence of the event pattern. The action can be in the form of invoking any program that is executable from the Yeast system shell. Yeast provides a mechanism for detecting two types of events: *polled* or *pushed* events.

As shown in Figure 1, Yeast is a client/server system. The server is a central entity accepting client commands from a number of (possibly remote) users. The primary function of the server is to accept, match and manage specifications on behalf of users. The user invokes client commands through the computer system's command interpreter (such as UNIX shell). Client commands are used for various activities such as register specifications with the server and to perform various querying and management chores. The server and client programs can reside anywhere in the network. The various client commands are described later in this section.

In Yeast, an event corresponds to change in the value of an attribute of some object class. The object definition database stores the definitions of various object classes and attributes. The database contains the definitions of object classes and attributes that are pre-defined or user-defined.

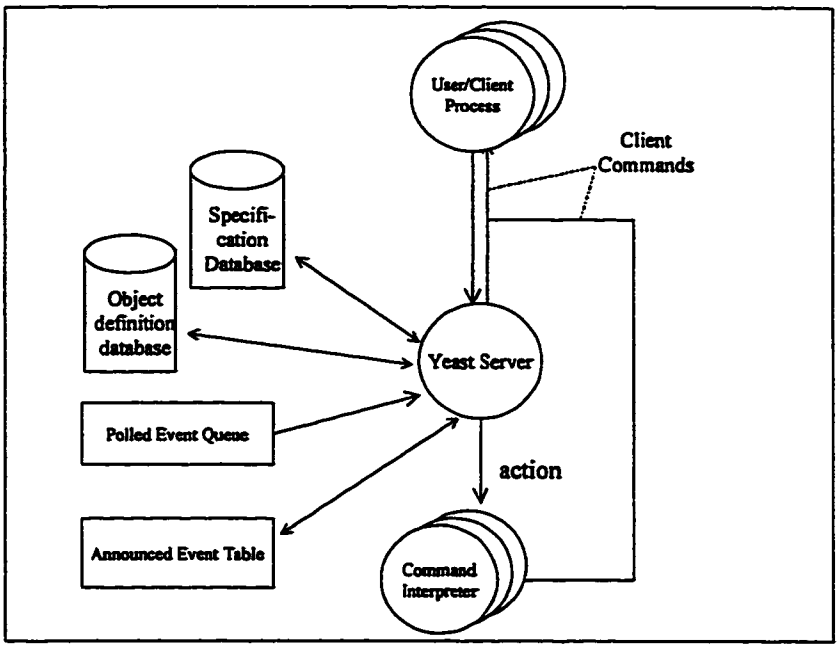

Figure 1: Architecture of Yeast System

As new Yeast specifications arise they are stored in a file system for interpretation. The specification database includes a persistent copy of the specification stored in the file system. If the Yeast server is restarted after a failure, the file system is restored from the database. A Yeast specification has the following syntax:

event_pattern **do** *action*

The event pattern contains primitive event descriptors formed that are combined using the connectives **then**, **and** and **or**. New specifications can be specified in Yeast using the ***addspec*** command as shown below:

addspec *event_pattern* **do** *action*

where *event_pattern* is a complex expression of event descriptors and *action* is any shell (that is running Yeast application) executable command. The three operators *and*, *or* and *then* are used to form the event pattern. The action part is not parsed by Yeast and is executed by the command line interpreter as it is.

As a case study, we will go through the sequence of steps to specify a policy in Yeast to monitor the "status of a project file and notify the project team when the project file is debugged".

Step 1: *defattr* file debugged boolean

The Yeast commands *defobj* and *defattr* are used to define new objects in Yeast. However, since the object "file" is a predefined object in Yeast, it exists in the object definition database and hence can be used in Yeast specification. We will define an attribute called "debugged" for the pre-defined object class file and then register a specification that notifies project personnel whenever file project.c is debugged:

Step 2: *addspec* file project.c debugged == true

do notify project.c debugged

The **addspec** command registers new specifications in Yeast. The action, *notify* has to be supported by the command interpreter running Yeast application. The above specification would be matched when the person responsible for debugging the file project.c generates the following announcement:

Step 3: *announce* file project.c debugged = true

Yeast's **announce** command is used to generate a *pushed* type event to Yeast thus matching any specification that monitored the user-defined attribute.

Yeast provides specification related commands such as *lsobj, lsattr, rmobj* and *rmattr* to manipulate Yeast objects. And *commands lsspec, rmspec, modgrp* to manipulate Yeast specifications.

2.2.2 Challenges and Limitations

By default, Yeast specifications have global scope, i.e., Yeast event-action specifications are applicable to all registered Yeast users. Yeast provides a mechanism to verify the authority of a user to monitor or initiate enterprise processes by providing three levels of control to Yeast objects: *read, announce, write* and *owner* accesses. A user is allowed to monitor an event if he or she has a *read* access to the event object. A user can initiate an enterprise process by instantiating an event object if he or she has an *announce* access to an object. Similarly, the user can define and remove attributes of an object class with *write* access. With *owner* access, the user can delete and remove the object class itself. Although, this helps to verify the authority of user to initiate a request, it will be cumbersome to maintain accesses at the object level for every entity for each enterprise process in a large enterprise.

Since the task of policy specification is not restricted to one user (e.g., administrator), conflicts may arise when two users input specifications that perform opposite actions on the occurrence of the same event. Yeast does not provide a mechanism to detect and resolve conflicts between its specifications. Yeast [Inverardi] discusses possible inconsistencies that can occur between Yeast specifications but does not provide a method to detect or resolve them.

2.3 Case Study 2: Karma

2.3.1 Overview

Karma [Sobieski] is a policy management application developed to support the collection, analysis and implementation of business policies at Fannie Mae, a large insurance company. However, Karma is domain independent system. Karma has three main components. They are:

- Data Dictionary Editor, where objects and attributes that are required to define business policies are defined
- Rule Editor, where policies are formally specified as business rules to the system
- Rule Browser, which is used to browse existing policies

Karma allows defining new event and action objects and their attributes through its data dictionary editor and provides the user with a rule specification language to define new rules using these objects. When new rules are defined, Karma stores it in the rule database. From this database, Karma generates executable ART-IM rules for the Karma rule server to interpret. Using Karma Rule Specification Language the policy rules are specified as:

IF <clause>
AND <clause>
THEN <clause>

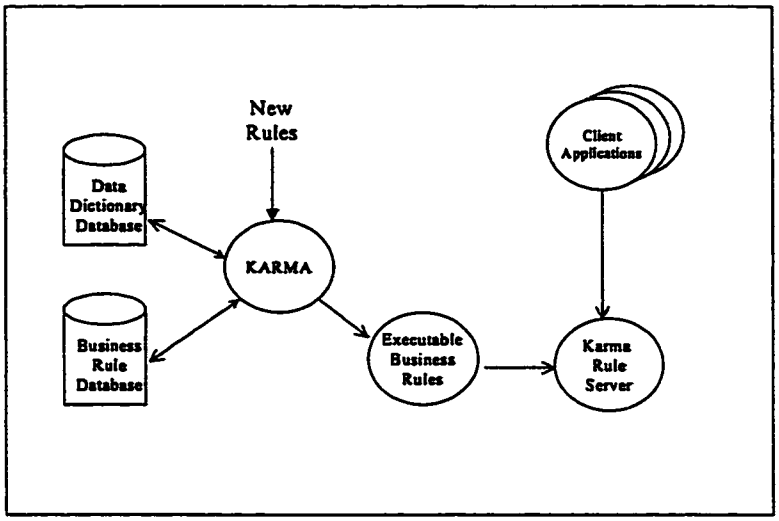

Figure 2: Karma Component Overview

Where the <clause> can be any of the following forms:

<attribute> <operator> <attribute> |

<attribute> <operator> <value> |

<attribute> <operator> <attribute-list> |

<attribute> <operator> <value-list> |

<object> <operator>

As a case study, we will go through the sequence of steps needed to specify a policy that monitors the *status* of a project file and *notifies* the project team when the *project file* is *debugged*. Using the Karma Data Dictionary editor, we first define a object *Project*, that monitors the file "project.c" and appropriately sets its attribute *debugged* to boolean value true, when the file is debugged. And another object Notify that will send appropriate notifications.

We will then define a Karma rule given below:

IF Project.debugged *is* true

THEN notify team

The clause *"Project.status is debugged"* is an *<attribute> <operator> <value>* clause in Karma (i.e., Project.status, is, debugged). The clause *"notify team"* is an <object> *<attribute><operator>* clause. Karma policy rule expressions consist of left-hand side and

right hand side clauses. They may have one or more clauses ANDed together in the left-hand side but have only one single clause in the right hand side.

2.3.2 Challenges and Limitations

Karma assumes a centralized rule administration, with one user (e.g., administrator) defining and maintaining policy rules and hence a Karma policy is applicable to all users of the policy system. Karma does not provide support to enterprise concepts such as unit, role or authority of a user. Also, Karma objects do not have an ownership as in the case of Yeast and this limits it from verifying the access of a user to initiate an enterprise process or to specify policies to monitor them. And hence any user with access to the Karma data dictionary can specify policies using the objects defined in them.

Karma has a good consistency checking mechanism. When new rules are defined in Karma they are checked for consistency with the existing rules and policy conflicts are detected and displayed. The consistency checking capability of Karma identifies the following relationships among new policy rules: inferred rules, redundant rules, conflicting rules and subsumed rules. However, when Karma detects a conflict it only displays the results and does not propose a method to resolve these inconsistencies either interactively with the user, or automatically.

Karma's provides a data dictionary editor to manage Karma business object model. However, when an object is deleted from Karma data dictionary there is still a possibility that Karma policy set has rules that continues to use this object. Since Karma does not recognize the need to maintain consistency between its object model database and policy database, there could be policies in the policy set that may never be executed or may throw exceptions during execution.

2.4 Case Study 3: Policy Service

2.4.1 Overview

Sloman, Lupu and Marriot's research on Policy Service [Marriot] relates to policy specification [Marriot] [Sloman], conflict analysis [Lupu] and architecture [Marriot] for policy-based systems. Policy Service provides a persistent data store for policies and provides a mechanism for distributing policies in a distributed system. Fundamental to their work on Policy Service, is the way the authors have classified policies into two types: as *authorization policies* and *obligation policies. Authorization policies* define what a set of users (*managers)* are permitted (or not permitted) to do in terms of operations on a set of resources. *Obligation policies* specify what activities a user must do (or must not do) to a set of available resources. Their policy model allows each type of policy (*authorization or obligation*) to be either positive or negative in nature.

Policy Service provides a high-level specification language, Policy Notation, to enable users to create new policies into the policy system. Policy Notation has the following syntax (note: optional attributes are within brackets):

[description] identifier mode [trigger] subject `{'action `}' target [constraint] [exception] [parent] [child] [xref] `;'

The mode of a policy is given either by an "O" for obligation policy or by an "A" for authorization policy, with a "+" or a "-" denoting a positive or negative policy. The notation supports the following types of policies:

- positive authorization policy (A+)
- negative authorization policy (A-)
- positive obligation policy and (O+)
- negative obligation policy (O-)

We briefly describe other attributes used in the notation:

- **description:** attribute is used to provide general comments about the policy such as its history, author or name
- **identifier:** is used to provide a label to refer the policy

- **mode:** of a policy is given either by an "O" for obligation policy or by an "A" for authorization policy, with a "+" or a "-" denonting a positive or negative policy.
- **Trigger:** is the definition of the event that may trigger actions of an obligation policy. However, authorization policies are consulted whenever relevant action is attempted.
- **subject:** of a policy specifies those objects which are obliged or authorized to perform the action specified. Users and automated managers are examples of typical subjects. An obligation policy is distributed to its subjects to be interpreted.
- **target:** The target of a policy specifies the objects on which actions are to be performed. Targets can be grouped as domains.
- **action:** specifies what action must be performed for obligations and what is permitted for authorizations.
- **Constraint:** defined by the **when** clause limits the applicability of a policy
- **Exception:** It is a exception mechanism provided for positive obligations to permit the specification of alternative actions to cater for failures which may arise in the system while executing policy actions. This failure maybe due to network failure or any other reason.
- **parent, child, xref:** High-level abstract policies can be refined into implementable policies. In order to record this hierarchy, policies contain references to their parent and child policies. In addition, a cross-reference (xref) from one policy to another can be inserted manually.

The Policy Notation for a policy that monitors the *status* of a project file and *notifies* the project team when the *project file* is *debugged* is shown below:

```
ProjectStatus O+ u:@developers {project.status = = debugged}
    Notify.team ("File Debugged");
```

2.4.2 Challenges and Limitations

Sloman's authorization policies define what activities a *subject* can perform on a set of *target* objects and are essentially access control policies to protect resources from unauthorized access. In an enterprise scenario, *targets* can be *subjects*, since multiple managers (*subjects*) may specify policies for employees (*subjects*) to follow. In this case,

there is a need to verify if a *subject* can specify a policy for another *subject* to follow. Sloman and Lupu [Lupu] have described Roles and Role Relationships in enterprises but do not recognize the need to verify new policies based on roles and role-relationships between policy creator and policy subjects.

Policy Service imposes a constraint that every obligation policy action has to be authorized by an authorization policy otherwise the action is prohibited. This can create inconsistencies in the policy set with an obligation policy existing without appropriate authorization policy. Such inconsistencies can be prevented if obligation policies are verified for appropriate authorizations before their acceptance for enforcement.

In Policy Service, conflicts occur when the following policies apply for same set of *subjects* and co-exist in the policy set:

1. positive and negative authorizations (A+/A-)
2. positive and negative obligation policy (O+/O-)
3. obligation policy with negative authorization (O+/A-)

Marriot [Marriot] in his initial work on Policy Service has used modalities ("+" or "–") to establish precedence between conflicting policies. This provides higher priorities to negative modality ("-") over positive modalities. Therefore, if a positive authorization policy and a negative authorization policy for the same set of subjects co-exist in a policy set, then the negative authorization policy is given precedence and retains its place in the policy set. The positive authorization policy is retracted and discarded. Although, in this method precedence is easier to establish, it is not suitable when multiple users with varying authorities create policies. In an enterprise scenario, conflict resolution needs to consider the authority of the users involved in creating conflicting policies and not just the modality of the policy.

