A HIGH-LEVEL FRAMEWORK FOR POLICY-BASED MANAGEMENT OF DISTRIBUTED SYSTEMS

A THESIS SUBMITTED TO THE
UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jovan Cakic
December 2003

"You see things; and you say `Why?` But I dream things that never were; and I say `Why not?`"
George Bernard Shaw

ABSTRACT

Policy-based management of distributed systems, which uses specification of behavioural constraints as means of management control, has received considerable attention in recent years. Policies, which are derived from the goals of management, define the desired behaviour of distributed heterogeneous systems and networks, and specify means to enforce that behaviour.

The requirements that management goals put on the behaviour of distributed systems today tend to be complex, sophisticated and to change rapidly. In order to enable fast and simple adaptation of the behaviour, flexible, built-in mechanisms need to be available in distributed systems. We see management policies as the best way to seamlessly integrate dynamic management goals into the behaviour of distributed systems.

With the ever growing number and complexity of large distributed systems and networks, the problem of complexity is the limiting factor to the applicability of existing policy-based management solutions. At the beginning of this research, our impression was the existing policy specification languages lacked the power to express complex management models in a sufficiently abstract way, and the existing policy-based management solutions lack completeness. The ambition of this thesis is to demonstrate new application possibilities that policy-based management presents, by providing the necessary supporting framework and better understanding of the background itself.

The focus of the research was on design and implementation of a new policy specification language, which can be used for specifications of high-level management policies. Since the policy specification language itself is not enough to support the whole concept of distributed system management, a conceptual framework for policy-based management is proposed too, together with a complete management process.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisors, Professor John Derrick and Dr Gill Waters, who had helped me to develop a sense of discipline and taught me to express myself concisely and precisely. I'm particularly thankful for their guidance, patience and belief in me.

I would like to acknowledge the support from BTexact, and Paul McKee in particular.

Thanks to my good friend David Elwood and his nice parents for pleasant moments I had with them. I also thank all the friends from around the world that I have made during my time at UKC.

I would also like to thank all the people I had pleasure to meet in the Computing Lab and all the nice people I had pleasure to work with at the Computing Help Desk and the Computing Service.

Finally, I gratefully acknowledge the financial support from BT and the Computing Laboratory at the University of Kent. I also thankfully acknowledge support from Microsoft's MSDN Academic Alliance.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

1 INTRODUCTION

In this thesis we will firstly introduce the reader to the idea of policies and policy-based management, present some related concepts and technologies, and then describe the conceptual management framework that has been developed in the course of the research project.

This chapter will begin by presenting basic concepts of distributed systems and distributed system management, which define the context for this research. Then, we will give an introduction to the concept of policies and policy-based management, present the motivation and objectives of the research, and finally give the structure of the thesis itself.

The requirements that management goals put on the behaviour of distributed systems today tend to be complex, sophisticated and to change rapidly. In order to enable fast and simple adaptation of the behaviour, flexible management instruments need to be available in distributed systems. We see management policies as the best way to seamlessly integrate dynamic management goals into the behaviour of distributed systems. The ambition of this thesis is to demonstrate new application possibilities that policy-based management presents, by providing the necessary supporting framework and better understanding of the background itself.

1.1 Context of research

Distributed systems exhibit a number of inherent characteristics: system components may be spread across space with either local or remote interactions (remoteness), system components can execute in parallel with any other components (concurrency), the global state cannot be precisely determined (lack of global state), any component may fail independently of any other components (partial failures), communication and processing activities cannot be assumed to take place at a single instant (asynchrony), components are built using different technologies that will change over time (heterogeneity), processing resources and associated devices are under the control of separate autonomous management authorities (autonomy), constant changes happen as a result of technical progress and new strategic decisions (evolution), programs and data, processing nodes, and users may be mobile to optimise performance (mobility), etc. The main difficulties facing distributed systems are heterogeneity, asynchrony, limited local knowledge, and delays and failures. The explosive growth of distributed systems makes it imperative to understand how to overcome these difficulties.

1.1.1 Distributed system management

Management of a system is concerned with supervision and controlling the system so that it fulfils the requirements of both the owners and users of the system [Sloman 1994]. It has been common to discuss distributed system management in terms of the following five functional areas:

- *Fault management* encompasses fault detection, isolation and the correction of abnormal operation. It includes functions to maintain and examine error logs, accept and act upon error detection notifications, trace and identify faults, carry out sequences of diagnostic tests, and correct faults.

- *Accounting management* enables charges to be established for the use of resources, and for costs to be identified for the use of those resources. It includes functions to inform users of costs incurred or resources consumed, enable accounting limits to be set and tariff schedules to be associated with the use of resources, and enable costs to be combined where multiple resources are invoked to achieve a given communication objective.
- *Configuration management* identifies, exercises control over, collects and provides data for the purpose of management. It includes functions to set the parameters that control the routine operation of the system, associate names with managed objects and sets of managed objects, initialize and close down managed objects, collect information on demand about the current condition of the system, obtain announcements of significant changes in the system, and change the configuration of the system
- *Performance management* enables evaluation of the behaviour of resources and the effectiveness of communication activities. It includes functions to gather statistical information, maintain and examine logs of system state histories, determine system performance under natural and artificial conditions, and alter system modes of operation for the purpose of conducting performance management activities.
- *Security management* purpose is to support the application of security policies by means of functions which include creation, deletion and control of security services and mechanisms, distribution of security-relevant information, and reporting of security-relevant events.

Current management systems derive from traditional networked environments, where devices lacked resources to execute non-trivial management software, management data and functions were relatively simple, and organizations could devote the human resources needed to handle operations. As a result, the management is engaged in abnormally fine-grained and complex process interactions, which leads to significant barriers on manageability due to great semantic heterogeneity of operational behaviours among resources.

Within large, heterogeneous distributed systems, where devices incorporate significant processing power while human resources are scarce, the situation is different. To meet their functional requirements, system management processes must adapt, customize, and refine the capabilities of the available resources [Maullo 1993]. The management strategy needs to provide a range of different management objectives and policies, a variety of hierarchical and federated organizational structures, and the ability to manage a very disparate collection of entities supplied by multiple vendors and interconnected by many different networks [Langsford 1993].

1.1.2 Policies and policy-based management

Policy is a much-overloaded concept in network and system management; it is used to describe everything from the specification of corporate policies on computer use to a data driven configuration of a router [Casassa Mont 1999]. There are many definitions of policies, some of them very different in their nature:

- Policy is a rule governing choices in behaviour of the system [Damianou 2000].

- … a policy is about the constraints and preferences on the state, or on the state transition of the system. It is a guide on the way to achieving the overall objective which itself is also represented by a desirable system state [Goh 1997].
- Policies are a means of influencing management behaviour within a distributed system, without coding the behaviour into the managers [Sloman 1997].
- Policy is essentially a matter of allocating resources in terms of business decisions [Mahon 2000].
- A policy defines the desired behaviour, i.e. it is a restriction on the possible behaviour [Wies 1995].

Our definition of a policy is given in section 1.4.

Policies are often seen as a link between corporate management and technology management. They provide a means of specifying and dynamically changing management strategy without coding policy into the implementation. The separation of management policies from the resources and activities being managed allows, on one hand, abstract and uniform view of the managed resources that hides their heterogeneity and, on the other hand, implementation of a uniform and generic model for the management activity.

The use of management policies enables specification, control and enforcement of required system behaviour in a flexible and dynamic way. It enables management tasks specification at different levels of abstraction, allows better control and allocation of resources, increases automation and provide the rapid response needed for dynamic environments, and simplifies management operations. According to [Damianou 2002a], the main benefits from using policy are improved scalability and flexibility for the management system. Scalability is improved by uniformly applying the same policy to large sets of devices, while flexibility is obtained by separating the policy from the implementation of the managed system. Policy can be changed dynamically, thus changing the behaviour and strategy of the system, without modifying the implementation or interrupting the system's operation.

It is a common approach to represent policies in a form of rules. Rule-oriented representation of policies can take different forms, but the least common denominator of the major policy representation formats are:

- *Policy conditions*, which define a situation that activates (triggers) management actions specified by the policy. Some approaches relate this situation to the events in the system, while the others express it in terms of the system state.
- *Policy actions*, which define actions to be taken in response to the triggering situation. In most cases, the policy actions are expressed as steps to achieve state transitions in the system.

The central idea behind policy-based management is to develop an infrastructure that provides abstraction of capabilities of individual devices. These abstract actions elevate the level of control to the system level and represent a transition from managing services realized within individual devices to managing services realized within the system itself. The real value of the policy-based management lies in the development of a unified and coordinated information model that represents the interfaces to an infrastructure made up of services and other manageable aspects of systems.

The key enabling components of a policy-based management solution may be identified as follows:

- **Language** for policy specification.
- **Graphical user interface** (GUI), which hides the complexity of the system infrastructure and details of the policy specification language. The GUI is designed to support policy specification and usage (policy-based management).
- **Policy execution environment** (framework), with its:
    - *Architecture*, which consists of two main logical elements: policy decision point (PDP) that acts as a "judge" who makes decisions based on the policies retrieved from the policy repository, and policy enforcement point (PEP) that acts as a "police officer" who enforces policies.
    - *Mechanisms* that include storage, which is supported by:
        - Centralized or distributed policy repository.
        - Triggering, which may be event-based (reactive) or state-based (proactive).
        - Refinement, which is a process of mapping a "human oriented" abstract policy to the appropriate resource configuration.
        - Conflict resolution, which should prevent contradictions in actions prescribed by simultaneously satisfied policies.
        - Enforcement, which involves applying management actions specified by policies.

As the number of resources to be managed grows, the task of managing devices and applications becomes more dependent on numerous system and vendor specific issues. To prevent the operators from drowning in excessive details, the level of abstraction needs to be raised in order to hide system specifics. Policies enable the abstraction of management, and the benefits of policy-based management will grow as distributed systems become more complex and offer more services.

1.2 Motivation

At the beginning of this research, our impression was the existing policy specification languages lacked the power to express complex management models in a sufficiently abstract way, and the existing policy-based management concepts lack completeness (for a detailed analysis, see Chapter 3). With the ever growing number and complexity of large distributed systems and networks, the problem of complexity has became the limiting factor of the applicability of existing policy-based management solutions.

The general idea of policy-based management is not new, but practical implementations of policy-based management are, however, mainly at a low level at the present [Casassa Mont 2000]. Although many approaches exist that address various management issues, there is a lack of policy concepts that adequately cover the problem of system-wide, high-level policy based management. While appropriate parameters can be set in individual devices, there are usually no system-wide mechanisms available to control its overall performance. It is also hard to relate these mechanisms to the service experienced by end users of the system.

The management problems in large distributed systems are essentially caused by the heterogeneity of the system components and a low abstract level of the available management tools and concepts, which

makes the heterogeneity even more obvious. According to [Robins 2002], the problem with the existing management solutions is that they:

- ∉ Do not operate well in an open, heterogeneous environment. Management systems are designed for tight coupling with (often embedding into) managed systems, which results in an inability to follow the typically dynamic life-cycle of distributed systems (lack of flexibility) and interoperability problems because of low-level management interactions.
- ∉ Do not deliver a business-centric or process-centric view into transactions. The focus in on a structural, resource-oriented view of the system, which typically includes far too many details.
- ∉ Were not designed for the pure volume of service and event management. Management models are mainly structure-based and do not scale very well, because they are rigid and include far too many details.

1.3 Objectives

The essence of the problem of policy specification is well captured by Koch in [Koch 1996]. According to Koch, the biggest problem with the formalization of management knowledge is the great semantic gap between the demands of users, usually given in plain text on a very abstract level, and the formal description of appropriate procedures in terms of executable commands. Therefore, we need not only to capture the abstract management knowledge using a policy specification language, but also to define a management framework capable of bridging the semantic gap between the specified management knowledge and the particular management capabilities of management systems.

The key to a successful management of large distributed systems is in cooperative work of human manager and supporting management infrastructure, each with its own role: human manager offers decision-making intelligence, while the management infrastructure reduces the complexity of management activities. The ultimate goal is to relieve human manager by moving the complexity of management activities into the management infrastructure and to make distributed systems as autonomous as possible.

The aim of this research was to put more power into the hands of system managers, by abstracting away the complexity of real systems and shifting management activities to a higher level of abstraction. The focus of the research was on the design and implementation of a new policy specification language, which might be used for specifications of high-level management policies. Since the policy specification language itself is not enough to support the whole concept of distributed system management, a conceptual framework for policy-based management will be proposed too, together with a complete management process.

According to [Balcer 2002], the history of software development is a history of raising the level of abstraction. Each higher layer of abstraction is introduced only as a concept, but over time the new layers of abstraction became formalized, and tools were constructed to support the concepts and to map from one layer to the next automatically. This has the effect of hiding the details of how the lower layers work. This idea has been used to address the problem of management complexity in large-scale distributed systems, by creating a conceptual management framework and placing it on top of existing solutions.

This research was particularly motivated by the need for a high-level solution for the problem of management of large distributed systems. The main goal of this research was to offer a solution for the problem of complexity in distributed systems management, by tackling the following issues:

- *High-level system modelling*: How to represent behaviour of a system at the high level of abstraction
- *High-level specifications of management activities*: How to use such abstract system representations for high-level policy specifications
- *High-level management process*: How to make such high-level policy specifications operational

Goals of the project were derived directly from the identified requirements and justified by the incapability of the existing solutions to satisfy them (a detailed analysis is given in Chapter 3). These research goals were identified as follows:

- Address the problem of management complexity in large distributed systems.
- Provide enough flexibility to support ongoing system changes.
- Enable automation of the management process.
- Preserve independence of management from the managed system.
- Deliver concise but expressive policy specification.
- Assure formality of policy specifications.

1.4 Policy-based management framework

In this research, the high-level policy-based management is seen as a solution for the complex management problems in large distributed systems. It was envisaged that the proposed concept will be able to simplify system representation from the management point of view, by presenting it at a higher level of abstraction, and to reduce the management complexity, by classifying and generalizing system management activities.

The management framework proposed by this research was based on our vision of policies and their role in distributed systems management.

Definition

Policies specify declarative knowledge (know-how, expertise) about how to use the behaviour of a system to manage its state. The management knowledge is captured in an explicit form, so it can be changed.

We see management policies as specifications of generalized management experience, which guide management operations in the policy-based management system towards the goal. Policies do not specify management decisions themselves, but rather provide knowledge that may be used to make such decisions. The purpose of a policy specification is to state management knowledge so that others (humans and machines) can understand it and (re)use it to control the behaviour of a system.

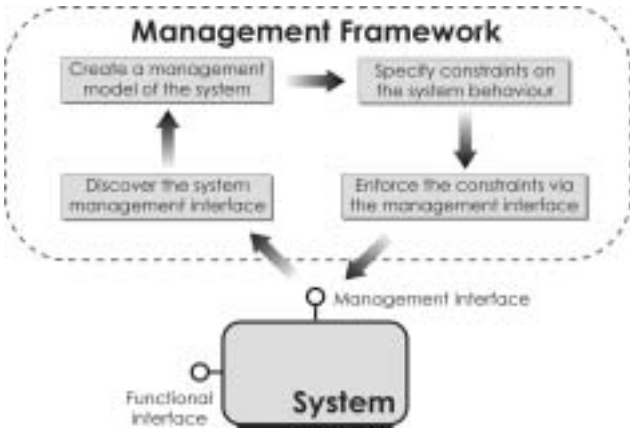


Figure 1-1 The model of the management framework

Our solution for high-level policy-based management of distributed systems was envisaged as a conceptual framework (see Figure 1-1), defined in terms of its:

- *Architecture*. The design of the framework architecture had two main goals: to be independent of systems it was supposed to manage, and to be compatible with the latest development in the domain of distributed systems technologies. To achieve the goal of independence, the managed systems were considered as Open Architectures (see 1.4.2) from the outside, encapsulated behind their management interfaces. To achieve the goal of compatibility, the managed systems were considered to be internally designed as Service-Oriented Architectures (see 1.4.4), with management interfaces defined in terms of (high-level) management services.
- *Processes*. In order to implement a complete management solution, the framework processes were designed to implement the Management Control Model (see 1.4.1). The processes work together to establish a complete, autonomous, workflow-oriented management process, with workflows seen as collections of coordinated activities designed to achieve management goals.
- *Information model*, which was designed as a policy specification language able to support all the framework processes. The language design was based on the principles and criteria for the design of modelling and programming languages (see 1.4.6), but also on the principles for the design of modern information languages based on XML (see 1.4.3). The policies themselves are seen as mobile, self-described specifications of management workflows (see 1.4.5).

In the rest of this chapter we will briefly introduce the basic concepts the design of our management framework is based upon.

1.4.1 Management Control Model

Theorists as well as practicing executives agree that good management requires effective control. Control is a function through which the executive is able to identify change, discover its causes, and provide decisive action in order to maintain a state of equilibrium within the system. The purpose of control is to provide the business system with effective information, decision rules, and means to take corrective action in such a way as to attain objectives.

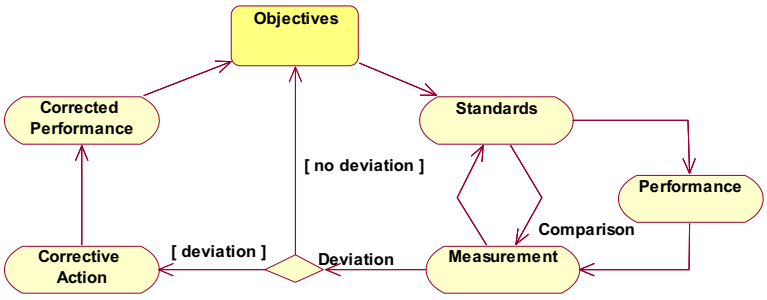


Figure 1-2 The Management Control Model

One of the most widely accepted approaches to managerial control involves the closed-loop model shown in Figure 1-2 (see [Strong 1968] for more details). This model begins with the establishment of objectives for the organization and completes its cycle with their attainment. The control cycle typically consists of the following phases:

- *Development and communication of goals*, which provide a common direction within the organization. Without objectives there are no criteria for evaluating performance and, in fact, there is no real need for management.
- *Development of realistic standards of performance*, as a minimum level of output that is expected. To state where a system ought to be is to set a standard. Standards, in turn, are based upon the objectives desired.
- *Measurement of performance*, to determine whether or not the standards have been met. Effective measurement of performance requires that the executive determine the type of management information which he needs from each of the strategic elements within his area of responsibility. Information becomes his key to control.
- *Corrective action*. When measurement indicates that the system or one of its elements not operating within the established standards, the control process has performed its function of identifying abnormal variation. Management should have a contingency plan which will lead to the elimination of the excessive variance from standard.

A sound executive control system provides information, evaluates this information, gauges actual performance against established plans, and finally assures that corrective action is taken when performance is not directed toward the achievement of organization goals.

1.4.2 Open Architectures

The reason "black-box" abstraction doesn't always work is that in some cases, the best implementation strategy for a module can't be determined unless the implementer knows exactly how the module will be used. In other words, the client often knows better how the module should be implemented. "Black-box" abstraction forces the implementer to decide early on what the implementation will be, and then locks that decision into the "black box". This results in conflicts when the implementer makes a choice that a client can't tolerate.

Open architectures may provide elegant solutions to the design and implementation of highly adaptable software [Oliva 1998]. According to [Kiczales 1996], module implementations must somehow be opened up to allow clients to control these issues as well. Open implementation is a design approach in which the module's interface is created in such a way that the client can assist or participate in the selection of the module's implementation strategy. Open architectures are based on the existence of an additional interface that allows them to dynamically adapt to new requirements presented by their clients. The primary interface provides the functionality, while the meta-interface allows the client to adjust the implementation strategy decisions that underlie the primary interface.

Reflection is the process of reasoning about and acting upon the system itself, and the role of reflection is to provide a principled means of achieving open implementation. In distributed systems, reflection may provide two important facilities: to separate management and functional logic, and to model management policies dynamically and easily.

Goals of policy-based management are very similar to the concept of open architectures. Open architectures encourage a modular design where there is a clear separation of policy, that is, what a module has been designed for, from the mechanisms that implement a policy, that is, how a policy is materialized. Automated policy-based management is, in effect, one of the possible implementations of reflection.

1.4.3 eXtensible Markup Language (XML)

The word markup means the insertion of information into a document. In a structured document, the markup is designed to highlight the structure by putting the semantics directly into the syntax in order to reduce complexity of document analysis. XML provides a standardized language to exchange data for automated consumption. It offers the most precise way to describe and create information objects and to make those objects available to process–oriented workflow software.

One of XML's greatest values is that it provides the basis for developing information representation languages. These languages are used to facilitate information exchange between systems, or between humans and systems, and to provide for storage and recall of information. By defining a specific set of tags, it is possible to create the markup language in terms of a particular problem set. XML offers a very flexible environment for a language design, because it enables easy language evolution through extensions. New language parts can be plugged-in anywhere in the language, preserving its background compatibility – new elements will be ignored if they are not understood.

1.4.4 Web Services and Service-Oriented Architectures (SOA)

The fundamental goal of services is to enable software developers to seamlessly integrate an assortment of applications into the infrastructure of a system without any knowledge about the underlying technology. Web services technology comes as an Internet version of the traditional, LAN-based concept of services. Web services are fundamentally one-way, asynchronous messages mapped onto executable software programs. The core characteristic of a Web service is the high degree of abstraction that exists

between the implementation and consumption of a service. By using XML-based messaging as the mechanism by which the service is created and accessed, both the Web service client and provider are freed from needing any knowledge of each other beyond inputs and outputs. The intelligence for understanding how to map a message into a software program is not contained within the interface itself. However, service descriptions contain sufficient information to allow service requesters to bind to and invoke Web services.

Web services define a powerful layer of abstraction on top of the existing computing infrastructure, capable of bridging technology domains. Instead of simply exposing APIs, Web services expose dynamic service descriptions. The service-oriented approach takes the perspective that the entire software environment consists of dynamically described services that can be located and invoked on the fly when needed. If the underlying API changes, service descriptions adjust automatically, while the other components of the system can adjust to the changes at run-time.

Web services and SOA represent a fundamental shift in the design of enterprise software. SOA represent a model in which small, loosely coupled pieces of application functionality are published, consumed, and combined with other applications over a network. Application design becomes the act of describing the capabilities of services to perform a function and describing the orchestration of these collaborators. At run-time, application execution is a matter of translating the collaborator requirements into input for a discovery mechanism, locating a collaborator capable of providing the right service and orchestrating message sends to collaborators to invoke their services. These new applications, themselves, become services, thus creating aggregated services available for discovery and collaboration. According to [Sleeper 2002], SOA are marked by four essential characteristics:

- SOA are distributed.
- Systems are characterized by loosely coupled interfaces. Loose coupling requires much simpler degree of coordination and allows for more flexible (re)configuration.
- Connections are based upon vendor-independent standards.
- Systems are conceived from a process-centric perspective. Services, designed with a task-orientation, function as discrete step in a larger workflow of business process.

A common problem in building distributed applications is difficulty in evolving the application [Agrawal 2001]. Most of SOA intend to reduce application rigidity by allowing the application to locate services dynamically through service registries that describe service input and output interfaces, among many other details. Supporting dynamic run-time discovery and binding to Web services at run-time is one of the key points of flexibility in a service-oriented architecture.

The key open and interoperable technologies that support Web services are: Universal Description, Discovery and Integration (UDDI), which provides a standard way to publish and discover Web services using XML, Web Services Description Language (WSDL), which provides a standard way to describe Web services using XML, and Simple Object Access Protocol (SOAP), which provides a standard way to consume Web services using XML messaging.

SOA are based upon the interactions between three roles: service provider, service registry, and service requestor. Together, these roles and operations act upon the Web services artefacts: services and service descriptions. Service provider is the platform that hosts access to a service. Service provider or publisher is generally responsible for providing the business logic, generating a WSDL description of the interface(s), providing the HTTP invocation mechanism, providing marshalling between SOAP and the business object or XML document, and publishing to a UDDI-compliant registry. Service registry is a searchable registry of service descriptions where service providers publish their service descriptions. Service requestors find services and obtain binding information for services during development for static binding or during execution for dynamic binding. Once a service requestor has located the service provider, the interaction is directly between the requestor and provider [Haneef 2002].

For an application to take advantage of Web services, three behaviours must take place: publication of service descriptions, lookup or finding of service descriptions, and binding or invoking of services based on the service description. To be accessible, a service description needs to be published so that the service requestor can find it. The service requestor retrieves a service description directly or queries the service registry for the type of service required. The find operation can be involved in two different lifecycle phases for the service requestor: at design time to retrieve the service's interface description for program development (static binding), and at run-time to retrieve the service's binding and location description for invocation (dynamic binding). Finally, the service requestor invokes or initiates an interaction with the service at run-time using the binding details in the service description to locate, contact and invoke the service.

There are three methods for binding to a specific service:

- *Static binding* is built at design time by locating the service implementation definition. The service definition contains a reference to the service interface, which will be used to generate the service proxy code.
- *Design time dynamic binding* is defined in a service interface definition. Using the service interface definition, a generic service proxy can be generated. This service proxy can be used to access any implementation of the service interface. The generic service proxy will contain code to locate a service implementation by searching a service registry.
- *Run-time dynamic binding* is similar to design time dynamic binding, only the service interface is found at run-time. After the service interface is found, the generic proxy code is generated, compiled, and then executed.

1.4.5 Code mobility

Code mobility can be defined informally as the capability to dynamically change the bindings between code fragments and the location where they are executed. It is an approach where programs are considered as documents, and should therefore be accessible, transmitted, and displayed (i.e., evaluated) as any other document. By freeing the behaviour of distributed components from their location, mobile code has the potential to change radically the way distributed applications are developed and deployed. Mobile code systems address a wide range of needs and requirements, such as load balancing, service

customization, dynamic extension of application functionality, autonomy, fault tolerance, and support for disconnected operations.

The idea of decentralizing management functionality has been followed by the approach called management by delegation. Instead of exchanging bare Client/Server messages the management station can specify a task by packing into a program (management policy) a set of commands to agents and sending it to the devices involved thus delegating to them the actual execution of the task. A portion of the functionality of the management station is actually decentralized and put directly onto the devices. Moreover, since the code fragments (management policies) are not statically bound to devices they can be changed and re-sent by the management station at any time. This enables more flexibility because the management station can customize and enhance dynamically the services provided by the agents on the devices. Policies can be considered as pieces of mobile abstract code, which is distributed and executed on demand, where the policy-based management system implements one of the mobility code paradigms. For more details on the code mobility paradigms, see [Carzaniga 1997].

1.4.6 Language design principles

A great deal of effort has been spent on studying and producing principles and criteria for the design of programming languages. According to [Paige 2000], techniques, criteria, and principles for the design of modelling languages should be produced as well. As a summary from the literature (for more details, see [Raymond 2001], [van Lamsweerde 2000], [Meyer 1990], [Hoare 1989], [Hoare 1981], [Meyer 1992], [Paige 2000], [Norman Ramsey], [Fischer 1993], [Cardelli 1994], [Cardelli 1990], [Stansifer 1995]), the following presents the most important design principles that should be respected in the process of a language design:

- *Simplicity*. Simplicity of a language means few clear constructs, each with unique meaning.
- *Orthogonality* (uniqueness) in a language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures. Orthogonal features encourage uniformity in language structure that makes a language look like a toolkit of standardized parts that connect together in predictable ways.
- *Generality* refers to the existence of only the necessary language features, with others composed in a free and uniform manner without limitation and with predictable effects.
- *Clarity* of the semantics could be explained as the extent to which is obvious what was stated by the language - language constructs should suggest their semantics. Intuitive semantics can be achieved using simple syntax, familiar keywords and intuitive syntax constructs.
- *Consistency* means that there is a purpose to the design of a language. In [Hoare 1981], Hoare states that a language should include only those features which will be needed for every single application of the language. Then extension can be specially designed where necessary for particular applications.
- *Seamlessness* allows mapping of abstractions in the problem space to implementations in the solution space without changing notation, thus avoiding the impedance mismatches.
- *Expressivity* of a language is the ration between the amounts of useful properties versus the amount of extra information included - it is a measure of what it can be used to say.

- *Scalability*. The principle of scalability states that a modelling language should ideally be useful for both small and large systems. The modelling language must provide the means to collect abstractions, name them, and hide their details.
- *Abstraction mechanisms*. The gap between the abstract data structures that characterize the problem domain and built-in language types should be minimal.
- *Verification mechanisms* are ways in which domain experts can check whether the model matches the application.
- *Formality*. According to [van Lamsweerde 2000], a specification is formal if it is expressed in a language made of three components: rules for determining the grammatical well-formedness of sentences (the syntax), rules for interpreting sentences in a precise, meaningful way within the domain considered (the semantics), and rules for inferring useful information from the specification (the proof theory).

1.5 Thesis structure

The thesis outline is represented in Figure 1-3. This chapter has presented the scope of our research. Chapter 2 examines various existing policy-based management frameworks. Chapter 3 demonstrates the main steps of the analysis and design used in the course of this research. Goals of the research were derived from the analysis of existing work in the domain of policy-based management and other related fields, operationalised in a number of requirements and then realized as a number of design decisions. Chapter 4 presents the framework for policy-based management, built on the analysis and design presented in the previous chapter. The framework will be presented at a conceptual level, from two viewpoints: structural (framework components) and functional (framework processes). Chapter 5 presents the proposed policy language in more details, by giving the formal definition of the language. Chapter 6 presents the work on an implementation project, whose goal was to investigate some practical aspects of our ideas. A policy modelling environment was designed and implemented in order to explore possibilities for visual representation and simplification of our policy specification language, and to give some ideas about the look and feel of the language in practical use. Chapter 7 presents a critical analysis of solutions proposed by this research. We will separately present critical analysis of the policy framework and the policy specification language, which was based on the general quality factors derived from related literature. Chapter 8 summarizes our work on policy-based management, describing the contribution made and suggesting areas for future work.

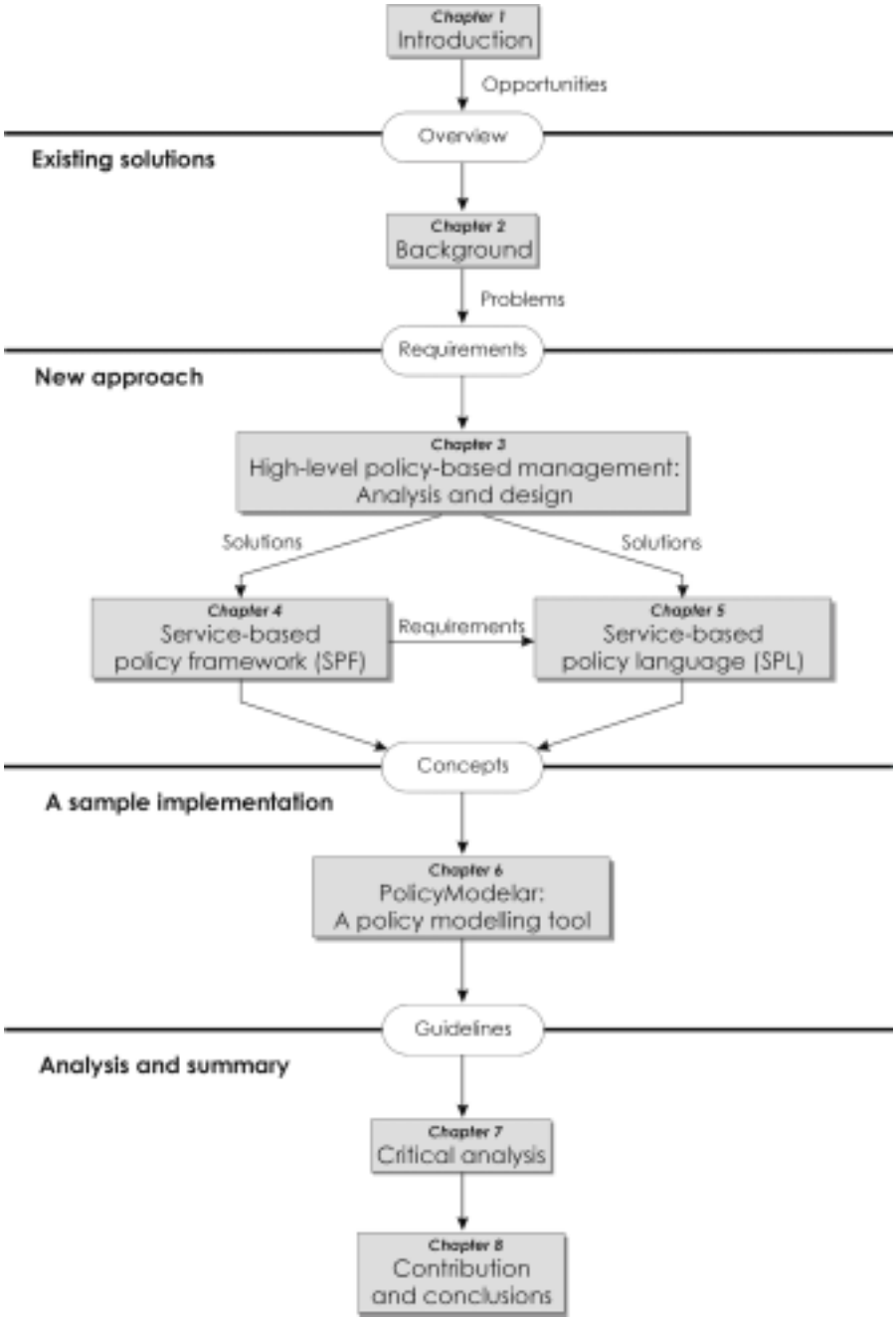


Figure 1-3 Thesis outline

Chapter 2

2 POLICY-BASED MANAGEMENT CONCEPTS

In this chapter we will present a number of policy-based management frameworks. Our goal is not to cover all the existing concepts, but only to indicate the complexity of the problem of policy-based management in general, to present the diversity of possible approaches, and to identify various requirements for a potential solution of the problem. A more comprehensive survey of different policy specification approaches may be found in [Stone 2001] and [Damianou 2002].

With the automation of many aspects of management in distributed systems and computer networks, there is the need to represent management policy within the computer system so that automated managers can interpret it in order to influence their activities [Moffett 1994]. To handle the complexity of policy decisions and to ensure a coherent and consistent application of policies network-wide, the policy specification language should ensure unambiguous mapping of a management objective to a set of policy actions. Policy-based management frameworks monitor and manage a distributed computing environment as a whole, rather than configuring and reconfiguring individual pieces of equipment. They use policies as basic management instruments to correlate information about each user and the application running, taking into account the three major aspects: security, business priorities and system characteristics.

A number of policy-based management concepts exist, and in this chapter we will have a look at the most representative among them. This will include the Policy Description Language (PDL) and Network Management System from Bell Labs (see [Kohli 1999], [Lobo 1999], [Virmani 2000]), Internet Engineering Task Force (IETF) Policy Framework (see [Moore 2001], [Judd 1998], [Westerinen 2000]), Reference Model for Open Distributed Processing (RM-ODP) from ISO/ITU (see [ISO/IEC 1996], [ISO/IEC 1995], [ISO/IEC 1995a], [ISO/IEC 1997]), and Ponder and Imperial College Policy Framework (see [Lupu 1997], [Lupu 1997a], [Lupu 1996], [Marriott 1996]).

2.1 PDL and Network Management System

The Network Management System from Bell Labs is based on the policy specification language called PDL. PDL was developed for the purpose of specifying event triggered network management activities. The language can be described as a real-time specialized production rule system to define policies [Lobo 1999]. A policy in PDL is composed of a set of rules, which specify the desired behaviour of some resources in the network. The rules may be expressions of two types:

- *Policy rule propositions*, which are event-condition-action expressions in the form:
  event **causes** action **if** condition
- *Policy defined event (pde) propositions*, which are event-condition-event expressions in the form:
  event **triggers** pde(m_1=t_1,…m_k=t_k) **if** condition

Policies are triggered after a pre-determined stream of primitive event instances is detected. Primitive events in PDL may be generated by the network or by the policies, according to the pde propositions. In

a pde proposition, the m_i's are the attributes of the policy-defined event, with the t_i's being their values. The m_i's can be attributes from other primitive events that appear in the event, they could be constants or the result of operations applied to the attributes of the other primitive events in the proposition. Complex events can be defined as disjunctions or conjunctions of primitive events. Policy actions represent simple device-specific actions to be executed at a given device/system or workflows of simple actions.

The following represents an example policy specification in PDL:

We have assigned the number 5559991 to a Customer, and we would like to allow a maximum of 15 connections at the beginning of each day.

CoarseTimeEvent **causes** ModemPoolAssigment(5559991; 15) **if** (CoarseTimeEvent: Time = "morning"):

2.1.1 Workflows

In PDL, a script language of workflows is used for specification of complex management activities. Each workflow is implemented as a PDL policy, where the actions are the component actions of the workflow and the events are the success, failure, and other events raised by the execution of these components. In contrast to policies in PDL, where the events triggering rules are coming from devices around the network, the events in a workflow are local to the workflow and its sub-activities.

The workflow script language consists of four types of symbols, for: actions or activities, exogenous events (trigger the workflow), endogenous events (generated by the workflow), and local workflow events (generated by the workflow activities). Local, exogenous and endogenous event symbols may have attributes. Using those symbols, four types of statements (propositions) may be specified: dependency workflow propositions (starts activities), alert activity propositions (send alerts to activities), instance termination propositions, and conflict workflow propositions (specify activities that are in conflict). The following is an example of a workflow:

A workstation named S, that also acts as a router, is connected with a serial line to a similar router named P. S is located in a small branch office while P is a concentrator node at a central location. Suddenly there is a burst of traffic through S that causes the memory allocated for networking to overflow and the system crashes. P detects that the link has failed and sends a critical network event to the network failures policy manager. The policy manager after receiving the event starts the following process:

- *It tries to determine if the serial interface still has a carrier signal.*
- *If there is no carrier signal the manager can decide the link is not operational.*
- *To check further the state of the link, the serial interface in P is put in a loop back to check P's side of the link. In addition, other hardware level tests are run on P to check the status of the link. After all the checks complete, the manager reports that there is a failure in S or the S side of the serial link is broken.*

Workflow: SerialLinkFailure(P; S)
Start CarrierSignal(P; S)
Terminate_with success("link"; P; S; "down") **after** $\text{CarrierSignal}_{\text{fails}}$
Start LoopBack(P; S) **after** $\text{CarrierSignal}_{\text{succeeds}}$
Start PhysicalLayerTest(P; S) **after** $\text{CarrierSignal}_{\text{succeeds}}$
Terminate_with success("link"; S; P; "down or"; S; "down")
after $\text{LoopBack}_{\text{succeeds}}$; $\text{PhysicalLayerTest}_{\text{succeeds}}$

Terminate_with success("problems with link in"; P)
after $\text{LoopBack}_{\text{fails}}$ | $\text{PhysicalLayerTest}_{\text{fails}}$

2.1.2 Network Management System

The PDL management model is presented in [Kohli 1999]. The distributed policy-based management system consists of a collection of cooperating policy enabled nodes called Policy Elements. These nodes communicate using events and actions, and cooperate to enforce given policies. The leaf elements of the hierarchy interface various hardware/software elements in the network to the policy server. Interior elements coordinate between leaves and/or other interior elements.

Every Policy Element consists of several sub-systems (services). Each of these performs a specific policy management role: Policy Engine manages the registration and execution of policies, while Action/Workflow Manager processes action/workflow requests. Actions triggered by the Policy Engine are routed to the Action/Workflow manager that determines where the action is to be executed and sends the appropriate request. In addition to the common parts of the policy element, interior elements have Domain Distributor, which determines the appropriate policy elements to receive particular action requests and policy registrations. Every policy specifies a set of domains to which it applies as property-based specifications of the target elements.

Leaf elements have several specialized services that allow them to interface to devices and systems in the network: Action Mapper, which maps action requests from policies to actual device specific commands to be performed by the device, Event Mapper, which converts device-specific events into policy events, Event Filter, which only passes events that are desired by a policy, and Domain Filter, which dynamically determines if a leaf element is part of the action or event domain specified by a policy.

When a new policy is introduced in the system, it is analyzed and a policy element is chosen as the root of the hierarchy that will enforce this policy. The policy is then registered with this element, and decomposed into a local (coordination) part and several sub-policies that will cooperate to realize the policy goals. Actions for the local policy are registered with the Action/Workflow manager and the local Policy Engine of the element. The remaining sub-policies are distributed to policy elements that form the next level in the policy server hierarchy for the policy. Each of these elements performs the same registration steps specified above. This process terminates when the element being registered with is a leaf element. Leaf elements register events desired by the local policy with their Event Filter. They also register the domain expressions specified in the policy for actions and events with the local Domain Filter.

Policy execution is initiated by the device-specific events at leaf elements. These events are mapped to a canonical form called policy event. Policy events are passed through the Event Filter to the Domain Filter, which discards all events whose domain membership constraint is not satisfied.

Any event that survives both Event and Domain Filters is delivered to local Policy Engine. When the event expression and condition of any rule in the policy are satisfied the Policy Engine requests the

Action/Workflow manager to process the associated action for that rule. It may represent an action to be executed or an event to be delivered. In the case of a leaf element these actions may map to some sequence of actions that would act on hardware/software entities in the network. This behaviour is repeated at all elements in the hierarchy to which events are delivered in this step.

The invocation of an action corresponding to a workflow results in the Action/Workflow manager registering the policy implementing it, and then raising the appropriate start event to the element where the workflow is to be executed. The execution of the workflow proceeds by invoking the actions triggered by the start event and feeding back the success and failure events of these actions to the PDL program. This results in the execution of other component actions. This process repeats until the workflow is completed, which results in its success or failure event.

Different parameter configurations of the policy elements, event schemas, action schemas and device specific event mappings to policy events are stored in a directory. System manager specifies a policy in two steps. First, the manager will consult a directory server to obtain events that the system is able to monitor, actions that can be invoked by a policy and functions that system supports to evaluate the status of the environment. Then the manager will write policies by combining events, actions and functions from the directory and store them back in the directory.

2.1.3 Remarks

The strength of the PDL language is in its message- (event-) based flow control, which makes its management model very flexible. However, for a higher-level management of bigger systems it lacks the expressive power and support for proactive management. PDL has no abstraction/refinement mechanisms, and the specification of complex management activities tends to be long and hard to follow. This is likely to produce the scalability problems. Workflows are written as scripts that are translated into sequences of PDL policies for execution. Every step of a workflow is implemented by a PDL policy. Workflows are executed step-by-step, and each step depends on the events generated by the previous step. The events that a PDL workflow manipulates are local to the workflow and its sub-activities. As a result, workflows are encapsulated, with a predefined structure and sequential execution.

2.2 IETF Policy Framework

The IETF Policy Framework encompasses a policy framework definition language, a policy architecture model, policy terminology, and a policy model. We will firstly look at the general IETF policy architecture, and then we will present the concept of directories and Directory Enabled Networks (DEN). Finally, we will present the policy model and related policy specification approaches.

2.2.1 General architecture

Centralised policy management architecture proposed by the IETF includes four key elements: policy management interface, which supports the specification, editing, and administration of policies, dedicated

policy repository, which provides storage and retrieval of policies as well as policy components, policy decision point(s) (PDP), which is responsible for handling events and making decisions based on those events, and policy enforcement point(s) (PEP), which enforces policy it has received from the PDP.

A PDP generally takes the form of policy servers. They are responsible for assessing the conditional criteria of policy rules. Policy servers communicate with a directory server to obtain configuration and other information about the network, its users, and applications, in order to make policy decisions. The PDPs are gathering all relevant information, make a decision based on the administrator's predefined set of policies, and then communicate that decision to the relevant network node(s).

A PDP instructs a PEP to behave according to the policy rules which the PDP itself received from a repository. The PEP takes care of the actual implementation of the rule. The PDP evaluates the policy rules inherent in policy definitions and generates appropriate instructions for the PEPs. Policy conflicts may be detected at the time the policy is entered into the policy repository or when the policy decision is made at the PDP.

Policies need both static and dynamic information about the system. In the IETF architecture, a directory serves as the central location for storing policies, profiles, user information, network configuration data, and IP infrastructure data such as network addresses and name server information. The directory server is used to store relatively static information about the network, whereas policy servers and policy-enabled network devices interpret this information in the context of the current state of the network.

2.2.2 Common Information Model and Directory Enabled Networks

With the arrival of policy-based network management, there is an increasing demand for a unified model that can represent the common components of a network device. Common Information Model (CIM) is an approach to the management of systems and networks that applies the basic structuring and conceptualization techniques of the object-oriented paradigm for describing overall management information in a network/enterprise environment. The CIM model defines the vocabulary and grammar for abstract policies. It defines two hierarchies of object classes: structural classes, which represent policy information and control of policies, and relationship classes, which indicate how instances of the structural classes are related to each other.

In parallel to the development of CIM, work to construct a model for networking equipment has been carried out as well. The result of that effort has called Directory Enabled Networks (DEN). DEN was designed to provide the building blocks for more intelligent management by mapping concepts from CIM (such as systems, services and policies) to a directory, and integrating this information with other elements in the management infrastructure.

A primary goal of CIM is the presentation of a consistent view of the managed environment independent of the various protocols and data formats supported by those devices and applications. In contrast, DEN focuses more narrowly on user, network device, and service/application management. The DEN

specification enhances CIM, by providing a network model for representing network elements and services, and extends CIM, by adding models for defining and enforcing policy and network services.

2.2.2.1 Directories and DEN

At its core, a directory is an information repository. Directories provide a convenient means for storing static data that can be arranged in a hierarchy. Although a directory associates attributes with objects, what makes it different from a database are four key properties:

- Directory objects are essentially independent of each other, whereas database objects are interrelated.
- Directories organize their information using the notion of containment, which is not naturally implemented in databases.
- Directory can have specific access controls assigned to an object and even attribute of an object.
- Directories, unlike databases, are optimized to perform a high number of reads vs. writes.

General-purpose directory services provide a universal way of naming, finding, accessing, and protecting resources across both space and time. They also provide the foundation for adding, modifying, removing, renaming, and managing system components. A directory service is characterised by two aspects: the namespace, denoting the range of information it exposes, and the programming interface which determines how the namespace's content is accessed by clients.

Five main limitations of directories adversely affect their use in networking applications:

- No standard information model is in use for existing directory objects, much less for network objects. DEN purpose is to provide a single information model and schema in which network information can be shared.
- Information stored in a directory is bound to a single name representation and organization. The power of DEN comes from its capability to relate different objects to each other, and to map these (purely) object-oriented constructs into a representation that the directory can accommodate.
- Directory services are incapable of differentiating between different network resources. The DEN specification addressed lack of metadata needed to describe objects and their roles by including important metadata in the object classes that model and manage network elements and services. DEN (as well as CIM) also used a consistent naming methodology so that attributes of different objects that served a similar purpose were named similarly. Finally, DEN introduced the notion of roles.
- Directory services usually are not used to model objects; they are used to represent them. New classes and attributes need to be dynamically associated with the device. DEN defines fundamental knowledge domains to represent this information. DEN further describes ways in the model to link information from one domain to information in another domain.
- Directories cannot represent behaviour associated with objects (interactions). Traditional directories are used as static warehouses to store information. Current and future applications will require more intelligent interaction between directory information and the network environment.

In DEN, a directory is a special purpose database that contains information about the nodes, or devices, attached to an enterprise network. The goal of directory-enabled networking is to establish a common

management interface for all resources in an enterprise, including applications, systems, services, and users [Clark 1999]. DEN presents a single mechanism to represent how network elements provide services, and how policies can be used to control them. According to [Judd 1998], the integration of the network infrastructure with the directory service allows the applications and users to discover the existence of devices and relationships by querying the directory service, a single authoritative place to obtain the information of interest, rather than contacting the individual devices and aggregating the results.

Managing network services through DEN allows administrators to assume a network-centric view of services and provision them on an end-to-end basis as opposed to managing individual network elements. Directory integration also provides a foundation for policy-based management of network services on a per-user basis. Because DEN integrates network information into the directory, it is possible to associate specific users with network properties. This combination enables dynamic configuration of network to support each user's needs.

2.2.3 IETF Policy model

The IETF do not define a specific language to express policies but rather a generic object-oriented information model, derived from CIM/DEN, for representing policy information. The philosophy of the IETF is that business policies expressed in high-level languages, combined with the network topology and the QoS (Quality of Service) methodology to be followed, will be refined to the policy information model, which can then be mapped to a number of different network device configurations [Damianou 2002b].

The IETF policy model considers policies as rules that specify actions to be performed in response to defined conditions. Each policy rule consists of a set of conditions and a set of actions:

if <condition(s)> **then** <action(s)>

The condition part of the rule can be a simple or compound expression specified in either conjunctive or disjunctive normal form. The action-part of the rule can be a set of actions that must be executed when the conditions are true. Policy rules may be aggregated into policy groups. These groups may be nested, to represent a hierarchy of policies. Policy rules themselves can be prioritised.

For example, in the policy presented in Figure 2-1 a “class of service” mechanism is used to differentiate between types of network traffic, and then a specific “quality of service” is then applied to one of the resulting service classes. The policy consists of three rules, each of which defines one class of service.

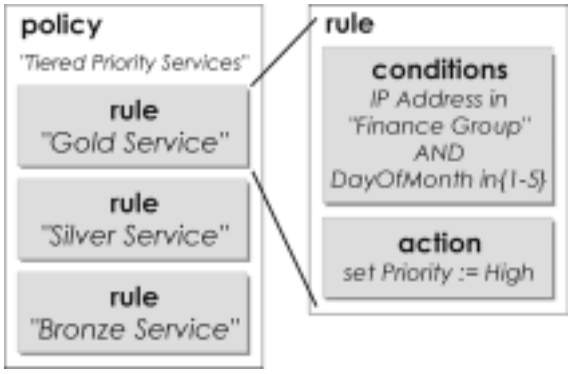


Figure 2-1 Example IETF QoS policy

Applicability of policies to resources is controlled by the mechanism of roles. A role represents a functional characteristic or capability of a resource to which policies are applied. Role is used as a selector – it selects a particular set of policies assigned to the behaviour it identifies. A policy administrator assigns each resource to one or more roles, and then specifies the policies for each of these roles. The Policy Framework is then responsible for configuring each of the resources associated with a role in such a way that it behaves according to the policies specified for that role. Thus, roles provide a level of indirection that allows binding policies to any interface.

2.2.3.1 Policy Core Information Model (PCIM)

The Policy Core Information Model (PCIM) defines an object-oriented information model for representing policy information, developed as extensions to CIM. This model is described as a "core" model since it cannot be applied without domain-specific extensions.

PCIM extends CIM with classes to represent policy information. It defines two hierarchies of object classes: structural classes, representing policy information and control of policies, and association classes, which indicate how instances of the structural classes are related to each other. The classes comprising the PCIM are intended to serve as an extensible class hierarchy (through specialization) for defining policy objects that enable application developers, network administrators, and policy administrators to represent policies of different types.

In PCIM, policies are seen as business goals and objectives. These high-level descriptions of network services must be translated into low-level, vendor and device independent specifications. The PCIM classes are intended to serve as the foundation for these vendor and device independent specifications. It is expected that business policies expressed in high-level languages will be refined to PCIM, which can then be mapped to a number of different network device configurations. The policy specification language proposed by this research has defined mappings to PCIM (see Appendix A).

2.2.3.2 Policy Framework Definition Language (PFDL)

PCIM is not a policy specification language. However, it can be used as a basis for development of object-based policy specification languages, such as the Policy Framework Definition Language (PFDL).

The purpose of the language is to translate from a business specification to a common vendor- and device-independent intermediate form, as a way to ensure that multiple vendors interpret the policy the same way while enabling vendors to provide value-added services.

The design of the PFDL is based on satisfying PCIM. The class and relationship hierarchies of the PCIM help define the structure of the PFDL grammar [Strassner 1998]. In PFDL, a policy is a named object that represents an aggregation of policy rules. The policy describes the overall business function(s) to be accomplished, while the set of policy rules defines how those business functions will be met.

The following is an example of a policy specified in PFDL:

If anyone accesses the draft archive at IETF, rather retrieve the draft from our local mirror.

```
IF      HTTP_URL_HOST     EQ          www.ietf.org          AND
        HTTP_URL_PATH     EQ          "/internet-drafts/*"
THEN
        HTTP_URL_HOST     REPLACE     "chiba.ccrle.nec.de"  AND
        HTTP_URL_PATH     REPLACE     "/Mirror/ftp.ietf.org/internet-drafts"
```

2.2.4 Policy specification language from AT&T

AT&T proposes a policy specification concept with ambition to extend the ITEF Policy Framework (see [Dini 1999] for more details). The goal was to address the problem of difficulty to capture and solve policy conflicts, due to the lack of clear concern related to policy activation event and guaranteed post conditions, and the need for semantic difference between various types of policies. In this approach, a policy specification is based on the generic rule of the form:

```
policy ::=   IF <pre-cond>
             THEN {<> | <action> | <plan>}
             [ELSE {<> | <action> | <plan>} <action> | <plan>}]
             [<post-cond>]
             <policy_properties>
```

The policy body expresses the functional behaviour of a policy, while other static (management domain, time interval, target, scope, etc.) or dynamic (refinement status, trigger mode, administrative state, etc.) policy properties are used for the purpose of policy management itself.

An action has preconditions, post-conditions, and/or an activation event. Activation events and state predicates (expressions on the state variables) express preconditions to apply an action. An event can be issued by a managing object, by a managed object, or generated by a system operator. A plan is a set of related actions. Within a plan, some dependencies between actions may be prescribed, such as sequences, parallelisms, exclusion, and/or temporal constraints.

A policy can be derived from goals (proactive management), or applied when certain policy preconditions are satisfied by the state of the system (reactive management). In the former case the policy is goal-oriented, while in the latter is event-oriented. For an element in the system, conditions can refer to an

event occurring in the system. Actions are those triggered and directly performed by a well-defined protocol primitive (for example Simple Network Management Protocol - SNMP, Common Management Information Protocol - CMIP, Lightweight Directory Access Protocol - LDAP, etc.).

Management actions and plans are initiated by a policy. A policy is selected according to management goals or the type of activation event and its priority. The policy body focuses on particular system states according to management goals. There are three distinct situations concerning policy body:

∉ *Daemon-based policy*. The "If-Then" form can express this type, where post-conditions are guaranteed by the action. It is assumed that a daemon (policy server) captures the preconditions and triggers the action prescribed by the policy. For example:

If a managed object degrades its usage state, the polling frequency must be adapted according to a function depending on usage state.

IF (us_state = warning_active)
THEN modify_polling_frequency: 'polling_frequency = f(us_state)'

∉ *Plan-command policy*. In this case, it is assumed that a goal has been presented to the system and a plan or action has been identified. For example:

On M-GET.confirm event, receive a state list (SL) or an error list (ER). In case of an error, send a notification.

IF (M-GET.confirm)
THEN ((send: state_list = SL) 〉 ((send:"err_list = ER") | | (notify: "error_type = ET")))

∉ *Desirable-goal-oriented policy*. A goal is registered as desirable, but there are no guarantees that the goal will be satisfied. For example:

If there is no input from a mouse in 60 seconds, switch off the screen.

IF ((no_input_from_the_mouse) 〈 (counter> 60 sec)
THEN (turn_off: "screen_color = black")

Daemon-based policies and goal-oriented policies are passive objects, which are waiting for their preconditions to be satisfied to be triggered. Plan-command policies are usually received as inputs from a higher-level manager or from a system operator, which interprets the goals and activate a policy.

2.2.5 Remarks

IETF/DEN implements a bottom-up approach to standardisation. It was imagined as a common information model that can serve as a basis for the development of higher-level management solutions. Its focus is standardization of the low-level policy-based management. Higher-level management solution will be designed to plug into the standardized information base by the relation of abstraction/refinement.

IETF/DEN defines components of a policy-based management system, but not the policy-based management process itself. In the IETF policy model, there is no explicit event specification to trigger

the execution of policy actions, and no control and adaptation mechanisms. These are expected to be offered at a higher level of management hierarchy.

The DEN base schema is fairly straightforward to implement, but the concept of modelling network elements to fit into a hierarchy isn't. According to [Stevens 1999], the core policy schema provides a structure for organizing the storage of policy expressions but does not yet provide definitions for operands and operators. The operands of a policy constitute an abstract interface to the software or hardware elements with which the policy wishes to interact (the system management interface). Policy expressions must be capable of referring to the operands of the policy consistently and unambiguously.

The AT&T concept greatly enhances the IETF policy model, and it offers a solution for policy triggering by integrating events in policy specifications. However, it does not have the ambition to tackle other problems related to the management process, such as abstraction/refinement, control or automation. As a result, the focus of the concept remains on a low-level network management.

2.3 Reference Model for Open Distributed Processing (RM-ODP)

The presence of heterogeneous hardware and software represents the major difficulty in distributed systems. To solve the problem of heterogeneity in the underlying environment it is necessary to provide a level of independence from the infrastructure. Heterogeneity was the principal motivation factor for the development of open distributed processing, with a goal to enable interaction with services from anywhere, without concern for the underlying technology. Open systems are characterized by the fact that their key interfaces are published and standardized. They are based on the provision of a uniform inter-process communication mechanism and published interfaces for access to shared resources.

Standardization is essential for openness to be achieved. There are actually a number of open distributed processing standards and platforms. The ODP Reference Model is widely recognised as offering the most complete and internally consistent framework. It defines a generic architecture for open distributed systems. The definition is achieved of a minimal set of modelling concepts that constitute a very powerful and very general object model [Stefani 1995]. The RM-ODP itself can be used to specify specific distributed systems within enterprises, or used to specify other ODP component standards, so it should be thus considered a meta-standard for open distributed processing.

RM-ODP approaches the problem of describing distributed systems by expressing the effects of distribution as a number of distribution transparencies [Lewis 2000]. Transparency is the property of hiding from a particular user the potential behaviour of some parts of the system. The aim of transparencies is to shift the complexities of distributed systems from the application developers to the supporting infrastructure [Raymond 1995]. Transparencies defined in the RM-ODP are:

- *Access transparency*, which masks differences in data representation and invocation mechanisms to enable inter-working between the objects.
- *Failure transparency*, which masks (from an object) its failure and possible recovery to enable fault tolerance.

- *Location transparency*, which masks the use of information about location in space when identifying interfaces.
- *Migration transparency*, which masks (from an object) the ability of a system to change the location of that object.
- *Relocation transparency*, which masks relocation of an interface from other interfaces bound to it.
- *Replication transparency*, which masks the use of a group of mutually behaviourally compatible objects to support an interface.
- *Persistence transparency*, which masks (from an object) variations in the ability of a system to provide processing, storage and communication functions to that object.
- *Transaction transparency*, which masks coordination of activities among a configuration of objects to achieve consistency.

Aside from transparencies, RM-ODP provides system developers with a framework for an effective and disciplined approach to distributed specification. This framework is given by the RM-ODP viewpoints, which provide an orderly way to tame the complexity involved in specifying distributed systems. A viewpoint is a subdivision of the specification of a complete system, established to bring together those particular pieces of information relevant to some particular area of concern [Linington 1995]. The RM-ODP defines five complementary viewpoints at different levels of abstraction:

- *Enterprise viewpoint* is concerned with the purpose, scope and policies (business objectives).
- *Information viewpoint* is concerned with the semantics of information and information processing (meaning).
- *Computational viewpoint* is concerned with the functional decomposition, which involves specification of objects, interfaces and behaviour (logical partitioning).
- *Engineering viewpoint* is concerned with the system infrastructure required to support distribution (mechanisms).
- *Technology viewpoint* is concerned with the implementation (conformance).

A set of concepts, structures and rules is given for each of the viewpoints, providing a language for specifying ODP systems in that viewpoint. Models can be produced at different levels of abstraction within each of the viewpoints. A system specification may comprise one or more viewpoint specifications. As viewpoints are separate but inter-related views of the same system, the relations between terms in different views are subject to consistency constraints.

2.3.1 Enterprise viewpoint

Within the RM-ODP, the problem of policy-based management is addressed in the Enterprise viewpoint. The aim of the Enterprise specification is to express the objectives and policy constraints on the system of interest [Linington 1995]. The Enterprise language defines the concepts necessary to represent the behaviour expected of an ODP system and the structuring rules for using those concepts to produce an Enterprise specification. An Enterprise specification defines the purpose, scope and policies of an ODP system in terms of each of the following items: roles played by the system, activities undertaken by the system, processes in which the system participates, and policy statements about the system.

Role is one of the foundation concepts in the ODP standards, and it is defined as an identifier for behaviour, associated with an object. This definition means that a role is a placeholder for behaviour to be filled by an object that satisfies this behaviour [Lupu 1999a]. Roles are used to group specifications of the behaviour that is expected from objects assigned to them. The behaviour offered by an object can be partitioned into a number of separate services (referred to as roles) each visible at a distinct interface.

In ODP, the primary building blocks of an Enterprise specification are communities [Linington 1999]. Community represents a composition of objects formed to meet an objective. The objective is expressed as a contract, which specifies how the objective can be met. A community is defined from two aspects:

- *Collective behaviour of the community* is specified in terms of one or more of the following elements: roles of the community (including those roles which define how a community interacts with its environment), processes that take place in the community, assignment of roles to steps in processes, policies that apply to the roles and processes, and identification of those actions for which parties are accountable.
- *Structure of the community* is defined in terms of roles, policies for assignment of enterprise objects to roles, relationships between roles, relationships of roles to processes, policies that apply to roles and to relationships between roles, policies that apply to relationships between enterprise objects in the community, and behaviour that changes the structure or the members of the community during the lifetime of that community.

In RM-ODP, policy is a set of rules related to a particular purpose. Policy identifies specification of behaviour, or constraints on a behaviour, that can be changed. It is named place-holder for a piece of behaviour used to parameterise a specification in order to facilitate response to later changes in circumstances. Policies constrain the behaviour of objects in communities by governing interactions between objects fulfilling roles, creation, usage and deletion of resources, and configuration of objects and their assignments to roles. A policy rule can be expressed as permission, which states what can be done, prohibition, which states what must not be done, or obligation, which states what must be done.

2.3.2 An Enterprise language based on UML and Object-Z

The RM-ODP provides abstract languages of relevant concepts, but it does not prescribe the use of any particular notation. There have been a number of attempts to define specification languages that implement the abstract concepts of the Enterprise language. Most of those approaches are concentrating on the use of UML for structural modelling, with an appropriate set of extensions for policy specification.

An Enterprise language implementation, proposed in [Steen 2000], uses UML class diagrams for the structural specifications, while policies are specified using the Object Constraint Language (OCL). A translation mechanism is also provided, from the policy language to Object-Z (an object-oriented extension of Z language), in order to give formal semantics to policy specifications. According to [Steen 2000], the language enables policies to be specified concisely and precisely, consistency checks on the policies, verification on enterprise behaviour against a policy specification, and automatic derivation of implementation constructs for enforcing policies.

We will briefly look at the implementation of the three key RM-ODP concepts in the proposed Enterprise language, namely community, role and policy.

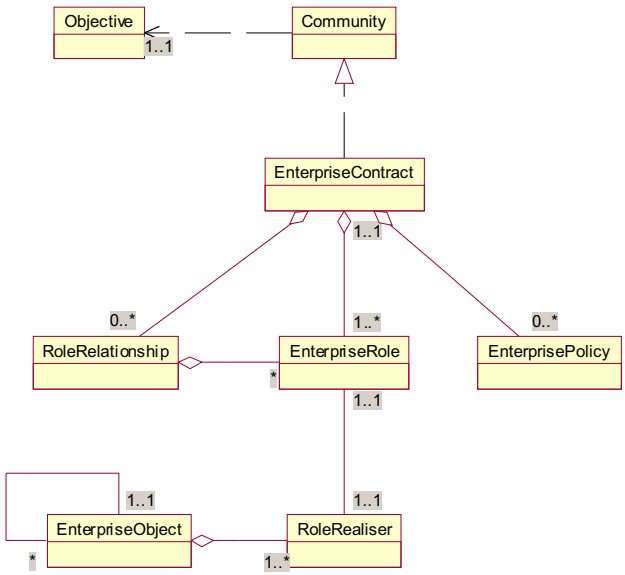


Figure 2-2 Community contract

In the proposed language, a *Community* is associated with an *Objective* (Figure 2-2) and it can be represented as an integral UML package or refined into an organization of enterprise objects and roles, specified as an UML class diagram. An *EnterpriseObject* is loosely coupled with a *Community*. The fact that an *EnterpriseObject* fulfils roles in a *Community* is given indirectly, by associations between his *RoleRealiser* and an *EnterpriseRole*.

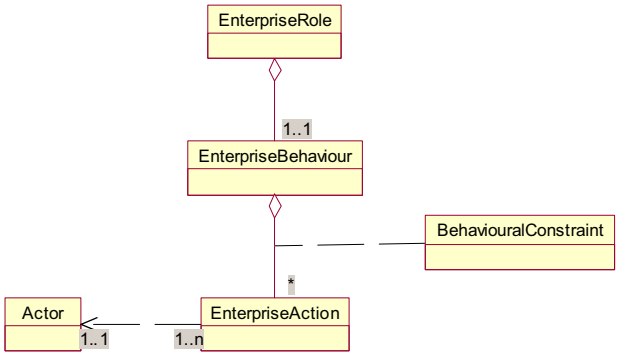


Figure 2-3 Role behaviour

In the proposed language, an *EnterpriseRole* (implements the RM-ODP notion of role) defines an *EnterpriseBehaviour*, which consists of a set of *EnterpriseActions* and a set of *BehaviouralConstraints* on these *EnterpriseActions* (Figure 2-3). At least one *EnterpriseRole* participates in each *EnterpriseAction*. In addition, each *EnterpriseAction* is initiated by precisely one *Actor* role. A *BehaviouralConstraint* describes the relation

between *EnterpriseActions* and an *EnterpriseBehaviour*. Enterprise actions are represented in the language using UML Use-Case diagrams.

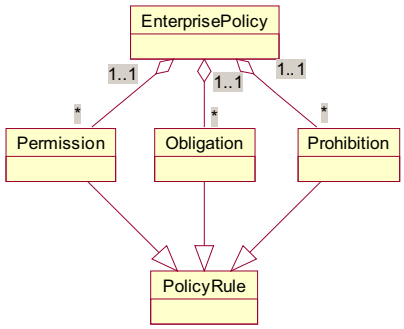


Figure 2-4 Policy structure

EnterprisePolicy (implements the RM-ODP notion of policy) is represented as an aggregation of *Permissions*, *Obligations* and/or *Prohibitions*, which are all specialization of *PolicyRule* (Figure 2-4). Logical expressions written in OCL are used to specify policy rules for roles defined in the UML class diagrams. The policy rules always have to be interpreted within the context of a related UML class diagram. A policy rule has the following form:

[R?] **A** <role> **is** (**permitted** | **obliged** | **forbidden**) **to** (**do** <action> [**before** <condition>] |
satisfy <condition>)[, **if** <condition>][, **where** <condition>][, **otherwise see** <number>].

The following example demonstrates the use of conditions to constrain the behaviour of a Library community.

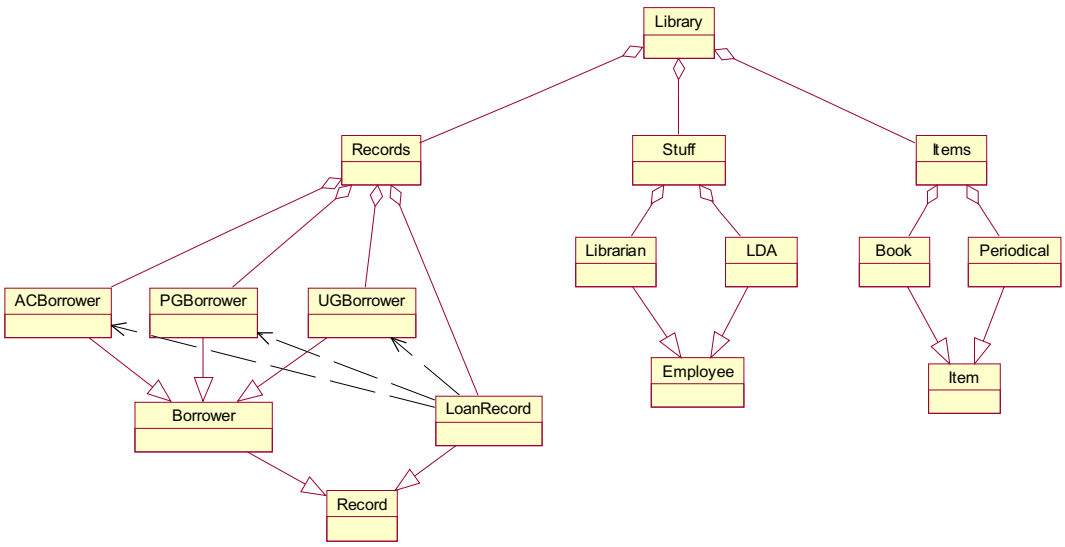


Figure 2-5 The Library community

[R1] **A** Borrower **is permitted to do** Borrow(item:Item), **if**(fines < 5*pound).
[R2] **A** UGBorrower **is forbidden to do** Borrow(item:Item), **where** item.isKindOf(Periodical).

[R3] **A** Borrower **is obliged to do** Return(item:Item) **before** (today > dueDate), **if** (loans ϕ exists(loan | loan.item = item)), **where** (dueDate = loans ϕ select(loan | loan.item = item).dueDate), **otherwise see** R4.

Policy statements R1 and R2 specify a permission and a prohibition respectively, while R3 is an obligation. The *otherwise* clause indicates an exception when the obligation is violated, with corresponding actions specified in another policy (R4).

2.3.3 Remarks

In contrast to the IETF/DEN bottom-up approach to standardization, which offers a common information model to build upon, the RM-ODP initiative is taking the top-down approach that offers the general concepts that can be refined into concrete management frameworks. RM-ODP defines only the basic principles of role-based management of distributed systems, not a management model for immediate use. RM-ODP doesn't provide:

- ∉ Design methodology. The RM-ODP does not explain how to proceed to build viewpoint models; it only states what these models should specify, and how they should be structured around the object concept.
- ∉ Notation or specification language for writing viewpoints models. Consequently, the lack of a standard notation makes talking about ODP models difficult.
- ∉ Mappings between viewpoints.
- ∉ Policy-based management process.

According to [Stefani 1995], the Reference Model remains largely unprescriptive with respect to the Enterprise and Information languages. This reflects both the relative absence of well-established principles governing the description of systems from these two viewpoints and the neutrality of the Reference Model with respect to the diversity of enterprise and information models.

Management at the system level (Enterprise viewpoint) is not precisely elaborated in RM-ODP. In an enterprise specification, an ODP system and the environment in which it operates are represented as a community. At some level of description the ODP system is represented as an object in the community [ISO/IEC 1995a]. From the definition of the community, one could conclude that it represents some sort of a composite object that defines a collective behaviour of the community. However, it isn't clear if the collective behaviour encompasses the encapsulation – if it represents an exclusive way of interacting with the community. One of the possible interpretations is presented in [Linington 1998] and [Linington 1999], and it is suggested that the collective behaviour of a community is represented in terms of roles and it doesn't automatically include the existence of an interface nor the encapsulation.

The language presented in 2.3.2 offers a rather simplified view of the RM-ODP enterprise language. By focusing only on the UML class diagrams, the language uses only a fraction of the UML expressive power. Even though UML does envisage the usage of OCL expressions with the class diagrams, the proposed language would gain more expressive power by using the UML state and/or activity diagrams to specify enterprise behaviour. The usage of the Use-Case diagrams to specify role interactions within

communities is not really appropriate, because the Use-Case diagrams in UML consider a system (in this case community) as a "black box" and capture only its interactions with its environment.

2.4 Ponder and Imperial College Policy Framework

Imperial College Policy Framework is based on a policy specification language, called Ponder, and a number of supporting services. Firstly, we will present the language, and then we will look at the supporting framework services.

2.4.1 Ponder

According to [Damianou 2000], Ponder is a declarative, object-oriented language for specifying different types of policies, grouping policies into roles and relationships, and defining configurations of roles and relationships as management structures. Key concepts of the language include roles to group policies relating to a position in an organization, relationships to define interactions between roles, and management structures to define a configuration of roles and relationships pertaining to an organizational unit such as a department [Damianou 2001].

Policies in Ponder are defined as ADTs (Abstract Data Types), where type definition introduces a new user-defined policy type (class) from which one or more policy instances (objects) can be created. All policy types can be parameterised. Ponder makes explicit distinction among the following policy types:

- **Basic policies** may be:
  - *Authorization policies*, which are essentially security policies related to access-control and specify what activities a subject is permitted/forbidden to do, to a set of target objects.
  - *Obligation policies*, which specify what activities a subject must do to a set of target objects and define the duties of the policy subject.
  - *Refrain policies*, which specify what a subject must refrain from doing and are similar to negative authorization policies but are interpreted by the subject.
  - *Delegation policies*, which specify which actions subjects are allowed to delegate to others. A delegation policy thus specifies an authorization to delegate.
- **Composite policies** are used to group related policy specifications within a syntactic scope with shared declarations, in order to simplify the policy specification task for large distributed systems. Four types of composite policies are provided:
  - *Groups*. A group is a syntactic scope used to declare a set of policies and constraints that are grouped together as they have some semantic relationship and should be instantiated together.
  - *Roles*. A role groups the policies specifying the duties and rights relating to a position within an organization. A role is thus a particular type of group in which all policies have the same subject domain.
  - *Relationships*. Relationships specify policies pertaining to the relationship rather than the individual participating roles (policies about interactions between roles and policies relating to shared objects and protocols for interaction).

- *Management structures*. A management structure defines the configuration of roles and relationships in organizational units in terms of the required instances of the roles. It may be used for instantiation of concrete configuration instances within an organizational unit.

∉ **Meta-policies** are policies about which policies can coexist in the system or what are permitted attribute values for a valid policy. They specify constraints on the permitted types of policies or their policy elements in order to handle semantic conflicts.

In Ponder, domains are used to group objects to which a common policy applies, to partition management responsibility in large systems or for the convenience of humans. According to [Lupu 1997b], an advantage of specifying the policy scope in terms of domains is that objects can be added and removed from the domains to which policies apply without having to change the policies. The subject and the target for a basic policy are specified using domain scope expressions (combine domains to form a set of objects) or by formal identifier of type subject or target. Actions represent the operations defined in the interface of a target object. They specify what must be performed for obligations and what is permitted for authorizations.

The following is an example of an obligation policy:

When the patient's temperature exceeds 37 degrees, a nurse should administer analgesics to that patient if he/she is not allergic.

```
domain a = /wardA
domain b = /sectionD/patient
constraint
        chart(s) = s.isNotAllergic()
type
        oblig drugsAdminT1 (subject s, target t)
        {
                on t.temperature > 37
                do administer(analgesics)
                when chart(s)
        }
inst
        oblig da1 = drugsAdminT1(a/nurse, b/stevens)
```

A positive authorization policy may be specified as follows:

Members of the NetworkAdmin domain are authorised to load, remove, enable or disable objects of type ProfileT in the Nregion/switches domain.

```
inst auth+ switchProfileOps
{
        subject /NetworkAdmin ;
        target <ProfileT> /Nregion/switches ;
        action load(), remove(), enable(), disable() ;
}
```

Example of a delegation policy:

The subject of the switchProfileOps policy can delegate the enable and disable actions on policies from the domain /Nregion/switches/typeA to grantees in the domain /DomainAdmin for 24 hours from the time of creation.

```
inst deleg+ (switchProfileOps) delegSwitchOps
{
        grantee /DomainAdmin;
        target /Nregion/switches/typeA;
        action enable(), disable();
        valid Time.duration(24);
}
```

2.4.2 Policy Framework

The policy framework was envisaged as a collection of the following services: role service, policy service, domain service, and event service.

Role service ensures the persistence of role objects and supports the basic role functions defined in the role interface: adding and removing policies, constraints and relationships, and assigning and enabling/disabling roles. A role is a collection of policies relating to the same subject domain, and it is implemented as an object. The corresponding role control object takes care of loading, unloading, enabling, and disabling all the policy instances. In addition, it handles the updating of the domain associations and the domain membership changes on behalf of all policy objects contained inside the role.

Policy service is used to manage and coordinate access to policy objects stored in the Domain service, and to instantiate new policies from existing policy types. It acts as the interface to policy management; it stores compiled policy classes, instantiates new policies from existing policy types and distributes new policy objects. Policies are implemented as persistent objects and maintained by servers. A policy object can be loaded into its enforcement components and, once loaded, it can be enabled, disabled or unloaded from its enforcement components by the Policy service. The fact that Ponder policies explicitly define their subjects and targets makes the automated distribution of policies possible.

Domain service creates and holds domains to be used for management. It manages a distributed hierarchy of domain objects and supports the efficient evaluation of subject and target sets at run-time. A domain acts as the unit of activation and distribution for the conceptually passive domain objects. Each domain object holds references to its managed objects but also references to the policy objects that currently apply to the domain. Any policies that are in a loaded or enabled state need to be informed of changes to the memberships of domains to which the policy refers.

Event service notifies special enforcement components of events which trigger its obligation policies. It collects and composes system events as well as those from the managed objects in the system, and forwards them to the subscribers. The Event service exposes a publish/subscribe interface whereby clients can subscribe to receive certain types of events.

2.4.3 Remarks

Ponder and RM-ODP have essentially different approach to the implementation of Role Theory (see 3.1). The difference is obvious in the way they implement two basic concepts, namely role and domain. RM-ODP is focused on the behavioural aspect of Role Theory. Role is an identifier for (reference to) a behaviour. The rights and duties of a role are given by the context in which it was defined. Domains are controlled groups of objects that have a representative behaviour, assigned to domain controllers. Objects that are organized in domains and communities are not just aggregated, but they offer a new quality – a collective behaviour.

On the other hand, Ponder is focusing on the organizational aspect of Role Theory, which makes it more suitable for design time (static) specifications, and access control. In Ponder, roles and relationships can be used to describe and analyse the organizational structure. Role is its rights and duties – a position in the system organization. According to [Lupu 2000], Ponder domains are merely a grouping construct, similar to the ODP group, so differ from ODP domains which have a single controller role that controls roles with respect to a specific aspect of their behaviour. Members of a domain should be somehow behaviourally compatible, to make their grouping meaningful from the management point of view. However, domains in Ponder are defined to be a loose grouping mechanism, so it is impossible to define what exactly it is possible to do with a domain in terms of management.

Ponder is a product of a very important work in the domain of policy-based management, because it is a result of a large experience and a detailed and precise analysis of the problems in the field. However, even though the language individually addresses the most important policy specification problems, its exclusive focus on details jeopardised the language integrity and simplicity. Ponder gives an impression of a collection of rather independent features that don't fit very well together. As a result, its syntax is very complex. According to [Sebesta 2002], orthogonality is closely related to simplicity, and too much orthogonality might lead to unnecessary complexity. For example, Ponder has 5 distinct policy types (basic policies) and 4 policy grouping concepts (composite policies), each with a different syntax.

The entire policy-based management concept revolves around the notion of modality: policy representation, policy triggering, conflict resolution, and policy enforcement are all modality-dependent. This might be a good solution for a low-level management scenario that involves direct management of object/roles, but, as remarked in [Wies 1995], the modality is of little use when trying to refine policies or apply them to management services.

Ponder claims that its structural representation of a system is dynamic. However, design of the policy life-cycle introduces lots of complexity in the management model. Even though policies themselves don't need to be changed every time something happens, role assignments and domain memberships certainly do. Constant cascading updates are required in the management system, which is likely to lead to serious scalability problems. Moreover, the lack of support for control and adaptation of the management process eliminates the possibility for its automation.

2.5 Summary

This chapter has presented a brief survey of the most interesting policy-based management frameworks, each with its own policy specification concept. The survey has demonstrated the complexity of the problem of policy-based management, and a number of different approaches that address certain aspects of it.

In the next chapter we will present a more detailed analysis of the existing solutions for policy-based management, identify their deficiencies, derive research goals and requirements, and make design decisions accordingly.

3 HIGH-LEVEL POLICY-BASED MANAGEMENT: ANALYSIS AND DESIGN

The outline of this chapter is presented in Figure 3-1. *It demonstrates the main steps of the analysis and design made in the course of this research. Goals of the research were derived from the analysis of existing work in the domain of policy-based management and other related fields, presented in previous chapters, operationalised in a number of requirements and then realized as several design decisions.*

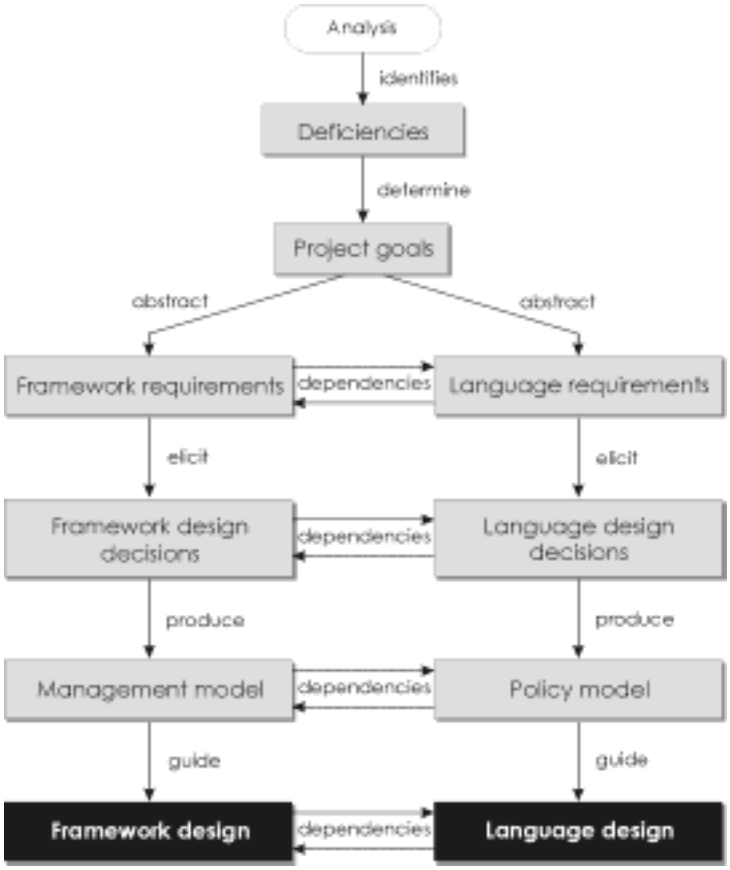


Figure 3-1 From the problem analysis to the framework design

All policy statements are made in a context [Steen 1999], defined by a model of the system (Figure 3-2). For the policy-based management to be possible a real managed system must be represented by some sort of management model, more or less abstract, and then policies are specified in the context of the model.