Sloman and Lupu [Sloman] propose a domain nesting based precedence mechanism to establish precedence between two conflicting policies. This gives precedence to policies that apply to more specific set of *subjects, targets* or both. In an enterprise, it is important to ensure that unit level policies (which are more specific to a domain) need to conform to global policies (which are less specific to a domain) of the enterprise. If applied in an enterprise situation, when conflict is detected between global and unit policies then domain nesting based resolution mechanism will give unit level policies precedence over global policies. This is not desirable. Also, domain nesting based

precedence cannot be established when the sets are equal or when *subject* sets are more specific but the *target* sets are less specific or vice verca.

The policy notation [Marriot] allows specification of exception clauses to specify alternative actions to cater for failures that may arise when trying to execute a policy action. An obligation policy with multiple targets may result in multiple exceptions. Exception mechanisms are useful to handle transient failure conditions (eg: network down time) that recover to normalcy after a period of time. However, when the failure is bound to be permanent (eg: subject entity deleted), exception clauses are not an efficient solution. This is because over a period of time as the number of permanent failures increase, the number of exceptions will continue to rise. It will be more appropriate to remove policies that create permanent exceptions from the policy set. This will free up some execution time spent in executing these exception clauses thus speeding up the processing of other events waiting for execution.

2.5 Case Study 4: Rule Based Management Architecture

2.5.1 Overview

Thomas Koch's management framework [Koch] [Kramer] is based on an object-based approach and hence all components of their environment are viewed as instances of different object classes. There are two types of objects in their model. They are *managed objects* and *managing objects*. *Managing objects* are objects that perform operations on *managed objects* and they can be grouped into domains so that policies may be set at domain level rather than for every object.

Thomas Koch [Koch] has adopted Moffets [Moffet] definition of a policy where every policy is a object with a core set of attributes and it may have additional optional attributes. Similar to Policy Service [Marriot] discussed in the previous section, policies are classified into two types: as *obligation policies* and *authorization policies*. However, unlike Policy Service, Koch's work does not further classify each type of policy into positive and negative modes. Koch's and others have implemented their proposed

policy framework with Marvel [Kramer]. Marvel was originally designed as an environment that assists software development and evolution. Koch's management policies are represented as rules and interpreted by Marvel. Marvel requires data model and a process model to function. Various objects in Koch's framework are represented in Marvel data model that contains the description of these objects.

Koch's and others [Koch] have proposed a three level policy specification hierarchy which are requirements, goal oriented and operational level. They are illustrated below:

- requirements level – at this level policies are typically described in natural language. It allows description of any desired behavior without having to have a prior knowledge of the technical details concerning it.
- Goal oriented level – define the use of management services.
- Operational level – operate at the level of managed objects (simple abstractions of managed resources)

There are different language constructs for each abstraction level thus enabling a clear distinction between levels. At the lowest level they provide a Policy Definition Language that enables the computer to check the syntax of a given policy description and translate policies into executable rules. The PDL syntax for obligation policy is given below:

```
policy name type obligation for subject {
<targets>
        action [eventname] [if precondition] {
        <operation>*
        } [success postcondition1]
          [nosuccess postcondition2]
}
```

As a case study, we will go through the sequence of steps to specify the same policy discussed in Yeast and Karma at operational level using PDL (Policy Definition Language) to monitor the status of a project file and notify the project team when the project file is debugged.

```
policy p1 type obligation for "/root/users/" {
```

```
        Action prject_c_debugged {
                Notify(team, "project.c debugged")
    }
```

The above policy is triggered when an event "project_c_debugged" is generated.

2.5.2 Challenges and Limitations

Koch's work follows the Moffet's [Moffet] definition of a policy and views a policy as an object with a set of mandatory attributes. Subjects and target objects are mandatory attributes of a policy and they are explicitly specified using the Policy Definition Language (PDL). It is possible to have domains as targets. Due to this feature, it possible to group entities into domains and have policies target these domains. However, since the framework essentially targets network management chores, it assumes that a single user, typically the administrator manages all policies in the system. Therefore, there is no suitable mechanism to verify the policy creator's authority to specify new policies for a domain or sets of entities. Also, domain memberships have to be updated as when new entities are added or deleted to the enterprise. This is cumbersome and error prone. It should be possible to obtain the target list for a policy based on the policy creator's authority in the enterprise.

Koch's management framework provides three different specification languages to allow specification of policies at different levels. However, conflict detection is critical when allowing policies to be specified at different levels. Since in some cases conflicts may not exist at a higher level but may arise at lower levels during the translation process. These conflicts have to be detected and resolved before further levels of translations are allowed to occur. In different policies they may occur at different levels. Koch's proposal to resolve conflicts based on differences in hierarchy levels between conflicting policies is not sufficient in an enterprise context. Since Marvel based framework is essentially targeted towards systems management it is limited in its conflict detection and resolution mechanism.

Marvel data model contains the description of managed objects. It does not discuss types of inconsistencies that can occur between its data model and process models and how they will be detected and consistency is restored. Any changes in

Marvel data model (such as deleting an object) could make the target policy irrelevant. In an enterprise scenario such a possibility is high since when target objects are employees and they assume new roles, or transfer to other groups the Marvel data model may require changes.

2.6 Chapter Summary

This chapter examined four popular policy systems, their specification, conflict detection and resolution mechanisms. Based on our survey, we realize that these systems and most other policy systems today are mostly targeted for systems management or network management tasks where single user namely, the system administrator creates and manages policies for all users to follow. However, in case of a large enterprise this is not sufficient and we identify below the design goals for a policy system suitable for enterprise management chores – it should be possible:

- for individual units within an enterprise to manage their own local policies
- to verify the authority of a policy creator on *targets* and *subjects* of a policy
- detect and resolve conflicts as relevant to an enterprise based on enterprise authorities of entities involved in creating conflicting policies
- maintain constant consistency of policy set by providing close interaction between available resources and policy set

In the next chapter, we have proposed an Enterprise Policy Specification Tool (PST) that takes into account above design goals. The tool uses our proposed Enterprise Authority (EA) Model that captures the relationship between various entities in an enterprise.

3.0 Enterprise Model Driven Policy Specification Tool (PST)

In the first section of this chapter, we propose an Enterprise Authority (EA) model that captures the structure of an enterprise or business organization and the authority of enterprise entities on enterprise processes. In the next section, we propose a Policy Specification Language (PSL) that provides constructs to denote the EA model in enterprise policies. In the following section, we discuss the syntax and semantics of PSL. We then examine policy conflicts that may occur when new policies are defined and how they are detected and resolved. Finally, we propose an architecture that will support the EA Model, PSL and Conflict Detection and Resolution mechanism discussed in this chapter.

3.1 Enterprise Authority (EA) Model

An enterprise is viewed as a collection of interconnected entities, called enterprise objects. These objects can be software entities, human users or interface objects such as transducers and actuators. An enterprise object may have authority over zero or more other enterprise objects in which case it may command that an operation be carried out. It is assumed that a commanded object will perform the requested operation or will respond to inform any abnormal situation. In this section, we propose an Enterprise Authority (EA) model that captures the relationships between enterprise objects in a business organization. As shown in Figure 3, we classify *enterprise objects* into three disjoint sets: *specifier*, *program* and *event objects*. *Specifier objects* (eg: humans, software agents) are entities that can command other enterprise objects. They use *program objects* (Eg: BudgetSanction or RouterConfigure) and *event objects* (Eg: BudgetRequest or FaultAlarm) to initiate or accomplish enterprise processes.

In an object-oriented paradigm, the EA Model provides a mechanism to define new *specifier*, *event* or *program* objects. These objects have the following common mandatory attributes:

- *entityID* – the value is assigned to the object at the time of its creation;
- *conflictID* – the value can consists of one or more *entityID* of other entities that may perform actions that conflict with it. If this attribute does not exist for an object, then it means it's

action does not conflict with any other object in the system. This attribute can take one or more values.

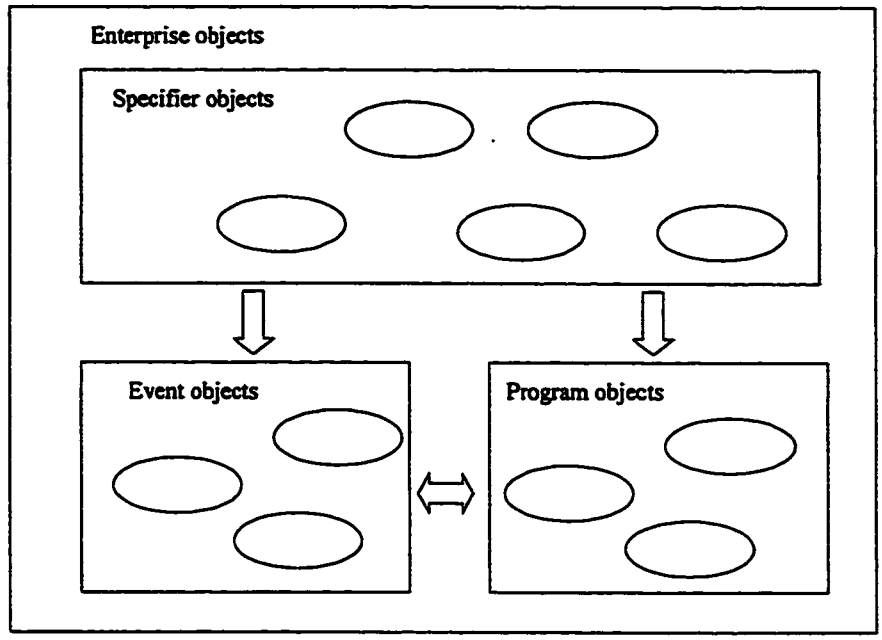

Figure 3: Enterprise Objects Classification

In addition to the above attributes, the *event objects* have the following mandatory attributes:

- *originatorID* - the value is assigned by the system; it is *entityID* of the entity that generates the event;
- *destinationID* - the value is assigned by the system; it is *entityID* of the entity that is intended to receive the event;
- *header* – value is provided by the entity that generates the event; its value is of type string.
- *priority* – value is provided by the entity that generates the event; it denotes the importance of the event and it can take any one of the three values (of enumerated type) namely low, high or critical.
- *timeStamp* – time (in hrs, minutes and seconds) of generation of this event; its value is provided by the system.

A typical Enterprise Authority (EA) model is shown in the Figure 4 below. Enterprise objects are represented as nodes in a Directed Acyclic Graph (DAG) and the edges between nodes represent

the *type* of authority of one node over the other. Event and program objects are placed in the leaf nodes of the DAG. The node from which an edge originates is called the *source node*. The node to which the edge terminates is called the *target node*. The source nodes are *specifier objects* and target nodes maybe a *specifier, event or program* object.

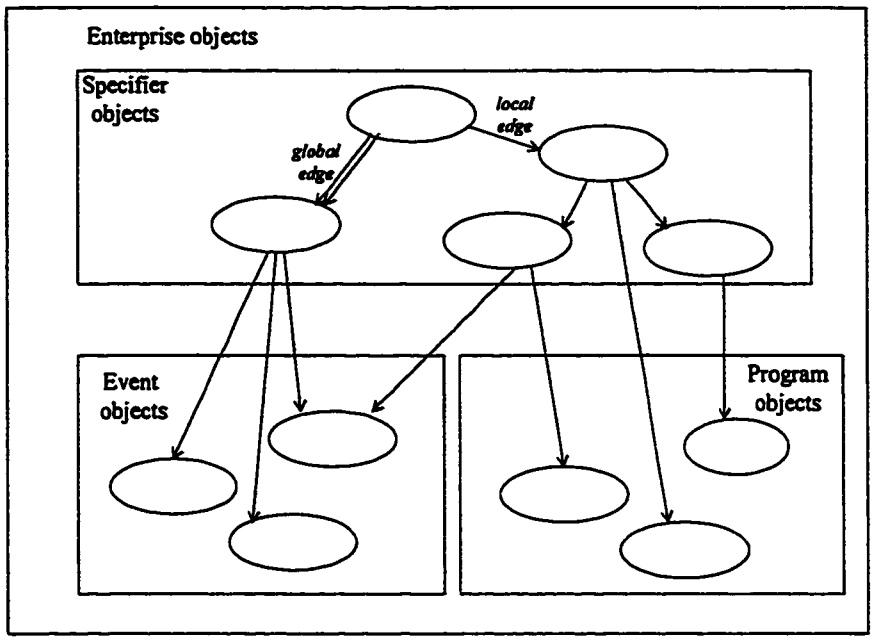

Figure 4: A typical Enterprise Authority (EA) Model

We classify authorities to be of two *types*: *local* or *global*, hence there are two types of edges – namely a *local edge* or *global edge*. In the figure above, *local edges* are indicated with single line arrows and *global edges* are indicated with double lines. A local edge between a *source node* to *target node* implies a local authority where policies set by the source node are local in nature- i.e., they are applicable only to nodes to which the source node has direct connecting edges. And a *global edge* implies a global authority where policies set by a *node* are applicable to all its descendent nodes. In other words, the *local* or *global* authority of a *specifier objecct* determines the *scope* of a policy. And the *scope* determines the other specifier objects that have to adhere to the policy.

3.2 Case Scenario: Role Based Policy Specification

We will consider a business organization, identify various enterprise objects in it and define relationship between them using our EA model. Employees have responsibilities, permissions and authorizations and play roles such as manager, salesman and vice-president etc. It is practical to represent roles rather than employees as nodes in the DAG because separating the roles and employees permits the assignment of new employees to a role without re-specifying the policies. When employees assume a role they obtain the various characteristics of that role. Also, employees may assume more than one role to accomplish their everyday tasks. The mechanism to related employees to roles is beyond the scope of this thesis.