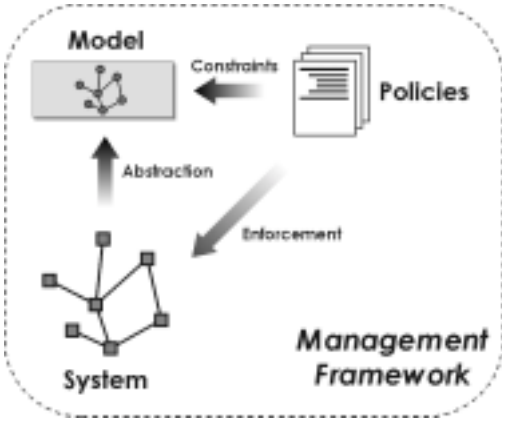


Figure 3-2 Policy-based management

According to [Steen 2000], there is a subtle distinction between a system model and its specification. A model is a description of how a real system can behave (potential behaviour, given by the design), while a specification describes how the system should behave (subset of the potential behaviour). A specification is defined in terms of policies, which are enforced by a management framework to ensure the consistency of the behaviour. The management framework transforms (refines) the abstract management policies, specified in terms of the abstract behaviour from the model, into the real (executable) management activities supported by the system.

3.1 Deficiencies of the existing solutions

According to [Damianou 2002b], a fundamental problem which remains unsolved in all work on policy-based management is the issue often referred to as policy validation. Given a policy, can it be instrumented in an existing hardware/software configuration, and is it possible to check and validate that this is the case before deploying the policy? Therefore, formal mapping mechanisms between a system and its model and between the model and its specification are required to guaranty validity (applicability) of specified policies in the context of the managed system.

Another problem with the existing solutions is related to the very nature of system modelling. The existing, role-based policy concepts use role-based models to capture the internal organization of managed systems. However, such models are inherently susceptible to changes, and therefore very hard to maintain over time. As a result, policies specified for such models are far too often exposed to changes. Moreover, the focus on details of the system internals makes both models and policies extremely complex.

Role-based management makes use of concepts developed in Role Theory. According to Role Theory, individuals in society occupy positions, and their role performance in these positions is determined by social norms, demands, and rules. A role is the set of prescriptions defining what the behaviour of a member holding a position should be, with position being a unit of social structure. Prescriptions are behaviours that indicate that other behaviours should be engaged in. More details on Role Theory may be found in [Richard 1999], [Marriott 1993], [Sandhu 1996], and [Biddle 1966].

Essentially, Role Theory is applied in Computing Science in the following way: Specifying a type (predicate characterizing a class of objects) in terms of roles allows the type specification to be reused, with different associations between roles and the objects that fill them. A role type describes the view one object holds of another object. An object, which conforms to a given role type, plays a role specified by the role type. At any given time, an object may act according to several different role types. A role model describes (possibly infinite) set of object collaborations using the role types. The role models are orthogonal and complementary to object models – they do not replace them.

The idea that an individual's behaviour could be constructed as role performance implied that role was one linkage between individual behaviour and social structure [Biddle 1966]. In Ponder and RM-ODP this idea was used to propose an organizational structure of distribute systems as hierarchies of roles and domains, which can be mapped to object-oriented models of those systems (structural specification) in order to constraint their behaviour:

- Individuals are objects fulfilling the roles.
- Social norms, rules and prescriptions are expressed as policies.
- Social groups are represented by domains.

However, there are two basic problems with applying Role Theory to object-oriented systems. First is complexity. An object-oriented model of a real system and its role-based generalization must be both created and constantly maintained up to date. Moreover, mappings among these two models must be also created and maintained up to date. In large systems, this involves lots of human work and strong support from management tools. Consequently, this would be a management at a high price.

Second problem are management capabilities of targets. Role Theory was originally created for humans as targets. All humans have more or less the same management capabilities (they are behaviourally compatible), which can adopt by learning. On the other hand, object as targets have very different management capabilities, which are defined by their interfaces (behaviour) and frozen at design time. Managing a group of people is easy, because they are at the same level of understanding when it comes to the execution of management tasks. Managing a group of heterogeneous objects (most are like that) is very hard, because there's only a limited amount of management interactions, if any, supported by each and every member of the group. Moreover, every object interface is likely to define its own namespace, so even interactions with the same semantics may not have the same names or parameter lists.

Role Theory, as it was implemented in distributed systems design, is far too complex to capture the dynamism in large systems. Management in large distributed systems is more process-oriented, and the role-based approach, which is essentially organization-oriented, is very hard to apply to behavioural modelling. The need to specify what goes where and who is related to whom at object/role level introduces lots of details, which results in complexity and rigidity of those management models. Moreover, the essential problems related to discovery and selection of objects, behavioural compatibility of objects/roles, and assignments of objects to roles (role-taking) practically remain unsolved.

3.1.1 Conclusion

In general, existing policy-based management solutions exhibit the following characteristics:

- ∉ *Low-level* in nature, without a strong support for abstraction. As a result, there are no mechanisms to face the problem of management complexity in distributed systems.
- ∉ *Procedural*, overloaded with details. The lack of generality affects the flexibility of the existing policy concepts. Too many details produce rigid specifications, susceptible to changes.
- ∉ *Embedded*, which produces dependencies and close ties between management and managed systems, and, ultimately, inflexibility.
- ∉ *Lack of a "big picture"* both in space (focus on isolated aspects and components of the system) and in time (tactical, short-term management decisions), as a result of incapability to face the complexity of the problem.
- ∉ *Discontinuous*, consisting of sequences of isolated management interventions. There is no a complete and continuous management process defined.

To address these issues, a high-level policy management system should ideally support centralized system configuration, which enables consistent policy enforcement. It should provide a system-wide view, and show the effect of a policy or parameter change on all of the affected components. System-wide management will also ensure that parameters are set consistently across multiple domains.

According to [Barros 1997], role-based and object-oriented models of the system internals can prescribe organizational infrastructure, thereby compromising conceptualization. The incorporation of an internal role-based and/or object-oriented model in a policy framework might well lead to a situation where the processing structure of the system prescribes the essential structure of the conceptual model. Therefore, it is seen as a major obstacle in a conceptual (high-level) approach to the policy-based management. High-level management needs to replace structural system modelling with a behavioural modelling approach that generally produces more abstract models, stable in time.

In addition to the behavioural modelling, one of the important steps towards a high-level policy based management would be encapsulation of the real system behind an explicit management interface, in order to break usually close ties between a system and its model. The encapsulation should be a design feature, either as a result of the initial design goals (for example SOA, see 1.4.4), or as a result of the wrapping existing systems into one of the new integration technologies (for example Web services). The encapsulation from the management point of view hides the details of the system internals and enables controlled influences on the system behaviour, where possible side effects are carefully considered by the design of the management interface.

The behaviour models also need abstraction mechanisms to cope with complexity. Besides the abstraction mechanisms, a possibility of having a number of interrelated viewpoints of the same system would additionally enrich the modelling concept. According to [Moffett 1993], different levels of abstraction need to be supported together with relationships between policy refinements. Abstraction can be a multi-step process, which ends when it reaches an acceptable level of complexity. Abstract policies (policies in terms of abstract operations) will be specified to manage an abstract system, created as a

representation of the management capabilities on the real system. Abstract policies will need to be interpreted into policies in terms of the real (implemented) management operation from the management interface, so they can be executed.

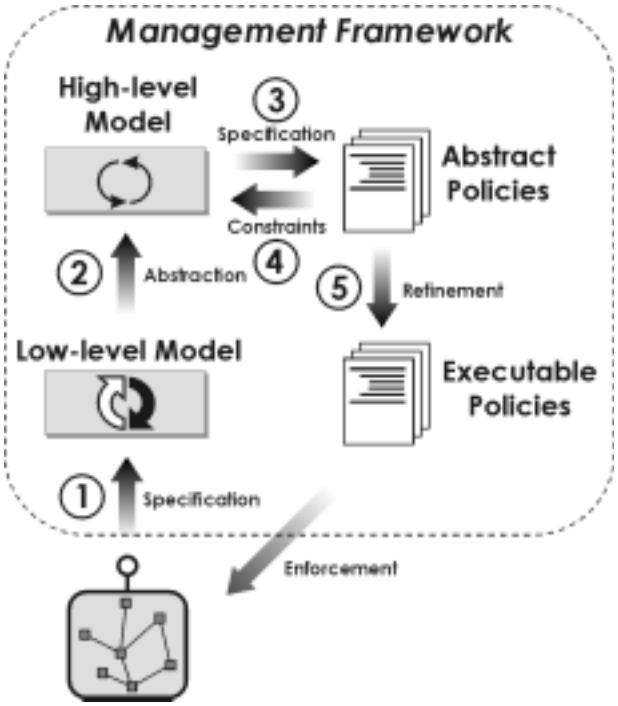


Figure 3-3 High-level policy-based management details

To conclude, a policy language should support information modelling for the purpose of both system and policy specification, more precisely (Figure 3-3):

1. Behavioural modelling of the system.
2. Abstraction mechanisms that enable simplification of the model, with a mechanism of explicit links between models at different levels of abstraction.
3. Specification of abstract policies.
4. Mechanism of links between the policies and the model, which enables interpretation of the abstract policies in the content of the model. This enables independence of abstract policies from the changes in the system, as a result of separation between specification of the system behaviour and specification of constraints on that behaviour. It also enables verification of policies within the context defined by the model.
5. Refinement of abstract policies into policies executable in the system.

In such a management model, the role of a policy management framework would be to make all the concepts from the policy language operational.

3.2 Research goal

The goal of this research was to propose a complete high-level solution for the policy-based management of distributed systems, and it was formulated as a response to deficiencies of the existing policy-based

management concepts. The inability of the existing object-oriented and role-based policy management approaches to address the problem of complexity in large-scale distributed systems is clearly showing the need for a radical paradigm shift. The existing policy-based management solutions are embedded within managed systems, and it poses a clear limitation to their capability to rise above the complexity to an enterprise-wide management level.

The aim of the policy based management is to apply integrated management systems so that system management, network management, and application management can cooperate [Lewis 1996]. This may be achieved by providing a common information model and an overall management process. A policy specification language in isolation cannot properly address the problem of policy-based management, because it can cover only one aspect of the management problem – the information modelling. In addition, we need to identify main roles in the management process and to define the flow of control and data among those roles. The presence of such a conceptual framework would impose a realistic set of constraints and requirements for the design of a policy specification language and it would define a context for its use.

With all that in mind, the goal of our policy framework, as an implementation of policy-based management, would be to integrate an information model, defined by a policy-specification language, into the management process – to make the management process policy-based. Such a management process needs to include:

- *Basic roles* in the management process, with a scope adjusted to the task of a high, system-level management of distributed systems.
- *Information model*, which will define the requirements for the design of an appropriate policy specification language. The role of the policy specification language is to integrate all the roles and their activities into a complete and consistent management process.
- *Flow of control and data* among roles, which defines, at a conceptual level, how different roles collaborate to achieve management goals and which mechanisms to control the flow of events are available.

The management framework should be defined at a conceptual level in order to be generic enough and to enable refinement that would suit the goals of particular projects. The management model should be able to provide management support for any distributed system, regardless of its size or complexity, and during its entire operational life. Such a high-level framework should be able to address the following problems related to the policy-based management:

- *Management complexity*, which requires powerful abstraction/refinement mechanisms. System management will be performed strictly from the outside, in a non-intrusive manner, and exclusively by means accessible from the outside.
- *Independence* of policy-based management. There must be a clear, high-level interface between the management framework and the system, and the communication between these two should be performed exclusively via the management interface.
- *System evolution*. Change is constant in organizations. Thus, we will need to step away from the level at which the changes in the distributed systems are frequent to an abstract level at which management view of the system is rather static.

- *Automation* of the management process, which requires mechanisms to support adaptability as the ability to flexibly determine the order in which tasks need to be performed. All management activities specified as policies represent only individual, but interrelated, steps in an overall automated and continual management process.
- *Minimal policy specification* with the maximal semantic information. Declarative policies state only what the system should do, while the system itself shall decide what would be the best way of doing it at any particular moment.
- *Formality* of policy specifications would, according to [Steen 1999], enable concise and precise policy specification, consistency checking on the policies, verification of actual system behaviour against a policy specification, and automatic derivation of implementation constructs for enforcing policies. Correctness of policy specification is checked against a model of the managed system (defines the management context), and it will guaranty executability of policies if the model of the system correctly represents the real managed system and its management capabilities.

3.3 Analysis of the requirements

Project goals, presented in the previous section, will become our design goals and they must be refined into concrete requirements in order to be correctly realized. Use-Cases are essentially a functional decomposition technique for describing behaviour at a high-level of abstraction, which offers a semiformal framework for structuring the requirements. We apply Use-Cases analysis to our problem to come up with an operational representation of our key requirements.

3.3.1 Management framework

Based on the analysis presented earlier in this chapter, we envisage that the goals of our research may be achieved by a management framework able to support:

- System independence, which requires a clear interface between the management framework and the managed system.
- Discovery of the system management capabilities, to enable adaptability of the management model, which is the key to automation.
- Abstraction of discovered management capabilities, to hide the complexity of the system management view and to provide scalability.
- Policy specifications for an abstract model of the system and their refinement into the real (implemented) management operations. This will ensure that a small number of simple policies will be able to precisely specify required system behaviour.
- Delegation of management policies to the system for enforcement, in order to provide independence and flexibility for the management framework.
- Seamless integration of proactive and reactive management, to enable compactness of the management process.
- Built-in control mechanisms, to enable automated and integral management process and to support adaptability.

In addition to the general requirements, we also need to consider some additional requirements specific for the problem of policy-based management. According to [Stevens 1999a], a policy system built upon the expression of rules must demonstrate at least three abilities: to define and update policy rules, to store and retrieve policy rules, and to interpret, implement and enforce policy rules.

Policy complexity can be reduced by adopting a global policy management approach [Follows 1999]. To fight the management complexity of large-scale distributed systems, it is not enough to specify high-level policies, but also the management model must be high-level too. Management operations, which are policy building blocks, must come from an abstract model of the system management capabilities. The modelling instruments provided by the framework itself should enable further abstractions of the system management interface as needed. The management framework should be able to support multiple views of the managed system, each at a different level of abstraction, together with a mapping mechanism between those levels.

Properties of encapsulation and abstraction are crucial for the design and construction of modular, evolutive systems in a heterogeneous environment [Stefani 1995]. We see a logical encapsulation at the system level, by means of a dedicated management interface, as the best way to face the problem of management complexity in large-scale distributed systems. This results in a clear separation of concerns, explicitly made by design, between a high-level (system-wide) management, and the internal system management.

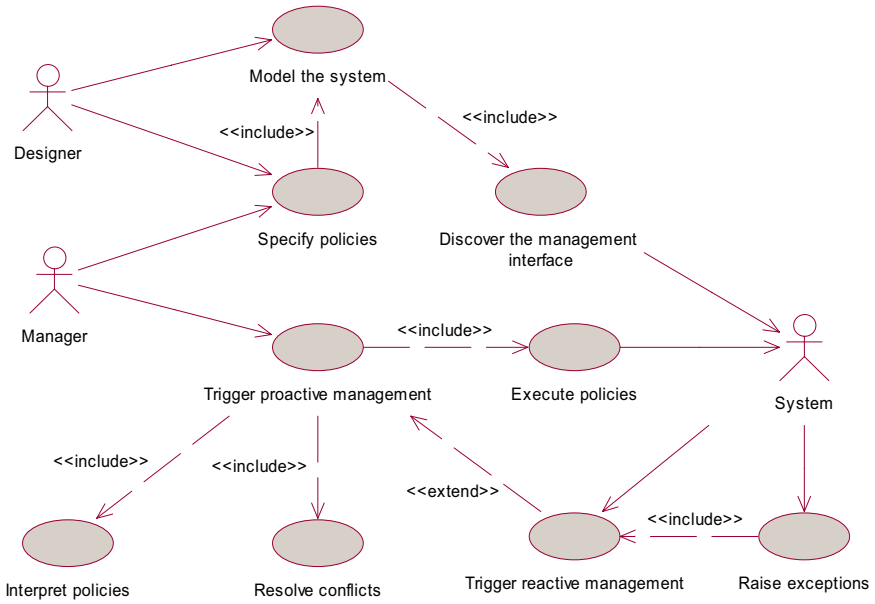


Figure 3-4 Operational view of the management framework design requirements

To summarize, our policy management framework should be able to support the following use-case scenarios (Figure 3-4):

- **System modelling**, which includes:

- *Discovery of behaviour*. Management specifications make sense only if they are feasible in (compatible with) the context of a particular managed system. To learn about the management capabilities of a system, a discovery must be performed by the framework.
- *Modelling (abstraction) of behaviour*. It should be possible to model the management capabilities of a system at various levels of details, with formal refinement relationships among them. A multi-level abstract hierarchy should provide the required scalability.

∉ **Policy (constraints) modelling**. It should be possible to specify and manage abstract and controlled management activities (policies) in the form of rules.

∉ **Policy-based management**, which includes:

- *Automation of the policy refinement*. A mechanism to refine abstract management specifications (in terms of abstract management operations) to executable management specifications (in terms of implemented management operations) should be available.
- *Automation of the management response*. It should be possible to specify triggering scenarios for reactive management (Event Handlers).
- *Built-in control mechanisms*. Exception handling and automated semantic and conflict analysis of the management specifications should be supported.

3.3.2 Policy specification language

The requirements for our language design are largely based on the application of the general principles for language design (see section 1.4.6). Many of the principles of programming language design that are presented by Hoare [Hoare 1989] and [Hoare 1981], Paige [Paige 2000], and Halpin [Halpin 1998] also appear to be applicable directly to policy specification languages.

The essential principles of programming language design may be summarized as the fact that language design is influenced by three mayor factors: by required expressive power, by required support for an envisaged specification methodology and by limitations imposed by the requirements of the language implementation. These three factors were our guiding light through the process of our policy specification language design.

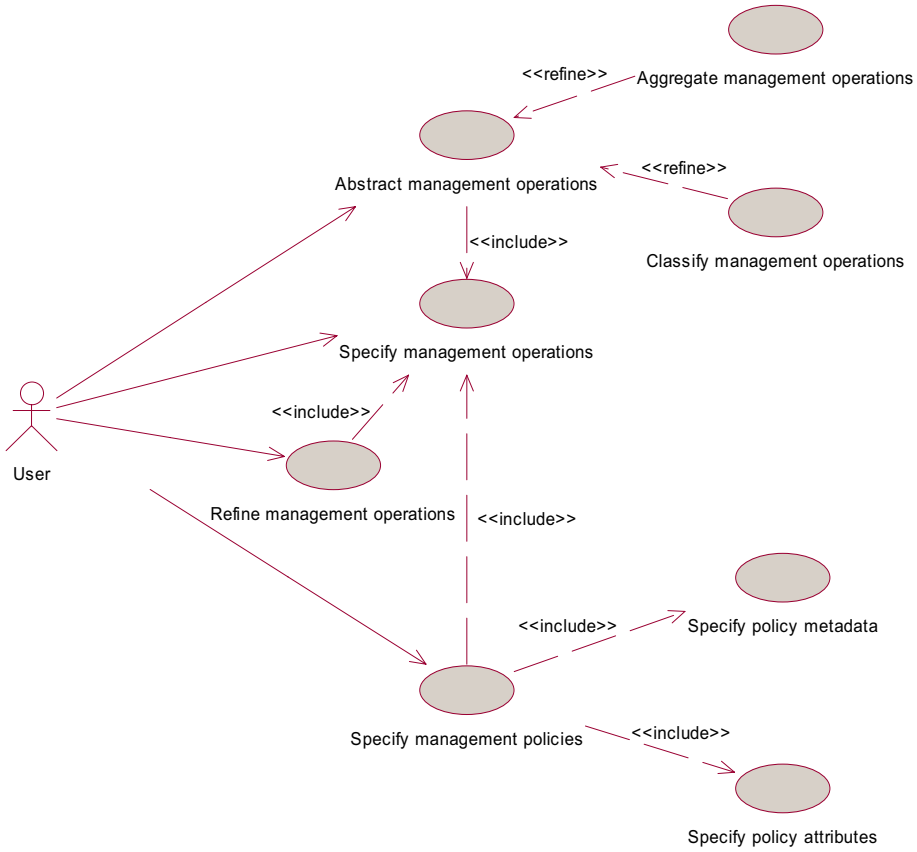


Figure 3-5 Operational view of the policy specification language design requirements

In the policy-based management of a system, there are four essential usage scenarios that need to be supported by the policy specification language (Figure 3-5):

1. *Modelling of the system management capabilities*. The actual management capabilities of the system are captured in a model as a set of supported management operations. Ideally, they should be dynamically discovered via interactions between the managed system and the framework.
2. *Abstraction of the model to a suitable level of complexity*. Complexity of the system management model needs to be addressed by abstraction mechanisms. If the complexity of the model is too high (large number of management operations), there will be a need to abstract the basic model in multiple steps. This will result in a hierarchy of different viewpoints, each at a different level of abstraction, together with mappings between them.
3. *Specification of the management policies for the abstract model*. The highest-level model of the managed system should be sufficiently simple to enable an easy policy specification.
4. *Refinement of the abstract policies* by the refinement of their abstract management operations. A refinement mechanism of the policy framework must enable transformations of the abstract policies into an executable form.

Therefore, a concrete set of requirements for our new policy specification language, which are based on the literature survey, analysis of the framework requirements and the comparative analysis of the existing policy specification languages, may be identified as follows:

- *Simple*. Simplicity of the language is reflected in its syntax and semantic. Simplicity of the syntax opens an opportunity to create a visual representation for the language, which increases the expressive power of the language, and makes the work on policy specifications much easier.
- *Modular*. Policies shall be assembled from predefined building blocks, defined in the system management model. This concept will offer a possibility for extensive semantic analysis, simpler language syntax, and reuse of basic specification constructs, which improves consistency of overall management activities in the system.
- *Self-explained*. Rich metadata and policy attributes enable semantic analysis of policy specifications and their classification.
- *Extensible*. Extensibility is an ability to add new or modify existing features without impacting existing specifications. It is a quality universally expected from any language. We will focus only on the semantic extensibility, because extensibility of the syntax is very hard to achieve. In [Hoare 1989], Hoare recommends the avoidance of any form of syntactic extension to a language.
- *Executable*. A formal language, with dynamic mappings of its building blocks to implementations, will enable automation of the management process.
- *Unambiguous*. The formality of policy specifications will be guarantied by the associations between system model (specification of the behaviour) and policies (specifications of the constraints on the behaviour).
- *Declarative*, which means that it will be able to produce high-level, implementation-independent policy specifications.

In our policy-based management process, policies will have dual function:

- *To express constraints* on the system behaviour. For this to be possible, the policy specification language must have sufficient semantic power, which has to be balanced with the equally important requirement for simplicity.
- *To enable automation* of the management process. For this to be possible, specifications in the policy language must be executable, which practically means that those specifications must be formal and must make sense in the context of the management capabilities of the system. Moreover, the language must be able to support automation of the management process by means of internal triggering and control mechanisms.

3.4 Policy-based management concept

Our policy-based management model and policy specification language were designed based on the previous analysis. In the approach we envisage, the problem of complexity will be addressed from two aspects:

- *Management model*: We will have an exclusive focus on the system management behaviour, assuming that the system internals are hidden.
- *Policy model*: We will implement an abstraction/refinement mechanism and try to achieve modular design of the policy specification language.

3.4.1 Management model

Systems need to be designed to be managed, not only to be used, and the extent of the management control is defined by the system design. The work on Open Architectures (see section 1.4.2) and Service-Oriented Architectures (see section 1.4.4), as well as the work on environments that support the development, deployment and management of distributed applications based on services (see [Bauer 1997], [Bakker 2000], [Bauer 1994b], [Bauer 1994a], [Bauer 1993], [Bauer 1994] for more details) give some principles of the system design which separates system manageability from its functionality. In such systems, constraints on system behaviour may be created:

- *By a designer*, at design time. These are system invariants, which express static constraints frozen by the design.
- *By a manger*, at run-time. Decisions that cannot or should not be taken at design time will be left to be made at run-time, by the management. The role of development process is to identify and to enable such management decisions by specifying a management interface, which defines management capabilities of the system.

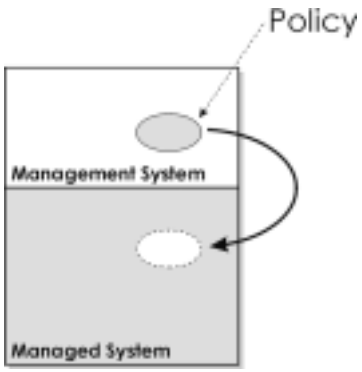


Figure 3-6 Management from the outside: Management specification appears to end up inside the system

Design time policies implement the actual system behaviour, whereas management policies control (guide) that behaviour. Management policies are external to the system, and they are designed to be used after the system is implemented. The role of a management framework is to automatically maintain constraints specified by the management (run-time) policies. In our management framework high-level management decisions are made centrally and sent as policies to the managed system for execution. Policies can be seen as small client programs in a high-level programming language, which are inserted into the system from the outside (Figure 3-6). The managed system has the full power in making any further decisions about how to accomplish policy enforcement. Every policy specifies a management task delegated to the managed system to be executed by means of its internal functionality. Therefore, while the control decision-making is centralised, the policy execution is delegated to the managed system to be carried out in a most appropriate way.

3.4.1.1 Abstraction of the managed system

One of the main inspirations of scientific research is the pursuit of simplicity [Hoare 2000], and the most important decisions taken in the course of our research was the choice of the abstraction level at which the problem of policy-based distributed systems management will be considered. Such a level must capture all the essential aspects of a high, system-level distributed system management, and yet be simple enough to produce a simple solution (complex ones already exist). We base our simplification strategy on the idea of encapsulation of the managed system from the management point of view, with the system internals hidden behind a management interface (Figure 3-7). One of the consequences of having a higher level view of the system as a whole is to be able to talk about system management in a global sense. Almost inevitably, this leads to the use of policy and high level goals to describe the desirable behaviour of an IT system which will support the business process of an enterprise [Goh 1998]. Encapsulation from the management viewpoint hides the internal management complexity of the system, and empowers the separation of concerns between the system design and the system management. Independence enables loosely coupling, minimizes the number of interaction points with the system, and provides scalability.

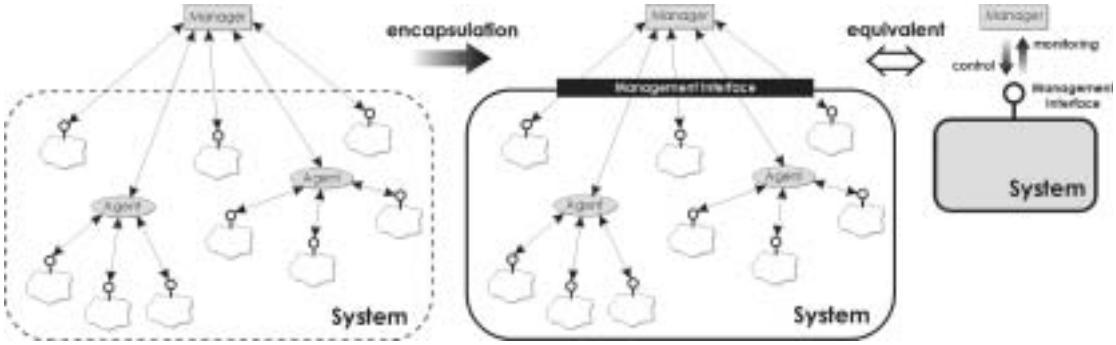


Figure 3-7 Separation between high- and low-level management made explicit

The idea of a system architecture designed to enable management through a centralized set of high-level management services is not new. CORBA, with its Common Facilities Architecture that offers System Management Common Facilities, is one example of such a design. Common Facilities provide higher level interoperable interfaces for Application Objects (correspond to the traditional notion of applications), which can be specialized for specific application domains. It was envisioned that hardware vendors will deliver ORBs with a standard set of System Management interfaces (facilities), and a capability to control, configure, and monitor system resources. Another example is work presented in [Bauer 1997], [Bauer 1994b], and [Bauer 1994]. This framework is based on a set of common management services which support management activities. The core of the proposed integrated management architecture is a set of management services that are logically organized in three subsystems: Repository services, Configuration services, and Monitoring and control services.

3.4.1.2 Management interface

System manageability is abstracted in its management interface just like its functionality is abstracted in its functional interface. According to [Maeda 1997], the management interface defines an abstract language for the system management. Vocabulary of such an interface language consists of management

operations implemented by the system, so it is made for the system to understand it. Actually, the language defines a vocabulary of a policy specification language. Management policies can then be written as small, declarative management scripts in that abstract interface language. Abstraction level of the vocabulary determines the abstraction level of specified policies, whereas the degree of the management control that can be achieved over a system (manageability) depends on the richness of the management interface.

Essentially, management interactions represent a sequence of Get/Set State operations on the system state. The management framework sees a managed system as a state machine, which may be controlled by means of the behaviour it exposes. Management goals may be to keep the system out of the forbidden states and/or to keep the system in the preferred states. Freedom (boundaries) of the behaviour is given by the system design, and it may only be restricted (narrowed) using management policies.

The system management interface defines potential behaviour of the system, given by the design. The real system state domain is unknown, but its abstract representation can be deducted using the exposed management behaviour (management interface). Such an abstract state domain is defined in terms of parameter values that are possible to receive from the system using Get State type of interactions (Figure 3-8). An example of such an analysis is given in Appendix C.

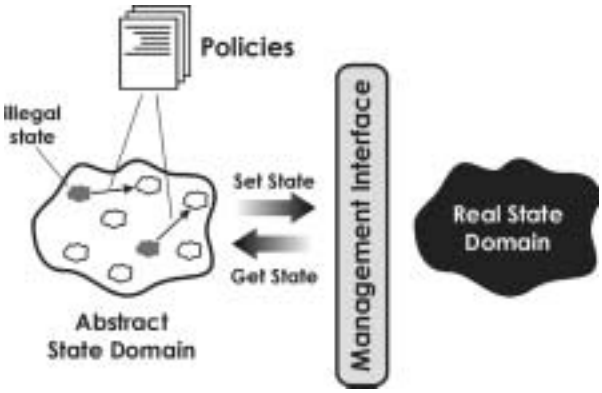


Figure 3-8 Indirect control of the system state

Management policies put constraints on (narrow) the potential behaviour of the system by creating interdependencies (relations) among elements of that behaviour. The purpose of such constraints is to enforce state transitions of type *illegal state* ⇓ *acceptable/preferred state* within the system. As a result, a portion of the system state space will be marked illegal (Figure 3-8), and on its occurrence corrective actions will be taken by means of Set State type of interactions.

According to Letier (see [Letier 2002]), two alternative styles of semantics are generally considered for an operational specification language:

- *Generative semantics*: Every behavioural change is forbidden, except the ones explicitly required by the specification. For example, this approach has been used in Ponder.
- *Pruning semantics*: Every behavioural change is allowed, except the ones explicitly forbidden by the specification. The specifications prune the set of admissible behaviour of the system.

We believe that the pruning semantics is more intuitive, because it borrows from real life (civil law, traffic law, etc.). High-level management policies constrain only a small fraction of the possible system behaviour. Therefore the pruning semantics seams to be more appropriate, since it relieves the specifier from the obligation to explicitly state everything that does not change in the system behaviour. In addition, we will apply an implicit frame rule saying that aspects of the system state for which no effect of policy actions are specified do not change their value in the state following the execution of the policy.

3.4.1.3 Management process

Modern component-based and service-oriented architectures are organized around processes, not hierarchically: Ad-hoc configurations of components and services are dynamically created to support processes and tasks, and resource allocation is performed much more on per-process then on per-client basis. Interactions among components are no longer hierarchical, but peer-to-peer in nature. Hierarchical interactions are often found in more stable, long-term relationships between partners, while peer-to-peer interactions reflect relationships that are often established dynamically on a per-instance basis. As a result, the need to manage the behaviour of a system, not its structure, becomes eminent.

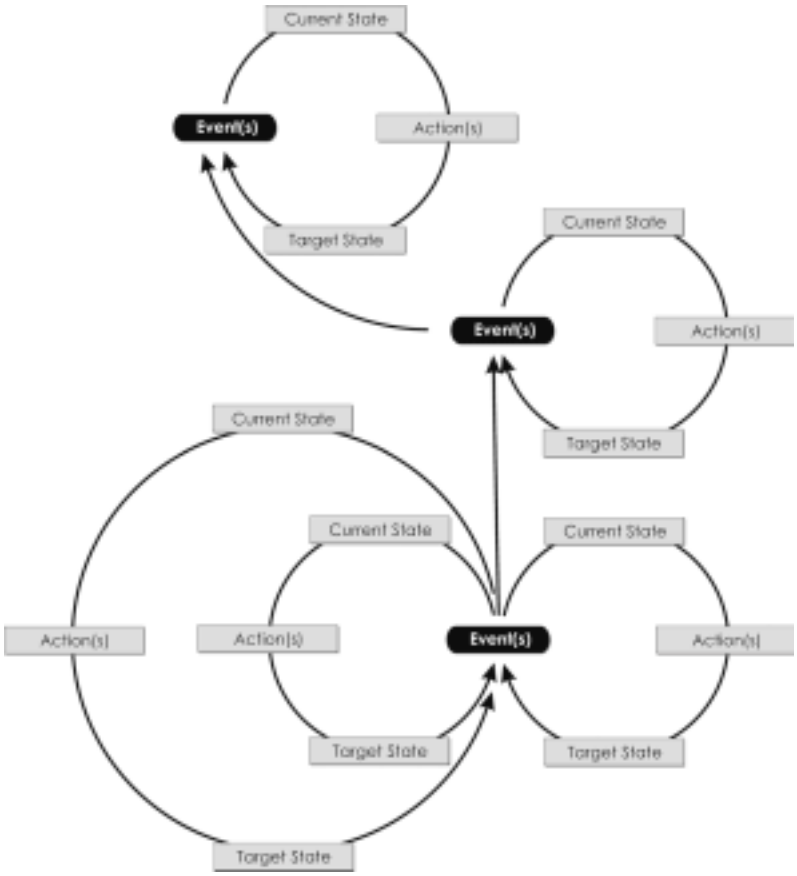


Figure 3-9 Management workflow

In our management framework, we want to apply process- (workflow-) oriented approach to system management (see 4.2 for details). In contrast to most of the existing resource-oriented management

concepts that model the system as a known hierarchy of objects and roles, our managed system is envisaged to be a collection of processes. The management process itself represents a flow of management activities, guided by polices (Figure 3-9). At the system level, the process-oriented paradigm appears to handle the problem of complexity much better. The process-oriented management approach promotes focusing on management goals, whereas the object- (role-) oriented approaches are more concerned with the structure and the state of the system.

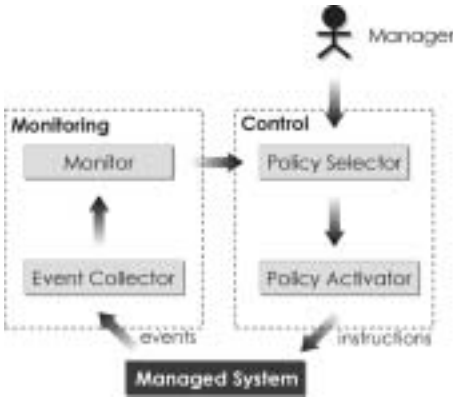


Figure 3-10 Basic model of a policy-based management process

Two essential components of our envisaged management process are (Figure 3-10):

- *Monitoring*, which collects information from the managed system. Two basic components implement monitoring: Event Collector, which interfaces with a notification service of the managed system, and Monitor, which operates at a higher level of abstraction and detects signals for action.
- *Control*, which is responsible for implementing management policies. Two basic components implement control: Policy Selector, which decides about the appropriate policy, and Policy Activator, which is responsible for enforcement of the selected policy and it works with the management facilities of the managed system.

The policy life-cycle is based on the fact that policies are very dynamic, both is terms of goals they enforce and methods they may use to achieve their goals. Hence, the enforcement of policies is not something one can define or derive once and code in some programming language. Policy enforcement is dynamic in that it responds to changes in the environment and is interdependent with other policies acting on the same resources [Wies 1997]. Approaches in which policies are objects that exist in time are very complex to implement, because the state of all policies must be updated constantly. In our management framework we implement a "use-and-waste" policy life cycle (see 4.2). When triggered, policies will be interpreted to reflect the latest changes in the management model and sent to the system, which will be able to independently execute them. Therefore, we see policy as a piece of mobile abstract code that is delegated to the managed system for execution.

3.4.2 Policy model

We see management policies as specifications of generalized management experience, which guide management operations in a policy-based management system towards the goal. Policies do not specify

management decisions themselves, but rather provide knowledge that may be used to make decisions. Knowledge is a set of syntactic and semantic conventions that makes it possible to describe things [Bench-Capon 1990]. It defines a mental model of a problem, and it is being specified in order to be shared. The purpose of a policy specification is to state management knowledge so that others (humans and machines) can understand it and (re)use it to control the behaviour of a system. In our management framework, policies specify declarative knowledge (know-how, expertise) about how to use the behaviour of a system to manage its state. Management knowledge is captured in an explicit form, so it can be changed.

3.4.2.1 Policy form

The basic policy information model was founded on the Management Control Model (see 1.4.1), and consists of:

- **IF** part (preconditions), which includes measurement of performance and comparison to the standards.
- **THEN** part (actions), which performs corrective action.
- **WITH** part (post conditions), which performs post action control (measure results) against the objectives.

Management policies express constraints on behaviour by establishing dependencies (relationships) among elements of the behaviour (management operations from the system management interface). The IF-THEN-WHILE structure extends the basic condition-action semantic with post conditions. Such an extension enables support for a controlled (monitored) flow of activities in our management framework (Figure 3-11). The required behaviour is specified by making relations between an initial state (IF part), a goal state (WITH part), and actions needed for the actual state transitions to take place between these states (THEN part): For each unwilling state that may occur in the system (IF part), we need to specify a set of activities that will perform a transition (THEN part) to a target state (WITH part). WITH part of the policy performs the management audit – it judges about the quality of the management activities performed in the THEN part.

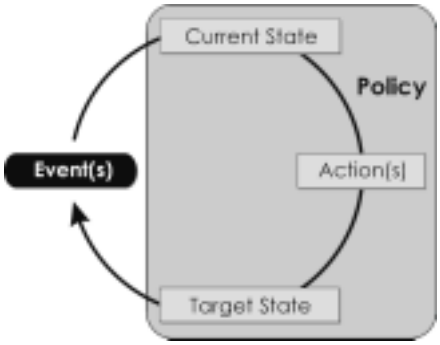


Figure 3-11 Policy workflow

The semantics of our policy specification is a conditional and controlled state transition, and it does not include events. Stating the rule in an event-free fashion is an important step in ensuring consistent and

complete enforcement across all events [Ross 2001]. According to [Barros 1997], from the point of view of the environment, the actual management process triggered by some event-based request is inconsequential for the formulation of the request. A technique should allow the formulation of event-based requests to be independent of their actual processing.

In the process of policy specification, two kinds of parameters need to be extracted from the requirements specification:

- *Monitored parameters*, for observing the behaviour.
- *Control parameters*, for changing the behaviour.

The process of policy specification usually starts by identifying operations relevant to goals and defining their pre- and post conditions. Goals refer to specific state transitions; for each such transition an operation causing it is identified; its domain pre- and post condition capture the state transition [van Lamsweerde 2001].

Distributed systems management is strictly a goal-oriented activity. Therefore, it is important to identify management goals and to state them explicitly. Post conditions (WITH part) in a policy express the policy goal as TRUE/FALSE propositions, as suggested in [Bearden 2001], and IF-THEN part specifies activities to achieve it. Often the policy goal is simply to eliminate the state expressed as policy preconditions (Figure 3-12). In such cases, post conditions would simply state the negation of the preconditions, so they can be omitted from policy specifications as obvious.

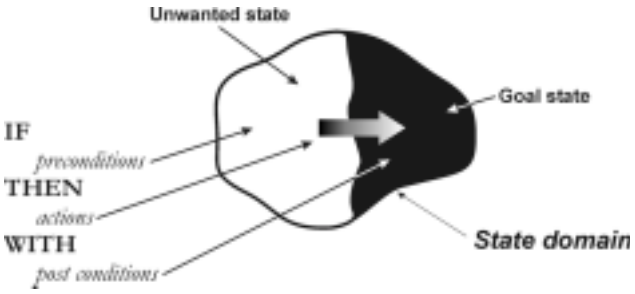


Figure 3-12 "Black and White" approach to the state control

Since the state domain is divided in two areas:

state domain ◊ preconditions ≅ post conditions ♥ post conditions ◊ !preconditions

the semantics of such policies would be:

```
IF
        conditions          // refers to the unwanted state
THEN
        actions             // provoke the state transition
WITH
        !conditions         // refers to the goal state
```

However, states of big systems are usually complex and hard to control, so we need to tolerate certain offsets from the ideal. There are cases in which we need to define a "grey" zone between optimal and unwanted states. Policies would be used to keep the system state within that grey zone, as close as possible to the optimal state. We don't want to intervene until the state of the system is within the grey zone, and when we do, our goal is to put the system in the optimal state (Figure 3-13).