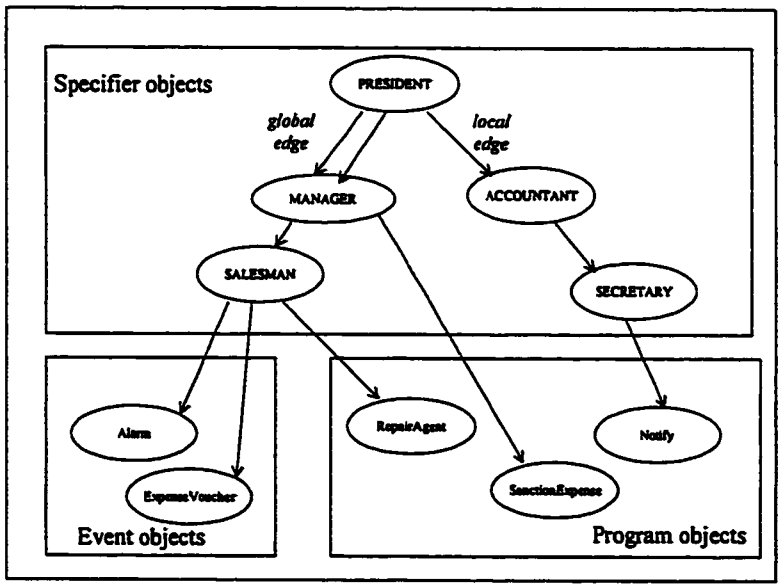

Figure 5: Role-based EA Model

Since employees initiate enterprise processes by performing operations on *event or program objects*, the EA model essentially captures their authority to initiate enterprise wide operation. When an employee initiates an enterprise process, events are triggered. Enterprise policies monitor these events and execute a set of activities when events occur thus enforcing policies on enterprise processes.

In the DAG shown in Figure 5, the "Salesman" has authority to initiate enterprise processes that use "Alarm", "ExpenseVoucher" and "RepairAgent". The "manager" inherits the

authority of "salesman" on enterprise object and in addition has a direct authority to "SanctionExpense". Similarly, the "secretary" has authority on "Notify" object and the "accountant" inherits this authority. Since the "president" inherits authority of "manager" and "accountant", it has authority on "Alarm", "Notify", "RepairAgent" and "SanctionExpense" and "ExpenseVoucher".

A *local edge* from a "president" to "accountant" denotes that enterprise processes initiated by "accountant" are bounded by policies specified by "president". Similarly, enterprise processes initiated by "secretary" are bounded by policies specified by "accountant". It is important to observe that enterprise processes initiated by "secretary" are *not* bounded by policies set by "president" since the "president" has only a local authority in that sub-tree.

A *global edge* from "president" to "manager" indicates that enterprise processes initiated by "manager" and "salesman" are bounded by policies set by "president". The "salesman" is also bounded by policies set by "manager". Several disjoint EA models can be combined together to form a Global Enterprise Authority (GEA – pronounced as 'Jeeya') model. The GEA model will capture the authority of the specifier objects of one enterprise over *event* and *program objects* of another enterprise. Essentially, it captures the authority of employees belonging to an enterprise or even a distinct unit within an enterprise to initiate enterprise processes in other enterprises or units. While creating GEA models is important to identify distinct units within the enterprise since a unit can exist in varying degrees, i.e., the smallest unit maybe a single employee and the largest unit is the enterprise itself.

The GEA model differs from the EA model in an important way. It does not allow *specifier objects* of one enterprise to create policies on *specifier objects* of another enterprise. The Figure 6 shows a GEA model that defines relationships between three EA models, namely EA-1, EA-2 and EA-3. The "president", "accountant" and "secretary" of EA-1 can use "Alarm" event objects in EA-2 and EA-3. In other words, this means that employees who assume the role of a president in EA-1 have the authority to trigger alarm events in other enterprises that may in-turn initiate other enterprise processes within the target enterprise.

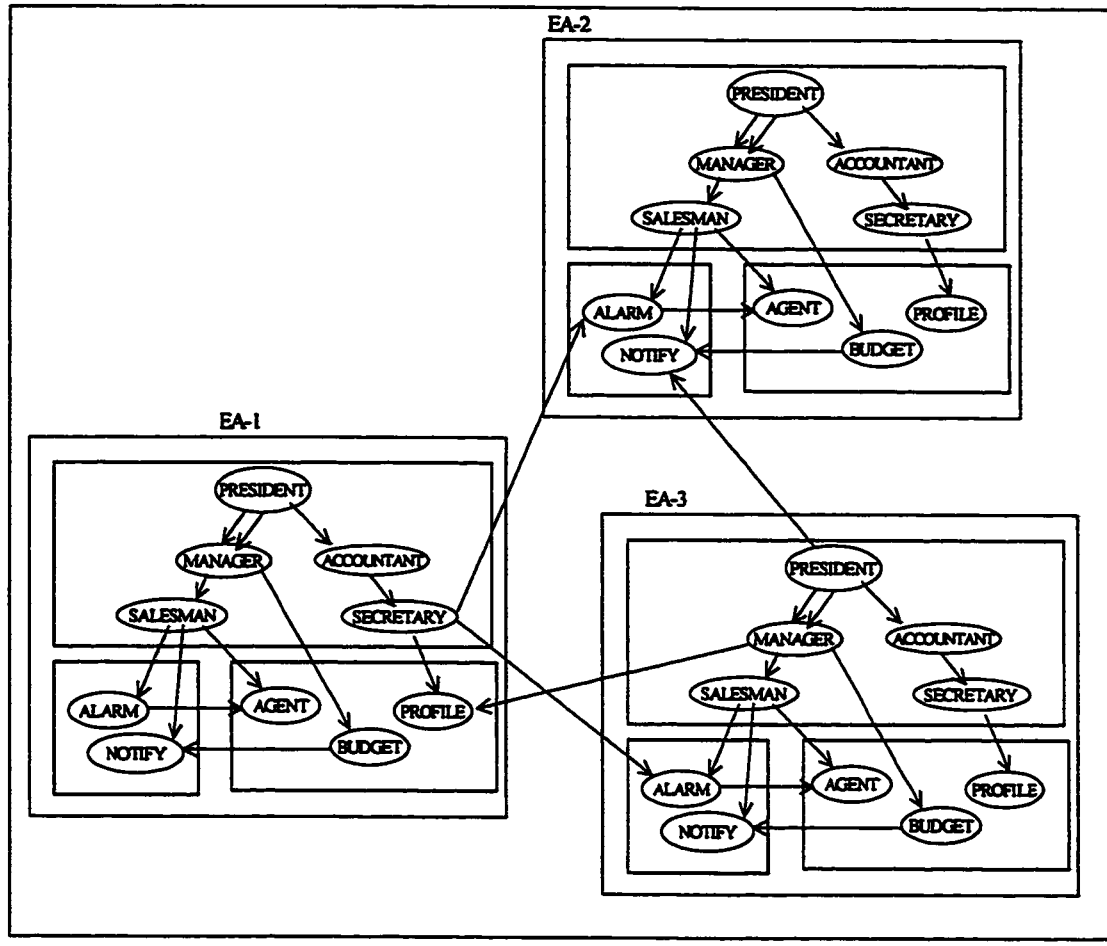

Figure 6: Global Enterprise Authority (GEA) Model

3.3 Policy Specification Language (PSL)

In this section we discuss the syntax and semantics of our proposed Policy Specification Language (PSL). It provides the necessary constructs to specify model driven enterprise policies. PSL formalizes the intent of the *policy specifier* into a form that can be read and interpreted by machines. A policy may give certain rights to entities (programs, users, etc) that fulfil some criteria, and deny certain other rights. A PSL rule can bind an entity to certain restrictions and obligatory actions.

3.3.1 Syntax and Semantics

The syntax of the Policy Specification Language (PSL) is given below:

<POLICY>	:=	**<policy-name> <policy-specification>**
<policy-name>	:=	String
<policy-specification>	:=	**<source> <content> <action>**
<source>	:=	<Policy-Specifier> <EA-Model>
<Policy-Specifier>	:=	String
<EA-Mode>	:=	String
<content>	:=	**When** <event-clause> **arrives If** <condition-clause>
<event-clause>	:=	<event-expr> | <event-expr> <logic-operator> <event-clause>
<event-expr>	:=	<EO-Occur> |
		<EO-Occur.attrib> <relation-operator> <EO-Occur.attrib> |
		<EO-Occur.attrib> <relation-operator> <constant>
<condition-clause>	:=	<cond-expr> |
		<cond-expr> <logical-expr> <constant> |
<cond-expr>	:=	<EO-Occur.attrib> <relation-operator> <EO-Occur.attrib> |
		<EO-Occur.attrib> <relation-operator> <constant>
<action>	:=	**then** <action-clause>
<action-clause>	:=	<action-expr> | <action-expr> <action-clause> | **forbid**
<action-expr>	:=	<EO> | <PO-Operation>
<EO>	:=	Instantiation of an Event Object
<EO-Occur>	:=	Occurred Event Object
<PO-Operation>	:=	Operation on Program Object
<logic-operator>	:=	AND
<relation-operator>	:=	= | != | < | > | >= | <=
<constant>	:=	String | Integer | Boolean

A PSL rule has four sections: *source, content, action and subject.*

- *source* contains name of the source or the *specifier object* that specifies the policy and the associated enterprise model referred by model-name.

- *content* is composed of two parts. The first part consists of <event-clause> that contains one or more monitored events. The second part is the <condition-expression> that evaluates to TRUE or FALSE.
- *action* contains the <action-clause> that can be an instantiation of an event object or operation on a *program object* or the terminal in nature by means of *"forbid"* primitive. The "forbid" is a PSL primitive that prohibits the event from creating any action, i.e., ignore the event.
- *subject* contains a set of *specifier objects* in the associated enterprise model that are required to follow the policy.

The first three sections, i.e. the *source, content* and *action* is obtained from the PSL syntax. The last section, the *subject,* is obtained from the Enterprise Authority model (EA) described in the earlier section.

Each policy expressed in PSL must have a policy name, a policy specifer, an EA model and the <event-clause> and <action-clause>. However, the <condition-clause> is optional. The *policy name* should be unique as polices are identified in the system based on their names. The name bears no influence on the interpreted semantics of a policy. However, it is advisable to adopt some standard conventions to name policies so as to be mnemonic. Since policies are specified by *specifier objects*, the *source section* contains the identity of the *specifier object* and its EA Model. The *specifier object* in the policy source must belong to the Enterprise Authority (EA) model that is provided. A policy's scope, or the set of *subjects* that are required to obey the policy, is automatically determined from the EA model. This relieves the user from identifying *subjects* for a policy. It also implicitly restricts the authority of *specifier objects* by permitting it to create policies that are directed only at entities that it has authority upon in the EA model (with a *direct* or *indirect* edge).

The content-section consists of the <event-clause> that can be used to monitor one or more event expressions. The <condition-clause> compare attributes of events monitored in the <event-clause> against constants or values of the same type. A policy is activated if the event-clause and condition-clause evaluate "TRUE". The condition-clause can refer to event attributes monitored in the event-clause. We use the popular "." (dot) notation to refer to the attributes of an event or program object. The <action-clause> operates on program or event objects to invoke their methods or forbid the monitored event from initiating any further actions into the policy system. When there is multiple <action-clauses>, they are executed in the order they are listed on

the <action section>. Since the EA model verifies the authority of entities on various objects unless explicitly forbidden by means of "forbid" using a PSL rule, actions by default is assumed to be authorized.

3.3.2 Two Level Specification of Authorization Policies

Authorization policies define what a set of entities are permitted ornot permitted to do. An entity's authority to initiate an enterprise process is controlled by issuing appropriate authorizations (permissions or prohibitions) on one or more program or/and event objects. When providing such authority we identify three possible cases:

Case 1: provide no authority to initiate an enterprise process (i.e., not allowed to send alarms)

Case 2: provide partial authority (eg:: allowed to send only "low" priority alarms)

Case 3: provide complete authority (eg: allowed to send all types of alarms like "critical", "high" or with "low" priority)

In our model, we manage the first and third cases (which are to provide "no authority" or "complete authority") by adding or removing an edge to an object in the EA model. We will henceforth refer these two cases as *simple authorizations*. And the second case as, a *partial authorization*. To specify *partial authorizations*, we use a combination of EA model and PSL in the following way:

Step 1: In the EA model provide an edge from the entity to the appropriate program or event objects (Eg: an edge between "employee" to "BudgetRequest" object results in authorizing the employee to request for budget clearance).

Step 2: Specify a PSL rule, and in it's <content-clause> monitor the event's pattern (Eg: BudgetRequest.amount > 100000) and apply primitive action "forbid" in the <action-clause>. This results in rejecting the event from initiating any further action. In this way, the PSL <content-clause> is used to filter event patterns thus providing a mechanism to specify *partial authorizations*.

Sloman's and Marriot's [Sloman][Marriot] Policy Notation supports positive (+) and negative (-) policies for each modality: authorization (A) and obligation (O). And hence there are four types of policies: A+, A-, O+ and O-. This was examined in detail in Section 2.1 of this thesis. Compared against Sloman's Policy Notation, Koch's PDL [Koch] supports only two type of policies: authorization (A) and obligation (O) policies. Our proposed PSL, is similar to Koch's

work and it supports two types of policies namely, authorization and obligation policies. However, we further classify authorization policies as *simple* and *partial* and provide a two level specification mechanism that uses the EA model to specify *simple authorizations* and a combination of EA Model and PSL to specify *partial authorizations* and *obligation policies*. This has the following advantages over Sloman's Policy Notation [Sloman] and Koch's PDL [Koch]:

1. *simple authorizations* can be managed at a higher level rather than at the level of operational policies, thus making it easier for policy creators (eg: managers)
2. managing *simple authorizations* in the EA model reduces the number of authorization rules at execution level. With lesser rules for execution, the policy set is smaller and hence the speed of execution is faster
3. Ensures that every PSL policy action has prior authorization in the EA Model. Unlike in Sloman's and Koch's work, in our case no redundant obligation policy can exist without a prior authorization.

In the next section we will examine types of conflicts that can occur in PSL, how they are detected and resolved.

3.4 Conflict Detection and Resolution among Polices

When a new policy is defined or when an existing policy is updated, it may create new inconsistencies in the policy set. Therefore before accepting a new or an updated policy, it is important to verify if it is consistent with the existing policy set. In this thesis, we will address one such inconsistency namely, policy conflicts. Other types of inconsistencies such as duplicate, subsumed, supersumed and inferred [Sobieski] policies that may arise is beyond the scope of this thesis work.

We give below definitions that we will use later in this section to define conflicts in PSL policies. The definitions are based on EA Model shown in Figure 5 (page 26) of this chapter.

Definition 1: (Identical Source) *Between two policies, the source sections are said to be identical if they contain the same specifier objects belonging to the same enterprise authority (EA) model.*

Example:	*Policy P1;*	*Policy P2;*
	Specifier: "Manager"; EA-1;	*Specifier: "Manager"; EA-1;*
	.....p1-content.....	*.....p1-content .....*

	....p1-action....	*.... p1-action ...*

In the above example: P1-source and P2-source are identical since they are specified by the *specifier object* "manager" belonging to enterprise authority model EA-1.

Definition 2: (Identical Content) *Between two policies, the content sections are said to be identical if they monitor the same set of events in their content sections.*

Example:	*Policy P1;*	*Policy P2;*
	.....p1-source ...;	*.....p1-source ...;*
	When Alarm1 arrives	*When Alarm1 arrives*
	....p1-action ... ;	....p1-action ... ;

Definition 3: (Source Contained) *Between two policies P1 and P2, the source of policy P1, is said to contain the source of the other policy P2, if P1's specifier object has authority (i.e., a direct or indirect edge in the EA model) over P2's specifer object in the source section.*

Example:	*Policy P1;*	*Policy P2;*
	Specifier: "Manager"; EA-1;	*Specifier: "Salesman"; EA-1*

In the above example: P1-source contains P2-source since in enterprise authority model EA-1 the "manager" has a direct edge to "salesman". Also, in this case, the *subjects* of policy "P1" is a subset of the *subjects* of policy "P2".

Definition 4: (Content Contained) *Between two policies, the content of one policy is said to be contained in the content of the other, if it monitors only a sub-set of events monitored by the other policy.*

Example:	*Policy P1;*	*Policy P2;*
	.....P1- Source ...;	.....P1- Source ...;
	When Alarm1 arrives	*When Alarm1 arrives*
	....P1- Action ... ;	*if Alarm1.priority = "high";*
		....P1- Action ... ;

In the above example, policy "P1" monitors all types of "Alarm1" events. However, policy "P2" monitors only "Alarm1" events with "high" priority. Whenever policy "P2" executes, policy "P1" also executes, but the reverse is not true. In the above example, P1-content *contains* P2-content.

The underlying mechanism for conflict detection is a two step process described below:

Step 1: identify conditions under which conflicts exist; i.e., to recognize ***conflict conditions.***

Step 2: determine the truth-value of the conflict condition – if it is TRUE then a conflict is detected. Else, conflict does not exist.

For **Step 1** of the above process, we identify two types of PSL conflicts that can occur: *authorization conflicts* and *action conflicts*. We apply the definitions discussed in this section to describe the two PSL conflict conditions:

Definition 5: (Authorization Conflict Condition) If a policy P1, has an *identical* or *contained source* **and** an *identical or contained content* section with another policy P2 and P1 has the primitive action *forbid* in its <action-clause> but P2 has in its <action-clause> an event or program object then an authorization conflict exists.

Example:

Policy P1;	*Policy P2;*
"Manager"; EA-1;	*"Salesman"; EA-1;*
When Alarm1 arrives	*When Alarm1 arrives*
If Alarm1.priority = "low";	*If Alarm1.priority = "low";*
Then Notify(entityID:1, conflictID:10;);	*Then forbid;*

In the above example, P1 and P2 monitor the same event and their *subjects* overlap since the "Manager" has authority on "Salesman" and they belong to the same EA model. Hence for the same set of *subjects*, P1 has a "Notify" action but P2 has a "forbid" action. Their authorizations are conflicting and hence only one of the two policies can exist at the same time in the policy set.

Definition 6: (Action Conflict Condition) If a policy P1, has *identical* or *contained source* **and** *identical or contained content* section with another policy P2 **and** if objects in P1's <action-clause> has in its *conflictID* attribute, the *entityID* of objects in P2's <action-clause> then an *action conflict* exists. Consider the example:

Policy P1;	*Policy P2;*
"Manager-1"; EA-1;	*"President"; EA-1;*
When StockAlarm arrives	*When StockAlarm arrives*
If Stock.Value < 100; then	*If Stock.Value < 100; then*
Buy(entityID:10 conflictID:5);	*Sell(entityID:5, conflictID:10);*

In the above example, P1 and P2 monitor the same event and their *subjects* overlap but their actions are conflicting (since they have each other's *entityID* in their *conflictID* attributes) and hence only one of two policies can exist at the same time in the policy set.

For **Step 2** of the Conflict Detection process, we determine the truth-value of conflict-conditions. We recognize that this can be established at any of the following stages:

- *at the time of accepting new or updated policies*: when new policies are specified and before they are accepted into the system their actions are verified if they are consistent with actions of existing policies in the policy set.
- *at the time of execution of policy actions*: policy actions that cannot be true at the same time or within a given time window [Lobo] are detected and resolved before policies are executed.

Detecting and resolving conflicts at the time of specification ensures that all policies in the policy set are consistent at any point of time thus making maintenance of policy set easier. In this thesis, we attempt to detect and resolve conflicts at the time of accepting new policies, that is, at the time of specification. At this level the detection mechanism actually detects*possible* conflicts because they are yet to be accepted for interpretation purposes. However, in our discussions, for ease of use we will use the term conflicts instead of *possible* conflicts.

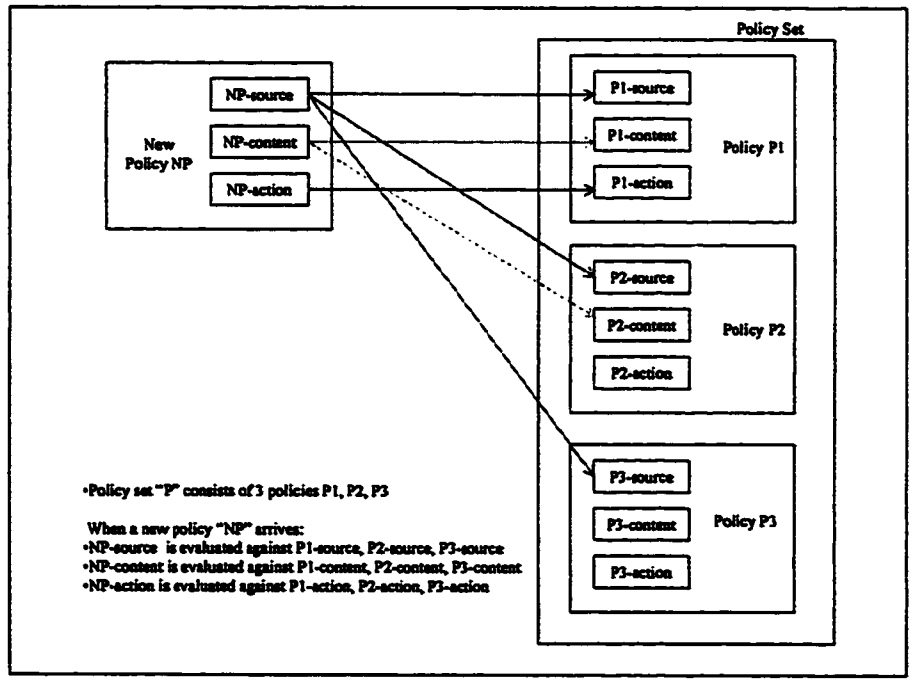

Figure 7: Conflict Detection Mechanism

Conflict conditions in PSL are detected by comparing sections (source, content and action sections) of new or updated PSL policy (NP) to respective sections of existing PSL policies in policy set. This mechanism is shown in Figure 7 above where all three sections (source, content and action) of "NP" is matched against respective sections of policy P1.

The algorithm for conflict detection is shown in Figure 8 below:

```
NP-reject = FALSE;                          // Is the new policy rejected?
I = 1;                                      // first policy in the policy set
Integer N;                                  // Number of policies in the policy set

WHILE ( (I <= N) && (NP-reject != TRUE) )
{

  WHILE ( NP-source <= Pi-source )          // If the <source-section> is identical or -
                                               contained (Definition 1 and 3).
  {
    IF (NP-content <= Pi-content)           // If the <content-section> is contained or -
    {                                          identical (Definition 2 and 4).
      IF ( NP-action X Pi-action)           // If the <action-section> Conflict.
                                               (Definition 5 and 6).
        PRINT ("Conflict Detected");        // Conflict Condition Exists
        NP-reject = ConflictResolve(NP, Pi); // Initiate Conflict Resolution Mechanism -
      }                                        it returns "FALSE" if new policy -
    }                                          is rejected else returns "TRUE"
  } // end of while - 2

I = I + 1;                                  // Index incremented to access next policy

}  // end of while - 1
```

Figure 8: PSL Conflict Detection Algorithm

When a conflict is detected, the conflict resolution function is initiated before proceeding to the next step. The resolution mechanism obtains the authorities of the *specifier objects* involved in the conflicting policies from the EA model. Based on the relative authorities, the resolution mechanism either rejects the new policy (NP) or replaces an existing policy with the new policy (NP). When relative authorities are obtained there can be four possible outcomes:

1. Greater than (>), i.e., NP-source has a higher authority than Pi-source
2. Less than (<), i.e., NP-source is less than authority of Pi-source
3. Equal to (=), i.e., NP-source has authority equal to Pi-source
4. Cannot be determined (??), i.e., the relative authorities of two sources cannot be determined

Based on the above outcome, the proposed resolution mechanism obtains the following result:

1. (NP-source > Pi-source) => NP replaces Pi

   In this case, the conflict resolution function returns "NP-reject = FALSE" and as a result the conflict detection mechanism continues to check the next policy as illustrated in the algorithm provided in Figure 9.

2. (NP-source < Pi-source) => NP is rejected

   In this case, the conflict resolution function returns the value "NP-reject = TRUE" and as a result the conflict detection mechanism is stopped.

3. (NP-source = Pi-source) => NP Rejected

   This case occurs when two "managers" who have the same authority in the EA model specify conflicting policies for an employee. The new policy is rejected and the conflict resolution function returns the value "NP-reject = TRUE" which stops conflict detection mechanism.

```
boolean ConflictResolution (NP, Pi)
{
   IF ( NP-source < Pi-source ) {
    PRINT ("Rejecting New Policy");
    return TRUE;
   }
   ELSE IF ( NP-source > Pi-source) {
    PRINT("Conflict Resolved - Replacing existing policy with New Policy");
    return FALSE;
   }
   ELSE IF ( NP-source = Pi-source) {
    PRINT ("Rejecting New Policy ");
   return TRUE;
   }
  ELSE {
    PRINT ("Rejecting New Policy ");
   return TRUE;
   }

} // end of function
```

Figure 9: PSL Conflict Resolution Algorithm

4. (NP-source ?? Pi-source) = > NP rejected

   The authorities between S1 and S2 cannot be determined when *specifier objects* are from different EA models. In this case, the new policy is rejected and the conflict resolution function returns the value "NP-reject = TRUE" resulting in the termination of conflict detection mechanism.

3.5 Policy Set Incremental Maintenance

An organization constantly evolves or due to re-organization, new business processes, employees or roles maybe added or deleted. Such changes involve modifications (eg: adding to new edge, deleting an edge, deleting a node etc) to the EA model. However, changes to the EA model will create new inconsistencies in the policy set as there maybe policies that are specified by an entity which now maybe deleted thus making the policy redundant. *Incremental Maintenance* involves a constant update to the policy set whenever the EA model changes. When a policy is specified by an entity, it is *owned* by that entity. Depending on the authority of the owner, its scope is established. Given an object and an activity on that object, the incremental maintenance function detects policies in the policy set that may now become obsolete or create new exceptions as a result of this activity on the object. There are three possible situationswhere there is a need for incremental maintenance function:

Situation 1: *when program or event objects are deleted:* When program or event objects are deleted, policies may exist in the policy set that use these now non-existent objects. These policies may never be executed at all or may create exceptions when an attempt to execute is made. When such policies accumulate in the policy set, they will slow down the efficiency of execution and also make the task of maintenance tedious.

Situation 2: *when specifier objects are deleted:* When an existing employee leaves the enterprise or when a new employees arrives, a *specifier object* may be added or deleted depending on the occasion. However, when deleted *specifier objects* continue to own policies in the policy set, it will gives rise to inconsistencies between the EA Model and policy set resulting in unexpected results during policy execution.

Situation 3: *when an authorization is removed:* When an edge in the EA model is deleted, it affects the existing policy set. For example, an employee may have created a policy to handle new alarms from network. If at any later stage, if the employee's authority to operate on "Alarm" objects is removed, then the existing policy becomes inconsistent with the EA model.

Therefore, in the above circumstances, there is a need to perform some maintenance tasks on the policy set such as removing *dead* and inconsistent policies that may exist after modifications to the EA model. The following maintenance tasks are recommended on the policy set to maintain consistency with the EA Model:

Recommendation 1: If a *specifier object* is deleted from the model, all policies that are owned by this deleted object are automatically removed from the policy set.

Recommendation 2: Similarly, if an *event* or *program* object is deleted, all policies that use these now deleted objects in their *content* or *action* sections are detected and removed from the policy set.

Recommendation 3: If a *specifier object's* authorization on an *event* or *program* object is removed, policies that are owned by this *specifier object* using these *event* or *program* objects are identified and removed from the policy set.

In this way, the EA model and the policy set can be kept consistent with each other. In the next section we examine an architecture that supports the EA Model, PSL, Conflict Detection and Resolution mechanism and Incremental Maintenance tasks discussed so far.

3.6 Enterprise Policy Specification Tool (PST) Architecture

The system architecture shown in Figure 10 is aimed to provide a framework to enable enterprise entities to manage policies, detect and resolve any policy conflicts that may arise at the time of specification and support incremental maintenance tasks on policy set. The architecture aims to, support specification, analysis (conflicts) and representation of policies.

The client/server architecture shown in Figure 10 has *six* modules: *specification module, EA model handler, conflict handler, policy handler, policy engine* and the *event handler*. There are three possible types of inputs to the system, they are:

- policies (add/delete/modify policies)
- EA Model changes (add/delete/modify enterprise entities or to add/remove authorizations)
- events (this may be generated either by system monitors or by a human user or as a result of any other policy action)

The *specification* module provides a graphical user interface to specify or update PSL policies and the EA model. The policy database provides a persistent place to store policies. When the policy system starts up, the *policy handler* extracts PSL rules from the *policy database* to create a list data structure of the policy set as illustrated in the Figure 11. Each section (*source, content* and *action* sections) of each PSL rule is stored as a separate node in the list. The links between the nodes are self-explanatory from the figure.