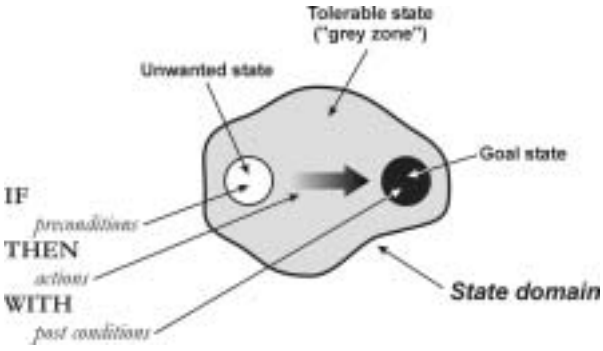


Figure 3-13 Flexible approach to the state control

In this case, the state domain is divided in three areas:

state domain ◊ preconditions ≅ post conditions ≅ "grey zone"

Post conditions are no longer obvious, and the semantics of such policies would be:

```
IF
        preconditions       // refers to the unwanted state
THEN
        actions             // provoke the state transition
WITH
        post conditions     // refers to the goal state
```

As a conclusion of the analysis of policy semantics, we suggest that a policy specification will need to specify the following facts about the related management activity:

∉ **Operational specification**, more precisely:
- *Preconditions*, which are predicates that a specification requires to be true for actions to occur (refer to a current state).
- *Actions*, which are specified in terms of an alphabet of operations that manipulate current state of the system in order to affect its behaviour and/or configuration.
- *Post conditions*, which are predicates that a specification requires being true immediately after the occurrence of actions (policy goal).

∉ **Explicit semantic specification**, which includes metadata to enrich the semantics of the operational specification, and consists of the following elements:
- *Natural language representation* in a free textual form.
- *Class*, which describes purpose of a management policy (fault management, configuration management, performance management, security, accounting, etc.).

- *Goal*, which describes the objective of management actions specified by policy (increasing bandwidth, load balancing, optimizing performance, etc.).
- *Course of action*, which describes the course of management actions specified by policy, and can be: change of object assigned to the role (software or hardware plug-in), change of object state (attribute values), change of object location, etc.
- *Priority*, which may determine precedence in case of semantic conflicts detected in a set of policies.

∉ **Policy attributes**, which may include a number of arbitrary information about a policy (subject, target, modality, validity, etc.).

3.4.2.2 Policy content

The basic idea of our policy specification language design is to express management activities in terms of operations from the system management interface, defined in the management model of the system. We envisage a policy to be constructed from:

∉ *Skeleton* of a policy specification, which is used to define the structure and the placeholders for the policy building blocks.

∉ *Building blocks*, which are operations from the system management interface.

From a cognitive perspective, directness in computing means a small distance between a goal and the actions required to achieve the goal. In our policy specification language, the language vocabulary establishes direct mappings between the policy namespace (management activities) and the managed system namespace (management capabilities). The policy namespace directly references the managed system namespace using vocabulary of management operations. System management operations are building blocks (Figure 3-14), shared both by management model of the system (behaviour) and policies (constraints on behaviour).

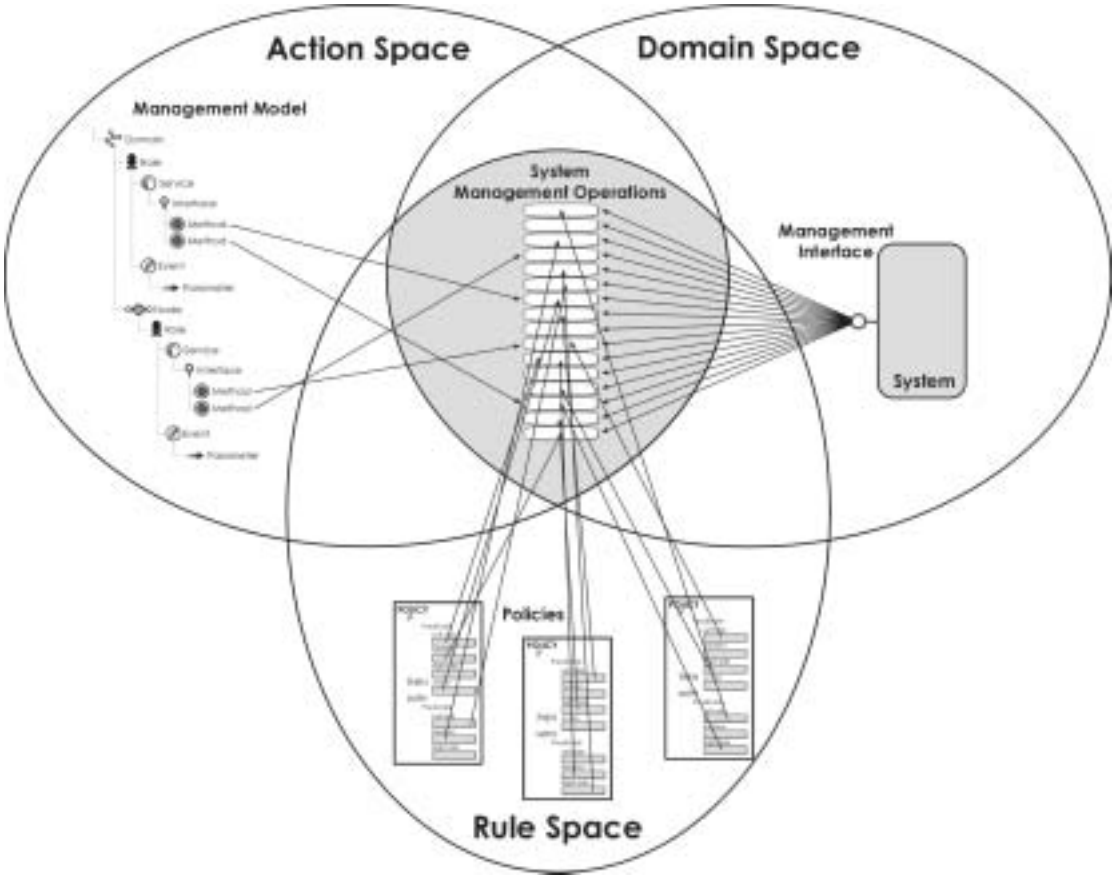


Figure 3-14 Logical components of the management model

Mappings between the policy and the system namespaces must exist in order to ensure the executability of policy specifications. Semantics that cannot be refined into a series of interactions with the managed system cannot be executed automatically. According to [Stevens 1999], it requires policies that contain no information unknown to the device that will execute it.

For our policy specification language we defined a universal syntax together with the rules on how to create custom, system-specific vocabularies (see Chapter 5). A vocabulary was imagined to be dynamic, and to evolve together with the changes in the corresponding managed system. The basic vocabulary can be abstracted in one or more steps in order to create more powerful building blocks for policies (abstract vocabularies). Having a language with adaptable vocabulary is very important, because of the requirements for flexibility imposed by the management framework. Our management framework will first discover what a managed system is capable of, and then use it to assemble an appropriate system-specific language vocabulary. The richness of the language vocabulary depends of the system management capabilities and it will define the expressive power of the policy specification language for every particular managed system. The richer system management interface is, the more sophisticated policy specification language will be.

In our language, policies do not incorporate specifications of the behaviour – they specify only relationships among them. Policies only refer to elements of the system behaviour, but the specification of the system behaviour itself is external to policy specifications (Figure 3-14). This is the key of the

proposed concept, which differs from any other policy specification language. In contrast to the existing policy specification concepts, which focus only on reuse of the form (templates), we believe that reuse of the content is more important, because the content tends to be more complex and in constant change. Behavioural specifications that are incorporated in policy specifications encourage redundancy of information, which is expensive in large distributed systems (maintenance) and leads to inflexibility and inconsistency.

3.5 Conclusion

In this chapter we have performed a detailed analysis of the existing policy-based management concepts, identified their deficiencies and derived requirements for a new, better solution. Basic ideas of the new management framework and policy language were presented, together with the most important design decisions.

In the following chapters we will present in details our policy management framework and the policy specification language.

4 SERVICE-BASED POLICY FRAMEWORK (SPF)

We have, so far, presented and analyzed the existing solutions for policy-based management, identified their weaknesses, derived requirements for a new, better solution, and, based on that, made the essential design decisions for a management framework and a policy specification language.

This chapter will present the framework for policy-based management, built on the analysis and design presented in the previous chapter. The purpose of the framework is to define a context for the practical application of our new policy specification language (SPL), which will be presented in full detail in the subsequent chapter. The framework will be presented at a conceptual level, from two viewpoints: structural (framework components) and functional (framework processes). In order not to be too restrictive, details of the framework design are left out of the picture, to be defined by the SPF implementation.

In general, a framework represents definition and organization of concepts to satisfy a set of requirements for a particular domain or an important aspect thereof. It can be seen as a reusable, "semi-complete" application. The main purpose of having frameworks is reuse of architectural design.

In order to quickly position our framework with the existing solutions for policy-based management, we have identified a conceptual, layered stack of basic concepts in the field. The purpose of the stack is to relate management efforts at different levels of abstraction and managed resources.

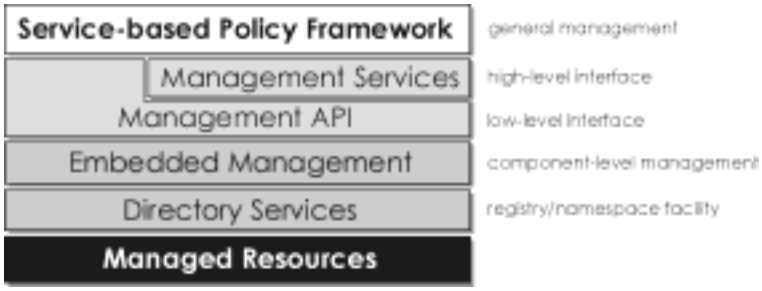


Figure 4-1 Policy-based management stack

In our simplified picture, the policy-based management stack (Figure 4-1) consists of the following layers:

- *Service-based Policy Framework* (SPF). We place SPF on top of the policy-based management stack, simply because it was designed to be a high-level, generic architecture that is concerned with the managed system as a whole.
- SPF will not interface system components directly, but only via a well defined management interface. *Management Interface* defines management capabilities of the system, and explicitly separates management framework from the system internals. It may come at different levels of abstraction: as a simple management API (low-level), or as a set of (high-level) management services that abstracts the management API.
- *Embedded management* offers specific management support at the component level. For SPF, it behaves as a management agent. The existing policy-based management solutions are, in most cases, at a low-level of abstraction and mainly suitable for an embedded, component-level management. The modular design of distributed systems includes various components and layers of different

granularity and complexity, with built-in (module-level) management facilities (Figure 4-3). Those module-level management facilities are parameterised (state), to enable reconfiguration, and generalized (behaviour), to enable control from the outside.

- *Directory services* control/facilitate access to the system resources. They define an interface to repositories of information about the system (for more details, see 2.2.2).

In the rest of this chapter, we will briefly present our framework from two major aspects: structural, in terms of the framework components, and functional, in terms of the framework processes.

4.1 Framework components

The proposed policy framework defines a generic architecture for policy-based management, which encompasses definitions of four basic logical components: Domain space, Action space, Rule space, and Policy driver (Figure 4-2). The architecture derives directly from the model presented in Figure 3-14.

Figure 4-2 SPF management model

The Domain space is defined by real attributes of a managed system (real management operations and resources). In the SPF, logical functions and dependencies from a management model are mapped to the real management operations and resources. Such mappings are realized by specifying the management model of a system (Action space) in terms of the real operations from its management interface (Domain space). The Action space is an abstraction of the Domain space – it is a management model of a system. In SPF, it consists of a hierarchy of Dictionaries (see 4.1.2). The Rule space (system specification) is defined in terms of policies, stored in a policy repository. The goal of high-level policy-based management is to define a minimal specification of desired system behaviour (Rule space) in terms of operations from the Action space (high-level, abstract policies), and then to refine it to specifications in terms of real operations from the Domain space (low-level, executable policies). The Policy driver (in SPF, it is called workflow engine) parses policies and transforms logical management interactions from the Action space into the real function calls from the Domain space.

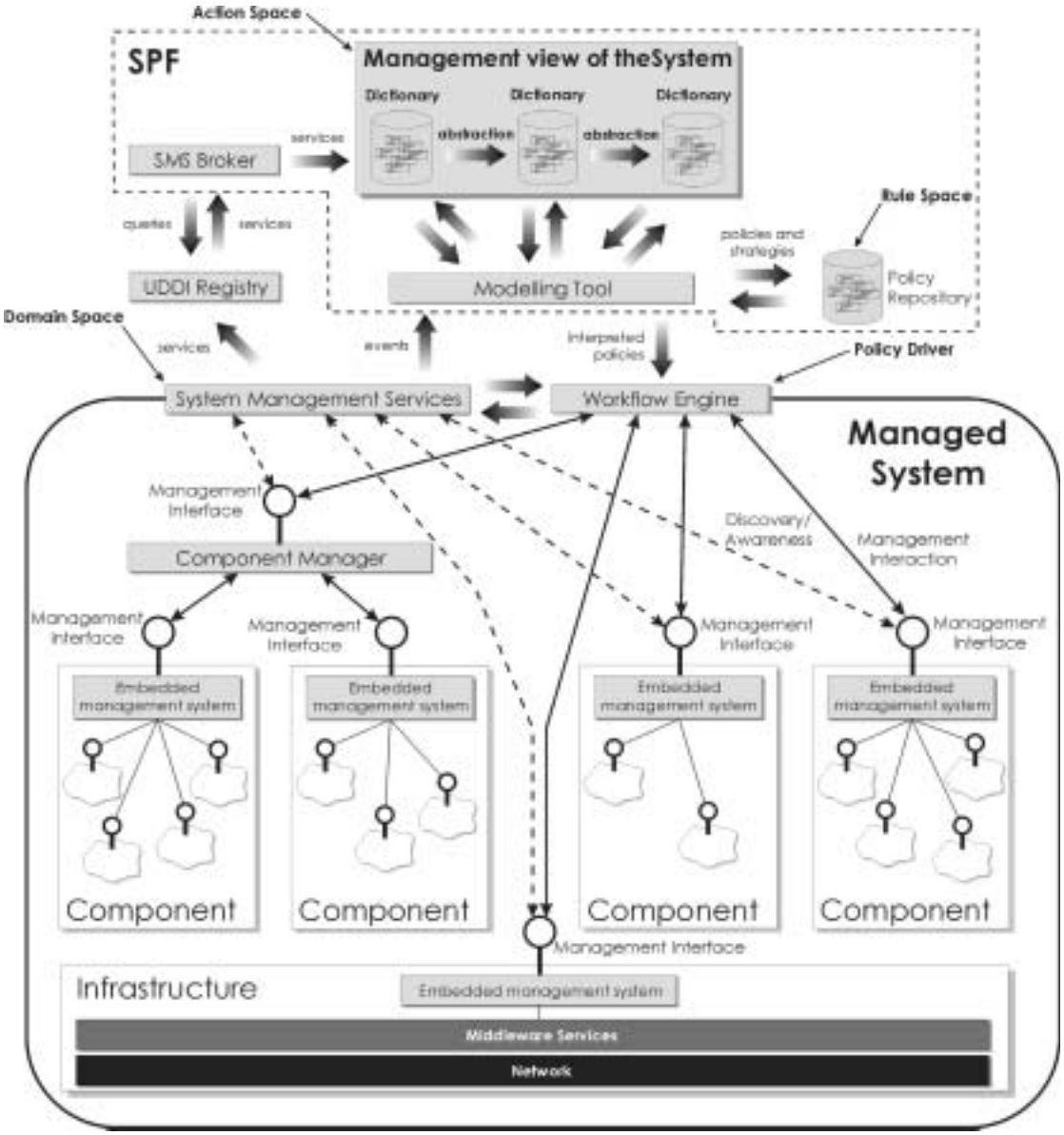


Figure 4-3 The SPF management model (details)

SPF was envisaged as a plug-in management concept that puts minimal requirements on managed systems design, which can be easily met even by systems already in operation. In general, a managed system should provide the following points of interaction:

- *Management Interface*, for usage of the system management capabilities. Every system has a management interface, only it is not always explicitly specified. The richness of the management interface will determine the extent and the quality of management provided by the framework. The system management interface enables and controls access to system resources, and it may be at different levels of abstraction. The system supporting infrastructure creates dynamic mappings between the management interface and software components that implement the interface (Figure 4-3). Providers of management services are hidden from the management framework, so that changes in the system at object and component level don't affect the management operations.
- *Discovery service* (registry), for recognition of the system management capabilities. Auto-discovery is one of the key technologies that enable management systems to be quickly customized to the

environments that they are intended to manage [Ramanathan 1999]. It can be implemented as UDDI registry, Directory service, or some implementation of the ODP Trader Function (for more details, see 4.1.3). Semantics of management operations needs to be published, so it can be discovered by SPF. If such functionality is not available, the discovery of the system management capabilities must be performed manually, by a human manager/designer.

- *Workflow engine*, for policy parsing and execution. Today, workflow engines come as an integral part of most application servers. Workflow engines are normally embedded in managed systems, and they execute management policies received from SPF.
- *Event notification service*, for delivery of the system events. If the system cannot provide a notification service, the full-scale proactive policy-based management will be still available.

In the rest of this section, we will have a more detailed look at individual components of the proposed framework.

4.1.1 System management interface

Every distributed system has some sort of management interface, but usually it is at a very low-level, and therefore very complex to use. Traditionally, customization of network and distributed applications has been through dialogues which provide direct access to program variables. However, this is of limited use when the abstractions are themselves complex and have little or no meaning to the user.

Perhaps the most significant trend in software engineering has been the drive toward more expressive forms of abstraction [Booch 1992]. Each of the following modelling approaches has moved the abstract level of the interaction model further up. With the object-oriented design, explicit distinction between interface specification (behaviour) and its implementation decouples clients from the evolution of a provider. In the component-based design, standardized interface specification enables decoupling between clients and the implementation technology of a provider. Finally, with the service-based design, interface is detached from its implementation, which decouples the interactions from the location and identity of the provider. Services have been created to hide the complexity and to soften the rigidity of the object-oriented model.

According to [Farrell 2002], Web services provide for a very dynamic, flexible, and reconfigurable execution environment. It is important that the management approach also support these attributes, that the general management architecture be correspondingly adapted, and that the application roles fit the Web services model. Web services are described and are accessible using interoperable standard protocols and transports. Therefore, applications based on Web services will be able to use services that execute in many diverse environments - systems, languages, platforms, and enterprises. Web services define a standard mechanism for publishing, finding, and interacting with other Web services at run-time, rather than having them statically bound or internalized during development. As a result, Web services, as well as management services, are more portable to different execution environments. All of these facts drive the need to develop a management approach that stays within the Web services paradigm [Farrell 2002].

Essentially, the system management interface may be at two levels of abstraction (see Figure 4-1). System Management API enables lower-level, RPC-style interactions, which are sensitive to perform since a tight coupling between involved parties is necessary. Because of the distributed nature of large systems, this usually results in complexity and susceptibility of management solutions. System Management Services (SMS) come as an abstraction of the System Management API, which enables simplified, message-based interactions with the system.

In the context of the distributed system management, the importance of the Web services technology is in its goal - to address the problem of complexity. Web services focus on WHAT semantics (purpose), whereas the WHO (service providers), WHERE (location of resources) and HOW (implementation details) is hidden from the user. SMS abstract System Management API by delivering an infrastructure for provisioning of management functions. The presence of SMS enables shifting of the SPF management operations to a higher level of automation and flexibility. Discovery and selection of the system management capabilities would be much easier to perform, because management services are described, and those descriptions are made public. Interactions with the system would be at a higher level of abstraction, which offers a simplification of management operations as well as a greater independence between SPF and the managed system. Management services are described, but their implementation and provisioning mechanism are hidden. As a result, it would be possible to focus on global, top-level system management, without thinking about technical details. Finally, decoupling between provisioning and the usage of management operations would enable independent evolution of both managed systems and the management framework. High-level management services are likely to encapsulate entire management workflows, which may be created, discovered and updated dynamically within managed systems.

SPF was designed to benefit from the flexibility of Service-Oriented Architectures (see 1.4.4), which are capable of offering management services. At the same time, the framework is compatible with the classic RPC-oriented architectures. The exact nature of management interactions specified in Dictionaries (see 4.1.2) is not of any importance for the policy specification itself. However, it affects the SPF management process in two major aspects. First, it determines the extent of discovery of the system management capabilities for the purpose of the system modelling. This affects the level of automation that can be achieved in the management process implemented by the SPF. If SPF cannot be automatically aware of the management capabilities of a system through a process of discovery, a great deal of human labour will be required to model and to maintain system management views.

Second, it influences the nature of binding to and triggering of management operations specified in policies, which requires the actual interaction with a real system. Policies will be sent to the system in a form that it understands – the form that can be transformed, by a workflow engine, into a series of interactions with the system management resources. Therefore, the actual execution of policies in more an internal problem of managed systems than it is a concern of SPF. If the management operations are implemented as management services, there will be a possibility for dynamic binding to the most appropriate service providers within the managed system, which will greatly increase the flexibility of the management process.

4.1.2 Dictionary

Prerequisite for building a policy-driven infrastructure is to gather a realistic assessment of the actual management capabilities of managed systems. Policies are rules we use to define an acceptable/favourable behaviour within the possible behaviour given by the system design. Since policies abstract what behaviours are available in a managed environment, it is obvious that we need to "see" our managed system and its management capabilities before we can start specifying management activities as policies. Because of the complexity of large distributed systems, it has to be at a high level of abstraction. Because of constant changes in distributed systems, it has to be decoupled from real systems and their implementation technology.

The main role of a Dictionary in SPF is to organize and simplify, by means of abstraction, potentially complex management operations from a system management interface. Abstractions of the system image onto metaphorical controls create an appropriate and useful mental model of the system. We use the Dictionary concept to build such a mental model of management capabilities of a system, with management operations as metaphorical controls. Available (potential) behaviour is abstracted in Dictionaries, and then constrained by policies.

Dictionary integrates different management concepts embedded in a system by providing a system-wide naming mechanism that comes as a part of the SPL information model. The information model represents a system by management information – the management view of those aspects of the system that are of interest to management. Typically, a Dictionary in SPF would be implemented as a database or a Directory service.

Semantics of Dictionary information is envisaged to be dual. On one hand, it should represent taxonomy of system management information at certain level of abstraction. Taxonomy is a hierarchical, structured presentation of information by categories [Burbeck 2000]. It provides semantics of management services categorized, so that managers can understand the system management capabilities and locate them quickly and use their services effectively. Essentially, we create Dictionaries to better understand management capabilities of managed systems.

On the other hand, a Dictionary should define a vocabulary of our policy specification language, because it represents an agreement on names. In SPF, the role of Dictionaries is to model managed systems in terms of their management behaviour. Since policies are specified in terms of the system management behaviour, Dictionaries indeed define a semantic domain for "words" (management operations) in a policy specification language. We use Dictionaries to formalize our policy specifications by creating a finite semantic domain for the available management interactions with a managed system.

The term informal means that the ways in which things are described depends to some extent on a common understanding of what is written among those people reading a statement. If that assumed common understanding was not present, the statements would become meaningless, ambiguous and/or incorrect. A formal specification makes clear the nature of all the terms to which it refers, which eliminates, or greatly reduces, the amount of intuition required to understand the specification. By

determining more clearly the meaning of the terms used in specifications, the scope for misunderstanding is reduced. The concept of Dictionaries is the key to the formality of policy specifications in SPF. Dictionaries precisely define the vocabulary of SPL, and, therefore, the meaning of all policy specifications. It assures that every component of a policy specification has been defined prior to its use, and that its definition makes sense within the context of a managed system.

A hierarchy of Dictionaries enables separation of concerns: top-level managers will work exclusively with the most abstract Dictionary and specify policies, while executives and technical personnel will maintain lower-level Dictionaries and mappings between them. Management policies will be specified in terms of abstract management interactions from the highest level Dictionary, while the rest of the Dictionaries will form an abstraction hierarchy all the way down to the lowest-level Dictionary, which specifies the real system management interface (see Figure 4-5).

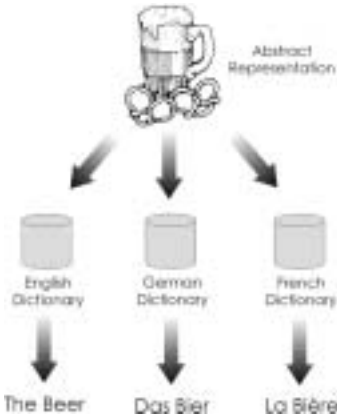


Figure 4-4 Interpretation of an abstract representation in different contexts (dictionaries)

The semantics of an abstract representation is immune to the choice of a context for its interpretation, or to the changes in the chosen context (Figure 4-4). Therefore, policies, as abstract specifications of management activities, will be immune to changes in their interpretation contexts (Dictionaries) or in the mappings between those contexts, which means that high-level management policies become independent from the evolution of managed systems.

Flexibility and dynamism of the policy specification concept are very important in the modern distributed systems, which constantly evolve. In SPF, flexibility of the policy specification concept comes as a result of the separation between specifications of the behaviour (Dictionary) and specification of constraints on that behaviour (policies). As a result, policy specifications are able to maintain their immutability during a very dynamic life cycle of distributed systems.

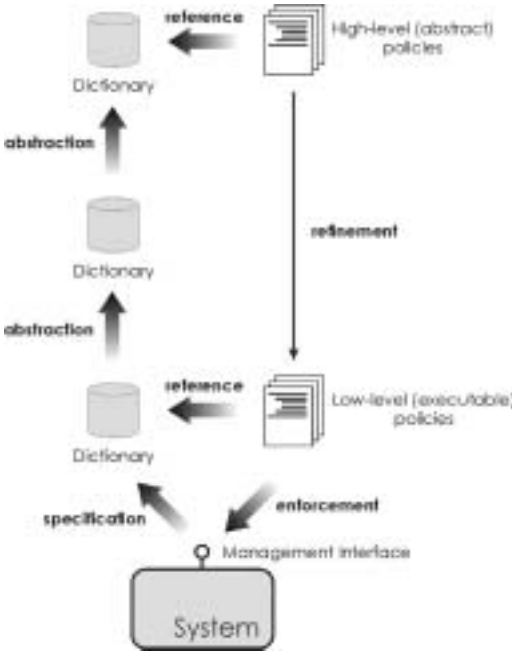


Figure 4-5 Abstraction/refinement mechanism in SPF

A hierarchy of Dictionaries (Figure 4-5) is needed to decouple a high-level management model of a system from changes in the real system, and to create a simple enough management view of the system by abstracting its management model. A simplified management view of the system will enable creation of a smaller number of abstract, high-level management policies, which will be easy to manage and to use. For the purpose of enforcement, those abstract policies will be refined to low-level specifications, which will be expressed in terms of the real management operations from the system management interface (for more details, see section 4.2.2.3).

4.1.2.1 Abstraction mechanisms in Dictionaries

According to [Mylopoulos 1999], a conceptual model in general comprises a collection of primitive terms, which specify a set of basic building blocks for constructing symbol structures, structuring mechanisms for assembling and organizing symbol structures, primitive operations, for constructing and querying symbol structures, and general integrity rules, which define the set of consistent symbol structure states, or changes of states. These are accompanied by interpretation rules and usage guidelines. A Dictionary represents a conceptual model of a system at an optional level of abstraction. The model is formally defined in SPL (for more details, see Chapter 5), and it includes:

- *Building blocks*, which are management operations (the Action type in SPL).
- *Structuring mechanism* based on aggregation/decomposition, which includes Interface, Service, Role, Node, Domain and System elements. Dictionaries support two models of organization: structured (hierarchical) and classified.
- *Classification/instantiation mechanism.*
- *Generalization/specialization mechanism.*
- *Integrity rules*, which are formally defined in SPL as rules of containment (see Figure 4-6).

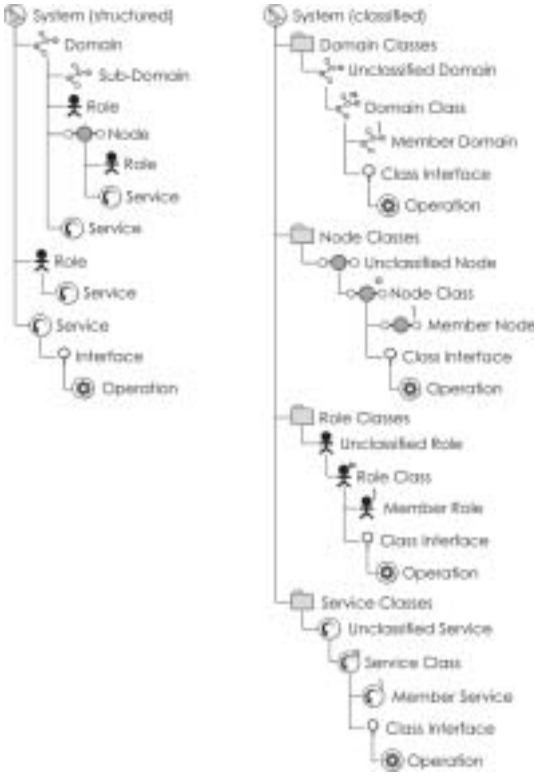


Figure 4-6 Containment rules in Dictionaries

Operations on Dictionary elements are left to be defined by particular SPL implementations, with respect to the general rules of referential integrity.

4.1.3 SMS Broker

System management requires dynamic definition. For policies to be implemented in a coherent fashion, it is necessary to have a mechanism that discovers and tracks resources and utilization [Aiken 2000]. According to [Maullo 1993], policy management process requires a common representative model of the system and its control structures in order to maintain context, coherence, and integrity. This model should maintain a real-time representation of the system being managed and the status of its constituent components. As the system changes and resources are added, deleted, and changed, the available primitives must also change in such a way that desired policy statements can be entered unambiguously. This would require the real-time evaluation of the management vocabulary. It is therefore suggested that the vocabulary should be dynamically established by the system model.

As an important part of the RM–ODP standard, the Trading Function (see [ITU/ISO 1995]) allows a service consumer (importer) to be "matched up" with a service provider (exporter) by a trusted third party (trader) dynamically, at run-time. ODP objects can be configured into an ODP environment without prior knowledge of the services or service providers within that environment. A consumer is able to use an appropriate service provider, even though the requirements of the consumer, the state of the system, and the services in the system may change dynamically. The Trader represents an infrastructure component of a distributed system that provides the Trading Function, a framework for "trading"

services in an open distributed computing environment. The real advantage of the ODP Trader is in large distributed environments where objects need to be made aware of the services available [Pratten 1995].

In general, brokering is a process of managing the flow of identity information between repositories. In SPF, the SMS Broker is essentially an information provider, whose goal is to help in creation and maintenance of system management models. The SMS Broker makes system management capabilities available to SPF through the process of discovery. A publishing/discovery mechanism makes transparent the changes in the system management interface, which may come as a result of components reconfiguration and upgrades (static changes) and/or service mobility (dynamic changes). The functionality of the SMS Broker was envisaged to consist of three tasks: interfacing with different sources of management information, making queries against those information sources, and preparing received data for use in SPF.

With its interfacing capabilities, the SMS Broker promotes SPF into a generic (system independent) plug-in component. Because the SMS Broker gives essential plug-and-play capabilities to SPF, it is desirable that it supports interactions with various sources of information (Figure 4-7):

- *UDDI Registries*, for the discovery of message-based management operations (management services). A Web service is fundamentally an interface accessible over a set of open standard discovery and invocation mechanisms. The services discovery and binding processes are driven by interface descriptions. The management interface will be most likely published in a system's private UDDI registry that caters to SPF as the client.
- *ODP Traders*, for the discovery of RPC-based management operations. Even though a tighter coupling between an embedded workflow engine and providers of management operations will be required, the overall SPF management process will not be largely affected by the lower-level nature of the management interactions within managed systems.
- *Directory services*, which offer generic information storage and discovery facilities. They may be used to implement UDDI Registries and other mechanisms of resources registration, tracking and discovery.

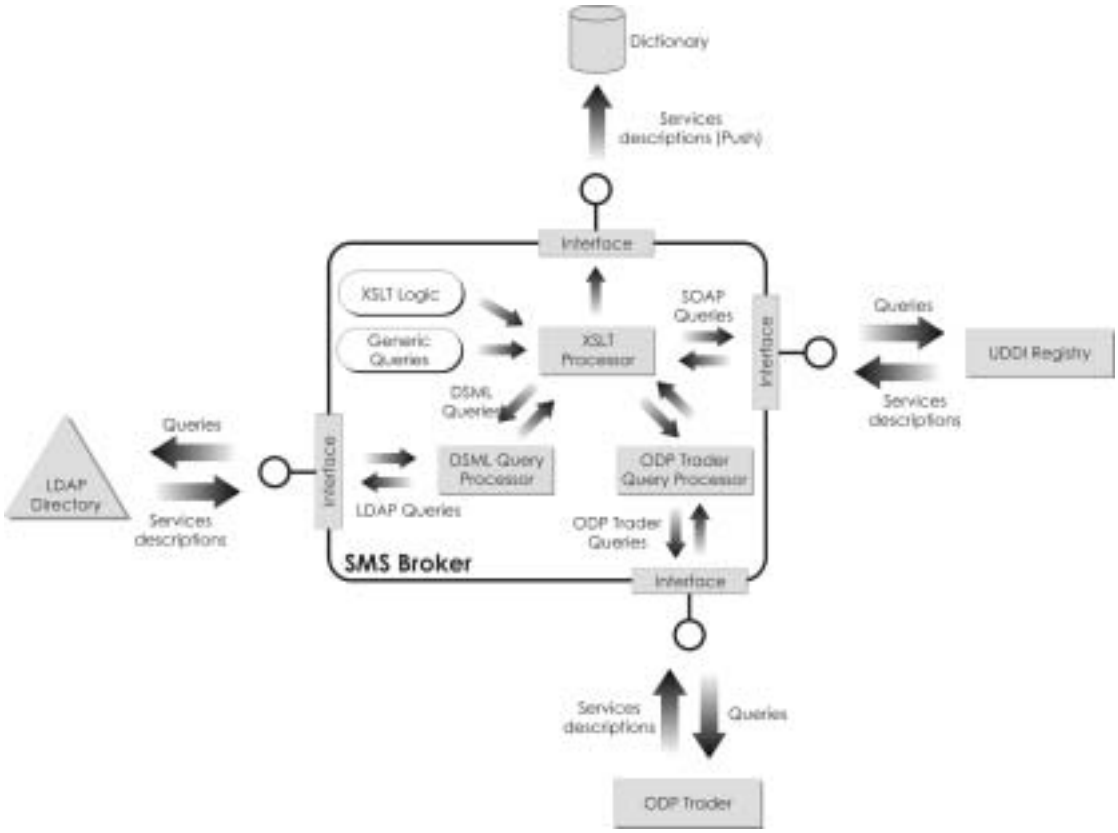


Figure 4-7 Anatomy of the SMS Broker

Generic queries for the discovery of management services will be created by human managers, transformed into an XML representation, and then parsed into appropriate formats using XSLT (eXtensible Stylesheet Language Transformations). XSLT currently represents the standard technology for XML transformations. It is a declarative language that uses pattern matching and templates to specify how to transform an XML document from one structure to another. Such queries would include information about management services of interest for SPF, which are very much like SQL queries for databases. Depending on the source of information available, the queries may include more or less details. In general, the richest possibilities for making sophisticated queries are offered by UDDI Registries and Web services technology because of their standardization. However, Directories and ODP Traders may also represent rich sources of information.

The SMS Broker will transform received replies to its queries into structured and explained information (XML) about management services, which may be easily parsed into the SPL information model and stored in Dictionaries. Even though most of the modern database systems are able to work directly with XML, classic SQL queries may be used too. If the management operations are available as Web services, every operation from the lowest-level Dictionary may have an associated WSDL file that describes it, and from which all the information about the service may be imported into the Dictionary: interfaces, operations, parameters and various service properties. The exact technology for making received information about management services available in Dictionaries will be left to particular implementations of SPF.

According to [Barros 1997], a modelling technique should provide an explicit notion of scenario for model validation. Validation is concerned with ensuring that a conceptual model is indeed a model of the managed system. In SPF, the content of the lowest-level Dictionary is obtained through the process of discovery, so it truly reflects management capabilities of the system. In cases when the discovery cannot be performed, due to the lack of support from the managed system, the responsibility for Dictionary modelling will be on human user.

4.1.4 Workflow engine

Workflow engine is a component embedded in managed systems, as a feature typically provided by application servers. In the SPF management model, a workflow engine transforms actions prescribed by a manger (policies) into triggering of management operations from the system management interface. In other words, it executes management policies in the system for SPF. Depending on the internal design of a managed system, the workflow engine itself may interact with the system via the management interface, or it may work directly with the system components.

Policy triggering may generate an ad-hoc flow of management activities in a system. In a proactive management scenario (Figure 4-8), initiated by a manager, a policy of choice is sent to a workflow engine for execution. Based on the outcome of management actions, a management event may be generated by the workflow engine or by the system, and sent back to SFP. In response, an Event Handler may trigger one or more policies, so the management process continues automatically until the satisfaction of management goals. For more details on policy triggering, see 4.2.2.1.

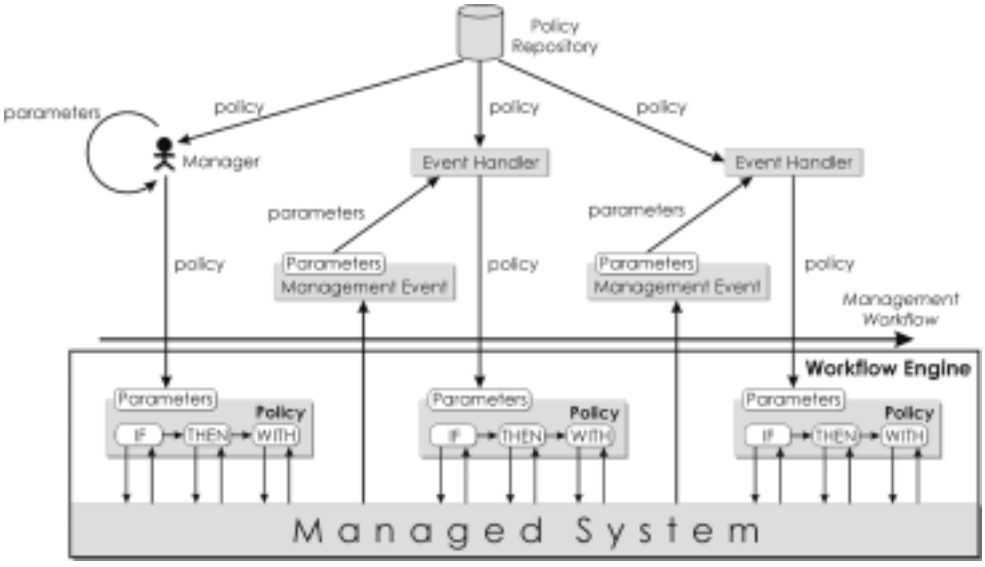


Figure 4-8 Proactive management workflow

Workflow engine will notify the management framework about policies that had failed – policies whose post conditions where not satisfied after the execution. The basic form of the management event is:

ManagementEvent(policyID)

Workflow engines that support more sophisticated reporting will be able to send events that include additional parameters. Management events will be sent back to SPF together with other system events.

The following Event Handler will trigger again the same policy ("Throughput manager") in case when the policy did not succeed (throughput is still under a limit):

ON
ManagementEvent(Throughput manager)
IF
GetThroughput() < limit
TRIGGER
Throughput manager
USING

Similar to the previous, proactive management scenario, a reactive management scenario can also initiate a management workflow. In this case, the workflow will be generated as a response to a system event, which will be handled by an Event Handler and appropriate flow of policies (Figure 4-9).