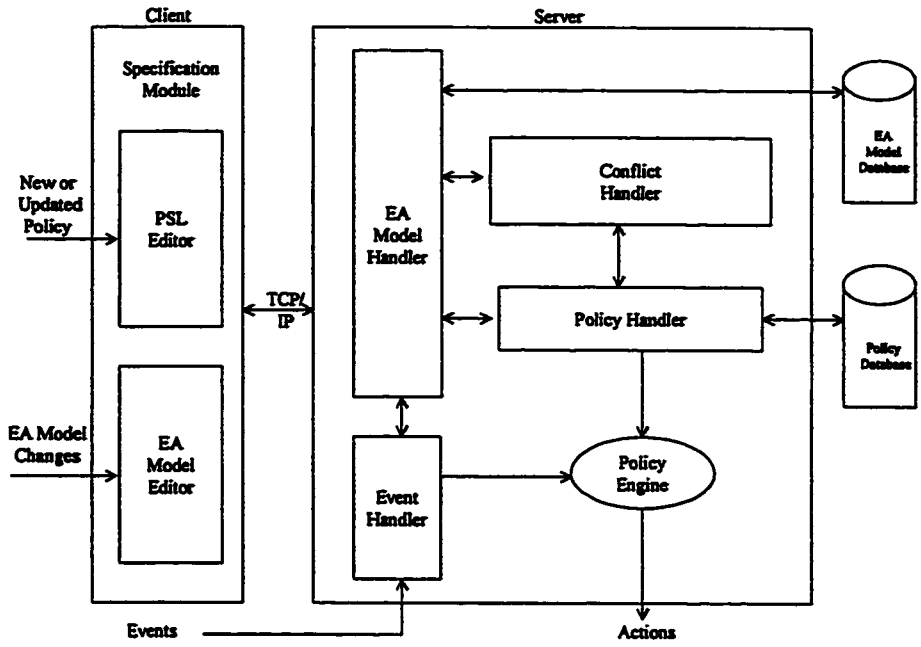

Figure 10: PST Architecture

The *object definition database* provides persistent information on all entities in the enterprise and relationship between them. The model handler interacts with the object definition database to perform verification tasks and to extract the authority relationships between various entities. When a new policy arrives or when an existing policy is updated using the specification module, the *model handler* verifies the authority of the *specifier object* over the *event* and *action* objects in the new/updated policy. If the verification fails, the policy is rejected immediately and the specification module informs the user of this decision. If the verification succeeds, then policy is passed over to the *conflict handler.*

The *conflict handler* identifies if the *authorization conflict condition* (Definition 5 of Section 3.4) is TRUE or FALSE. It compares source, content and action sections of the new/updated PSL rule with respective sections of existing PSL rules to ascertain the conflict condition. As a first step, its compares the *source* section of a new policy with the source section in the node in the top-left corner in Figure 12. If the two sections are *not identical*, it continues to traverse in a horizontal direction and perform similar check on the next policy. It continues to traverse in the horizontal direction until it encounters a node with an identical source section. If it completes the traversal without such an encounter it means that there is no possibility of conflict condition and the new policy can be accepted into the policy set.

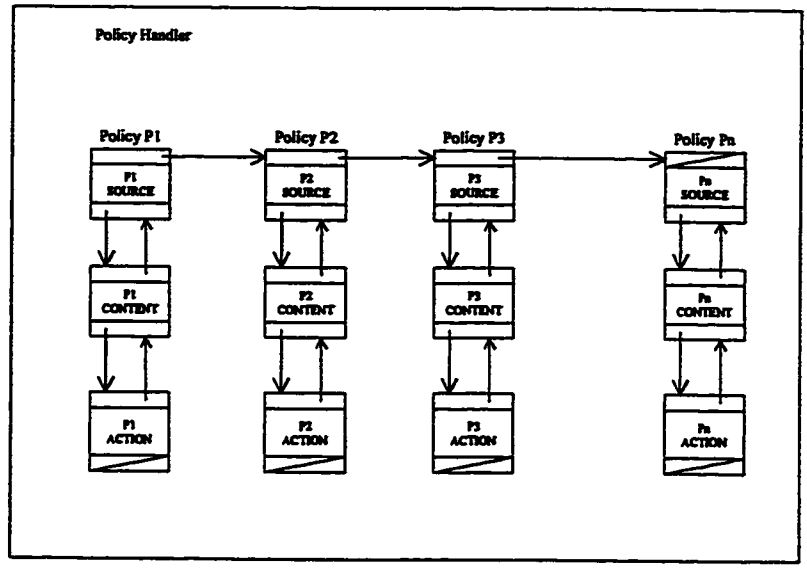

Figure 11: Policy Set Data Structure

However, if it detects an identical or contained section, the direction of traversal is changed and it continues in the vertical (depth-first) direction for the same policy. It then compares the PSL content sections. If the content sections are found to be different, then the vertical traversal stops to continue with the next policy as described earlier. If content sections are identical or contained, then the traversal continues in the vertical (downward) direction. From the Figure 12, it can be seen that if the traversal exits in a vertical direction, then the conflict condition is TRUE and the conflict resolution mechanism is initiated. If the traversal exits in the horizontal direction (breadth-first) then the conflict condition is FALSE and the new or updated policy is ready to accepted into the system. With some minor modifications, this mechanism can be used to detect *action conflict conditions* (Definition 6 of Section 3.4).

The conflict resolution mechanism obtains the authorities of the *specifier objects* involved in the conflicting policies from the EA model. Based on the relative authorities, the resolution mechanism either rejects the new policy or replaces an existing policy with this new policy. Once the policy is stored into the repository, the data structure in the conflict handler is updated to create new set of nodes for the new policy

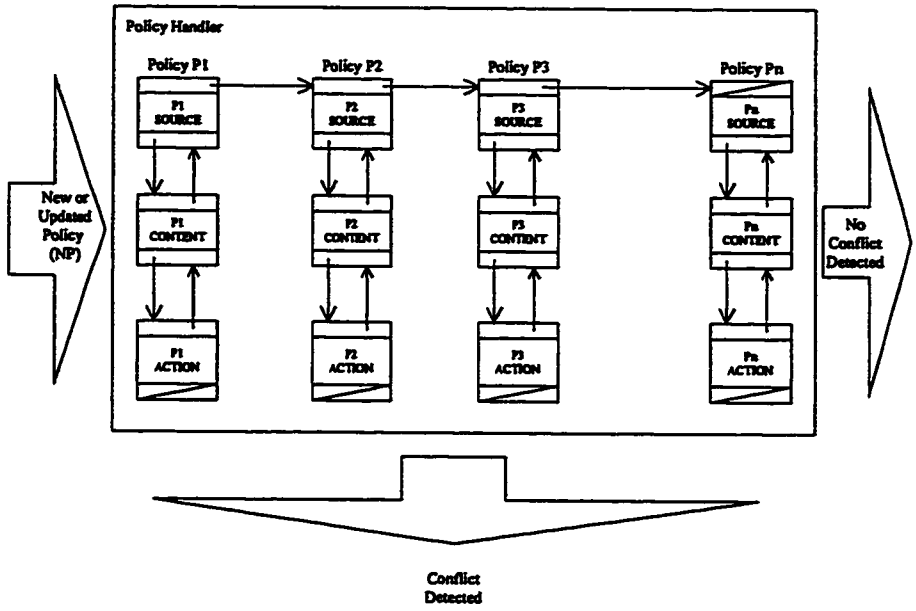

Figure 12: Policy Handler

The *policy handler* is also responsible to provide feedback to the EA model if a change to the model will create new inconsistencies in the policy set. When a request to delete an object or to remove existing authorizations arrives, the model handler first verifies the authority of the user to perform these actions. If the verification succeeds it interacts with policy handler to identify if any existing policies may become redundant if the action is performed. Policy handler identifies such policies by traversing the policy list. If no policies are affected then the task of maintenance is not required. However, if affected policies are detected, the policy handler removes them from the policy set and updates the policy database and the policy engine.

The *event handler* constantly monitors for events that may be generated by a human user or as a result of an operation on a program object or due to any other policy action. These events received are directly fed into the *policy engine*, which is the heart of this architecture. The *policy engine* is responsible to match events against policies and initiate policy actions to be executed in the system environment. EA model handler module is also responsible to propagate the changes in the EA model to the policy handler module, thus working to maintain the consistency of the policy set and EA model.

3.7 Chapter Summary

In this chapter the EA Model has been introduced and demonstrated with examples. A Two-Level Specification mechanism was defined that uses the EA Model to specify simple authorizations and a combination of EA Model and Policy Specification Language (PSL) for partial authorizations and obligation policies. When a policy is specified using PSL, the*subjects* that are required to interpret the policy is obtained automatically from the EA Model based on the policy creator's authority in the EA Model. Furthermore, we have identified and defined policy-conflict conditions in an enterprise and proposed a mechanism to detect and resolve them. Our conflict resolution mechanism identifies precedence between two conflicting policies based on relative authorities of entities that created the conflicting policies. We identified situations in the enterprise that will require incremental maintenance of policy set and recommended maintenance actions for them. Finally, we proposed an enterprise Policy Specification Tool (PST) architecture is based on the proposed EA Model, Policy Specification Language (PSL), Conflict Detection and Resolution Mechanism and Incremental Maintenance needs.

4.0 Application of Enterprise Policy Specification Tool

In this chapter we will describe the application of PST to a Message Notiification System (MNS). Message Notification Systems [Curamessage] [Keryx] [Telalert] can deliver messages in real-time using multiple delivery media (by voice, text or numeric pager etc). The term real-time is used to distinguish a Message Notification System from a traditional email system, which uses the store and forward technology where the time of delivery is not guaranteed. In the first section of this chapter, we will describe Message Notification Systems (MNS) and illustrate with an example, a message notification process. In the following section, we will discuss advantages and disadvantages of real-time message notification in an enterprise scenario and derive the need for policy-based message notification. In the next section, we propose an architecture that integrates PST with Message Notification System (MNS) to provide a policy-based message notification solution. Finally in the last section, we demonstrate the application of our integrated architecture for policy-based notification of messages in the context of real-time communication between two enterprises in a *helpdesk* scenario.

4.1 Message Notification System (MNS) Description

Inputs to a Message Notification System (MNS) may come from network events, system monitors or a human user. When inputs arrive, the system generates a notification to the recipient and is sent through one of the delivery media like pager, or voice mail. The recipient would call back to MNS to acknowledge that the notification was received. A 1-800 number is provided by the system for recipient acknowledgements. If the intended recipient does not acknowledge the receipt of a message, the MNS server would then send another notification to the next receiver on a list who is supposed to receive that message as an alternate receiver. This action is called an *escalation*. Escalations help to ensure that important messages do not remain unattended by the recipient.

Users of a Message Notification System (MNS) assume three different roles while interacting with the system. They are:

- as a *sender (maybe a human user or a system event)*
  - who is registered with the MNS
  - who sends requests to MNS
  - who is allowed to define service parameters to be met by the system
- as a *recipient*
  - who is a registered user with the MNS
  - who receives notifications sent by the senders
  - who acknowledges a message delivery
- as a *administrator*
  - who maintains the MNS Server for user registration
  - who maintains the MNS Database

A request to MNS consists of a message component and an escalation list. The following Figure 13 shown below illustrates different parts of a notification request:

Text/Numeric Message	Priority	Severity
Recipient name 1	Delivery Medium 1	Delay 1
Recipient name 2	Delivery Medium 2	Delay 2
Recipient name 3	Delivery Medium 3	Delay 3

Figure 13. Parts of MNS request

The top row in the Figure 13 shown above is called the *message component*. The remaining three rows constitute the *escalation list*. The *message component* consists of the following, mandatory elements:

- *Text message:* a text message indicating the content of the notification being sent. For example, a typical message could be "SQL Database Server B has experienced a fatal hard disk drive failure". If the recipient has a text pager, the contents of the text message will be transmitted to the pager. If the recipient is using voice mail, the text message is synthesized into speech before being transmitted to the recipient.
- *Numeric Message:* A sequence of numeric digits that is interpreted by the recipient. This could be the phone number that the recipient can call back.

- *Message Severity:* A numeric value (1 - high severity, 2 – low severity) indicates the importance of the message. A severity level of 1 indicates to MNS that if all of the recipients on the escalation list have been notified and no one has acknowledged the notification, the escalation list will be cycled again. This will continue with progressive delay in retrying. For example, if no one acknowledges an escalation during the first pass, MNS will cycle through the list of recipients again according to the delay value set in the escalation, say after 5 minutes. If the notification is still not been acknowledged after this second pass, a third pass will occur in another 10 minutes and so on.
- *Message Priority:* A numeric value (1-10) indicates the importance (eg: important customer) of the message with respect to the MNS system. That is, a request with a higher priority value would be given preference over other requests waiting to be processed.

- *Escalation list:* It lists of one or more message recipients contained in a notification is referred to as an *escalation or an escalation list.* An *escalation sequence* refers to the order in which the recipients of a message are notified. The escalation list of a notification consists of the following mandatory elements:
  - *Recipient name:* The name of the recipient for the message. MNS checks the name provided in the escalation list against a list of registered users contained in a database external to the MNS software. An escalation list can contain one or more recipients.
  - *Delivery medium:* The delivery type indicates what type of delivery method or service is used to contact the individual named as the recipient. This can be any of the following delivery mediums:
    - Text pagers, Numeric pagers or Voice mail systems
  - *Delay:* The value indicates how much time to allow for the named recipient to respond to the notification with an acknowledgement. If the first recipient in the escalation list does not acknowledge the notification with this delay, the notification is escalated to the next recipient in the list, if one is available. If the notification is not acknowledged once the escalation list is exhausted, MNS checks the severity level of the notification to determine whether or not to cycle through the list again. If the severity level is set to "1", the escalation list is cycled through again in 5, 10, 20 and 60 minute increments. If the severity level is set to "2", there are no further attempts to cycle through the escalation list.

In the Figure 14 shown below, we trace a message notification process from its origin to delivery.

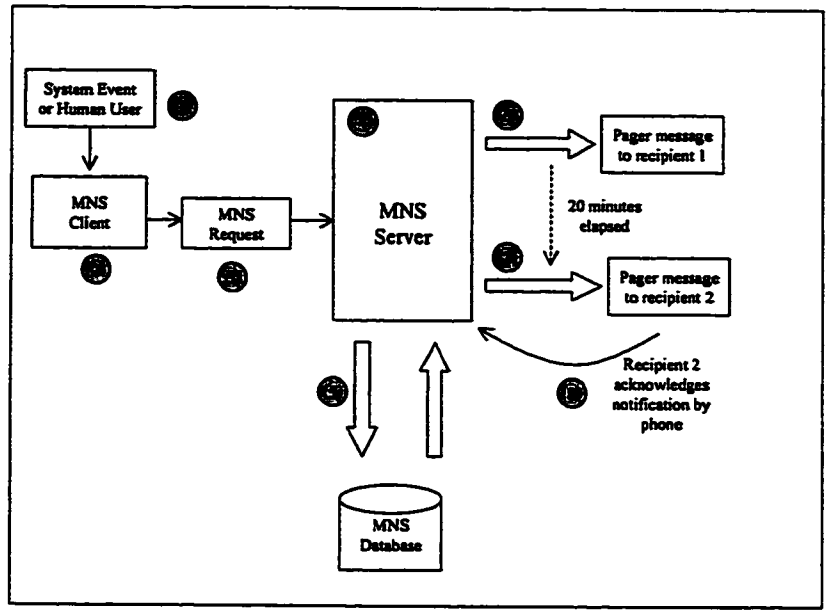

Figure 14: An Example Message Notification Scenario

When notification requests are received by MNS Server, it looks up the MNS database to verify if the sender and intended recipients are registered users. If the verification succeeds it sends notification to the first recipient on the escalation list. MNS cycles through recipient list if the first recipient does not respond within allotted delay time. The notification remains in the MNS delivery queue until the escalation list is exhausted or until the notification is acknowledged or cancelled. When first recipient acknowledges a notification, the notification is removed from the queue and it is marked as delivered and the activity is logged.

4.2 Need for Policy Based Notification

When deployed in an enterprise, message notification systems are used for communication within the units of the enterprise, between units or with clients. Customer support groups rely heavily on notification systems for quick response to problems in mission critical applications.

Enterprises use notification systems to be able to bring to a recipient's attention an important message in a guaranteed time frame. However, to effectively use MNS, there is a need to ensure that messages are sent to the right recipient who has the necessary expertise and is presently available to attend to a request. It is also essential that employees are not flooded with high priority messages with short delay time between escalations.

An enterprise's available resources to attend to a task at any point of time is constantly changing as employees are assigned to new tasks or are presently on vacation or no longer belong to the enterprise. Due to these reasons, *senders* of notification requests are not in a best position to identify the escalation list that determines the alternate recipient if the first recipient does not acknowledge the receipt of notification. Also, when senders compete for recipient's attention, notifications that are perceived by recipient as *low priority* in nature maybe sent as *high priority* with *severity one* and short *delay* values. This may result in the recipient being flooded with incoming messages making it difficult for him or her to prioritize given tasks. In some cases it may require that a delivered message needs to be forwarded to any other recipient who has the appropriate expertise to address the problem. Such scenarios introduce unpredictable delays in the notification process and also affect the efficiency of other users (as a result of managing unwanted escalated messages) of the system thus compromising the timeliness in *real-time* message notification.

Enterprises need to ensure that employees are assigned tasks, not by notification senders but rather by enterprise managers who are aware of the type of problem, the expertise required and available resources that can be deployed at that point in time and place. In the following section, we propose an architecture that integrates PST (described in Section 3.5 of this thesis) with message notification systems to provide a policy-based message notification solution. This integrated architecture allows different units within an enterprise to manage their own policies to handle incoming notification requests.

4.3 Integrated Architecture for Policy Based Notification

As shown in Figure 15, in our proposed integrated architecture, PST serves as a front-end to message notification systems (MNS). In PST, a notification request is modeled as an incoming *event*. Enterprise policies to handle these events are stored in the *policy database*. The user initiates a notification request by generating the notification event that is received by the *event*

handler module (refer figure 10, page 40) and propagated to the *policy handler* module for policy enforcement.

In the rest of this chapter, we will call notification requests to PST as "Contact" requests so as to be able to distinguish it from parts of a notification request to MNS (refer Figure 13 on page 45) discussed earlier.

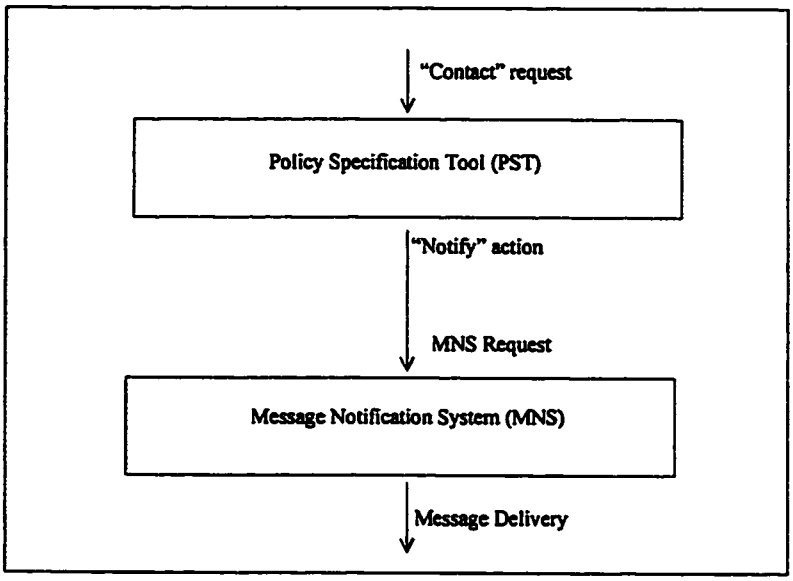

Figure 15: PST and MNS Integrated Architecture

A "Contact" request provides the users with only sub-set of attributes when compared to a MNS request. The parts provided by a "Contact" request is given below:

- text/numeric message, recipient –1, delivery –1 and priority

The remaining parts that are required to generate MNS request is supplied policies written in PSL. This is mechanism is illustrated in Figure 16. The additional parts provided by PSL are the following:

- severity, delay 1
- recipient list 2 (recipient-1, delivery method-1, delay-1)
- recipient list 3 (recipient-1, delivery method-1, delay-1)

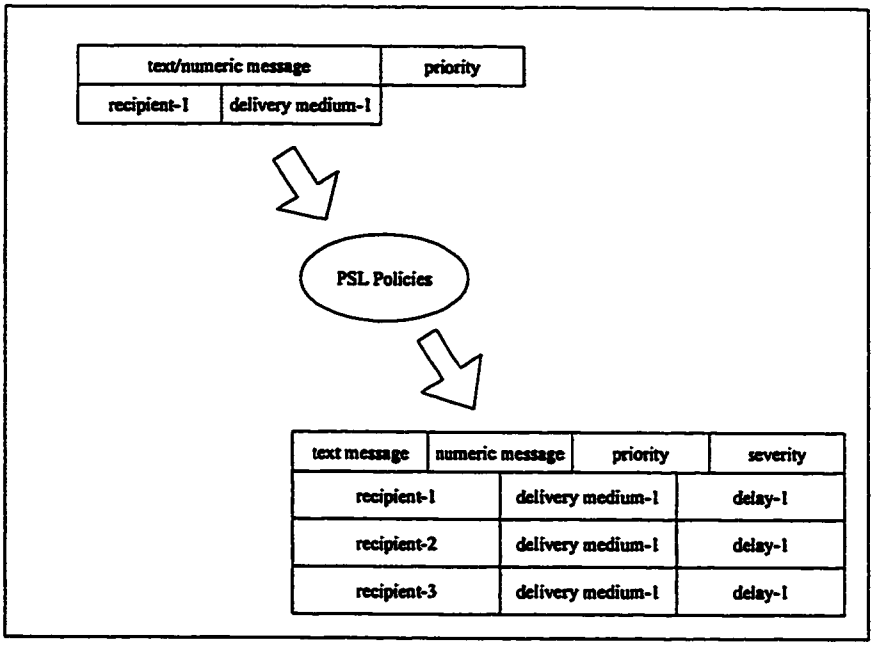

Figure 16: Policy-based Message Notification Mechanism

In this mechanism, not the *sender* but the PSL policies that determine the severity, escalation list and delay value for a notification. PSL policies, for example, may compare the message component, priority or time of day attributes of a "Contact" request to recognize the request to be of severity "one " in nature and identify the escalation list and delay values. In addition to performing the task of identifying parts for a MNS requests, the PSL policies can also determine whether to accept or reject a "Contact" request. If the recipient of a "Contact" request is presently not available, then PSL policies can overrule the *recipient* attribute and in turn create MNS requests that are intended for any other recipient. If there are no PSL policies for a "Contact" request then by default, the MNS request generated is a copy of the "Contact" request. In this case, the MNS request will have no escalation list (a "Contact" request does not have one), it is delivered with a severity value of "2" (low) and delay of "0". The *delay* value is of no significance here since there is only one recipient and the severity is low. In a policy-based MNS system, since the parameters of a notification request (like escalation list, severity, priority) are determined by the PSL policies it allows the unit heads to control (by creating PSL policies) the message notification process and ensure that notifications reach the right person at the right time using the most appropriate delivery media.

4.4 Case Scenario: Application of Integrated Architecture

We will consider two enterprises "Enterprise A" and "Customer A" where "Enterprise A" is a computer service company that provides software and hardware support to employees of "Customer A". In a human driven *helpdesk* environment, to seek the services of "Enterprise A", an employee belonging to "customer A" calls up a published telephone number. The call is answered by an employee belonging to "Enterprise A" who identifies the problem and assigns it to one of the available experts who in turn contact the originator of the request to seek additional information or to resolve the problem or to re-direct it to to any other expert.

In this *helpdesk* scenario, message notification systems maybe deployed to prevent avoidable delays at "Enterprise A" in directing calls through a centralized human operator and to provide customers with the flexibility to directly reach intended experts in *real-time* by reaching out to them via multiple delivery media. In the earlier section, we discussed the disadvantages of message notification systems and derived the need for policy-based message notification. In this section, We will now demonstrate how the two enterprises in discussion, namely "Enterprise A" and "Customer A" implement a policy-based notification solution.

4.4.1 GEA Model

As a first step, the two enterprises namely, "Enterprise A" and "Customer A" model their entities and resources using the GEA Model as described in Chapter 3 of thesis. The GEA Model shown in Figure 17 describes the relationships between enterprise entities involved this case scenario.

Since our aim in this section is to demonstrate the policy specification and enforcement issues we denote employees (instead of their role) as nodes in the GEA model. However, for large enterprises role-based modeling is recommended due to reasons discussed in Section 3.1 of this thesis. The mechanism to relate employees to roles is beyond the scope of this thesis.

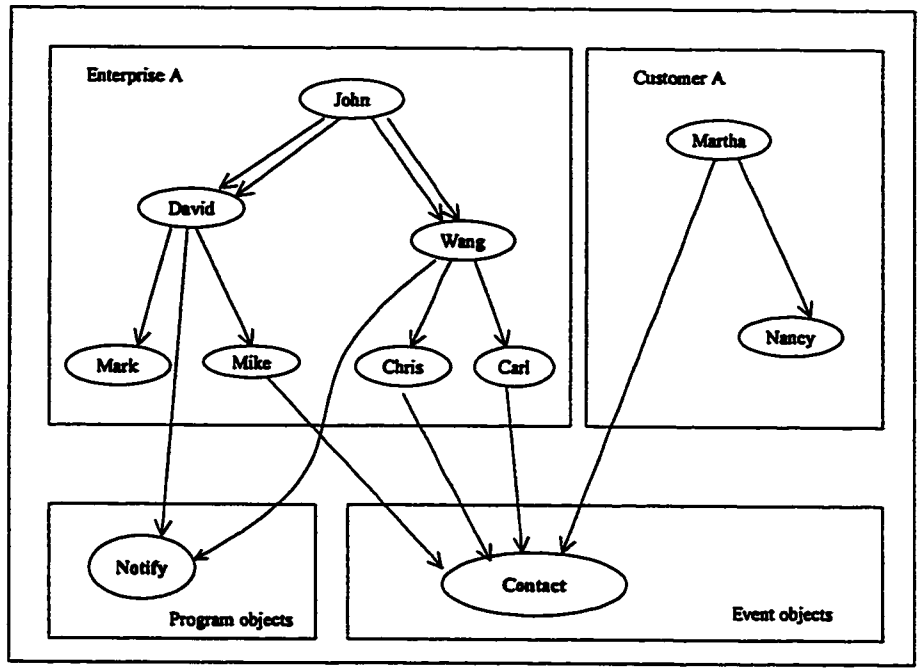

Figure 17: Case Scenario: GEA Model

In the Figure 17 shown above, unit heads ("David, "Wang" and "John") of "Enterprise A" have edges to both "Contact" and "Notify" objects and hence have the authority to specify policies to handle incoming "Contact" requests and to initiate "Notify" actions on them. Policies specified by these objects are applicable only to entities to which they have direct or indirect edges. Hence, policies specified by "John" are applicable to all entities in the enterprise. However, policies specified by "David" are applicable only to employees "Mark" and "Mike" and those specified by "Wang" are applicable to employees "Chris" and "Carl". An *edge* from the employee "Martha" in Customer A" to the "Contact" entity gives her the authorization to initiate "Contact" requests.

4.4.2 Policy Specification

In this section, we will demonstrate how unit heads of "Enterprise A" ("David", "Wang" and "John") use PSL to create policies that determine the escalation list and severity of an incoming "Contact" request and initiate a request to MNS.