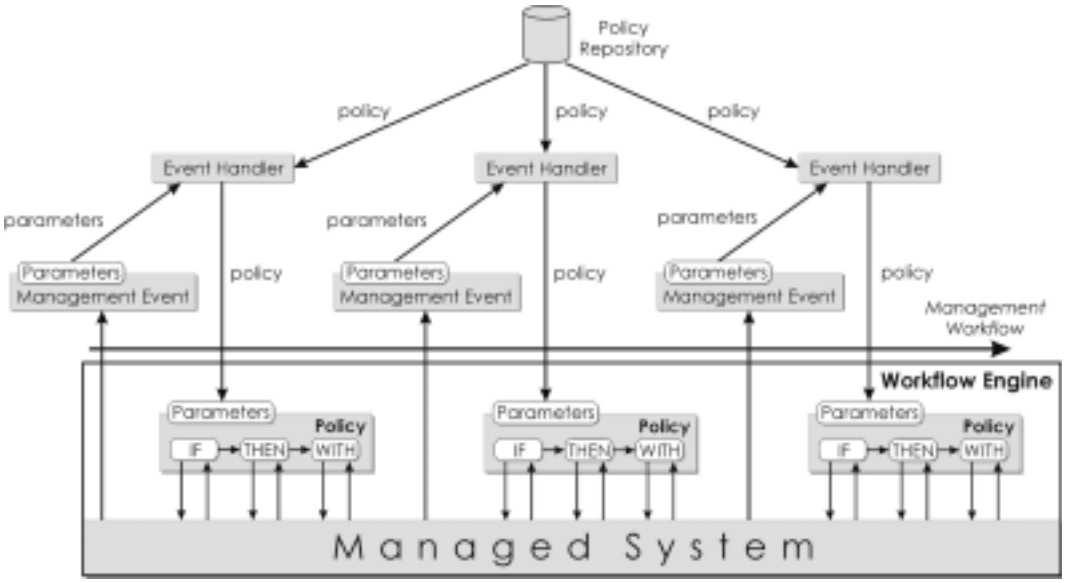


Figure 4-9 Reactive management workflow

In addition to the flexibility within SPF itself, the design of the management framework assures the flexibility of policy execution as well, by supporting late binding. Policies in SPL don't specify service providers (implementation) for the management operations they reference. As a result, the actual service providers may be defined (discovered) at the triggering time, by the workflow engine. SPL enables a formal portrayal of the management services semantics to be attached to management operations that should be used to find services. Because of this rigor, the service requester does not require human intervention at run-time to choose which service to bind to.

Mappings between management services and services providers are made dynamic, and can be changed at any time. For example, in case of a WSFL-compatible workflow engine (can execute WSFL

specifications), interactions with an UDDI Registry are required for the discovery of service providers for policy management operations. SPL implements the Path member in the policy Action type (see 5.1.2), which may be used to attach a:

- *Particular WSDL file* that describes the corresponding service, which includes the operation that a policy action is referring to. In this case, workflow engine will use the service provider specified in the WSDL file to trigger the appropriate operation (static binding).
- *UDDI query*, to dynamically locate a service provider in the UDDI Registry during the parsing of policies in a workflow engine (dynamic binding).

Similar functionality may be achieved with Directory services, in CORBA environments, or other architectures that implement ODP Trading Function or some equivalent trading service. In all the cases, the binding information will be kept external to policy specifications, and used at run-time by a workflow engine. Nevertheless, the exact nature of interactions performed by a workflow engine in the course of policy triggering is technology-dependent.

Decoupling of SPF from the managed system enables support for service mobility, system reconfiguration, faults management, and other internal processes that may affect system behaviour and configuration. Service mobility is a resource management strategy based on decoupling users from the system resources by abstracting resources into ubiquitous services. In such a model service mobility is the act of change of service providers, and it is based on their replication and/or relocation. Service mobility represents a very important aspect of a system flexibility, which may be used as an instrument for load balancing, resources management, operating costs control and QoS optimisation.

4.1.5 Policy Modelling Tool

Policy modelling tool represents the central component of SPF, and it is responsible for various aspects of monitoring, analysis and control. It includes the following major functional components (Figure 4-10):

- *Policy modelling environment* essentially represents a GUI, which enables human managers to interact with the management framework. A detailed analysis of the component, based on a prototype implementation, will be presented in Chapter 6.
- *Event listener* is, together with the SMS Broker, an essential part of the SPF plug-and-play interface with managed systems. It is responsible for receiving, filtering and handling system events.
- *Policy parser* simply performs policy transformation from SPL to particular XML formats, required by the workflow engines.
- *Triggering Pool* is a component that temporary stores policies scheduled for triggering (policy cache).

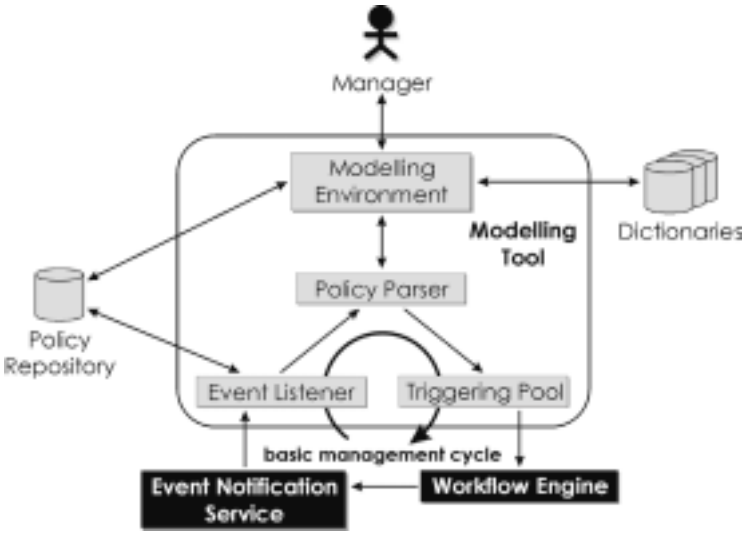


Figure 4-10 Anatomy of the Policy Modelling Tool

Apart from the system event notification service, which interacts directly with the Event listener, the Triggering Pool and the Event listener may access the system management interface only indirectly, via the workflow engine.

We will look now more closely at the main components of the Policy Modelling Tool.

4.1.5.1 Event listener

Events may be signals of important changes in a system; indications that certain aspects of the system state should be checked and that some management interventions may be required. They are only signals for attention, not necessary for intervention. The publish/subscribe communication paradigm connects together information providers and consumers by delivering events from sources to interested clients. A client expresses interest in receiving certain types of event by submitting a predicate defined on the event contents. In addition, filters allow clients to receive only the events they are interested in, and to tell suppliers which events are in demand. SPF proactive management model requires the publish/subscribe communication paradigm to be supported by managed systems.

In SPF, Event Handlers are used to decouple generic, high-level policies from the system and management events. Event Handlers behave like high-level call-back methods that respond to events by triggering management activities (policies). For more details on policy triggering, see section 5.1.3.4. Upon receiving an event, the Event listener looks for an appropriate Event Handler. The Event listener keeps a record of the past events in order to be able to process scenarios of events (sequences of ordered or unordered events) that may be defined to trigger Event Handlers. If an appropriate Event Handler exists, its conditions will be checked against event parameters and the current state of the system. If the conditions are satisfied, corresponding policies will be selected from the Policy Repository and sent to the Policy parser.

4.1.5.2 Policy parser

The Policy parser receives policies from two sources: the Event listener, in the reactive management scenario, and (indirectly) from a human manger, in the proactive management scenario. Its task is to transform policies from the storage to the operational (executable) format. The Policy parser will perform policy interpretation (for details, see 4.2.2.3) and transformations from the SPL representation, via an internal XLM representation to a target XML dialect of the system's workflow engine (see Figure 4-11). Policy interpretation requires access to Dictionaries, which may be achieved via the Modelling Tool, or even directly, which depends on the particular SPF implementation.

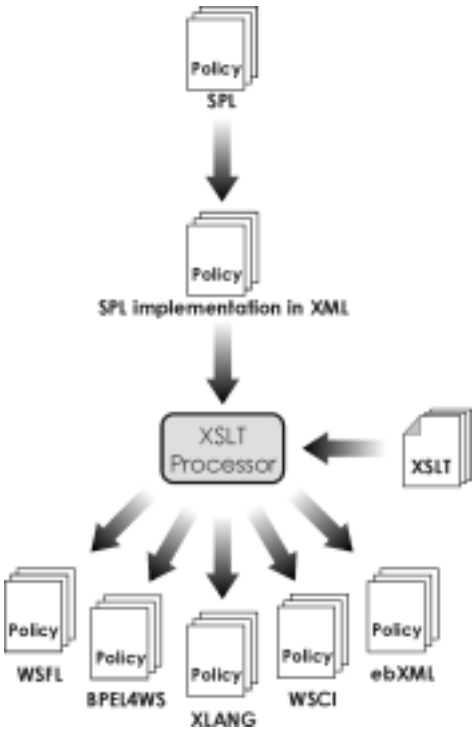


Figure 4-11 Policy transformations within the Policy parser

In SPF, policies get prepared for the execution in a workflow engine in a two-step process. First, SPL representation of policies is parsed to an internal XML format. Internal SPL implementation in XML gives to SPF independence from the current XML technologies, which are in constant flux. An example of such a SPL implementation in XML may be seen in section 6.4.3. Also, transforming policies to a neutral XML representation enables the use of the standard XML transformation technology (XSLT). In the next step, XML policies are transformed to a desired language using XSLT. Commercial workflow engines support different XML-based languages for specification of workflows they process. A workflow interoperability standard has yet to be decided on, even though there are several proposed workflow standards around.

After the transformations policies are ready to be executed, and they will be sent to the Triggering Pool.

4.1.5.3 Triggering pool

The Triggering Pool represents a cache for policies scheduled for triggering. Policies are cached before triggering to enable control of the SPF management process by means of semantic rules. Semantic rules are applied to triggered policies before they can join the Triggering Pool in a two-step process, which includes semantic analysis and conflict handling. A detailed presentation of that process is given in section 4.2.2.2.

4.1.6 Policy Repository

The efforts for defining policy-based management, especially in DEN, have focused on the data representation and properties of a repository for that information. The use of a repository is important to support reusability of data across managed resources, as well as allowing a manager to edit existing management data (both are forms of reuse). According to [Mahon 2000], information distributed from a centralized repository also aids in consistency of information throughout the managed environment. In SPF, the Policy Repository is just a simple repository service, which may be implemented as a database or a Directory service.

For the Policy Repository, SPL defines an information model that describes how to store policies and policy-related information (for more information, see Chapter 5). It encompasses identification, naming and definition of elements, classification of elements (taxonomy), relationships among elements, their cardinality and constraints, and element attributes (detailed element design). Policy Repository may also implement a number of stored procedures for automated information management. For example, the stored procedures would play an essential role in the implementation of a policy scheduling mechanism, because policy attributes, which determine triggering time and intervals, need to be maintained up to date.

4.2 Framework processes

Common understanding of a process is a series of actions, activities or executions that are in a certain relation or context to each other, in order to produce products or services. They involve operations (process steps) and operands (units of information). Process-oriented models are often derived from basic concepts of condition/action ideas and Petri Nets (state/transition diagrams). In this formalism, a specific condition turning true triggers the corresponding activity. Different conditions may be necessary to trigger an action and an action can cause several succeeding conditions. Hence this approach models a context between initial conditions which trigger an activity, the following conditions, and succeeding new activities.

Procedural models stress the tasks-orientation in the sense that each procedure is designed to perform a certain complex task. The SPF management process encompasses a number of processes, with policies as the basic unit of management information, which work together in order to achieve two goals. First is system management, which is the main goal of the SPF management process. It includes system and policy modelling, policy enforcement, and information management. Second goal is management of the

management process itself (self-management), which aims at enabling a high degree of automation. Event Handlers and semantic rules are used to create, monitor and to control flow of management activities in SPF.

The entire SPF management process has been designed to support a particular policy life-cycle (see Figure 4-12), which consists of the following stages:

- **Specification**. Policy modelling is the initial stage of the policy life-cycle, which begins with a preliminary idea and finishes with policy enabling.
- **Enabling**. Policy enabling is the step at which policies become operational. Policy enabling/disabling may be performed using different mechanisms, which completely depends on an implementation. For example, policies may have attributes, such as enabled/disabled, activation date, validity period, etc., which may be used by the framework to control their status.
- **Enforcement** includes several cycles of:
  - *Triggering*
  - *Semantic analysis*
  - *Conflict analysis*
  - *Interpretation*, which transforms policies from their storage to an operational (executable) format, and consists of:
    - Instantiation
    - Refinement
    - Transformation to an XML representation
  - *Execution* of interpreted policy(ies).
  - *Deletion* of interpreted policy(ies)
- **Modifications**, which come as a result of changes in the management environment and/or changes in the management strategy.
- **Deletion**, when policy becomes obsolete

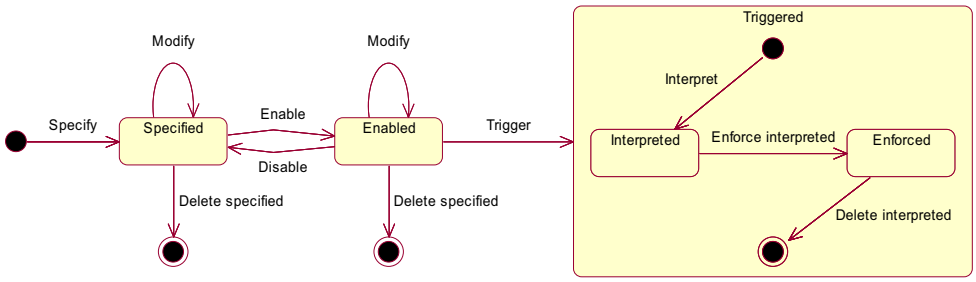


Figure 4-12 Policy life-cycle represented in a state transition diagram

In the rest of this section we will focus on the policy specification and enforcement, which represent the key processes of the overall SPF policy-based management process.

4.2.1 System modelling and policy specification: An informal methodology

In general, a methodology should define steps to support a predictable development progress towards a solution. The purpose of our methodology is not to guarantee the best modelling outcome, but to improve the chances of achieving an effective policy-based management solution. It provides a basic plan about how to discover the essential requirements and to present them using a modelling language, analyse the requirements, divide the model into manageable units, model the system in terms of its principal components, and translate the models of a problem into a model of a solution (Figure 4-13).

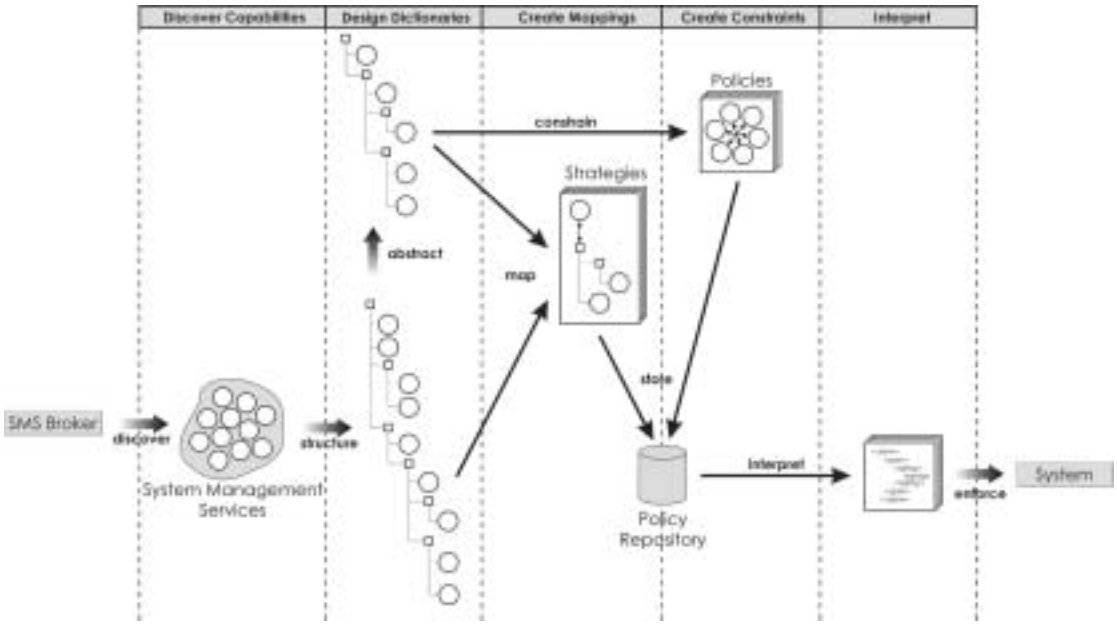


Figure 4-13 Informal methodology for modelling in SPF

Modelling for the purpose of policy-based management has two goals in SPF: to specify and abstract an existing behaviour (Dictionary and strategy modelling), and to specify constraints on it (policy modelling):

- *Dictionary modelling* specifies system management capabilities by building a management model in terms of operations from the system management interface. Dictionary modelling results in a hierarchy of Dictionaries, each of which is at a different level of abstraction.
- *Strategy modelling* defines mappings between Dictionaries at different levels of abstractions, as refinements of abstract, higher-level operations in terms of operations at a lower level of abstraction.
- *Policy modelling* uses management capabilities of a system, specified in Dictionaries (what can be done in terms of management), to specify the required behaviour of the system as a collection of management policies.

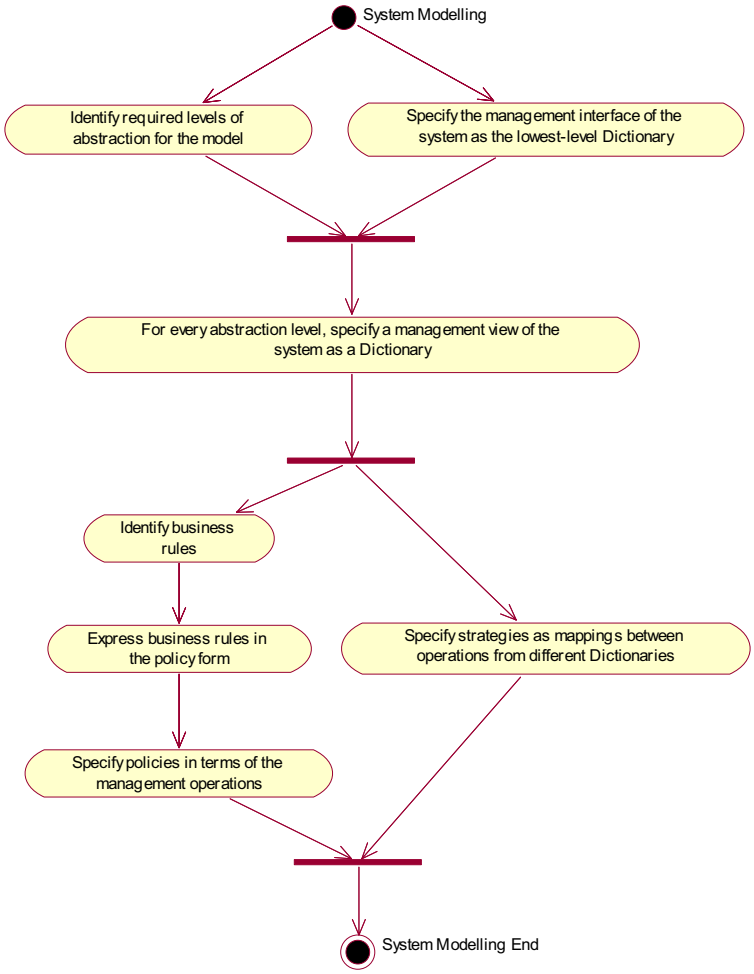


Figure 4-14 Basic steps in SPF methodology

The basic steps in the SPF modelling methodology are presented in Figure 4-14. Modelling of Dictionaries at different level of abstraction is a perquisite for policy modelling, because policy building blocks come from the Dictionaries. Strategy modelling is a prerequisite for policy interpretation, because the meaning of high-level operations (policy building blocks) needs to be refined in terms of real management operations, specified in the Dictionary at the lowest level of abstraction.

4.2.1.1 Dictionary modelling

It is important to bear in mind that the Dictionary modelling considers a system exclusively for management purposes. Therefore, a Dictionary should contain only those aspects of a system that are interesting from a management point of view. The first modelling step should be to determine a level of complexity for a Dictionary (level of abstraction for a system management model). Dictionaries may be specified to include different levels of details – to be at different levels of abstraction.

Management view of a system in SPF, especially if it is a complex one, should be created as a hierarchy of Dictionaries, and tied together using strategies as mappings between them. The Dictionary modelling may begin from the top or the bottom of the management hierarchy, but the Dictionary at the lowest level of abstraction shall be specified in terms on the interactions from the real system management interface. Once we establish a clear picture of the system management capabilities, we may begin with constraining its behaviour (policy modelling).

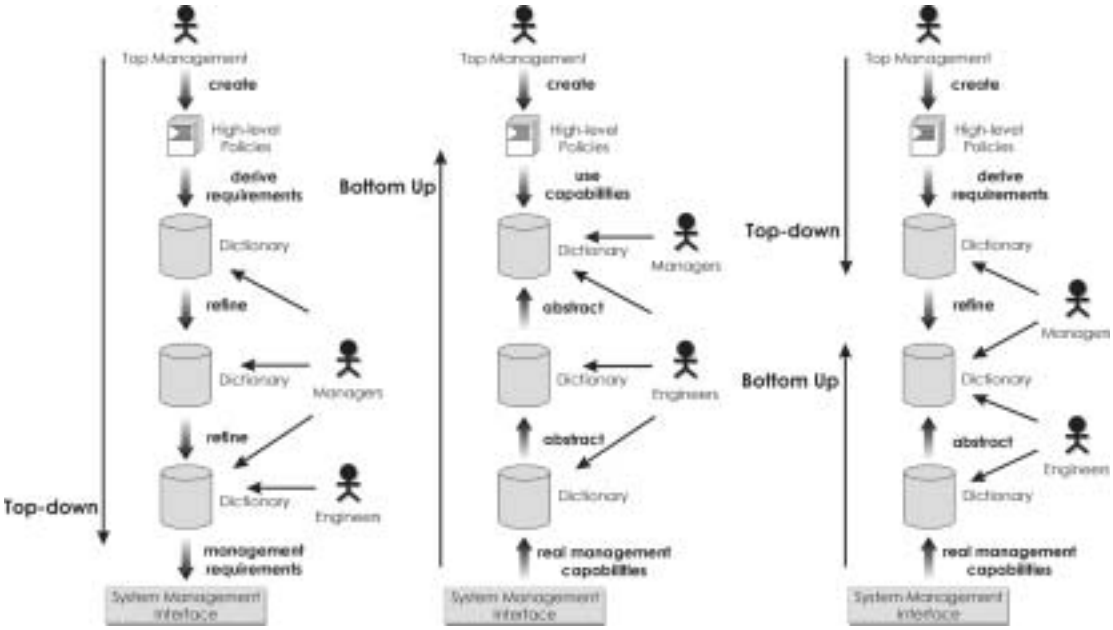


Figure 4-15 Dictionary modelling

In general, policy-based management of a system may be realised via three basic scenarios (see Figure 4-15). Bottom-up approach (start from limitations) is concerned with synthesis of existing components to make new structures. Discovered primitive management operations are assembled into larger components by primitive combinators. The focus is on what can be done with the system, taking into account its management capabilities. System management capabilities limit the expressive power of policies that might be specified/enforced when there are no possibilities to upgrade the system to meet the management requirements. Technical personnel play the major role in the system modelling, by specifying what is possible to do with the system in terms of management, whereas the management must adjust its goals to the given limitations.

Top-down approach (start from requirements) is concerned with analysis and decomposition of the problem into manageable pieces. It starts with an attempt to understand a system as a whole, by modelling external aspects of the behaviour of the system, such as might be observed by a high-level manager. The focus is on what needs to be done with the system in terms of management. Real system management capabilities might need to be upgraded and adjusted in order to support the management requirements. Managers have the full control over the management view of the system, whereas technical personnel work on the real system to meet the management requirements.

In the combined approach (compromise), the merits and deficiencies of top-down and bottom-up approach, which are entirely complementary, are combined. Management requirements and management capabilities meet in the middle, where a compromise has to be made with the help of both management and technical personnel. Both management requirements and the management capabilities need to be adjusted in order to attain realistic goals. Flexibility of a system design will determine to what extent the management capabilities may be changed to meet the management requirements.

A conceptual Dictionary modelling may be performed in UML, using a set of custom designed stereotypes to represent SPL types (see Appendix C for details). Such conceptual models may be further refined in a dedicated policy modelling tool, such as PolicyModeller (see Chapter 6). If carefully designed, resulting hierarchical model of a system in terms of Dictionary specifications at different levels of abstraction will be able to offer an insight into the system organization and behaviour from different viewpoints, with a well controlled level of details.

4.2.1.2 Strategy modelling

In SPF, strategise tie together Dictionaries at different levels of abstraction in one integral management model of a system (Figure 4-32). A strategy is very much like a function definition in programming languages. It has a name, a signature and a body. Every strategy is specified for a particular Dictionary operation, to define its refinement, and it borrows its name and signature from the operation it refines. Strategy body specifies the meaning of a strategy in terms of control structures (IF-THEN-ELSE and WHILE-DO) and lower level operations, and it is essentially very much like a specification of an algorithm in pseudo code.

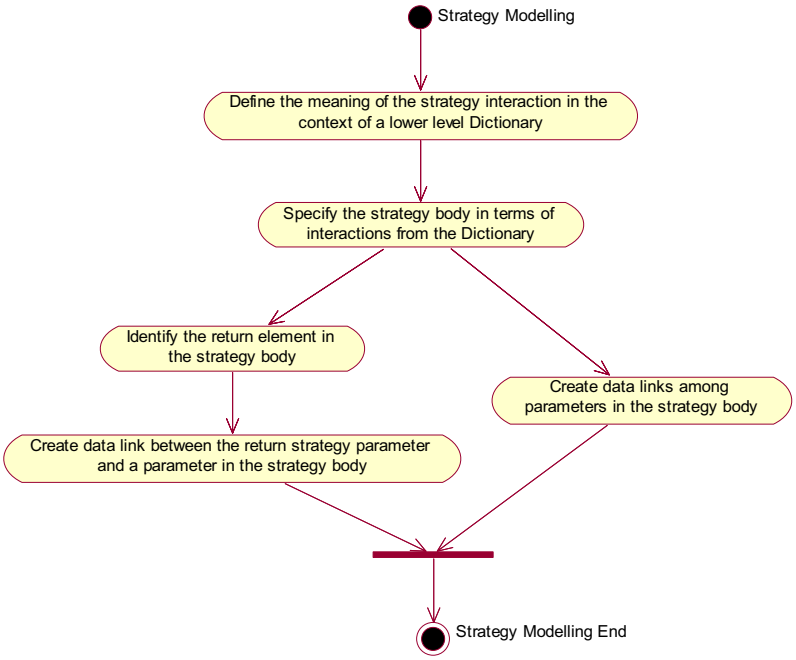


Figure 4-16 Basic steps in strategy modelling

Strategy modelling begins with selection of an abstract operation for the refinement (Figure 4-16). The next step is to express its semantics in terms of management operations from a lower-level Dictionary. In general, operations from a strategy body may be from different Dictionaries. A strategy body can contain Dictionary operations and/or one of the strategy control structures (see Figure 4-17), which in turn contain their operations. IF-THEN-ELSE and WHILE-DO control structures may be further expanded, by placing other control structures in their action parts (THEN, ELSE and DO parts). When a strategy structure is defined, management operations are selected from Dictionaries and placed into the structure of the strategy body.

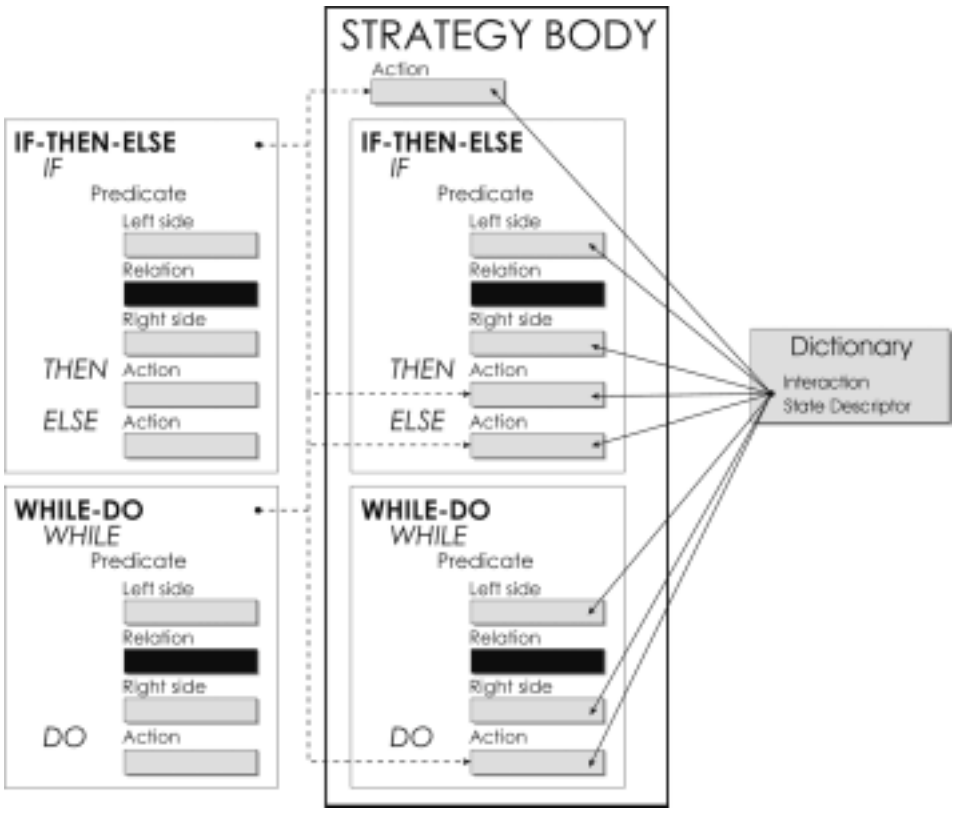


Figure 4-17 Strategy assemblage

Finally, it is necessary to specify the flow of data within a strategy, by defining strategy links. Mechanism of links is used in strategies to connect parameters that share same values. Links specify how parameter values are assigned in three scenarios: Strategy input parameters pass their values into the strategy body, values returned by operations are passed to input parameters of other operations within the strategy body, and the return parameter is passed from the strategy body to the return parameter of the strategy (Figure 4-18). Examples of strategy specifications are given in the Library example in Appendix C.

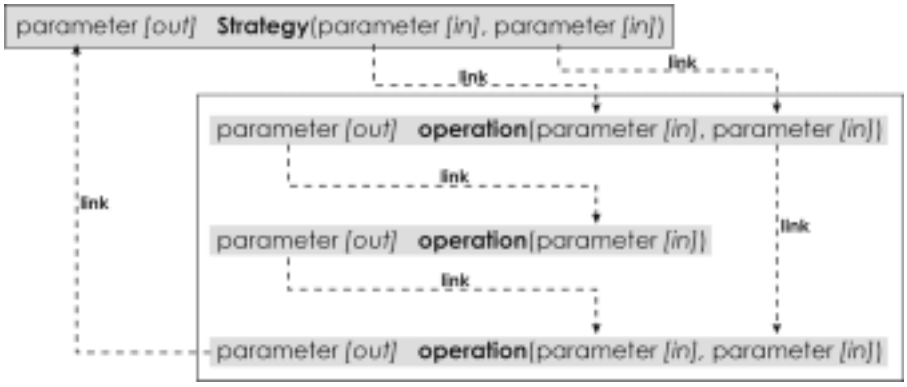


Figure 4-18 Flow of data in a strategy

When all the necessary mappings between Dictionaries are defined in terms of strategies, the management model of a system is complete. Such a specification of the system management behaviour defines a context for specification of management policies.

4.2.1.3 Policy modelling

In SPF, each policy specification includes of two basic parts:

- *Policy declaration* (semantic specification) includes Metadata and Accounting information, which explicitly states the essential policy facts and attributes (for more details, see 5.1.2).
- *Policy definition* (operational specification) is in the IF-THEN-WHILE form, which specifies management interactions with the system.

It is a good practice to start a policy specification with the policy declaration and to specify its purpose and semantics (Figure 4-19). Then, the operational part comes as an implementation of the intentions stated in the policy declaration.

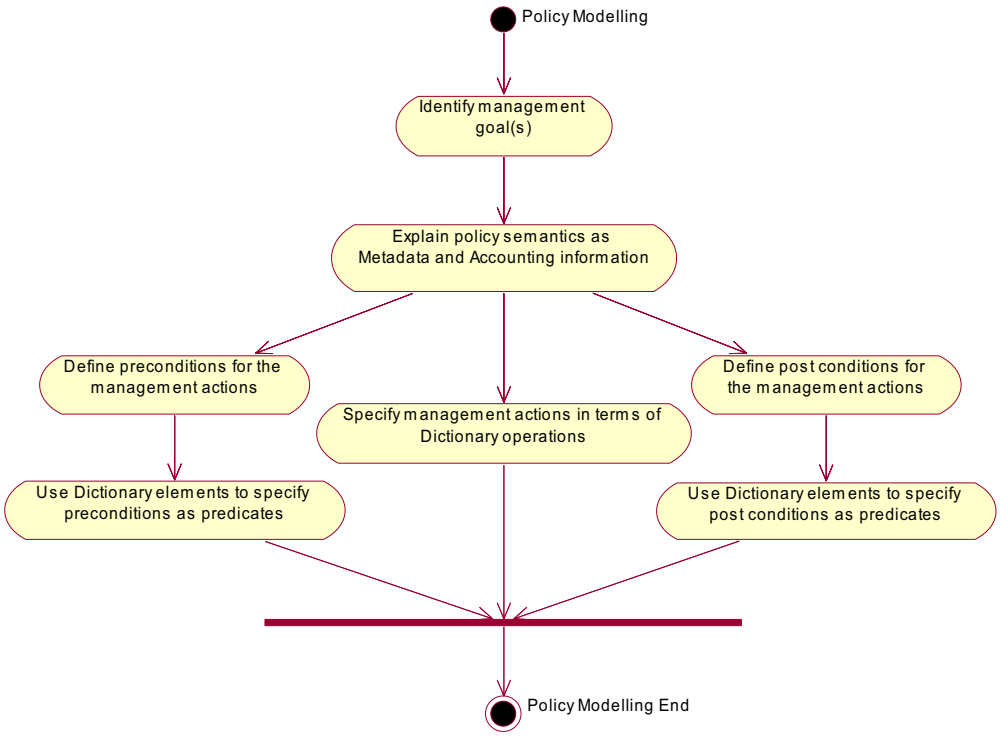


Figure 4-19 Basic steps in policy modelling

Policy definitions are assembled by selecting Dictionary operations (building blocks) and putting them into placeholders in the predefined IF-THEN-WHILE policy structure (Figure 4-20). Examples of policy specifications are given in Appendix C.

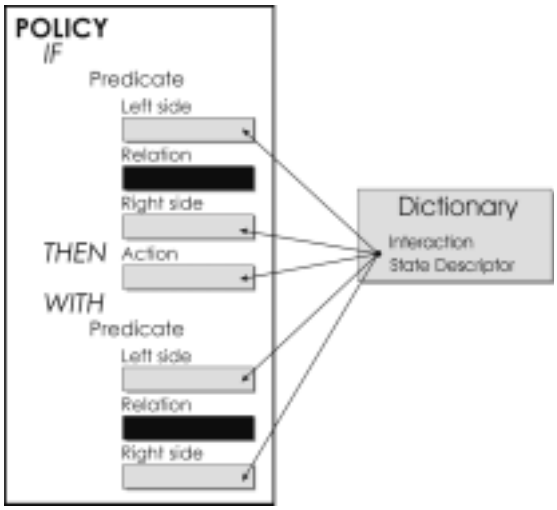


Figure 4-20 Assemblage of policy building blocks

When the policy modelling is finished, the modelling process for SPF is over. The system and its behaviour are fully specified, and SPF will able to enforce defined constraints on a real system.

4.2.2 Policy enforcement

Policy enforcement is the core process in the SPF management model. Its basic steps are shown in Figure 4-21. The entire policy framework and the policy specification language were designed to enable and support the policy enforcement process.

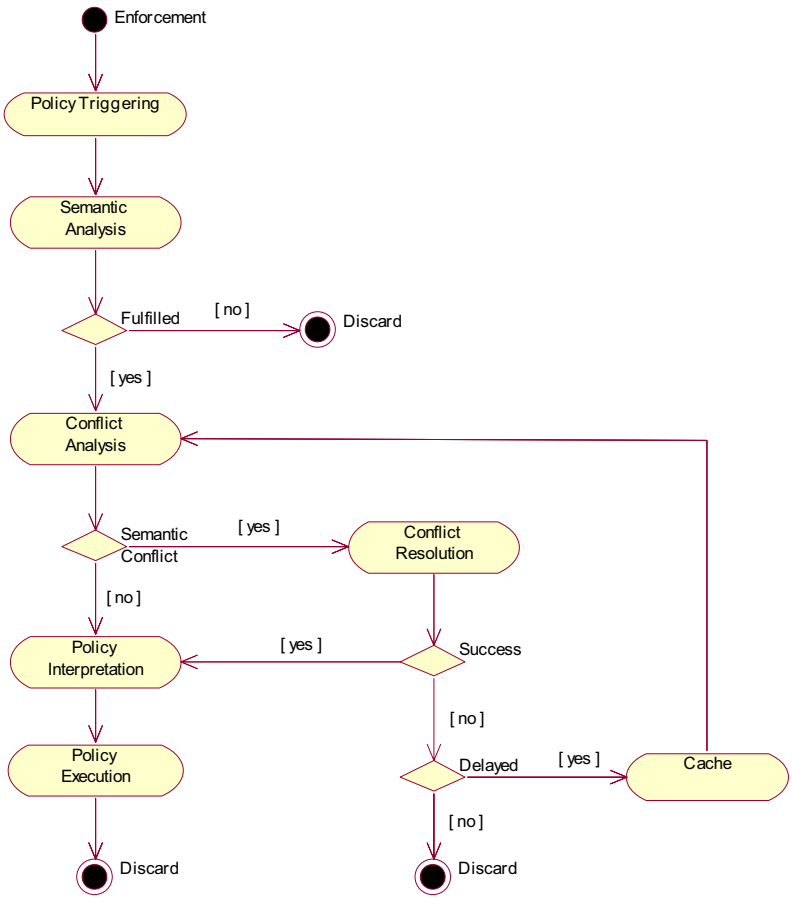


Figure 4-21 Basic steps in policy enforcement (policy life cycle)

Details of the policy enforcement process will be given in the rest of this section. We will start with policy triggering, which represents the initial step of the policy enforcement process.

4.2.2.1 Policy triggering

In our management model, policy-based management will be performed by means of interactions between SPF and a managed system. Based on the side that initiates these management interactions, we can make distinction between active and passive management scenarios (Figure 4-22).

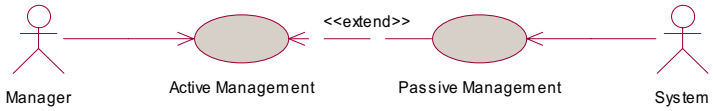


Figure 4-22 Basic management models

Active (proactive) management is a state-driven management model, in which management actions are initiated by managers. A manager initiates management actions periodically or when an opportunity arises. Manager takes an active role in the SPF management process, monitors and analyses some aspects of the system's state, and makes management decisions (triggers management policies) accordingly. The active management approach is based on a long-term, strategic planning by anticipation, and its goal is to prevent problems by adapting management activities.

Passive (reactive) management is an event-driven management model, in which management actions come as a response to events related to changes in the state of a system (Figure 4-23). Essentially, it includes the same steps as the active management; only the analysis is simplified and automated. The management response is also automated, performed without active participation of a human manager. SPF gets notified about the changes in a system by means of events, and it triggers appropriate management policies in response. The goal of passive management is to repair deficiencies in a system, typically on a short term.

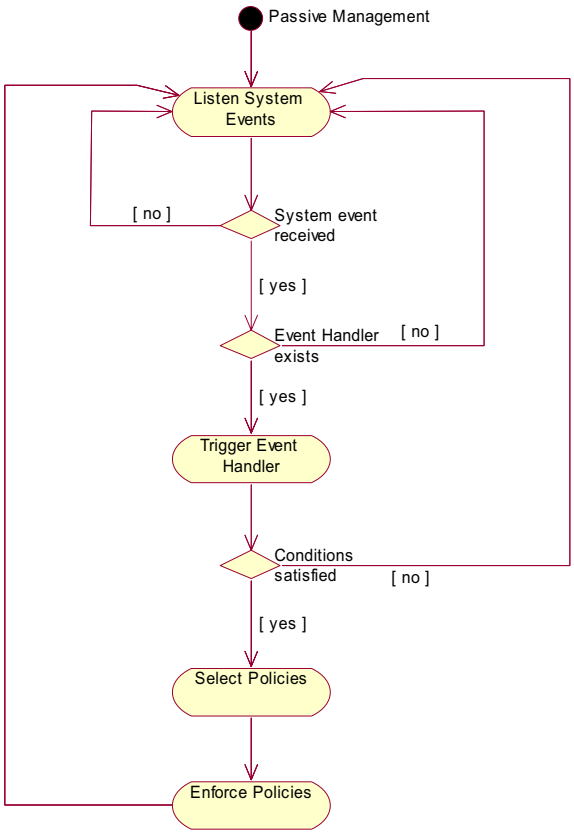


Figure 4-23 Basic steps in the passive system management

The passive policy-based management produces automated, reflective behaviour, in which SPF operates according to the rules prescribed by management policies. Automation and control of the process is achieved using management events and Event Handlers, together with a semantic and conflict analysis of triggered policies (see 4.2.2.2). In the passive management scenario, Event Handlers perform the same task as a manager would in the active management scenario:

- Select one or more policies.
- Possibly check certain conditions.
- Trigger policies if conditions are satisfied.

Event handlers

According to [Marriott 1993], the three main components of active mechanisms are events, conditions and actions. The policy conditions are referring to the system state that needs to be checked in order to decide how to act in response. The general idea is that an active mechanism invokes an action when one of its events is detected and its specified conditions are met [Marriott 1993]. Events come as a notification that some aspect of the system state may be changed. In response to events, we still need to check the system state before we take an action. Therefore, an event comes as a signal for a management action, but we don't act in a direct response to the event. Instead, we trigger one or more policies.