Example 1: "David" uses the following PSL statement to specify a policy to handle "Contact" requests with "high" priority to employees ("mark" and "mike") under his authority:

Policy P1;

David; Enterprise A;

When Contact arrives;

If Contact.priority = "high";

Then **Notify** (message = Contact.message; priority = 2; severity =2;

recipient1 = "Contact.recipient"; delivery1 = pager; delay1 = 10;

recipient2 = "David"; delivery2 = pager; delay2 = 20;

recipient3 = "Wang"; delivery3=pager; delay3 = 30;

)

Example 2: "Wang" uses the following PSL statement to specify a policy to handle "Contact" requests with "critical" priority sent to employees ("Chris" and "Carl") under his authority:

Policy P2;

Wang; Enterprise A;

When Contact arrives;

If Contact.priority = "critical";

Then **Notify** (message = Contact.message; priority = 1; severity =1;

recipient1 = "Contact.recipient" ; delivery1 = pager; delay1=10;

recipient2 = "Wang"; delivery2 = pager; delay2 = 20;

recipient3 = "David"; delivery3 = pager; delay3 = 30;

);

Example 3: We now include an example to demonstrate the two level specification mechanism. In the GEA Model shown in Figure 17, it can be observed that the employee "Martha" is provided with authority to initiate "Contact" requests. However, "Wang" creates the following PSL policy that aims to restrict "Martha" from initiating "low" priority "Contact" requests:

Policy P3;

Wang; Enterprise A;

When Contact arrives;

If Contact.priority = "low" AND Contact.originator = "Martha"

Then **forbid;**

In this section, we have demonstrated with three examples how the unit heads create policies to control the activities of entities under their authority. Since the scope of a PSL policy is obtained from the EA Model, the authority of an entity to specify policies on other entities is verified implicitly. Also, it can be observed that PSL does not rely on the policy creator to list the target entities for a policy. In the next section, we will demonstrate a case where the new policy conflicts with some existing policies in the policy repository.

4.4.3 Conflict Detection and Resolution

We assume policies P1, P2 and P3 described in Section 4.4.2 exist in the policy repository. In this example, "John" uses PSL to specify the following policy that aims to reject "Contact" requests with "high" priority.

Example 4:

Policy P4;
John; Enterprise A;
When Contact arrives;
If Contact.priority = "high";
Then **forbid;**

This new policy, P4, conflicts with existing policy P1. The conflict is detected because the *content-sections* of the two policies is identical, *source-sections* is *contained* (by definition 3 in Section 3.4, page 33) and *action-sections* conflict. The conflict is resolved by removing policy P1 from the policy database and by accepting policy P4 for enforcement. This is because the creator of policy P4 has a higher authority than the creator of policy P1 in the GEA Model shown in Figure 17. The conflict resolution mechanism is discussed in detail in Section 3.4 of this thesis.

4.4.4 Policy Enforcement

To demonstrate the policy enforcement examples, we assume that policies P1, P2 and P3 (described in Section 4.4.2) exist in the policy repository. In this section, we demonstrate four scenarios where P1, P2, P3 are enforced on entities "Mark", "Chris" and "Martha" (refer to Figure 17 for the GEA Model).

Scenario 1: "Martha" who belongs to "Customer A" initiates a "Contact" request with "high" priority to "Mark" in "Enterprise A". This is shown below:

Contact (originator = "Martha"; originator_model = "Customer A";
recipient = "Mark"; recipient_model = "Enterprise A";
message = "Router Down"; priority = high;
)

When the above request is received at the PST Server, the following action (a notification request to MNS) is executed:

Notify (message = "Router Down"; priority = 2; severity =2;
recipient1 = "Mark"; delivery1 = pager; delay1 = 10;
recipient2 = "David"; delivery2 = pager; delay2 = 20;
recipient3 = "Wang"; delivery3=pager; delay3 = 30;
)

It can be observed that escalation list for the "Notify" action is enforced by **Policy P1**. This is because the intended recipient "Mark" is under the authority of "David", and hence events to "Mark" are bounded by policies specified by "David", which in this case is Policy P1.

Scenario 2: "Martha" who belongs to "Customer A" initiates a "Contact" request to "Chris" in "Enterprise A".

Contact (originator = "Martha"; originator_model = "Customer A";
recipient = "Chris"; recipient_model = "Enterprise A";
message = "Router Down"; priority = high;
)

When the request is received at the PST Server, the following action (a notification request to MNS) is initiated:

Notify (message = "Router Down"; priority = 1; severity =1;
recipient1 = "Chris"; delivery1 = pager; delay1 = 10;
recipient2 = "Wang"; delivery2 = pager; delay2 = 20;
recipient3 = "David"; delivery3=pager; delay3 = 30;
)

The parameters of the "Notify" request is provided by **Policy P2** since the intended recipient "Chris" is under the authority of "Wang", the creator of Policy P2.

Scenario 3: Similarly, when "Martha" initiates a "Contact" request of "low" priority to "Chris" in "Enterprise A":

```
Contact ( originator = "Martha"; originator_model = "Customer A";
          recipient_name = "Chris"; recipient_model = "Enterprise A";
          message = "Router Down"; priority = high;
        )
```

When the request is received at the PST Server, Policy P3 created by "Wang" that seeks to reject "low" priority requests from "Martha" is enforced. Subsequently, "Martha" is displayed the message "Forbidden to send "low" priority requests".

Scenario 4: In the earlier scenario, Scenario 3, "Martha" had the authority to initiate "Contact" requests but was forbidden by PSL level policies to initiate "low" priority requests. In this scenario, we illustrate a case where the request is rejected at the GEA Model level. The user "Nancy" belonging to "Customer A" initiates the following "Contact" request:

```
Contact ( originator = "Nancy"; originator_model = "Customer A";
          recipient_name = "Chris"; recipient_model = "Enterprise A";
          message = "Router Down"; priority = high;
        )
```

When the request is received at the PST, it displays the following message to the user: "Authority to initiate a notification request is denied".

4.4.5 Java Implementation and Testing

In this section, we verify the feasibility of our proposed architecture by means of an implementation prototype and demonstrate the four policy enforcement scenarios (for P1, P2, P3) discussed in the previous section.

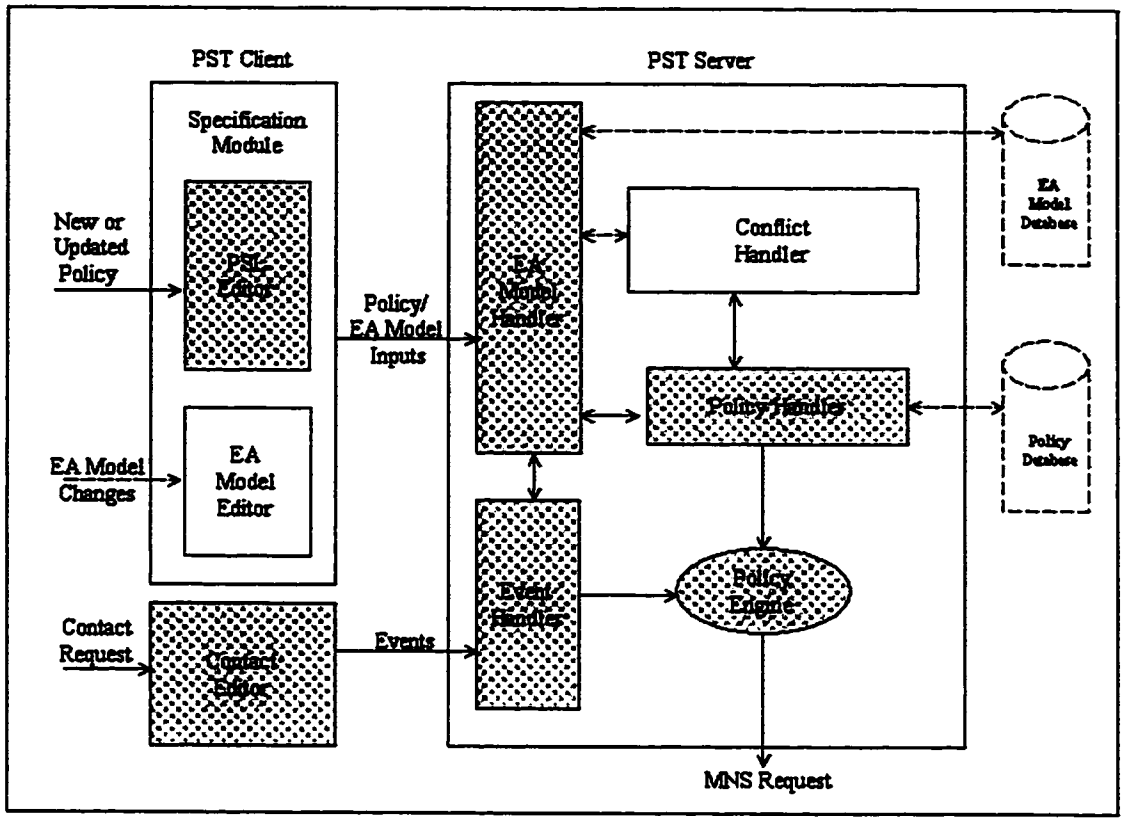

Figure 18: Scope of Implementation of Integrated Architecture

In Figure 18 shown above, the *shaded* modules and the arrows shown as continuous lines denote parts of the architecture that is implemented in the prototype. We have also implemented a "Contact" editor that provides a graphical user interface to generate "Contact" requests to PST. When "Contact" requests are received by Event Handler module, it is propagated to EA Model Handler and Policy Handler for further processing. The five implemented modules (PSL Editor, Event Handler, EA Model Handler, Policy Handler and Policy Engine) enable us to demonstrate the ease of use of our proposed policy specification language (PSL) and the application of EA Model to manage simple authorizations and to provide the scope of a policy. The PST graphical user interface is shown in Figure 19 in the next page.

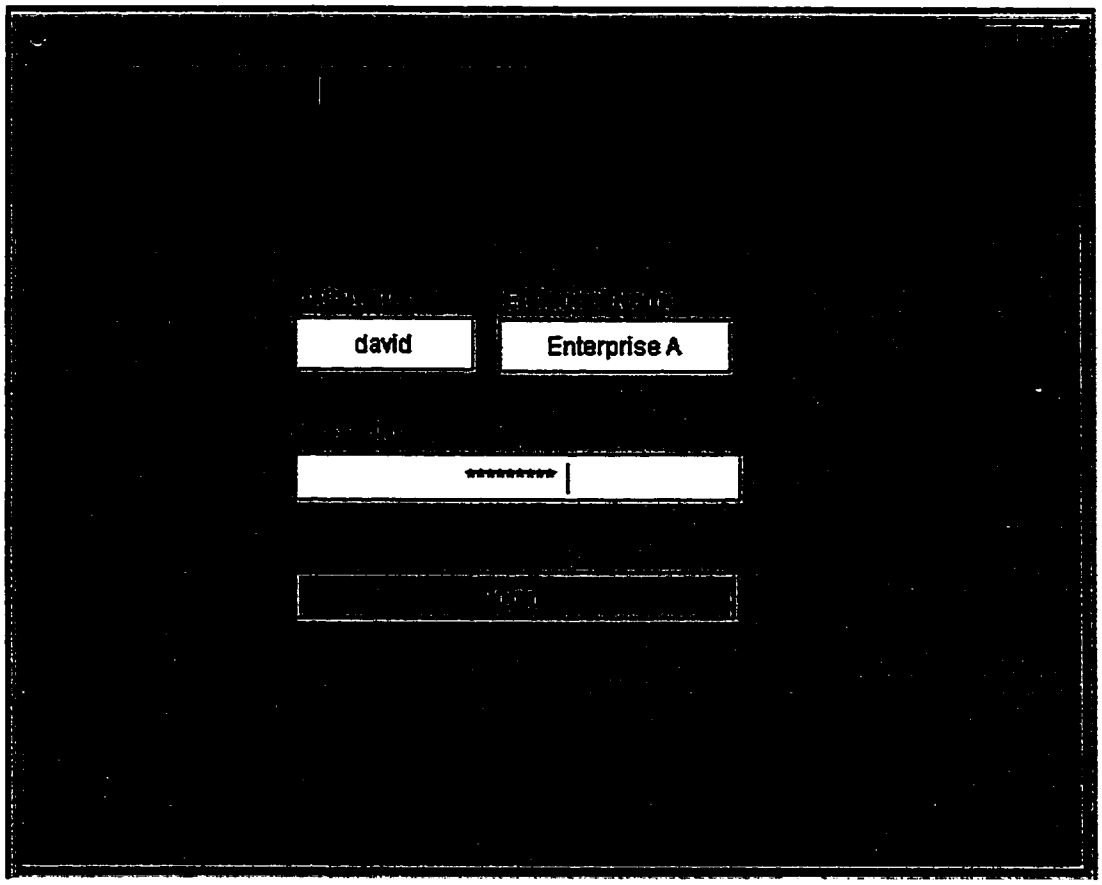

Figure 19: PST Login Window

4.4.5.1 PSL Editor and Policy Handler

The PSL Editor shown in Figure 20 provides a graphical user interface to create policies using PSL statements. The *source* field in the editor window is of non-editable type and its value is automatically obtained from the login window. When the enable button is pressed, the new policy is received by the EA Model Handler to verify the authority of the specifier entity on event and program entities used in the policy. If verification succeeds, the new PSL policy is sent to the conflict handler module. In our prototype, since we have not implemented the conflict handler module, when the enable button is pressed the new policy is transferred to the *policy handler*.