In SPF, management policies are seen as state handlers. They usually act on the system state without being concerned with how that state was reached (what was the cause). If the cause, however, is important for making a management decision, it will become a part of the system history (past system states), which can be referred to in policies and analysed using management operations.

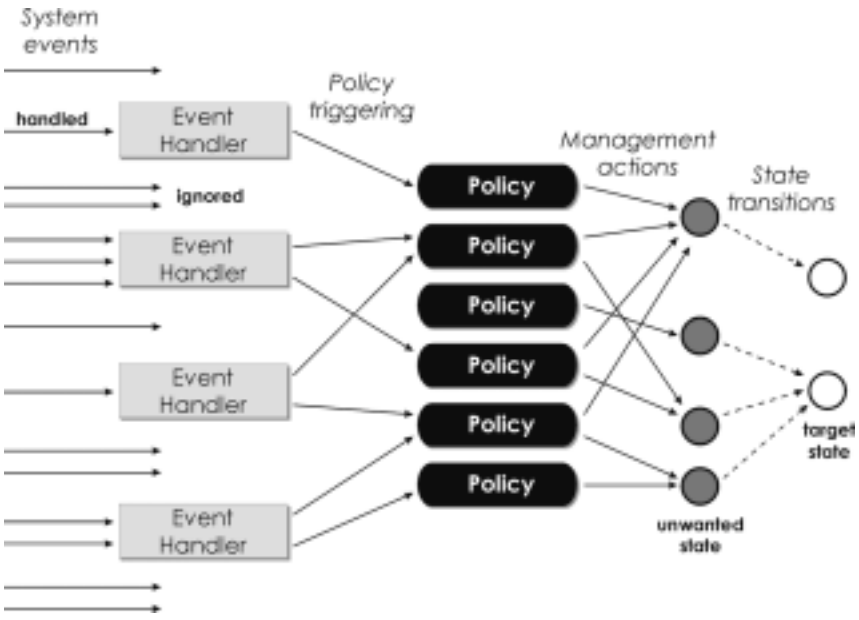


Figure 4-24 Event-based policy triggering

Management policies are associated with events by means of Event Handlers. The Event Handlers are used to decouple management policies from particular system events, in order to enable generic policy-

based management (see Figure 4-24). Policies control (respond to) changes in the system state by prescribing corrective state transitions regardless of the cause, so they can be used for both reactive and proactive management. Leaving events out of policies enables policy reuse in different management (triggering) contexts. Mapping between events and system states is not 1:1. Generally, one state may be changed as a result of many different events in a system. Instead of specifying a separate policy for every possible event that may cause the change, we simply want to specify only one policy to change the state of a system and one or more Event Handlers to trigger it.

In SPF, policy instantiation includes passing event parameters to corresponding policies, in order to simplify management operations and provide more flexibility for the SPF management model. Links (data flow) between event parameters and policies may be defined in Event Handlers, just like it was the case with strategies. For more details, see section 5.1.3.4.

The concept of using event parameters for policy instantiation:

ON
Event(parameters)
TRIGGER
Policy(parameters)

In order to introduce even more control and flexibility in the triggering model, SPF gives to Event Handlers a possibility to check event parameters and system state before triggering policies.

The concept of conditional, state-based policy triggering:

ON
Event(parameters)
IF
Condition(parameters, state)
TRIGGER
Policy(parameters)

Both Event Handlers and policies have the possibility to check system state before taking actions. Event Handlers may trigger policies only under certain conditions, and therefore perform filtering of events. This feature of SPL offers a possibility to balance control between Event Handlers and policies by deciding where to put most of the responsibility for management decisions. Extensive control on the Event Handler side would make them less generic, and it would result in a more complex triggering mechanism. However, such filtering would speed up the management process (avoiding potentially expensive policy instantiation, interpretation and triggering), and policies, as more generic, would become simpler and easier to reuse.

As a general rule, Event Handlers should focus only on checking some general conditions that are related to events they are controlling. First of all, this would include checking the soundness of parameters passed by events. More sophisticated, management-specific state analysis certainly belongs to policy specifications. It is important to bare in mind that events are generated by a managed system, and SPF must accept them as they are and as many as they are. Therefore, a filtering mechanism integrated with

the triggering mechanism offers another flexible instrument of control over the way the management framework interfaces with a managed system.

Management Events

Policies invoke mechanisms external to the management process, which might either succeed or fail, so they typically produce two mutually exclusive sets of effects – success of failure. In SPF, if a policy succeeds no further action will be necessary. In a case of a failure, however, an exception needs to be raised in the managed system (workflow engine, more precisely) and handled by the management process. According to [Sutton 1997], there are two main models of exception handling: block-oriented (exception handling block is attached to the scope in which the exception may occur) and rule-oriented (exceptions trigger exception-handling rules). In SPF, we use a mixed approach, in which exceptions are handled in the context of policy execution (policy is seen as a small management workflow specification, which provides an implicit context) by exception-handling rules. Because exceptions in SPF have the form of events, they will be handled by means of Event Handlers.

By using the policy triggering mechanism (see Figure 4-24), it is possible to generate a flow of management events and policies (see Figure 4-25), which is automatically guided by both state transitions in the managed system and the management events, and terminated when management goals are achieved.

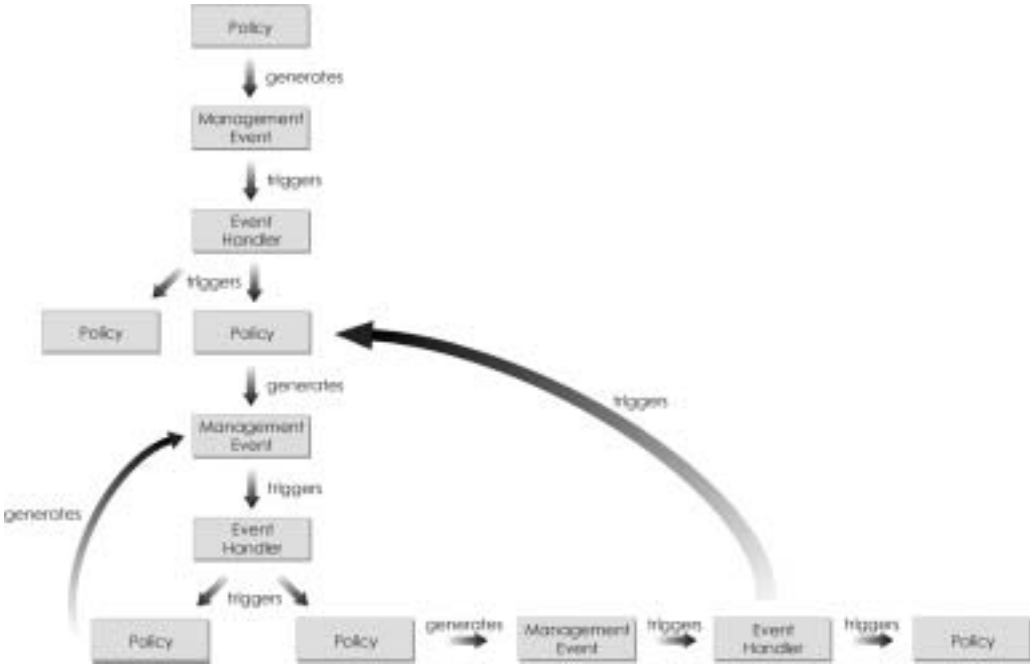


Figure 4-25 Flow of policies and Management Events in the SPF management process

Enabling automation of an action associated with events is the key aspect to move to a proactive mode of operation [OMG 1995]. The management events are control messages that are exchanged between a system and the management framework, causing policy instantiation, interpretation and triggering. The use of management events and Events Handlers dynamically generates ad-hoc management processes,

with sequences of actions that aim at achieving management goals. A simple sequence is too confining to handle a dynamic decision-making most management processes require, so the flow of activities in the SPF management process can follow a variety of patterns (see Figure 4-25):

- *Choice*, where an evaluation of policy post conditions is made and the next policy is chosen based on the result.
- *Split*, where multiple policies are spawned from one policy (via management events) and the child policies run in parallel.
- *Fault handler*, a sub-process launched to compensate for errors detected by the process.
- *Loop*, where a sequence of policies repeats for a number of iterations or until some condition is met.

4.2.2.2 Semantic and conflict analysis

Semantic analysis and conflicts handling are considered to be an optional step of the SPF management process. However, the possibility to handle conflict significantly enriches the expressive power of a policy specification language and sophistication of the management model in general, by enabling specification of policies that are logically non-monotonic. This basically means that, in certain situations (conflicts), one policy can override one or more other policies - the authority of policies is not absolute.

In SPF, the process of policy interpretation doesn't change policy semantics or policy attributes. Since there is no need to interpret policies that will be not executed, semantic and conflict analysis is performed at run-time, just before the refinement procedure for selected policies starts (Figure 4-21). Semantic rules use the same syntax as management policies, and they may be stored together in the Policy Repository.

The semantic rules may be seen as metapolicies. The idea of using meta-rules for conflict resolution is not new. However, in order to have an effective solution, the meta-rules must be well integrated into the overall management framework and they should be able to refer to a rich set of policy semantics. In Ponder, for example, meta-policies have only a limited set of information available, and mostly rely on the analysis of the domain expressions and the names of policy actions (see [Damianou 2001]). A meta-rule language was also proposed in [Jagadish 1998] for the purpose of management of rule interactions, which can be applied to conflict detection and resolution. Such meta-rules consider policies as "black boxes", and relations created by the meta-rules represent static dependencies. In contrast, the semantic rules in SPL examine policy semantics in order to detect conflicts and they do not create predefined relations among particular policies. Policies in SPF are atomic, independent specifications, and semantic rules may create only temporary relations among triggered policies, mainly as a result of particular system states. Semantic rules allow dynamic, run-time analysis of policy semantics. They provide a powerful and expressive mechanism for general control of policy semantics and make this control explicit and formal.

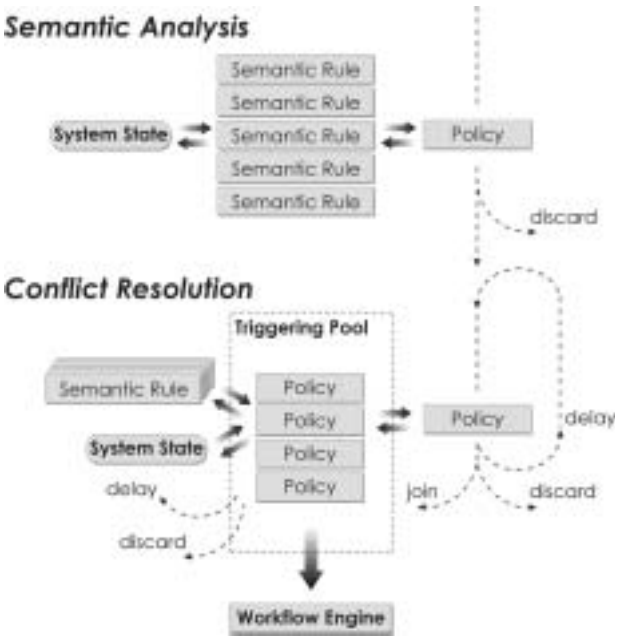


Figure 4-26 The concept of semantic analysis and conflict resolution in SPF

In SPF, the concept of semantic analysis and conflict handling was envisaged as a two-step process (Figure 4-27), based on the policy metadata and attributes. Each of these steps is implemented as a multilevel filtering mechanism, where policy metadata and attributes are checked against a series of semantic rules with regard to the current system state. Policies qualify for triggering only if they satisfy all the semantic rules (see Figure 4-26).

In the process of semantic analysis, a new policy, whose preconditions are satisfied, will be first checked against the current state of the system using semantic rules. In contrast to policy preconditions, which focus on specific policy-related aspects of the system state, these are general rules that apply to all policies before triggering. If a policy satisfies the general semantic rules for the current state of the system, it will be checked against the policies already scheduled for triggering. This is the conflict resolution process. In order to be possible to perform conflict detection and resolution, the process of policy triggering must be periodical ("stop-and-think") instead of continuous. A form of a policy cache must be established, which will contain a pool of policies scheduled for triggering. Such a pool, called the Triggering Pool (see 4.1.5.3), and the current state of the system will define a context for conflict detection and resolution by means of semantic rules. Collection (pool) of policies in the Triggering Pool defines a management response to the current system state. The pool will be flushed and its content triggered periodically. Every policy to join the pool must be checked for conflicts against the policies from the pool, and a policy may join the pool only if the semantic rules for conflict resolution are satisfied for the current system state. At the same time, policies already being in the pool may be replaced with the new, more favourable ones.

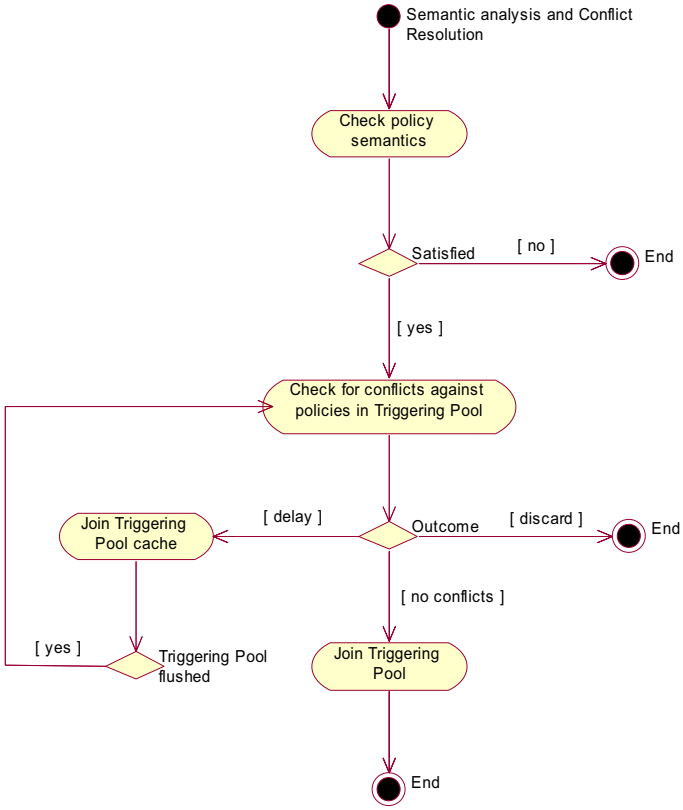


Figure 4-27 The process of semantic analysis and conflict resolution

A policy may replace one or more policies already in the pool, depending on the resolution strategy specified in the semantic rules. If the decision was made to delay the policy, it will be moved to the Triggering Pool cache to wait for the next triggering cycle. Once all the policies from the Triggering Pool are executed (Triggering Pool is flushed), policies from the cache will repeat the conflict resolution process, one by one.

The semantic rules may inspect every partition of a policy specification using the "dot" notation. Whatever might be the exact method of such an analysis (it is largely implementation dependent), the formality of SPL language enables a detailed analysis of the entire policy specification (see Appendix D for examples).

The content of a policy may be investigated in a following way: If there is a policy action that has the subject "Web Cache", do something (pseudo-code).

```
for(i = 0 to end)
        if( Policy.THEN.Action(i).Subject.name = "Web Cache")
            do something
```

The scope of the semantic and conflict analysis in SPF is dynamically defined by the semantic rules. By default, the scope encompasses the whole managed system, but the predicates of the semantic rules may

be used to narrow it to a particular organizational and/or functional scope. The organizational scope can be defined in terms of policy attributes (for example Subject, Target, or other attributes used to specify management domains), while the functional scope can be defined in terms of policy semantics (class, goal, course of action, etc) or in terms of the semantics of particular policy actions (see 7.2.3.4).

As an example of a general semantic rule, we may look at the following logic for management control.

In busy hours, we want to focus only on essential management, so we may specify the following rule. The rule will access every policy scheduled for triggering, check its Metadata, and discard it if it has a low priority.

IF
    AND,
    (GetTime() = busyInterval,
    Policy.Metadata.Class = "Accounting",
    Policy.Metadata.Priority < 4)
THEN
    Discard(Policy)
WITH
METADATA
    Description: "In busy hours, don't perform accounting management unless it is of the top priority",
    Class: Semantic analysis,
    Goal: Management optimization,
    Action: Elimination,
    Priority: 1

Another problem that may be addressed using the semantic rules is the policy life-cycle management. For example, a user may create ValidFrom and/or ValidUntil policy attributes as a part of the policy Accounting information, and use it for an explicit policy life-time management. The attribute values may change dynamically, as required, so there will be no need for removing policies from the management system when their validity is only temporarily suspended (the validity may be extended later on).

A semantic rule may be specified as follows: Discard all the policies that did not become valid up till now, or whose validity has expired.

IF
    OR,
    (Policy.Accounting.ValidFrom > now,
    Policy.Accounting.ValidUntil < now)
THEN
    Discard(Policy)
WITH
METADATA
    Description: "Discard all the policies that did not become valid, or whose validity has expired",
    Class: Semantic analysis,
    Goal: Policy life-time management,
    Action: Elimination,
    Priority: 1

Event though SPL also supports an individual explicit enabling and disabling of policies through its mechanism of policy attributes (e.g. Policy.Accounting.Validity = TRUE/FALSE), the presented approach to versioning control based on the time component is more flexible. General semantic rules

made for the versioning control enable consistent versioning control across the management system by elimination of obsolete policies (policies automatically cease to be valid at some moment in time) and by scheduling policies for the future use (policies automatically become valid at some moment in time). More examples of the policy life-cycle management can be found in Appendix D.

Conflicts detection and resolution

Conflicts between policies come as a result of inconsistencies among their management activities, so particular management policies should be validated in the context of the overall management process. According to [Moffett 1994], the major distinction is between conflict of modalities and semantic conflicts (conflict of goals). Conflicts of modalities can be recognized without reference to the meaning of the policy goal, whereas semantic conflicts depend upon the semantics of the goal, or are application-dependent.

Modality conflicts are inconsistencies in policy specifications that may arise when two or more policies with modalities of opposite sign refer to the same subjects, actions and/or targets (see Appendix D for examples). There are three types of modality conflicts:

- $O+/O-$, when the subjects are both required and required not to perform the same actions on the target objects.
- $A+/A-$, when the subjects are both authorized and forbidden to perform the actions on the target objects.
- $O+/A-$, when the subjects are required but forbidden to perform the actions on the target objects (obligation does not imply authorization).

Application-specific (semantic) conflicts refer to the consistency between what is contained in policies (subjects, targets and actions involved) and the state of a system (for example limited resources). These conflicts cannot be determined directly from the policy specifications, since additional information is needed to specify the conditions which result in conflicts. According to [Moffett 1994], there can be several types of application-specific conflicts. For example, conflict of resources occurs when the amount of resources (target objects) available is limited. Policies obliging and authorizing managers to use these resources must therefore have a limited number of objects in their target scope. Multiple management may occur when different managers are allowed to manage same objects. This may constitute a conflict when the management operations to be performed on the target object are not independent. Another potentially problematic scenario would be the self management, when a manager may not be allowed to retract policies that he/she is supposed to perform.

Automation of the detection and resolution of policy conflicts will be essential for effective automated distributed system management [Moffett 1994]. As automated managers cannot enforce conflicting policies, a precedence relationship must be established between the policies in order to resolve the conflicts [Lupu 1999]. Conflicts between management policies can be detected statically (at specification time) or at run-time. Precedence based on modality resolves all the conflicts in a deterministic way, which is not flexible. There can be several principles for establishing this precedence (see Appendix D for examples):

- *Negative policies always have priority*. It is quite common for negative authorization policies to always override positive ones so that a forbidden action will never be permitted.
- *Assigning explicit priorities*. Meaningful priorities are notoriously difficult assign, though.
- *Distance between a policy and the managed objects*. In the concept of calculating the distance between a rule (policy) and the objects it refers to, the distance between the policy and the (class of) objects to which it applies indicates the relevance of the policy to those objects. Priority is given to the policy applying to the closer class in the inheritance hierarchy.
- *Specificity related to domain nesting*. A more specific policy (policy applying to a sub-domain) refers to fewer objects so overrides more general policies applying to an ancestor domain. This concept is a particular case of the previous concept of distance.

According to [Moffett 1994], conflict analysis must be both formal, because then it will be possible to treat it rationally by creating simulation models or by reasoning in a formal logic, and generic, because all kinds of policy have to fit into a compatible framework if conflicts between them are to be discussed. The limitation of static analysis is that it may not be possible to evaluate policy constraints, as they depend on run-time state. In contrast to most of the existing solutions, which tend to define static (predefined) precedence rules, SPF semantic rules enable dynamic conflict detection and resolution in order to implement a formal and generic conflict resolution mechanism.

Consistency is not a property of a formal system per se, but depends on the interpretation which is proposed for it [Hofstadter 1999]. In SPF, such an interpretation context is defined by the semantic rules and the current state of the system. Semantic rules analyse the current state of a system and the goal structure and action semantics of policies in the Triggering Pool to determine temporal precedence relationships for policy triggering. The extent of such conflict resolution concept depends mainly on the quality of the metadata used – the extent by which metadata is able to capture the semantics of policies. SPL syntax enables creation of an extensive set of semantic data to accompany management policies in a form of metadata and policy attributes. In addition to policy semantics explicitly specified as metadata, policy attributes may be used to specify Subject, Target, Modality, etc. in order to support a detailed semantic analysis and conflict resolution (see section 5.1.2. for more details). Examples of semantic and conflict analysis in SPL may be found in Appendix D.

Once we establish the fact that triggered policies are consistent with the overall management goals, the management process may continue with policy interpretation.

4.2.2.3 Policy interpretation

The policy interpretation process, which includes instantiation and refinement, translates policies from storage (abstract) to an operational (executable) format. In the process of policy interpretation, static policy structure is merged with its dynamic content from Dictionaries in order to produce valid and executable policies. The essence of the policy interpretation is shown on example in Figure 4-28. A policy specification is modular, consisting of a skeleton and references to a number of (external) building blocks from a Dictionary. This is the format in which policies are stored in the Policy Repository. The policy interpretation interprets the meaning of references, by replacing them with the actual content they are

pointing to. When a policy is triggered, it will be interpreted so that all references to Dictionary operations will be replaced with the actual operations. In this form, policy may be sent to a workflow engine for execution.

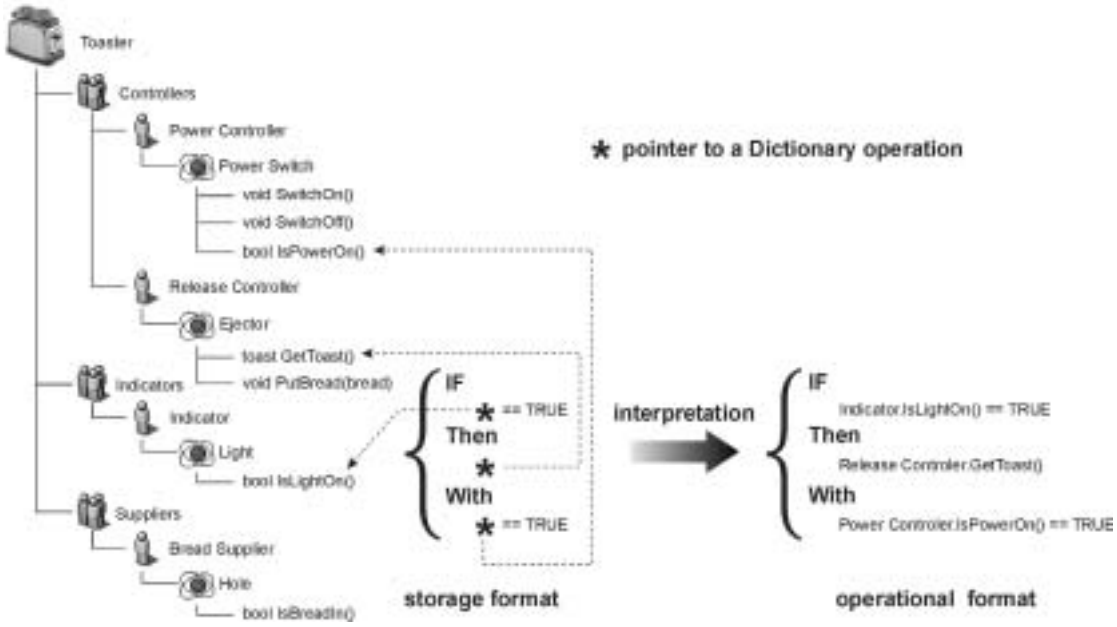


Figure 4-28 Concept of policy interpretation

With such an approach, the management model of the system (hierarchy of Dictionaries) is kept and maintained centrally, so that all changes propagate instantaneously to all existing policies. In effect, policies in the Policy Repository specify only the structured relations among Dictionary operations.

Policy instantiation

The objects grouped together in the classifications should have something of heuristic value in common; they should be "similar" in a useful sense; they should depend on relevant or essential features [Minsky 1963]. In SPL, classes group elements of the same type (Domains, Nodes, Roles or Services) that have, to a certain extent, a compatible behaviour – a collective behaviour that members will borrow to their classes. For instance, a class of printers is a generic printer that represents a collection of concrete printers, and it may be used to represent the collection. In that case, *SignalFailure()* for example may represent a common behaviour for all the members of the Printer class, and as such become part of the class behaviour.

Virtual operations for a class interface are selected among ("borrowed" from) the operations supported by the class members. A class interface defines a representative (collective) behaviour for the class, but it doesn't have to be implemented by all of its members. Some members may implement the entire class behaviour, while other may implement different parts of it (Figure 4-29). As a result, there will be a possibility for implicit creation of subclasses – a group of the class members that implements the same subset of the class interface.

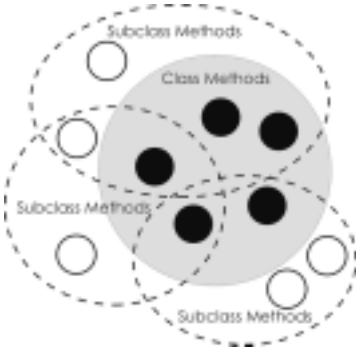


Figure 4-29 Concept of subclasses

A policy may be specified to reference a number of operations from a class interface. Such an abstract policy would be instantiated for all class members that implement all the referenced operations. That may include all members of the class, or just some of them (a subclass). Figure 4-30 represents a case in which an (abstract) operation from a class interface is implemented by three member elements of that class. A policy that references that abstract operation needs to be instantiated for all three members of the class, each with the abstract operation being replaced with the corresponding concrete operations implemented by the class members.

When a new element joins a class, abstract policies don't need to be updated. They will be always instantiated by the SPF for the current members of the class that implement required operations from the class interface. Moreover, since not all the member elements of a class need to implement all the interactions from the class interface, the concept allows a separate evolution of class members.

The actual mechanism that may be used to relate virtual operations from a class interface to appropriate concrete operations of the class members is implementation-dependent, but it may be based on two concepts:

- *Explicit links* between virtual and concrete operations. In this case, the model designer has to explicitly link virtual operations to their implementations in the class members. Names of the operations can be independent, but this complicates the model, makes it less intuitive and requires more work.
- *Implicit links* between virtual and concrete operations. In this case, the names of operations could be used to link virtual to operations to their implementations in the class members – names of the virtual and corresponding concrete operations must be the same. Having the same names for the virtual and concrete operations is a good design practice, which promotes naming consistence and easier understanding of the model. For example, PolicyModeller (see Chapter 6) implements such an approach.

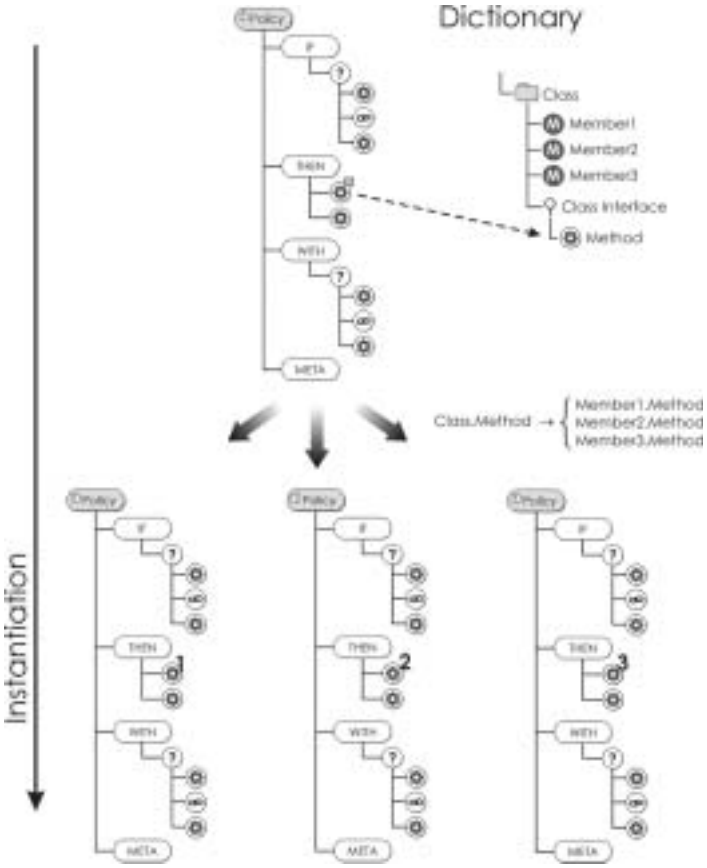


Figure 4-30 Concept of policy instantiation

In the following example, we will look at a system with three equivalent nodes, each of which is exposing a same set of management operations (Figure 4-31, on the left). We will apply abstractions of aggregation and abstraction of classification to simplify the management model of the system.

The first abstraction step in this flat model, which is a simple set of management operations, is to organize management operations into abstract nodes and to group the nodes into an abstract Domain. This is a model of the system at a medium level of abstraction (Figure 4-31 in the middle).

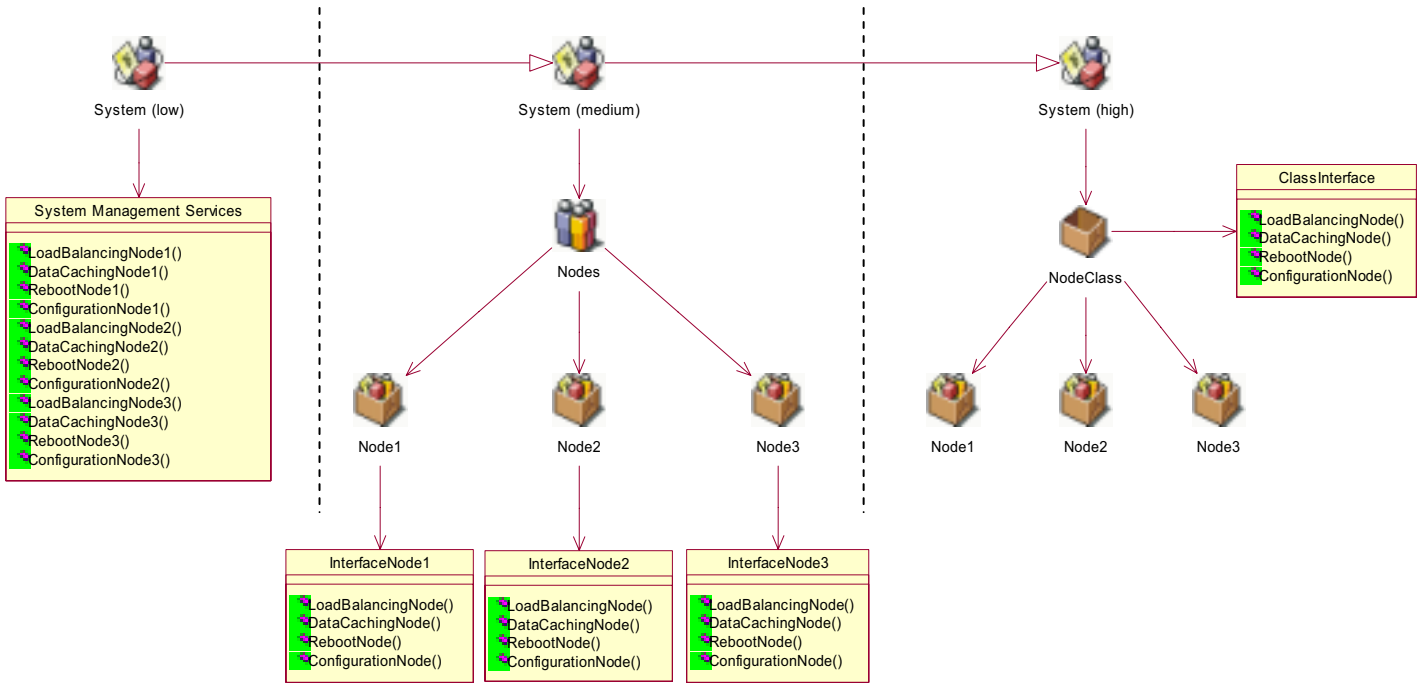


Figure 4-31 The SPF classification concept

Now we can use the abstraction of classification to create even simpler model of the system, for the purpose of node management. We will create a node class with a representative interface for the class and its members (Figure 4-31 on the right). With such an abstract model of the system, it is now possible to simplify management operations on classified nodes. For example, if we want to reboot all the nodes in the *NodeClass* we can specify a policy in terms of *RebootNode()* operation (from the *NodeClass* interface) instead of referring to individual nodes and their reboot operations:

$$\left.\begin{array}{l}\text{RebootNode1()}\\ \text{RebootNode2()}\\ \text{RebootNode3()}\end{array}\right\} \blacklozenge\!\!\lozenge \quad \text{NodeClass.RebootNode()}$$

Policy that refers to such a management operation would be an abstract policy, which needs to be instantiated for all the members of the *NodeClass* and then triggered on each and every one of them.

$$\text{AbstractPolicy(NodeClass.RebootNode())} \lozenge \left[\begin{array}{l}\text{Policy(RebootNode1())}\\ \text{Policy(RebootNode2())}\\ \text{Policy(RebootNode3())}\end{array}\right.$$

Element classification offers a possibility to implicitly specify relations of generalization/specialization among policies. Abstract (general) policies, specified for classes of elements, will be refined (specialized) into member-specific policy instances. This opens a possibility to manage class members using abstract policies, although (in the background) all those members are individually managed using the appropriate policy instances.

After the policy instantiation, we will end up with one or more abstract policies, specified in terms of (abstract) high-level management operations. Those policies will need to be refined into executable policies in terms of real management operations from the system management interface.

Policy refinement

Refinement is a process that transforms a specification at certain level of abstraction to a specification at a lower level of abstraction. Refinement adds more details to a specification either by including more constraints or by deciding among implementation alternatives. The process of moving from one level to another in the policy hierarchy can be viewed as a process of decoding pieces of fragments of policy [Maullo 1993].

According to [Damianou 2002], there are few examples of practical approaches to policy refinement. One such example is presented in [Casassa Mont 2000], as the POWER Prototype. Policies are specified in two steps, first as abstract policy templates, which are then refined (instantiated) into concrete policies. Policy templates are policies specified for classes of objects and generic operations, and policy refinement is a selection of particular class members among predefined options. The approach is suitable for the specification of lower-level policies (high-level of reuse and large number of available objects/roles), where concrete objects are manually assigned responsibilities to perform policy actions, very much like assignments of concrete resources/objects to roles. The POWER refinement concept represents a simple manual instantiation of the parameterized policies, and it is similar to the mechanism of element classification and policy instantiation in SPF, described in previous section. However, whereas in POWER a class of elements represents only a pool from which a manager can manually select an element, in SPF policy specified for a class automatically applies to all compatible class members.

In SPF, the process of policy refinement is implemented as an automated, multi-step process of behavioural specialization. Lower-level policies refine the semantics of corresponding higher-level policies by replacing high-level policy building blocks with the equivalent lower-level ones. Policy building blocks are management operations, and high-level policies refer to operations from Dictionaries at higher-levels of abstraction, whereas low-level policies refer to operations from lower-level Dictionaries. The fact that policies are stateless (they are not objects) simplifies the process of policy refinement. As a result, the policy refinement is based on operation refinement, in which higher-level operations are refined into (substituted by) equivalent series of lower-level operations. Semantically equivalent operations can replace each other in policy specifications, which enable policy refinement. Ultimately, a management view of a system has to be mapped to the real managed system, so it has to include only the functionality that is really implemented. The Dictionary at the lowest level of abstraction specifies the real management operations from the system management interface, in terms of which all the abstract operations from other Dictionaries will be refined (see Figure 4-32). As a result, refined policies will all refer to the real management operations from the system management interface – they will be executable.

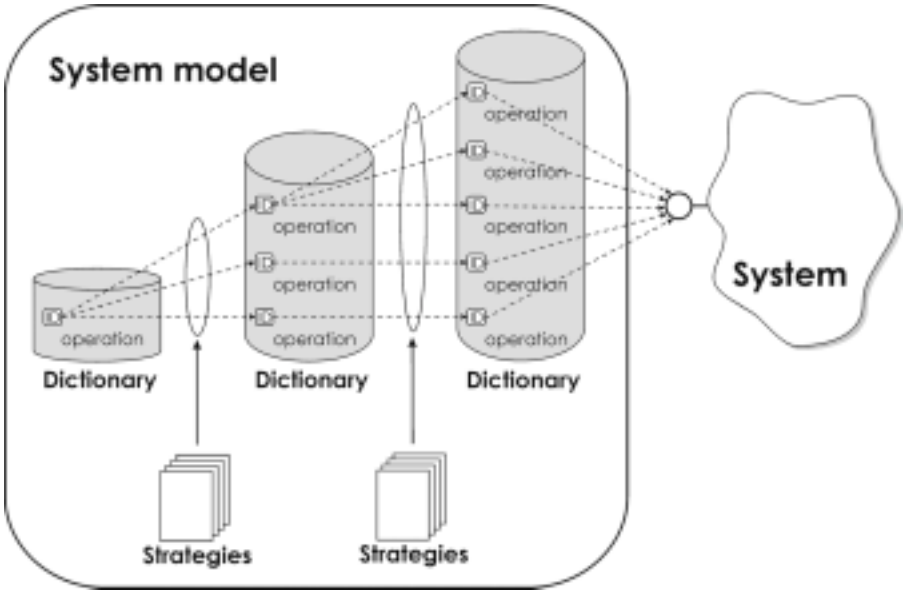


Figure 4-32 Mappings between real system management operations and abstract operations from Dictionaries

Work presented in [van Lamsweerde 1996], [van Lamsweerde 1999], [van Lamsweerde 1995], and [Darimont 1996] introduces patterns of goal refinement that allows high-level goals to be specified as a combination of lower level ones. The idea is to provide formal support for building generic goal refinement patterns that are once for all proved correct and complete by the pattern designer. They can be used for guiding the refinement process. The concept of refinement patterns was applied in SPL to specify collections of management operations that have the equivalent effect on the system state as some other, high-level management operations. Such refinement patterns are called strategies (of refinement), and they enable formal definition of behavioural specialization in SPL. Strategies explain higher-level operations in terms of lower-level ones, implicitly creating mappings between Dictionaries at different levels of abstraction (Figure 4-32).

As Wirth suggested in [Wirth 1971], every refinement step implies some design decisions, and it is important that these decision be made explicit. Explicit refinement of policy building blocks, implemented using the strategy concept, gives to managers full control over that process. A strategy simply gives to an operation it refines a definition (body), so that the relation between the operation and the strategy is practically the same as between a function declaration and a function definition in programming languages. The process of establishing relations of semantic equivalence is essentially the process of procedural abstraction, which includes aggregation, classification and/or generalization of management operations. Since management operations are only referenced from (not contained in) policies and strategies, the process of refinement essentially barely changes references in policies from operations in one Dictionary to operations from another Dictionary. Strategies are only one possible interpretation of all possible solutions. The interpretation logic may change in time, but that won't affect specified policies, only their interpretation.

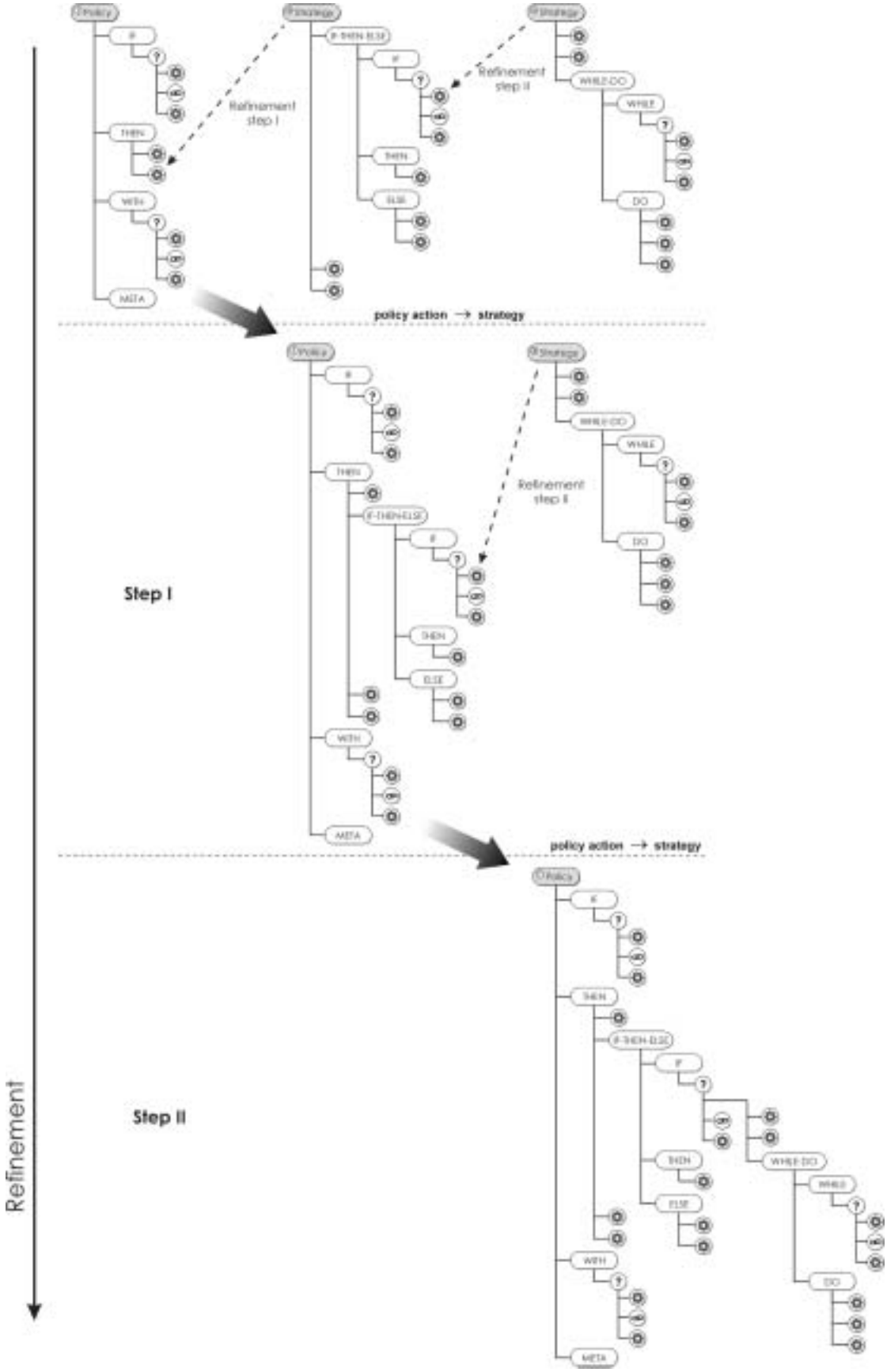


Figure 4-33 Concept of policy refinement

Referential transparency is the property of substitutivity of equals for equals [Meyer 1990]. It means that each expression denotes a single value which cannot be changed by evaluating the expression. Evaluation of the expression simply changes the form of the expression but never its value. All references to the value are therefore equivalent to the value itself. The value of an expression depends only on the values of its constituent expressions (if any) and these sub-expressions may be replaced freely by others possessing the same value.

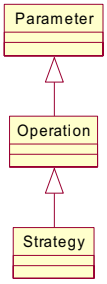


Figure 4-34 Basic relations of specialization in SPL

In SPL, referential transparency is the instrument which enables policy refinement: two equal expressions can substitute each other without affecting the meaning of a policy specification. The referential integrity was implemented in SPL by defining explicit relations of specialization between parameter, operation and strategy types (see Figure 4-34). It means that a policy operation may be replaced with an equivalent strategy body, where the equivalence means that they have the same effect on a system and they return the same value, and that a parameter may be replaced by a equivalent operation, where the equivalence means that operation returns a value same at the parameter's. Operations and strategies may be represented by the values they return, and thus may be treated like any other value.

Refinement of the *loadLimit* parameter changes the form of the expression, but not its value. *GetLoadLimit()* operation returns the value of the load limit (*loadLimit*), and the strategy performs a sequence of operations and finally returns the value of the load limit too:

```
                loadLimit
                    <
              GetLoadLimit()
                    <
⌈ capacity | CheckCapacity()
|⌈ price | Check Price()
||⌈ limit | CalculateOptimalLoad(capacity, price)
||⌈ return    limit
```

Using the presented refinement concept, SPF is able to refine policies via a multi-step procedure, which is illustrated in Figure 4-33.

When specifying a strategy, it is the task for a user to establish a relation of semantic equivalence between a high-level management operation and its refinement. The task is somehow equivalent to the task that a programmer has when writing a function definition (body) in typical programming languages. The user is

responsible for the correctness of the strategy specification. A correct strategy specification guaranties that a high-level management operation and its refinement can be used interchangeably. The SPF management process relies on the property of semantic equivalence to replace high-level management operations in policies with their equivalent refinements.

Regarding the refinement operation, the involved SPL constructs may be defined as follows (see section 5.1 for the SPL syntax):

The Parameter type may be considered to be atomic.

&Operation : Dictionary,) $Parameter_j$: Dictionary, i : 1..n
Operation ::= <$Parameter_1$, $Parameter_2$, … $Parameter_n$>

&Strategy : Policy Repository,) $Operation_j$: Dictionary, j : 1..m
Strategy ::= <$Operation_1$, $Operation_2$, … $Operation_m$>

&Policy : Policy Repository,) $Operation_j$: Dictionary, j : 1..k
Policy ::= <$Operation_1$, $Operation_2$, … $Operation_k$>

The refinement in SPL translates parameters and operations from a Dictionary at higher level of abstraction D to parameters and operations from a Dictionary at a lower level of abstraction D', and it is defined as follows:

Dictionary D, D'
Refinement f: D ⇓ D'

&x : D,) y_j : D', i : 1..n
f(x) = <y_1, y_2, …, y_n> 〉 f(x) = x

Refinements are explicitly defined by users in the form of strategies. Please note that not all the parameters and operations from D need to have a refinement.

An action is a refinement of a parameter if it explains how the parameter values are obtained from a system. The refinement establishes a relation of semantic equivalence between the parameter and an operation, which means that the parameter and the operation may be used interchangeably in the SPL specifications – they don't affect the system state and the operation's return value is identical to the parameter value:

refinement(parameter) = operation ♥ parameter ◊ operation

A strategy is a refinement of an action if it explains the modus operandi of the action and returns the same parameter value. The refinement establishes a relation of semantic equivalence between the operation and the strategy, which means that the operation and the strategy may be used interchangeably in the SPL specifications - they have the same effect on the system state and they return the same parameter value:

refinement(operation) = strategy ♥ operation ◊ strategy

User is responsible for establishing the relations of semantic equivalence among parameters, operations and strategies at different levels of abstraction. SPF will perform the policy refinement procedure based on the specified relations of semantic equivalence and the assumption that the policy refinement in SPL is distributive - it consists of the independent refinements of policy components. For example, the property of distribution for the operation 〈 may be defined as follows:

p 〈 (q 〉 r) ◊ (p 〈 q) 〉 (p 〈 r)

In the case of policy refinement, the property of distribution may be defined as follows:

refinement(Policy(x_1, x_2, …, x_n) ◊ Policy(**refinement**(x_1), **refinement**(x_2), …, **refinement**(x_n))

Strategies and compound actions may be also refined, and the property of distribution applies too:

refinement(Strategy(x_1, x_2, …, x_n) ◊ Strategy(**refinement**(x_1), **refinement**(x_2), …, **refinement**(x_n))

In summary, the property of distribution of the refinement operation in SPL is defined as follows:

Dictionary D, D'
Policy Repository P
Refinement f: D ⇓ D'

&p : P,) x_i : D, i : 1..n
p = <x_1, x_2, …, x_n> ♥ f(p) = <f(x_1), f(x_2), … , f(x_n)>

The property of distribution is based on the independence among parameters and operations form Dictionaries.

We have to prove that the following holds (policy and strategy refinement): Based on the SPL syntax and the definition of the refinement, the refinement of policies and strategies may be defined as follows:

Dictionary D, D'
Policy Repository P
Refinement f: D ⇓ D'

&p : P,) x_i : D, i : 1..n,) y_j : D', j : 1..m
p = <x_1, x_2, …, x_n> ♥ f(p) = <y_1, y_2, …, y_m> 〉 f(p) = <x_1, x_2, …, x_n>

Please note that not all the policies and strategies need to have a refinement.

Using the definition of the refinement, the previous may be expanded as follows:

Dictionary D, D'
Policy Repository P
Refinement f: D ⇓ D'

$\&p : P,) x_i : D, i : 1..n,) y_j : D', j : 1..m$
$p = <x_1, x_2, \ldots, x_n> \heartsuit f(p) = <f(x_1), f(x_2), \ldots, f(x_n)> = <y_1, y_2, \ldots, y_m> \rangle f(p) = <x_1, x_2, \ldots, x_n>$

Therefore, if the user-defined strategies:

$$f(x) = <y_1, y_2, \ldots, y_n>$$

are semantically correct, the policy refinement in SPL will be semantically correct too.

Now we will look at one example of an operation refinement. Let's assume that there is a management decision to reboot all file servers under the management control. We can identify three levels of abstraction:

1. Top level of abstraction hides the fact that both software and hardware aspects of the file server functionality need to be restarted. Step I refinement will be performed with the following strategy:

Server reboot includes shutting down services, hardware restart and restoring services.

```
        void Reboot()
EQUIVALENT
        ShutDownServices(),
        Restart(),
        RestoreServices()
USING
```

This strategy means that: $Reboot() \Diamond \lceil ShutDownServices() \lceil Restart() \lceil RestoreServices()$

2. Next level of abstraction hides the fact that server restart needs to be performed both internally (server state) and externally (server connections). Step II refinement will be performed with the following strategies:

Shutting down services includes:
- ∉ Notifying clients and waiting for their response (operation returns TRUE when it is done).
- ∉ Disconnecting clients and waiting for their response, and saving the server state in the database and backing it up.

```
        void ShutDownServices()
EQUIVALENT
        WHILE
                NotifyClients() != TRUE
        DO
        WHILE
                DisconnectClients() != TRUE
        DO

        SaveState()
        BackUp()
USING
```

Restoring services includes:
- ∉ Restoring the state from the database and waiting for it to be finished.
- ∉ Restoring connections and waiting for it to be finished.

```
        void RestoreServices()
EQUIVALENT
        WHILE
                RestoreState() != TRUE
        DO
        WHILE
                RestoreConnections() != TRUE
        DO
USING
```

3. The lowest level Dictionary specifies the real management operations, which reveal that the software restart needs to be performed both internally (state) and externally (connections).

Mappings between operations at different levels of abstractions, specified by means of strategies, are indicated in Figure 4-35.

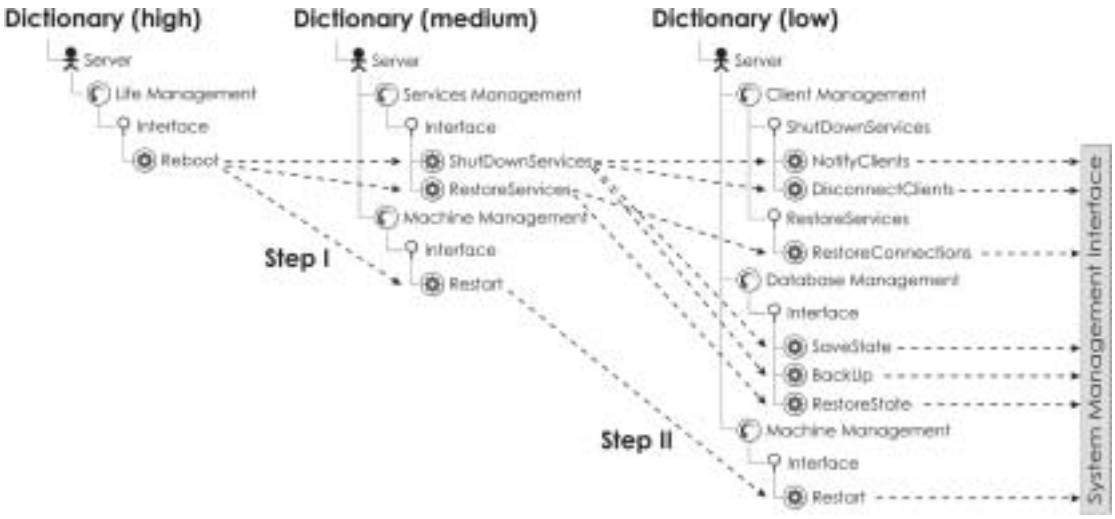


Figure 4-35 Example refinement

According to [Wies 1995], the end product of policy refinement are not policies that act directly on managed resources but rather specifications on how to apply management tools and how to utilize management functions or management services offered by a management system. It is a process of merging the results from a top-down approach (the refinement of policies) with the results from a bottom-up approach (the abstraction of available management functionality). In SPF, instead of having policies simple enough to be executed directly on the system resources, we abstract the management capabilities of the system to be powerful enough to support (execute) high-level management interactions. The level of abstraction in polices depends only on the level of abstraction and amount of details in Dictionaries (SPL vocabulary) they refer to. Dictionaries that are more abstract produce policies that are more abstract. By hiding the potential complexity of their building blocks, policies become easier to create and understand.

4.2.2.4 Policy transformation to XML, execution and deletion

Once the policy specifications that include all the information necessary for their execution are generated, SPF must transform them into a representation that will be understandable by a managed system. In the light of the latest developments in the domain of distributed systems technologies, this will require a transformation into an XML dialect.

In SPF, policy transformation to an XML dialect is a two-step parsing procedure. In the first step, SPL representation of policies will be converted to an XML implementation of SPL. Once expressed in XML, policies will be further transformed using standard XML technology (XSLT) to whatever XML dialect

managed system understands. The purpose of the two-step transformation is to decouple SPL from technology-specific information formats. The intermediate format is chosen to be XML, because it allows very simple conversion between different formats using the standard tools. For every new (workflow engine-specific) XML format, only a new XML Style Sheet will be required (for more information, see Figure 4-11).

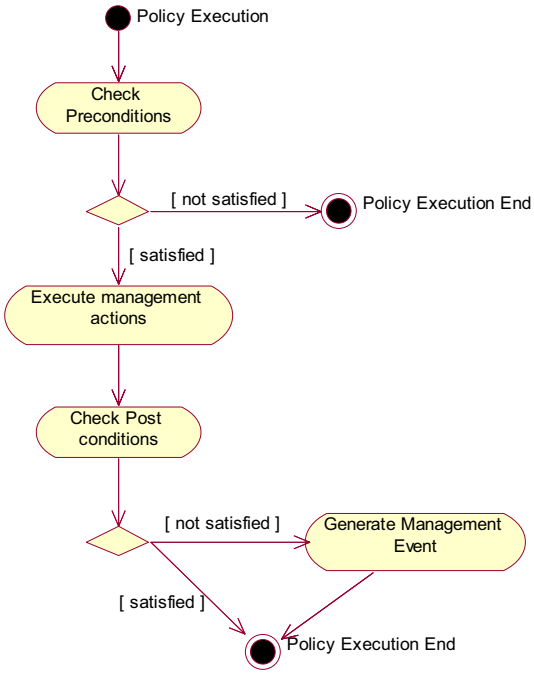


Figure 4-36 Basic steps in policy execution

Policy execution is the final step in the policy enforcement process (Figure 4-36). Policies, which are now interpreted to be executable, self-contained pieces of information, will be set to the system for execution. In a workflow engine, these pieces of XML data will be treated as small workflow definitions, parsed and then executed. A policy workflow specification includes three steps: assessment (IF part), management actions (THEN part), and monitoring (WITH part).

Workflow engine follows a simple rule to validate the outcome of management actions triggered by a policy, and in case that post conditions are not satisfied it throws a simple exception in a form of a management event in order to inform SPF that policy didn't achieve its goal.

For example, such a rule may be specified as follows (pseudo-code):

```
if(policy.WITH = FALSE)
        throw(policyID)
```

As a result, a management event will be delivered to SPF together with other system events. If there is an Event Handler in SPF defined to handle generated management event (exception), it will trigger one or more corresponding policies.

After being sent to the system for execution, all intermediate policy formats generated in the course of policy interpretation will be deleted. If a policy should need another triggering, the whole process of policy enforcement will be repeated to reflect all the changes in the management system since the last triggering.

4.3 Summary

This chapter has presented an environment in which our policy specification language was designed to operate, and whose processes and information model it was designed to support. The complexity of the policy management operations was presented at a conceptual level, together with the interactions and flow of information among various components of the management process. Because SPF is a conceptual framework, we didn't go much into details of its design. That was left as a task for its implementation. Too may details at this stage would make the framework too restrictive and inflexible, which certainly was not our goal.

In the following chapter we will present our policy specification language – SPL. Now that the context for its use is defined, we will be able to look at the details of its syntax and semantics.

5 SERVICE-BASED POLICY LANGUAGE (SPL)

The previous chapter has defined a context for the application of our policy specification language. SPL was designed to fully support the SPF management process and it has a double role: it defines a storage format (information model) and a policy specification language.

This chapter will present the language itself in more details. The expressive power of the language will be illustrated by example in Appendices C and D. Examples will cover three important aspects of the language usage: system modelling, policy specification, and control of the SPF management process.

5.1 Formal language definition

In this section we will present a formal definition of the proposed policy specification language. A formal syntax definition has three distinct uses. It:

- Names the various syntactic parts (i.e. nonterminal symbols) of the language.
- Shows which sequences of symbols are syntactically valid sentences of the language.
- Shows the syntactic structure of any sentence of the language.

The SPL grammar will be defined by specifying its syntax in EBNF (Extended Backus-Naur Form), and, additionally, using an abstract grammar specified by means of the UML Class diagrams. In general, it is easier to define language semantics on the basis of their abstract syntax. For comprehensive language descriptions, both kinds of syntax are needed [Mosses 1992]. The SPL semantics will be given in a free-text form.

Extended BNF	***Meaning***
unquoted	Non-terminal symbol
"..."	Constant value
bold	Terminal symbol
(...)	Grouping
[...]	Optional symbols
{ ... }	Symbols repeated zero or more times
::=	Defining symbol
|	Alternative
,	Concatenation
//	Single-line comment

Table 5-1 Extended BNF notation

The most important features of EBNF used in this document are summarized in Table 5-1. Although the terminal symbols should be represented quoted, they will be represented as bold text to improve the readability of the specification. The grammar syntax rules are indicated in the `Courier New` font type.

For the purpose of the SPL grammar definition, we will extend EBNF to enable definition of parameterized (generic) types. This will enable a concise definition of SPL grammar. An alternative would

be to specify separate types (identifiers) for every usage context, but such an approach would make the grammar definition unnecessary complex.

Some of the SPL types are parameterized, which was indicated in their definition:

type(parameter) ::= …, parameter, …

Parameterized types are abstract, and must be specialized before use by specifying a value for their parameters. In the formal definition of SPL, we will use the specialization to emphasize relations among language elements (types). This is similar to the concept of parameterized type in UML, whose specialization is performed by assigning a concrete value to a type parameter (Figure 5-1).

type(type) ♥ …, parameter = type, …

Parameterized types may have defined a default type for their parameters, so that explicit specialization is not mandatory:

type(parameter = type) ::= …, parameter, …

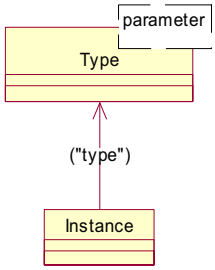


Figure 5-1 Instantiation of the parameterized type in UML

In SPL we use the dot notation, such as a.f, to refer to an attribute or an operation f provided by an abstraction a, or in other words, to the field named f of a module a. Qualified names that can be used with the dot notation are defined by the SPL abstract grammar.

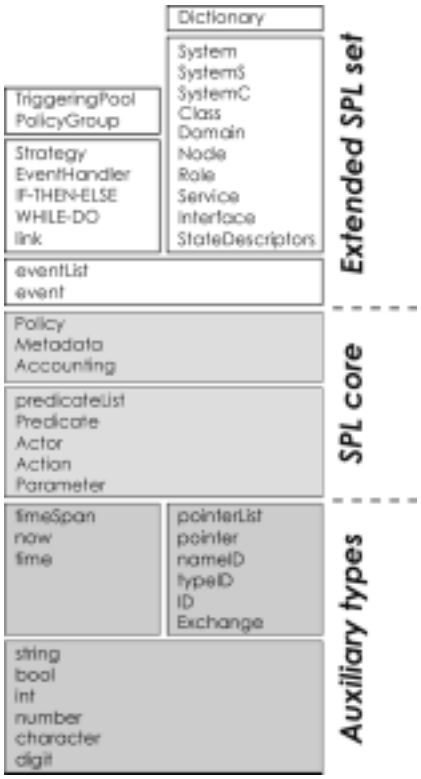


Figure 5-2 SPL types

A frequent dilemma in programming language design is the choice between a language with a rich set of notations and a small, simple core language [Cardelli 1994]. In SPL, we have tried to achieve both qualities. The language has three components (Figure 5-2):

- *Auxiliary types*, which define the essential building blocks of the language.
- *SPL core* (SPLc), which, together with the auxiliary types, represents a full-featured language for policy specification. It represents an information format for data exchange between the management framework (SPF) and a managed system. For that purpose SPLc, or it subset, can be mapped to other data representation formats: other policy specification languages, XML, PCIM, etc. One of the possible XML implementations of SPLc is a language called PolicyML, presented in section 6.4.3. Mappings of SPLc to PCIM are defined in Appendix A.
- *Extended SPL set*, which is designed to fully support SPF. It enriches the language core with a support for system modelling, policy grouping, triggering and refinement. Integration of the modelling capabilities into the language empowers the formality of policy specifications, whereas the grouping, triggering and refinement mechanisms enable its practical application.

5.1.1 Auxiliary types

```
character ::= "based on the Unicode character set"
```

```
digit::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
string ::= {character}
```

```
int ::= digit {digit}
```

```
number ::=  [- | +] int [. int]
```

```
bool ::= TRUE | FALSE
```

```
Exchange ::= [in] | [out]
```

The Exchange type we use to specify how parameter values are sent and received in interactions. It exists for compatibility with IDLs (Interface Definition Language), WSDL, and SOAP. If the value is "[in]", parameters are used in interactions to send values to the target, while "[out]" parameters are used in interactions to receive values from the target.

```
time ::= [digit digit / digit digit / digit digit digit digit]
         [/ digit digit / digit digit / digit digit]
```

The time type stores date/time in the following format:

DD/MM/YYYY/HH/MM/SS

Either parts (time and date) are optional, and if only one is specified its semantics will be determined based on the number of digits in the last part (4 for date and 2 for time). If one of the parts in a time type instance is 00 (or 0000 for the year), it refers to appropriate portion of the current date/time.

To refer to the current month, a time value should be specified as follows:

10/00/2002

Current (today's) date is specified as:

00/00/0000

Current moment in time may be also specified using the following constant value:

```
now ::= 00/00/0000/00/00/00
```

```
timeSpan ::= [time,] [+ | -]time
```

A time interval may be specified as:

- ∉ Time between two absolute moments in time (beginning, end).
- ∉ Time from a moment in time and + (to the future) or – (to the past) an offset (beginning, duration).
- ∉ Time span not related to a particular beginning, when the first part is omitted (, duration).

Working hours can be specified as:

09/00/00, +08/00/00

or as:

09/00/00, 17/00/00

Three months form now can be specified as:

now, +00/03/0000

One day before now (yesterday) may be specified using the timeSpan type as follows:

now, -01/00/0000

A time interval may be compared with another time interval or with a time value.

To check if a moment in time is before a time interval, one would write:

time < timeSpan

To check if a moment in time is within a time interval, one would write:

time = timeSpan

To compare the length of two time intervals, one would write:

timeSpan1 = timeSpan2

```
ID ::= int
```

ID uniquely identifies instances of SPL types. IDs will be automatically assigned to instances by a policy modelling tool, to ensure their uniqueness.

```
nameID ::= ID, [string]
```

The nameID extends the ID type with a name member, to create another type of a unique identifier:

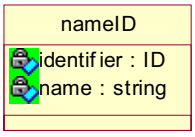


Figure 5-3 Abstract definition of the nameID type

```
typeID::= string
```

The typeID identifies a type in SPL as a type name.

```
pointer(typeID) ::= ID, typeID
```

A pointer points to another element using its ID as a reference. It uniquely identifies an element of a certain type. In SPL, the pointer type is parameterized, and it must be specialize for a particular use.

A pointer to an instance of the Role type, identified as 125, would be specified like this:

pointer("Role") = 125, "Role"

```
pointerList(typeID) ::= ID,
                        typeID,
                        (pointer(typeID) {, pointer(typeID)})
```

The pointerList type defines a list of references to instances of the same type. In SPL, the pointerList type is a parameterized type, which must be specialized for a particular use.

A list of two pointers to instances of the Parameter type would be specified like this:

pointerList("Parameter") = 456, "Parameter", (15, "Parameter", 36, "Parameter",)

5.1.2 SPL core (SPLc)

In this section, we will use previously defined auxiliary types as building blocks for the SPLc constructs.

```
Parameter ::= nameID,
              [string],
              string | number | bool | time | timeSpan | void, pointer,
              [string],
              [Exchange]
```

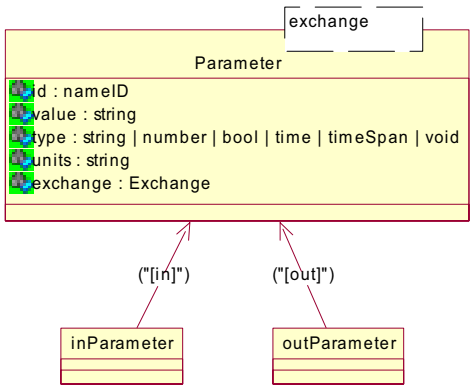


Figure 5-4 Abstract definition of the Parameter type

A parameter (identified as 92) that specifies load in terms of a number of users may be defined as follows:

92, load, 153, number, users, [in]

A parameter that holds a pointer to an instance of the Role type (identified as 133) would be specified as follows:

2, pRole, 133, number, , [out]

Before triggering, parameters receive their concrete values to complete the definition of policies. The values are supplied either by managers (proactive management scenario) or by Event Handlers (reactive management scenario).

```
Actor ::= "System" | "Domain" | "Node" | "Role" | "Class"
```

Actor is a Dictionary element that may be a subject and/or a target (service provider) of an Action. It may initiate and/or perform actions.

```
Action ::= Parameter(Exchange = "[out]"),
           [pointer("Actor") ⇓ ] [pointer("Actor" | "Action").]
           [pointerList("Parameter")],
           [string]
```

Action is a symbolic communication with a system, which is specified in the form of an abstract call to an operation from the system management interface. In SPL, the Action type extends the Parameter type – an action is a parameter that receives its value as a result of an interaction with a system. Name and ID of the Action are specified in the Parameter portion of the type definition. Although the Action type has a form of RPC, it may be actually implemented in a system as a pair of messages (SOAP messages for example). The form of a function call was chosen for actions because of its expressive power, formal syntax and familiar semantics.

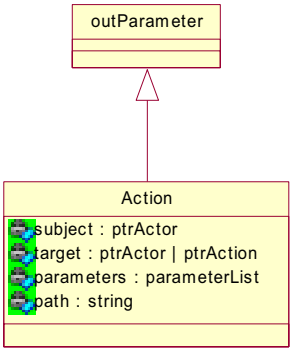


Figure 5-5 Abstract definition of the Action type

In SPL, every action must have a return type, but it can be void. As the result of a non-void action, a value will be returned. The value may be of a simple type (string, number, bool, time, timeSpan), or it may represent a pointer to an instance of a complex type.

String value will be returned as a result of the interaction initiated on Employee by the Supervisor. The userID parameter is defined elsewhere, and only referenced by the action:

string Supervisor ⇓ Employee.GetUserName(userID)

A subject (operation client) initiates a management operation on a target (operation provider). Depending on the level of details in a Dictionary, subjects and targets can be abstract elements from a system model or real entities from the system itself. More details on actions may be specified using strategies, and included in the process of policy refinement.

An action may be specified to include more or less details about an interaction. We may want to specify a subject and a target of an interaction as references to actors and other actions:

pointer("Actor") ⇓ pointer("Action" | "Actor")

Since pointers may reference other actions as well, the language offers a possibility for specifying a flow of control (delegation of authority) among actors. For example, one domain may ask another domain to ask one of his nodes to perform an operation on one of its roles:

DomainA ⇓ DomainB ⇓ NodeX ⇓ RoleY.Operation()

In this example we simply use nested pointers:

DomainA ⇓ (DomainB ⇓ (NodeX ⇓ RoleY.Operation()))

In a high-level management model, where the system internal organization is unknown from the outside, interactions between a system and its environment are simplified to be between a manager and the system, via the system management interface:

Manager ⇓ System.ManagementOperation()

In a lower-level management model, we may need to specify interactions that are internal to the system in the following way:

sourceRole ⇓ targetRole.Operation()

In the Action type, the Path element is optional, and its semantics depends of a particular SPL implementation. It may be used as:

- **Path in the system organization**. If the management model on a system, specified as a Dictionary, represents the actual system organization, a path may define the position of an operation provider in the system hierarchy.

> A path to a particular Web server in a system organization ("Site Server"), and its "File Cache" role may look like this:
>
> Servers.Web Servers.Site Server.File Cache

- **Path to a service description**. If the management operations are provided as Web services, service descriptions may be available somewhere on the Web for a workflow engine to use it. In general, a Web Service is a software application identified by a URI (Uniform Resource Identifier), which can be classified as:
  - *Locator*. URL (Uniform Resource Locator) refers to the subset of URI that identify resources via a representation of their primary access mechanism (e.g., their network "location").
  - *Name*. URN (Uniform Resource Name) refers to the subset of URI that are required to remain globally unique and persistent even when the resource ceases to exist or becomes unavailable.

> A path to a WSDL description of a Web service may be specified in a form of URL:
>
> http://www.acme.ac.uk/Services/Decsriptions/WSR-125.52.wsdl

- **Path to any external document**. In general, the path member of the Action type may be seen as an extension mechanism in SPL. Depending on a SPL implementation, various additional sources of information may accompany action specifications. Their location may be specified in the form of URIs, but that is not mandatory.

```
Predicate ::= Action,
              LeftSide
                  pointer("Action")
              < | <= | > | "= | = | !=
              RightSide
                  pointer("Action")
```

In SPL, the Predicate type is a special Action, extended with a predefined body that has an Action-Relation-Action form. It specifies an action of evaluating a logical expression, which consists of left hand side, a relation, and right hand side. The return value of a Predicate action represents the truth or falsehood of its relation (bool type). As a result, the Predicate and the bool types can be used interchangeably.

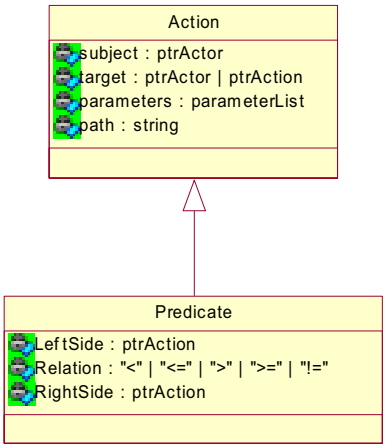


Figure 5-6 Abstract definition of the Predicate type

Load in a server may be checked against a predefined load limit in the following way:

Controller ⇓ Server.GetLoad() < loadLimit

Load in two system servers may be compared in the following way:

Controller ⇓ ServerA.GetLoad() < Controller ⇓ ServerB.GetLoad()

Just like actions can be replaced by their return values (results), predicates can be replaced by their validity (TRUE/FALSE). This enables nesting of predicate expressions during the policy interpretation. For example:

```
IsLoadLow() = TRUE              // checks the outcome of an analysis
      ↳
GetLoad() < loadLimit           // performs the analysis
      ↳
(GetLoad() < loadLimit) = TRUE  // refined expression
```

The interaction *IsLoadLow()* was refined as *GetLoad() < loadLimit*.

```
predicateList ::= Predicate,
                  AND | OR
                  pointerList("Predicate")
```

In SPL, the predicateList type is defined as a special (composite) predicate, derived from the Predicate type. Just like any predicate, each predicateList has a value as a whole – TRUE or FALSE. Members of a predicateList instance may be other predicate lists (nested predicate lists).

In the predicateList type, mode is used to specify semantics for lists of predicates. If the value is "AND" (stands for logical AND), then a predicate list represents a conjunction of predicates, so all the predicates in the list must be TRUE for the list to be TRUE. If the value is "OR" (stands for logical OR), predicate list is a disjunction of predicates, so only one predicate has to be TRUE for the list to be TRUE.

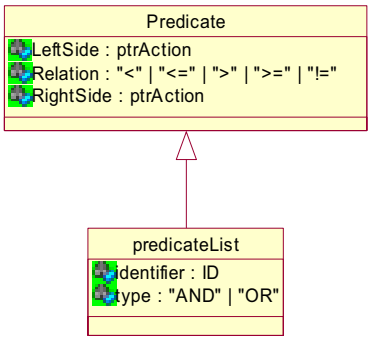


Figure 5-7 Abstract definition of the predicateList type

A predicate list may be used for specification of a complex check of a system state. The condition in the following predicate list will be satisfied only if all member predicates evaluate to TRUE:

AND
(Controller ⇓ ServerA.GetLoad() < loadLimitA,
Controller ⇓ ServerB.GetLoad() > loadLimitB,
Controller ⇓ Network.GetLoad() > loadLimitN)

```
Metadata ::= ID,
             Decsription: [string],
             Class: string,
             Goal: string,
             Action: string,
             Priority: int
```

The Metadata type contains non-functional information about policy semantics.

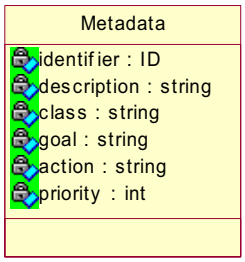


Figure 5-8 Abstract definition of the Metadata type

Policy metadata includes:

- *Natural language description* in a free textual form.
- *Class*, which specifies the semantics of the IF partition of a policy (general interest).
- *Goal*, which specifies the semantics of the WITH partition of a policy - the objective of the policy management actions.
- *Course of action*, which specifies the semantics of the THEN partition of a policy – its management actions.
- *Priority* determines precedence in case of semantic conflicts detected in a set of policies. A larger value indicates a higher priority.

ID	*Policy Class*	*Policy Goal (rationale)*	*Course of action*
1	Fault management	Bandwidth increase	None
2	Configuration management	Load balancing	Role reassignment
3	Performance management	Throughput increase	Change of state
4	Security	Network traffic reduction	Change of location
5	Accounting	Response time reduction	Change of structure
6		Cost control	Change of membership
7		New configuration	Access control
8		Secure access	Maintenance
9		State update	
10		QoS degradation avoidance	
11		Performance improvement	
12		Load redistribution	

Table 5-2 Example metadata information

In general, policy metadata is imagined to be used exclusively within SPF for semantic analysis. Interpreted policies, sent to a workflow engine for execution, usually do not require metadata information.

Metadata for a policy that performs maintenance for the purpose of load balancing may look as follows:

METADATA
Description: "If the load is high, perform load balancing",
Class: Performance management,
Goal: Load balancing
Action: Maintenance
Priority: 2

```
Accounting ::= ID,
               Author: [string],
               Created: time
               {, (string, [string,] = string)}
```

Accounting information is another non-functional part of policy specifications, which is introduced as a mechanism for extending the policy information pool. SPL requires only the basic accounting information about policies, but it can be extended to include a large number of additional policy-related attributes. The policy attributes can be used, just like Metadata, to enrich the semantics of a policy specification. SPL supports the extension of the Accounting information without any restriction.

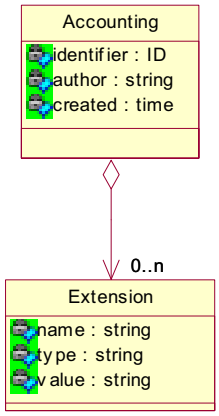


Figure 5-9 Abstract definition of the Accounting type

Accounting information may be used for maintenance and management of policies themselves.

The need to periodically trigger certain policies, for the purpose of continual control and maintenance, may be easily implemented using the policy Accounting information and a management function implemented within the Policy Repository. For such policies, Accounting information needs to include two new attributes: *nextTriggering* and *triggeringInterval*. Periodically, the Policy Repository management functions (stored procedures) would check policies and update (increment) the value of the *nextTriggering* for the value of the *triggeringInterval*. The following is a full policy specification that illustrates the concept:

IF
GetCacheSize >= upperLimit
THEN
PerformCacheMaintenance()
WITH
GetCacheSize >= lowerLimit
METADATA
Description: "Every week perform the cache maintenance to keep its size within the limits",
Class: Performace management,
Goal: Optimization,
Action: Maintenance,
Priority: 3
ACCOUNTING
Author: John Smith
Created: 08/02/1999
nextTriggering = 12/10/2003
triggeringInterval = +07/00/0000

Policy Accounting information may be also used to support versioning mechanisms, by introducing the Version attribute. Such data would allow tracking down the evolution of policy specifications in cases where it will be necessary to maintain and manage multiple policy versions. Again, semantic rules could be used to formulate and control versioning rules before the policy triggering.

```
Policy ::= nameID,
           IF
               [predicateList]
           THEN
               pointerList("Action")
           WITH
                [predicateList]
           META
                [Metadata]
           ACCOUNTING
                [Accounting]
```

Policies represent the most important units of information in SPL. They have a fixed structure that holds together policy building blocks.

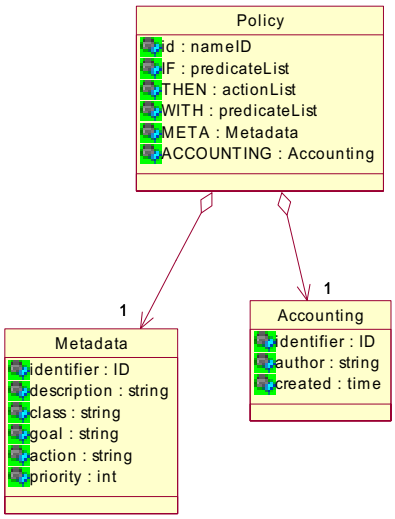


Figure 5-10 Abstract definition of the Policy type

Implicitly, the Policy type in SPL is also parameterized. Policy parameters are:

- *State Descriptors*, which essentially represent general, predefined constant values (for more information, see the definition of the State Descriptor type).
- *Action parameters*, which receive values as needed (at the triggering time).

Example of a policy specification

IF
OR
(Controller ⇓ ServerA.GetLoad() > loadLimitA,
Controller ⇓ ServerB.GetLoad() > loadLimitB)
THEN
Controller ⇓ Controller.LoadBalancing()
WITH
AND
(Controller ⇓ ServerA.GetLoad() < loadLimitA,
Controller ⇓ ServerB.GetLoad() < loadLimitB)
METADATA
Description: "Keep the load in servers under the limit",
Class: Performace management,
Goal: Optimization,
Action: Maintenance,
Priority: 3
ACCOUNTING
Author: Richard Miles
Created: 18/05/2001

5.1.3 Extended SPL set

In addition to SPLc (essential for policy specifications), the full SPL set provides language constructs to support the following mechanisms:

- System modelling, which ensures formality of SPL specifications.

- ∉ Policy refinement.
- ∉ Policy grouping.
- ∉ Event-based triggering.
- ∉ Semantic analysis and conflict resolution.

5.1.3.1 System modelling

```
stateDecsriptors ::= StateDecsriptors
                     [Parameter(Exchange = "[out]")
                          {, Parameter(Exchange = "[out]")}]
                     [, {string, [Parameter(Exchange = "[out]")
                          {, Parameter(Exchange = "[out]")}]}]
```

The real state of a system is hidden, but it can be modelled based on the system management interface (see 3.4.1.2). State Descriptors are parameters that define boundaries of an abstract state domain (important aspects of the system state from the management point of view). In policies, the actual system state is checked against those boundaries to decide if a management intervention is required.

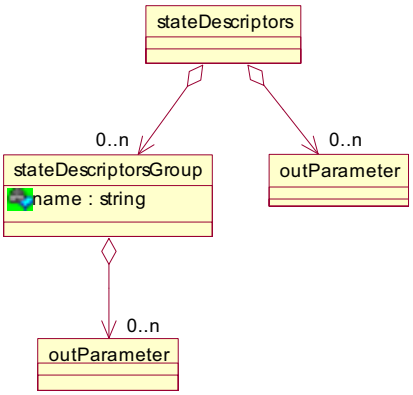


Figure 5-11 Abstract definition of the stateDescriptors type

State Descriptors can be structured in categories. Every category has a name and a list of member State Descriptors. State Descriptors can stay unclassified as well.

State Descriptors in a Dictionary may refer to some general system parameters (cache, storage size, etc.), load (load limits for various system components), response time, etc. In the following example, the actual parameter definitions are not shown for the sake of simplicity:

StateDecsriptors

```
        cacheSize,
        storageSize,
    Load,                          // group
        webServerLoadLimit,
        fileServerLoadLimit,
        applicationServerLoadLimit,
    Response Time,                 // group
        webServerResponse,
        fileServerResponse,
        applicationServerResponse
```

```
Event ::= nameID,
          [Parameter {, Parameter}]
```

The Event type specifies high-level management events, which may be generated in the management framework or within managed systems. Events may originate in various components of a model: systems, domains, nodes and/or roles. In the Dictionary on Figure 9-11 for example, events are generated by roles. Events are specified similarly to interactions, with a name and a parameter list. Event parameters, which carry additional information about the events, may be used by Event Handlers to instantiate appropriate management policies.