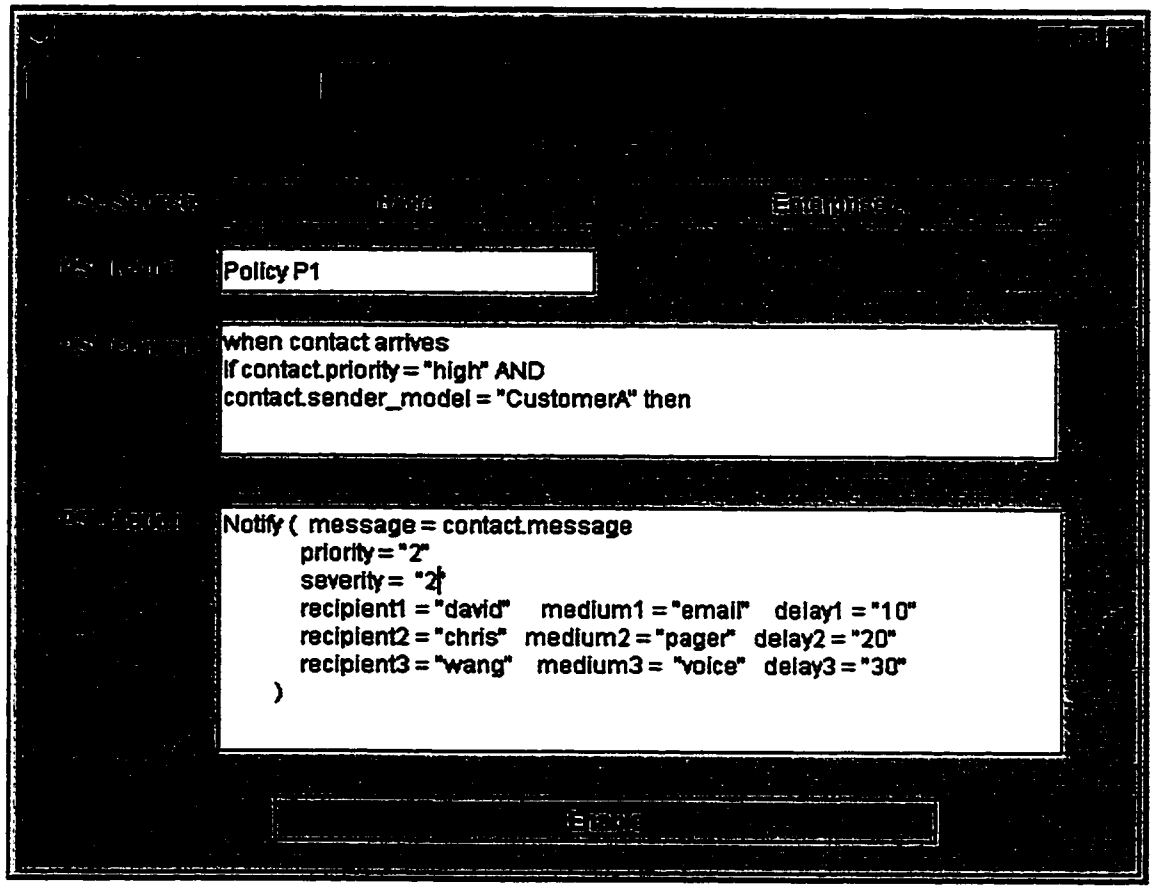

Figure 20: Graphical User Interface for Policy Specification

The *policy handler* module receives the new policy and uses the Java Expert System Shell [Jess] to enforce it. To enable this, the *policy handler* parses the new PSL statement and maps them into JESS rules, i.e., in the form that can be interpreted by the Java Expert System Shell. We illustrate below a policy in PSL notation and then present it as a JESS rule:

PSL Notation:

Policy P1;

David; Enterprise A;

When Contact arrives;

If Contact.priority = "high";

Then Notify (message = Contact.messaage; priority = 2; severity =2;

recipient1 = "Contact.recipient"; delivery1 = pager; delay1 = 10;

recipient2 = "mark"; delivery2 = pager; delay2 = 20;

recipient3 = "david"; delivery3=pager; delay3 = 30;);

When above PSL notation reaches the policy handler, it looks up the EA Model to obtain the scope of the policy and creates the following Jess rule to enforce policies on entity "Mark" (and in a similar way for "Mike"):

JESS Notation:

```
(defrule P1
      (Contact (originator ?)(senderModelName ''Customer A'')
      (recipientName ''Mark'')(recipientModelName ?)
      (priority ''high'')(messageContent ?mc))
=>
      (definstance
      Notify (new contact.fact.notify   ?mc          1        1
                                       ''Mark''    " pager"   10
                                       ''David''   " email"   20
                                       ''john''    " pager"   30))
)
```

The "definstance" keyword is used to instantiate event or program entity in the right hand side clauses of Jess rules. We assume th at "Notify" object performs the task of initiating a request to MNS system. The "defclass" command shown below is used to identify event and program entities to Jess:

```
(defclass Contact contact.fact.Contact)
(defclass Notify contact.fact.Notify)
```

4.4.5.2 Contact Editor

The "Contact" editor as shown in Figure 21 provides a graphical user interface to generate "Contact" requests. The sender field is non-editable and obtains its value from the user's login information. When the "send" button is pressed, the "Contact" request is generated and is received by the Event Handler module.

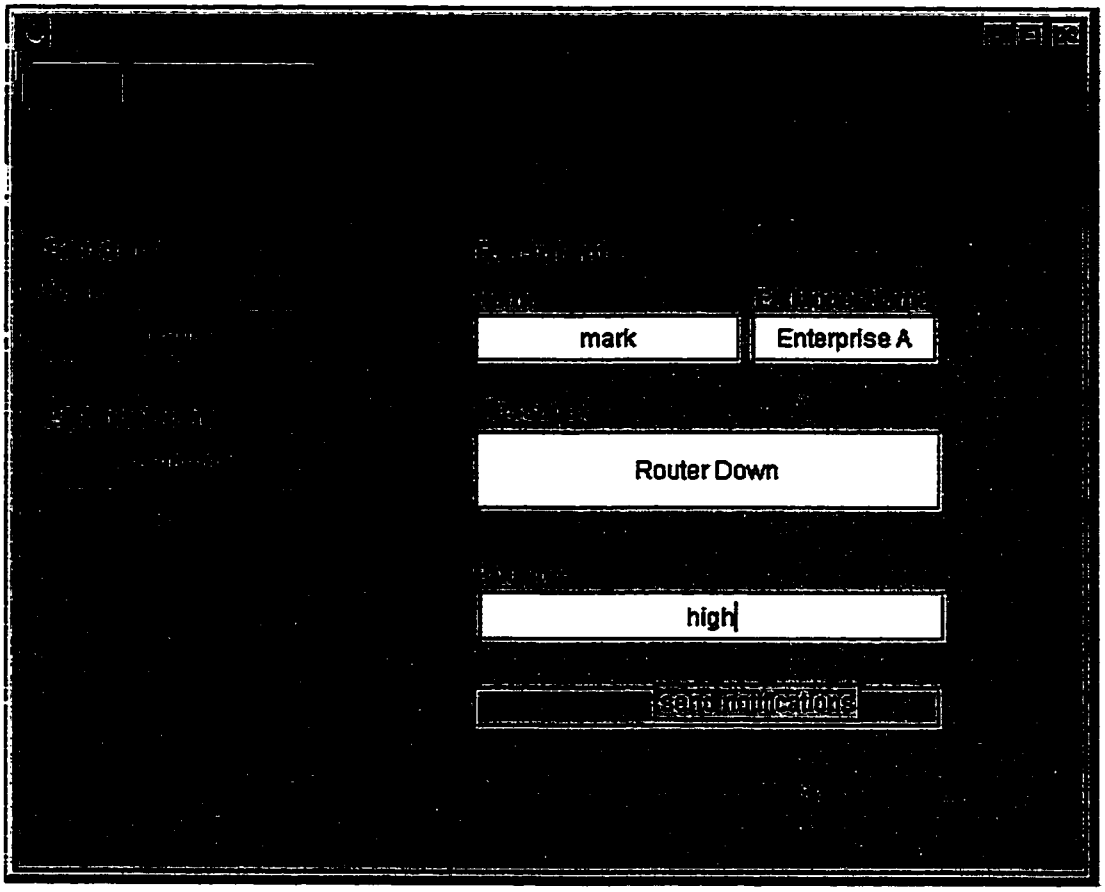

Figure 21: Contact Editor

4.5 Chapter Summary

Our goal in this chapter was to demonstrate the application of PST to accomplish policy-based management of enterprise processes. Although our proposal is domain independent, we chose Message Notification System for demonstration as they are being increasingly deployed in enterprises for inter and intra-office communications. We have derived the need for policy-based message notification system and illustrated the steps to accomplish it. The feasibility of our approach is validated by means of a prototype implementation of PST and subsequent demonstration of four case scenarios that achieve a policy based notification solution. The application prototype also demonstrates the benefits of a policy-based MNS system in which parameters of a notification request (escalation list, severity, priority) are determined by the PSL policies thus allowing unit heads to control (by creating PSL policies) the message notification

process and in turn ensure that notifications reach the right person at the right time using the most appropriate delivery media. Another significant contribution of the prototype implementation is to test the suitability of expert system techniques [Jess] for policy execution.

5.0 Conclusion

5.1 Contributions

The aim of this thesis is to provide an Enterprise Policy Specification Tool for managing enterprise policies. Since an enterprise's structure and available resources are constantly changing when multiple entities are allowed to manage enterprise policies the task of policy specification and maintenance is complex. We categorize our achievements in this thesis as *primary* and *secondary* contributions and summarize them below. As our *primary* contributions, we have:

- proposed an Enterprise Authority (EA) Model to capture the relationship between enterprise entities and their authority on available enterprise resources.
- introduced a Policy Specification Language (PSL) that denotes the EA model in its policies. This enables us to automatically obtain the scope of a policy from the EA Model and thus implicitly verifies the authority of policy creators to specify policies on other entities.
- identified two types of authorization policies: simple and complex. Simple authorizations are handled at the EA Model level leading to a simpler specification. For complex authorizations we have used the combination of EA Model and PSL rules.
- proposed a mechanism to detect conflicts between enterprise policies before they are accepted for enforcement. Further to this, we have proposed a mechanism to resolve conflicts based on enterprise authority of entities involved in the conflict.
- proposed an architecture for Policy Specification Tool (PST) to apply the above contributions in practice.

- Identified the need to ensure consistency between an enterprise's available resources, its structure and its policies. We have identified three situations that may create new inconsistencies and have recommended steps to prevent them.

As our *secondary* contribution we have:

- partially (modules that are shaded in Figure 18, page 55) implemented our proposed PST architecture using Java and Java Expert System Shell [Jess] to demonstrate using four policy enforcement scenarios the application of PSL, EA Model and Two-level specification mechanisms.

5.2 Further Work

We identify four areas where our work can be extended, they are: EA Model, PSL Syntax, Incremental Maintenance and the PST prototype. Presently, our EA Model supports enterprises that are organized as a directed-acyclic graph (DAG) structure. We chose to support DAG structure for our proposal as majority [Litterer] of enterprises are modeled this way. However, as enterprises continue to evolve or when they reorganize themselves there maybe new requirements for modeling non-standard structure in the form of a graph or matrix [Litterer] form. It will be useful to extend our proposal to support different forms of enterprise models. In the area of policy specification, our work can be extended to support different levels of policy specification languages. For example, policies maybe specified in a natural language form and then automatically transferred into one or more PSL policies. One other significant extension is to extend the PSL grammar to be able to support specification of temporal rules.

Our conflict detection and resolution mechanism has addressed one type of inconsistency, which identifies policies that monitor same set of events but perform opposite actions. However, other types of inconsistencies such as duplicate, inferred, redundant and subsumed policies may also occur and they have to be detected and resolved before they are accepted for enforcement. It will be useful to extend our work to detect these different types of inconsistencies. In our work, we have identified the need to maintain the consistency between available enterprise resources and the policy repository (policies accepted for enforcement). We recommend incremental

maintenance tasks to ensure this consistency. Our work on incremental maintenance can be extended so as to provide an algorithm to perform the same. Finally, our Java implementation prototype can be extended to demonstrate the feasibility of our proposed conflict detection and resolution algorithms and a graphical user interface to manage our proposed EA Model.

6. References

[Bala] *Balachander Krishnamurthy* and *D.S.Rosenblum*, "YEAST: A General Purpose Event-Action System" in IEEE Transactions on Software Engineering, Volume 21, NO.10, October 1995.

[Brites] *A.C.S.C.Brites, P.A.FSimoes, P.M.CLeitao, E.H.SMonteiro, F.P.L.B.Fernandes*, "A High-Level Notation for Specification of Network Management Applications", in proceedings of INET 1994/JENC5.

[Curamessage] www.curasoft.com

[Herbst] *H.Herbst, G.Knolmayer, T.Myrach and M.Schlesinger*, "The Specification of Business Rules: A comparison of Selected Methodologies", in System Life Cycle, Amsterdam et al.: Elsevier 1994, pp.29-46.

[IETF] "Policy Framework (draft)", www.ietf.org, September 1999.

[Ilog] ILOG Rules – www.ilog.com

[Inverardi] *P.Inverardi, B.Krishnamurthy, D.Yankelevich*, "Yeast: A case study for a practical use of formal methods" from IEI-CNR – Italy, Bell Labs – USA and University of Pisa - Italy.

[Jarvis] *P.Jarvis, J.Stader, A.Macintosh, J.Moore and P.Chung*, "What right do you have to do that? Infusing Adaptive Workflow Technology with Knowledge about Organizational and Authority Context of a Task", from AI Applications Institute, Loughbourgh University, UK.

[Jess] http://herzberg.ca.sandia.gov/jess/

[Keryx] www.hp.com

[Koch] *T.Koch, Krell, B.Kramer*, "A Policy Definition Language for Automated Management of Distributed Systems" in the International Workshop on Systems Management. June 1996,

Toronto, Canada.

[Kramer] *T.Koch, B.Kramer, G.Rohde,* "On a Rule Based Management Architecture", in the International Workshop on Systems Management (June 1995), IEEE Computer Society.

[Lee] *R. Lee,* "Bureaucracies as Deontic System" in ACM Transactions on Office Information Systems, Vol 6, No. 2, April 1988, pages 87-108.

[Litterer] *J. A. Litterer,* "Organizations: Structure and Behavior", Wiley publications.

[Lobo] *J.Lobo, R.Bhatia, S.Naqvi ,* "A Policy Description Language", in proceedings of AAAI 1999.

[Lupu] *E.Lupu and M.Sloman.* "Conflicts in Policy-based Distributed Systems Management", in IEEE Transactions on Software Engineering 1997 – Special Issue on Inconsistency Management.

[Marriot] *D.Marriot and M.Sloman,* "Management Policy Service for Distributed Systems", in IEEE third International Workshop on Services in Distributed and Networked Environments (SDNE 1996).

[Moffet] *J.Moffet, M.Sloman* Policy Hierarchies for Distributed Systems Management", in IEEE Journal on Selected Areas in Communications (1993). Vol 11(9): pp 1404-1414.

[Myrach] *H.Herbst and T.Myrach,* "A Repository System for Business Rules", in proceedings of the sixth IFIP TC-2 Conference on Data Semantics, London: Chapman & Hall 1995/96.

[Sloman] *M.Sloman, E.Lupu,* "Policy Specification of Programmable Networks", in proceedings of First International Working Conference on Active Networks (IWAN 1999), Berlin, June 1999.

[Sobieski] *Sobieski, S.Krovidy,* "KARMA: Managing Business Rules from Specification to Implementation", in proceedings of IAAI-9.

[Telalert] www.telalert.com

[Weis] R.Weis, "A practical approach towards distributed and flexible realization of policies using intelligent agents", from Department of Computer Science, University of Munich.