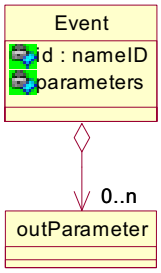


Figure 5-12 Abstract definition of the Event type

An event may be defined to carry an information about new users entering a system:

newUser(time, userID)

Abstraction elements

Terminology in the SPL abstraction model is chosen to correspond to the traditional organizational structure of distributed systems: System, Domain, Node, Role, Service, and Interface. However, the semantics of the structural building blocks can be freely chosen by users. Names only suggest semantics of structural elements, but their actual role in a model of a system depends on the model designer. For example, in a process-oriented model of a managed system, a system process may be modelled as a

System element (process is an independent functional whole) and a logical organization of its management services may include Roles, and possibly Domains.

```
Interface ::= nameID, [pointerList("Action")]
```

In a Dictionary, the Interface type represents a simple collection of operations. Interfaces are abstraction elements, used to logically organize service operations.

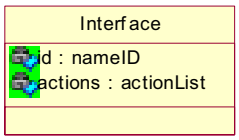


Figure 5-13 Abstract definition of the Interface type

```
Service ::= nameID, [pointerList("Interface")]
```

The Service type in SPL is defined as a collection of interfaces, each of which contains a number of management operations. It is used to specify functional part of management services. In the future, the Service type may be extended to include non-functional description of services (QoS, cost, etc.) as well, but that is not essential for policy specifications and it can be kept externally. A service has interfaces, each of which contains sets of operations, which matches the way WSDL describes Web services (see 1.4.4).

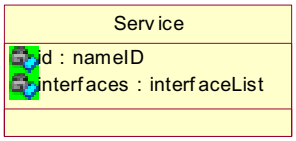


Figure 5-14 Abstract definition of the Service type

```
Role ::= nameID ,
         [pointerList("Service")],
         [eventList]
```

In SPL, the Role type is defined as an essential provider of management services. Roles offer management services, and may be abstract or correspond to the real objects and/or components that implement those services in a managed system. Roles can also generate role-level management events.

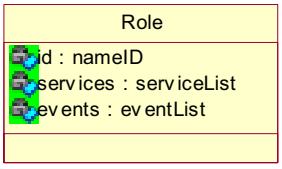


Figure 5-15 Abstract definition of the Role type

```
Node ::= nameID ,
         [pointerList("Role")],
         [pointerList("Service")],
         [eventList]
```

RM-ODP defines a node as a configuration of objects forming a single unit for the purpose of location in space. In SPL, Nodes are elements in the structure of a model, used to logically organize Roles and to provide node-level services. Nodes can also generate node-level management events. The behaviour of a Node as a whole (its interface) is defined by the Node-level management services.

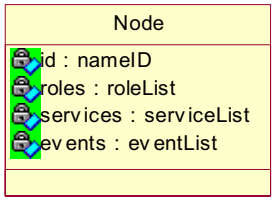


Figure 5-16 Abstract definition of the Node type

```
Domain ::= nameID,
           [pointerList("Node")],
           [pointerList("Domain")],
           [pointerList("Role")],
           [pointerList("Service")],
           [eventList]
```

Domains are elements in the structure of a model, used to logically organize nodes, roles, and to provide domain-level services. Domains can also generate domain-level management events. The SPL support for nested domains simplifies the specification of complex management models.

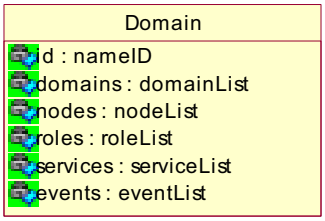


Figure 5-17 Abstract definition of the Domain type

```
SystemS ::= ID,
            [pointerList("Domain")],
            [pointerList("Role")],
            [pointerList("Service")],
            [eventList]
```

In SPL, elements of a model may be organized from two interdependent viewpoints:

- *Structured*, which models the standard hierarchical system organization by specifying relationships among system building blocks. The SystemS type was introduced for the purpose of structural system modelling, and its rules of containment are summarized in Figure 5-19.
- *Classified*, which groups system building blocks according to their type and behaviour. The SystemC type was introduced for the purpose of classification of system building blocks that are specified in SystemS instances. For more information, see the definition of the SystemC type that follows.

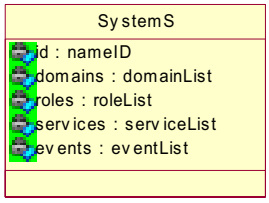


Figure 5-18 Abstract definition of the SystemS type

The SystemS type may be used to specify typical role-based models of managed systems. Those models may represent high-level abstractions of system management behaviour, or lower-level models of real system organization.

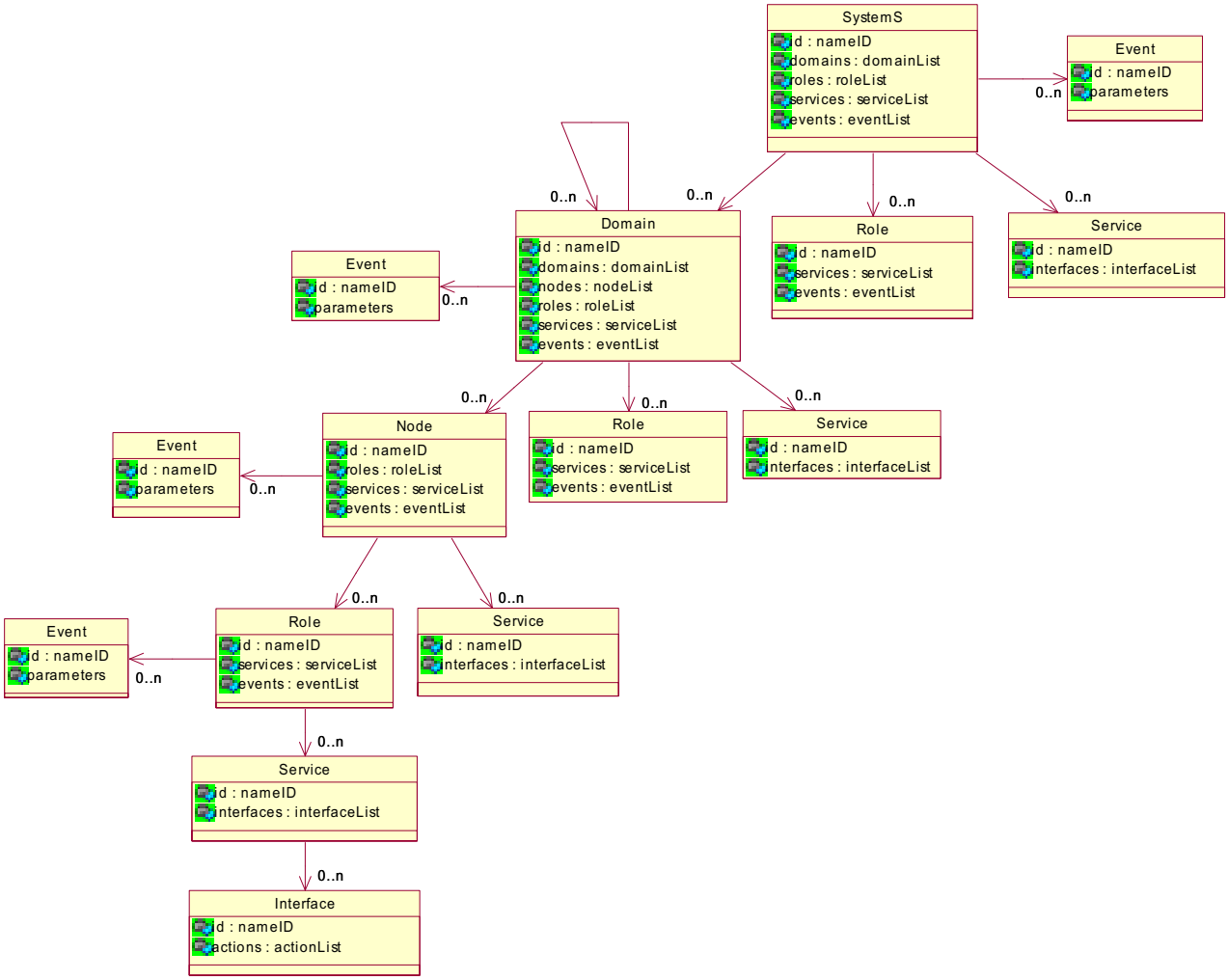


Figure 5-19 SystemS abstraction structure

```
Class(typeID) ::= nameID,
                  typeID,
                  pointerList(typeID),
                  pointer("Interface")
```

The Class type defines a collection of instances of the same type that has certain common (representative) behaviour, defined by the Class interface. Members of a class may fully or partially implement operations from the class interface. The purpose of element classes is to group subjects and targets for management operations. For example, a policy may be specified for a class of elements, refined for each of the class members (policy interpretation), and then executed on each of those class members.

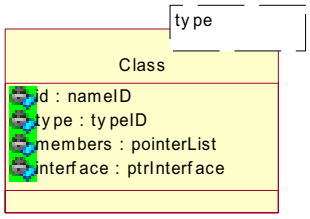


Figure 5-20 Abstract definition of the Class type

The SystemC type models the system by organizing its elements in classes. It is possible to specify classes of services, roles, nodes, domains, and events, each in its section of the SystemC type.

```
SystemC ::= ID,
            DomainClasses
                [Class("Domain") {, Class("Domain")}],
            NodeClasses
                [Class("Node") {, Class("Node")}],
            RoleClasses
                [Class("Role") {, Class("Role")}],
            ServiceClasses
                [Class("Service") {, Class("Service")}],
            EventClasses
                [Class("Event") {, Class("Event")}]
```

For example, all domains are put together under the DomainClasses section. If a number of domains have certain behaviour in common, they may become members of a domain class, which will have a name and a representative behaviour for the class defined in the form of an interface.

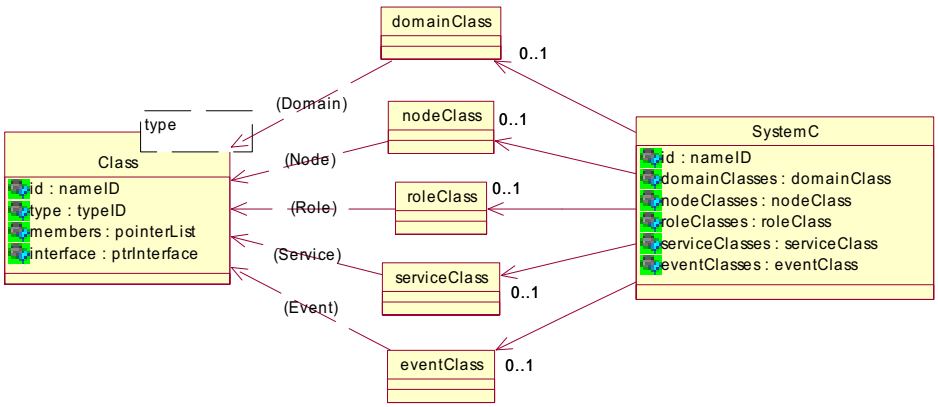


Figure 5-21 Abstract definition of the SystemC type

```
System ::= nameID,
           SystemS,
           SystemC
```

The System type includes system representation from both viewpoints, structural and classified, in order to facilitate modelling of complex systems. In general, the System type should model a self-sufficient

portion of a managed system (a subsystem). Such partitioning is very useful in case of large, complex models. However, the entire managed system may be as well represented only by one instance of the System type.

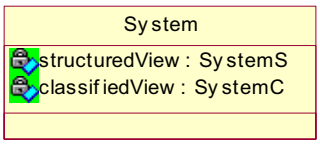


Figure 5-22 Abstract definition of the System type

```
Dictionary ::= nameID,
               time,
               string,
               [pointerList("Service")]
               [pointerList("System")],
               stateDecsriptors
```

In general, a Dictionary specifies a model of a managed system as a collection of (sub -) systems, global management services and State Descriptors. Every Dictionary also includes header information:

- *Abstract level of the model*. This is a descriptive statement about the level of abstraction used to model the system (for example low, medium, high, etc.). This information is important for relating Dictionary specification to other management models created for the same system.
- *Time component*. Dictionary specifies a management model of a system in certain moment in time. The time information represents a moment in time when the model of the system (represents the actual system) comes to existence. A succession of Dictionaries may be used to record an evolution of the system for the purpose of planning.

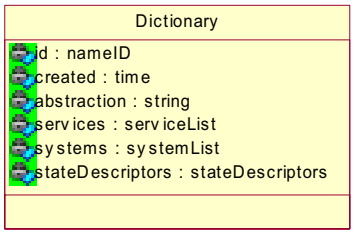


Figure 5-23 Abstract definition of the Dictionary type

Instances of the Dictionary type specify management models of a system at certain level of abstraction. The basic structure of a Dictionary is presented in Figure 5-24.

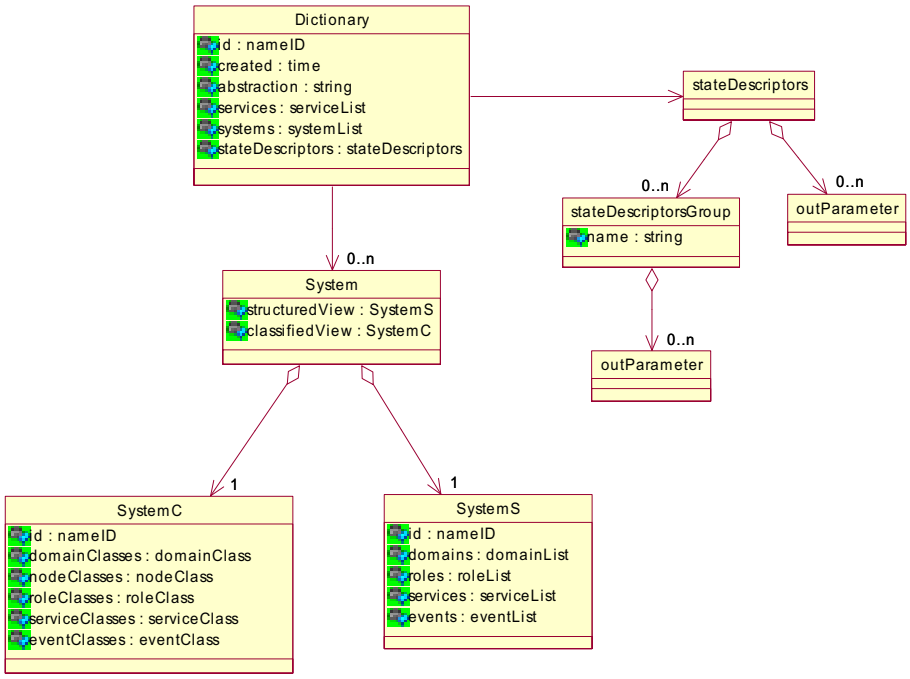


Figure 5-24 Dictionary structure

5.1.3.2 Policy refinement

```
IF-THEN-ELSE ::= Action,
                 IF
                     predicateList
                 THEN
                     pointerList("Action")
                 [ELSE
                     pointerList("Action")]
```

In SPL, IF-THEN-ELSE type is defined as a specialization of the Action type. It is used in strategies to specify alternative paths of execution (branching). Depending on the value of IF portion, THEN (for TRUE) or ELSE (for FALSE) paths may be taken.

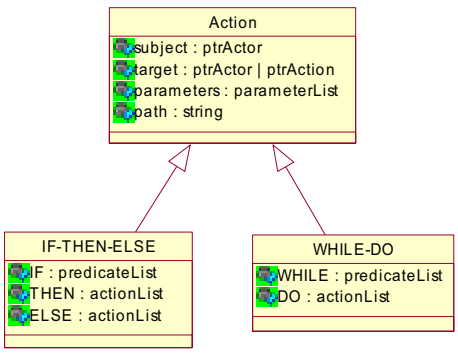


Figure 5-25 Abstract definitions of the IF-THEN-ELSE and WHILE-DO types

Course of management actions may depend on a certain state. For example, if a load in a system is low, we may want to do some maintenance, and if not we may need to perform a load balancing:

IF
GetLoad() < loadLimit
THEN
Maintenance()
ELSE
LoadBalancing()

```
WHILE-DO ::= Action,
             WHILE
                 predicateList
             DO
                 pointerList("Action")
```

In SPL, WHILE-DO type is defined as a specialization of the Action type. It is used in strategies to specify repetitive steps (looping), which will be performed as long as the WHILE portion is TRUE.

Some management actions need to be repeated until a certain state is reached. For example, we may need to perform a load balancing as long as the load in a sytem is high:

WILE
GetLoad() > loadLimit
DO
LoadBalancing()

```
link ::= pointer("Parameter") ⇓ pointerList("Parameter")
```

Link connects different parameters that share the same value. The first parameter has a value and passes it to one or more linked parameters. Links are used to indicate flow of data (parameter values) between SPL instances:

- In strategies, values are passed from actions to strategies that refine them (Action ⇓ Strategy)
- In Event Handlers, values are passed from events to policies that handle them (Event ⇓ Policy).

Parameter loadA passes its value to parameters loadB and loadC:

loadA ⇓ (loadB, loadC)

```
Strategy ::= Action,
             pointer("Action"),
             EQUIVALENT
                 [pointerList("Action")]
             USING
                 [pointerList("link")]
```

According to [Bergin 1994], a procedural abstraction names a segment of code so that it can be manipulated by giving its name. In SPL, we use the concept of procedural abstractions to create and use strategies (specifications of refinement logic). Strategies define transformation rules, which refine policies from one abstract level to another (lower) level, by refining their management operations.

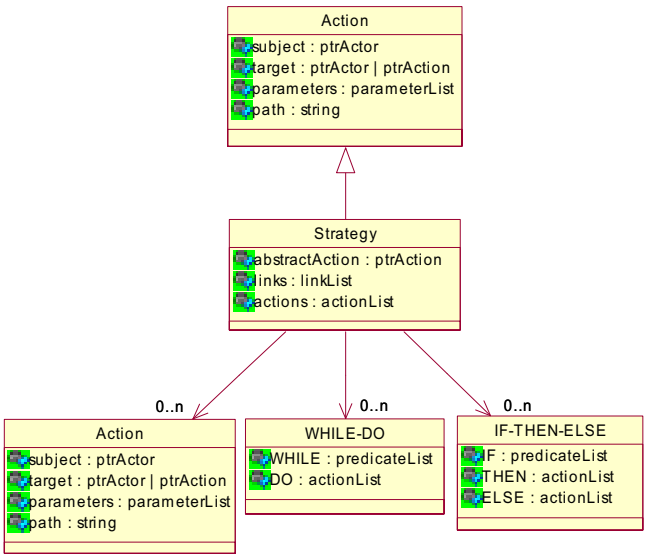


Figure 5-26 Abstract definition of the Strategy type

The Strategy type extends the Action type (it is a compound Action). It returns the same type and value as the action it refines, so an action and its refinement strategy can be used interchangeable. Effectively, strategy may refine an action or a parameter. If it refines an action, strategy provides details on included low-level actions, similarly to the definition of a function body in programming languages. If it is a refinement of a parameter, strategy defines how the parameter value can be obtained.

Discharging a server involves relocating its clients to other servers, redirecting traffic to other nodes, and rescheduling maintenance of the server cache. Such a refinement may be specified as follows:

DischargeServer(targetLoad)
EQUIVALENT
RelocateClients(newLoadLevel),
RedirectTraffic(),
RescheduleCacheMaintenance()
USING
targetLoad ⇓ newLoadLevel // link defines a value passing

How strategies relate to policies and actions may be seen on Figure 5-27.

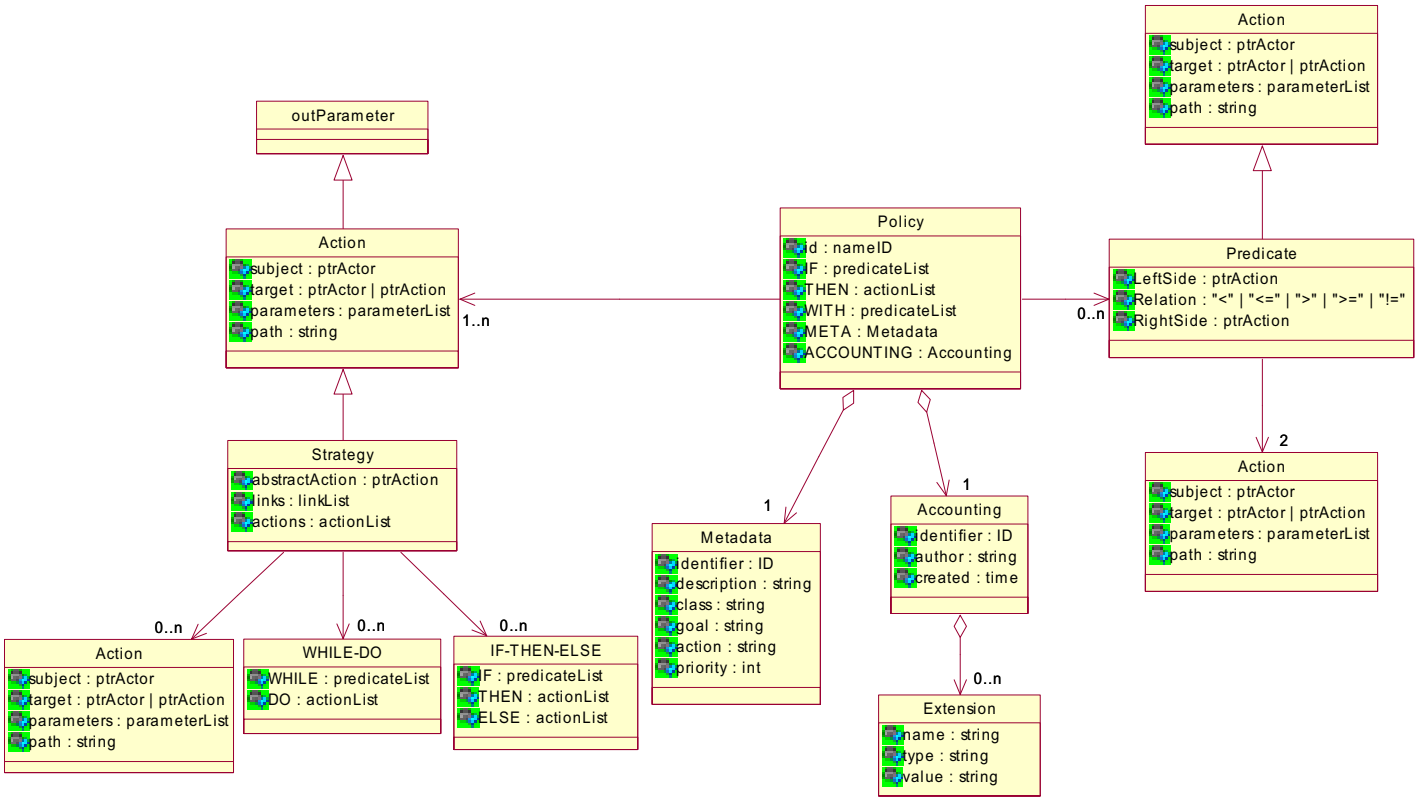


Figure 5-27 Policies and strategies

By extending the Action type, Strategy supports recursive composition. This, on the other hand, enables multi step policy refinement: strategy actions may be further refined using other strategies, until the required level of details is reached.

5.1.3.3 Policy grouping

```
PolicyGroup ::= nameID, [predicateList]
```

A policy grouping mechanism is introduced for the purpose of policy management. This is particularly useful for low-level management scenarios, where the number of policies may be significant. The grouping is dynamic, based on a specified criterion. Essentially, a policy group is an abstract query against the Policy Repository that defines a selection criterion. Policies are not explicitly listed as group members, but are dynamically selected at run-time, as needed. As a result, there is no need for expensive maintenance of policy groups and their memberships.

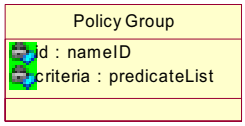


Figure 5-28 Abstract definition of the PolicyGroup type

For example, we may want to select for triggering all the most important load balancing policies. This can be specified as the following policy group:

AND
(Metadata.Goal = "Load balancing",
Metadata.Priority = 5)

A group of policies may be triggered in the same fashion as an individual policy. After a group is triggered, its member policies will be considered independently in the subsequent process of semantic analysis, conflict resolution and interpretation. Group membership doesn't prevent individual usage of a policy. Moreover, because of the dynamic grouping mechanism, a policy may be member of different policy groups.

5.1.3.4 Event-based triggering

```
eventList ::= ordered | unordered | OR,
              pointerList("Event")
```

The eventList type is an essential component of Event Handlers, used to define a scenario in which one or more policies shall be triggered. The order specifies semantics for a list of events in the following way:

- *Ordered*: The order of events in the list is significant. Only if events occur in the specified order, an Event Handler will trigger appropriate policy(ies).
- *Unordered*: The order of events in the list is irrelevant, but all the events from the list must occur in order to trigger appropriate policy(ies).
- *OR*: The list specifies a group of alternative events. Any of the events from the list will trigger appropriate policy(ies).

For example, a list of events may define a scenario that indicates possible illegal actions on a system database, carried out by a user:

ordered,
(newUser(time, userID),
databaseAccess(time, userID),
securityAlert(time))

```
EventHandler ::= nameID,
                 ON
                     eventList,
                 IF
                     [predicateList]
                 TRIGGER
                     pointerList("Policy"),
                     [pointerList("PolicyGroup")],
                 USING
                     [pointerList("link")]
```

In SPL, Event Handlers perform two functions:

- *Filtering*: Before triggering of appropriate policies, Event Handlers may check the values of event parameters and the state of a managed system. In some cases, we may want to ignore certain events, and a filtering mechanism may help preventing costly policy instantiation and triggering.
- *Triggering*: The essential use of Event Handlers is to link events and management policies. Event-free policies are independent of the triggering context, so they may be reused in different management scenarios. At the same time, Event Handlers may trigger multiple policies and policy groups, and may be changed without affecting referenced policies.

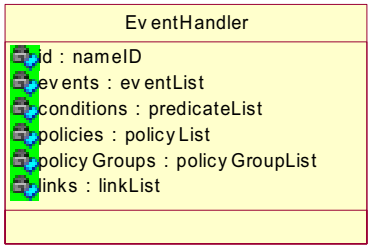


Figure 5-29 Abstract definition of the EventHandler type

In the following example, an Event Handler shall trigger a policy (Handle failure) only if the severity of a problem is high and the load in a system is high:

ON
failure(severity)
IF
AND
(severity > 2,
IsLoadHigh() = TRUE)
TRIGGER
Handle failure(severity)
USING

5.1.3.5 Semantic analysis and conflict handling

```
TriggeringPool ::= pointerList("Policy")
```

In SPL, the TriggeringPool type implements a policy cache, where policies scheduled for triggering are temporarily stored. Policies will be periodically triggered and then flushed from the TriggeringPool. The

reason for caching policies before triggering is to enable their semantic analysis and conflict resolution. The semantic analysis and conflict resolution will be performed against the content of the TriggeringPool (see 4.2.2.2 for more details).

In SPF, the policy management process implements two general operations for working with policies in the process of semantic analysis and conflict resolution:

- *Discard()* removes selected policy from the triggering cycle.
- *Delay()* schedules selected policy for the next triggering cycle. A new cycle begins immediately after the current triggering cycle is over (TriggeringPool will be flushed).

In a semantic rule, *TriggeringPool.Policy* refers to any policy within the TriggeringPool, while simple *Policy* refers to a new policy waiting to join the TriggeringPool.

The following policy specifies a conflict resolution rule, with the following logic: In one triggering cycle, policies of the Fault Management class must have the same course of action; otherwise the one with the lowest priority will be discarded.

IF
    AND,
    (Policy.Metadata.Class = "Fault management",
    TriggeringPool.Policy.Metadata.Class = "Fault management",
    Policy.Metadata.Action != TriggeringPool.Policy.Metadata.Action)
THEN
    Discard(min(Policy.Metadata.Priority, TriggeringPool.Policy.Metadata.Priority))
WITH
METADATA
    Description: "If two policies of the Fault Management class don't have the same course of action, discard the one with the lower priority",
    Class: Conflict Resolution,
    Goal: Management consistence,
    Action: Elimination,
    Priority: 1

The Accounting information enables specification of any number of policy attributes. This could include semantic of policy constraints (obligation, permission, prohibition, etc.), modality (positive, negative) and other semantic information. This additional semantic information may be used in the conflict resolution process, together with the policy Metadata.

We could specify the following semantic rule that performs modality checks:

IF
AND,
(Policy.Metadata.Class = TriggeringPool.Policy.Metadata.Class,
Policy.Metadata.Action = TriggeringPool.Policy.Metadata.Action,
Policy.Accounting.Modality != TriggeringPool.Policy.Accounting.Modality)
THEN
Discard(min(Policy.Metadata.Priority, TriggeringPool.Policy.Metadata.Priority))
WITH
METADATA
Description: "If two policies of the same class and course of action don't have the same modality, discard the one with the lower priority",
Class: Conflict Resolution,
Goal: Management consistence,
Action: Elimination,
Priority: 1

5.2 Summary

At the end of this chapter, we briefly summarize the essential SPL characteristics:

- *Family origins*: Declarative, transaction-based specification language.
- *Libraries*: Dictionaries correspond to the concept of libraries in the traditional programming languages. They define vocabulary of SPL.
- *Structure*: SPL is a modular language. Language constructs consist of a skeleton and a number of referenced building blocks, which are defined elsewhere. The presence of multiple namespaces (Dictionaries) reduces the central naming authority, which results in a simple, flexible model that promotes reusability and assures consistency.
- *Name resolution*: Every Dictionary defines a separate namespace. Mappings between namespaces are defined as strategies. Policies and strategies are main specification bodies, which refer to external building blocks, specified in Dictionaries.
- *Type system*: Strong static type system is enforced in the language.
- *Safety*: There are not user-defined types in the language, so there are no type safety problems. User is only able to assemble language constructs from predefined building blocks. Validity of policy specifications is guaranteed by the validity of the SPL vocabulary, defined by Dictionaries.
- *Data abstraction*: Data abstraction in SPL is used for hiding a potential complexity of a managed system, which would otherwise be reflected in the language vocabulary. A management model may include multiple viewpoints, each at different level of abstraction. The language supports abstractions of aggregation/decomposition, classification/instantiation, and generalization/specialization.
- *Dynamic semantics*: The basic IF-THEN-WITH policy form is expanded, during the process of policy refinement, into potentially complex specification of management actions, by replacing policy building blocks (management operations) with corresponding refinement bodies (token substitution).
- *Tools*: A modelling tool, in a form of an Integrated Development Environment, has been provided to support SPL policy modelling. The tool, called PolicyModeller, will be presented in the following chapter.
- *Precision*: SPL is formally defined in EBNF (tokens, syntax, and type system). Management operations are basic semantic elements, which are given by a managed system and considered unambiguous.

Policies in terms of such methods will be unambiguous as well. The process of policy refinement interprets the semantics of high-level (abstract) operations into the real management operations. Refinement rules are explicitly specified in terms of strategies, which makes the refinement formal as well.

- *Problem domains*: SPL is a problem-oriented language, designed to be used for the specification of management activities. It targets high-level policy modelling for the purpose of large-scale distributed system management. The language was designed to be used in SPF as a policy specification language, but it is sufficiently general in nature to be used for enterprise modelling, business rules specification, and low-level policy specification. The examples in Appendix C and D demonstrate some of the SPL expressive power. A more detailed analysis of the language scope and limitations may be found in section 7.2.

6 POLICYMODELLER: A POLICY MODELLING TOOL

This chapter presents our work on an implementation project, whose goal was to investigate some practical aspects of our ideas. A policy modelling environment was designed and implemented, in order to explore possibilities for SPL visual representation and simplification (through syntax hiding), and to give some ideas about the look and feel of the language in practice.

Appropriate tool support is essential to make management practical and feasible [Damianou 2002a]. Administrators and managers need to be isolated from the details of underlying implementations and policy representations. There is a number of commercial tools on the market created to support policy-based management of distributed systems and networks, which operate mainly at a low, resource level. For a brief overview of the commercial tools, see [Damianou 2002]. In this chapter we will briefly present PolicyModeller, a tool developed to support policy modelling in SPL

6.1 Design

The PolicyModeller project was envisaged as a reality check for the policy specification concept proposed by our research. The main goal of the PolicyModeller project was to check the viability of SPL in practical use. PolicyModeller should prove both that the proposed concept is feasible and that it is useful. In order to demonstrate what can be done with the language, an effort was made to implement different aspects of required functionality, but without going into an unreasonable level of detail.

6.1.1 Requirements analysis

PolicyModeller is essentially an implementation of the SPF Modelling Environment component of SPF. The principal requirements for the PolicyModeller design were based on the envisaged role of the Modelling Environment in SPF (see section 4.1.5), and on the design goals and syntax of SPL. Those requirements have been identified as follows:

- **Graphical User Interface** (GUI)
  - *Intuitive interface*, which is very important for software that imposes strict interaction rules. In PolicyModeller, the correctness of every modelling step will be immediately checked by the tool. An intuitive GUI functionality substitutes the need for in-dept knowledge of both the application logic and the SPL syntax.
  - *Visual modelling* (not coding) is in the spirit of SPL. The language is designed to be modular, and its implementation should support that feature by enabling assemblage of language constructs from previously defined building blocks. Elements from a predefined vocabulary shall be used to put together policies and strategies, just like creating sentences from words in natural languages. Such an approach enables active guidance of users, on-the-fly syntax checks, and discrete error reporting with explanations.
- **Functionality**:

- *System modelling*: The tool should support structural and behavioural modelling of managed systems at different levels of abstraction, as a collection of Dictionaries, just like it was imagined by the SPL/SPF design (see 4.1.2).
- *Constraints modelling*, which includes using the elements from Dictionaries to assemble policies (constraints on the system behaviour), and using elements from Dictionaries to assemble strategies (mappings between corresponding elements from Dictionaries at different levels of abstraction).
- *Data manipulation*, which includes policy interpretation, more precisely policy refinement and instantiation, and policy transformation to an XML representation.
- *Database storage and retrieval* of all persistent information.

In the rest of this chapter will illustrate how the identified design requirements where addressed in the PolicyModeller project.

6.1.2 Use-Cases

Use case modelling defines the functional requirements of a system. The purpose of a use case is to fulfil an actor's goal in interacting with the system [Cockburn 2000]. An actor is a role that a user or another system has, and the objective of use case modelling is to identify and describe all the use cases that the actors require from a system. The use case descriptions then are used to analyse and design system architecture that realises the use cases. Use cases are especially appropriate for highly interactive (behavioural) systems involving end users [Gottesdiener 2002], which was the case in the PolicyModeller project.

The primary purpose of PolicyModeller is to serve as a policy modelling tool. Therefore, the entire functionality of the tool has been designed around its interface. Most of the tool functionality focuses on the support for its GUI operations, more precisely data visualisation and storage. The software was designed to offer the following functionality to its users:

- **System modelling**, which includes work with one or more Dictionaries:
  - *Create Dictionary*. A Dictionary is implemented as a database. The tool will be able to create a new instance of the Dictionary database, together with its data model and supported stored procedures.
  - *Display Dictionary*. The complex data stored in a Dictionary must be visually and effectively represented to a user.
  - *Edit Dictionary*. The support is required for creation and changes of various data components that constitute a Dictionary. The strict checks for correctness must be enforced as the changes are introduced.
- **Constraints modelling**, which includes work with Policy Repositories, policies and strategies:
  - Policy Repositories:
    - *Create Policy Repository*. A Policy Repository is implemented as a database. The tool will be able to create a new instance of the Policy Repository database, together with its data model and supported stored procedures. By default, there will be only one Policy Repository for a system. However, it is be possible to create multiple

Policy Repositories for the purpose of replication and/or partitioning of the management information.

- Á *Display Policy Repository*. The data stored in a Policy Repository must be visually and effectively represented to a user.
- Á *Edit Policy Repository*. User must be able to create, delete, and classify policies and strategies within a Policy Repository.

- Policies:
  - Á *Create Policy*. Policies can be created in a Policy Repository.
  - Á *Display Policy*. Individual policies will be visualised in separate displays.
  - Á *Edit Policy*. Policy components will be created, changed and/or deleted in a policy display. The strict checks for correctness must be enforced as the changes are introduced.
  - Á *Interpret Policy*. Policy interpretation includes policy instantiation, policy refinement, and policy transformation to an XML representation.
  - Á *Delete Policy*. Policies can be deleted from a Policy Repository.
- Strategies:
  - Á *Create Strategy*. Strategies will be created in a Policy Repository for operations from Dictionaries.
  - Á *Display Strategy*. Individual strategies will be visualised in separate displays.
  - Á *Edit Strategy*. Strategy components can be created, changed and/or deleted in a strategy display. The strict checks for correctness must be enforced as the changes are introduced.
  - Á *Delete Strategy*. Strategies can be deleted from a Policy Repository.

A scenario is a set of input and output data or control flows and behaviours of a system brought about by an instance of an event. Use case scenarios for the envisaged functionality are represented in an UML Use-Case diagram in Figure 6-1. "Create Database", "Open Database", "Edit Database", and "Edit Policy Repository" are abstract use cases, used for generalization purposes only.

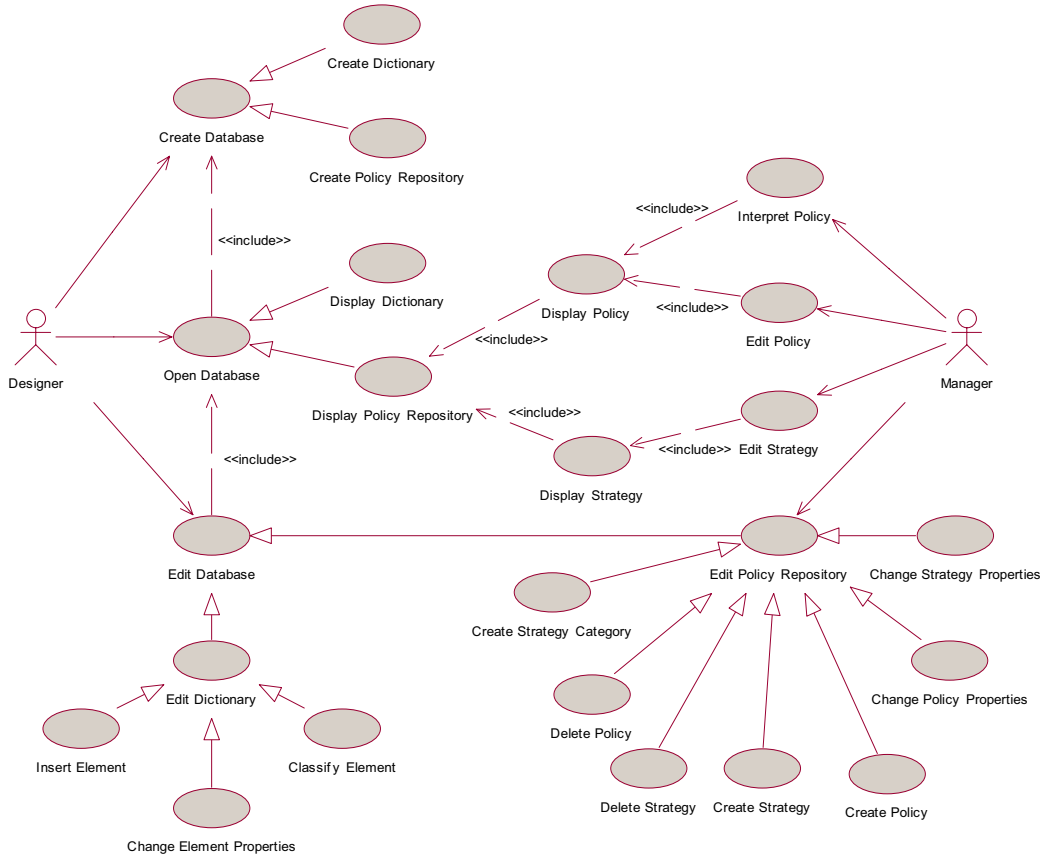


Figure 6-1 The Use Case-Model is realized by a detailed UML Use-Case diagram

It is expected that designers will work on system modelling, so that managers can focus on constraints modelling. However, such a separation of concerns in conceptual, and it will depend on particular application scenarios.

We will now present three major aspects of the implemented solution: its Graphical User Interface, its architecture, and the basic supporting functionality.

6.2 Graphical User Interface (GUI)

In this section, we will give an overview of users' perception of the tool. The PolicyModeller GUI is relatively complex, and we will not enter into a detailed presentation of all its features. Instead, we will focus on the basic elements of the GUI, their purpose and usage.

The PolicyModeller tool was imagined to be both an Integrated Development Environment (IDE) and a management portal, with Dictionary, Policy Repository, Connections, Properties and Desktop portlets (Figure 6-2). Portals are commonly used to provide people with access to information and applications in a condensed form. PolicyModeller was designed to work with information from various Policy

Repositories and Dictionaries, but it may be extended to work with other types of repositories in order to support access control and security. Moreover, the interface may be easily extended to include real-time monitoring of various aspects of a system performance.

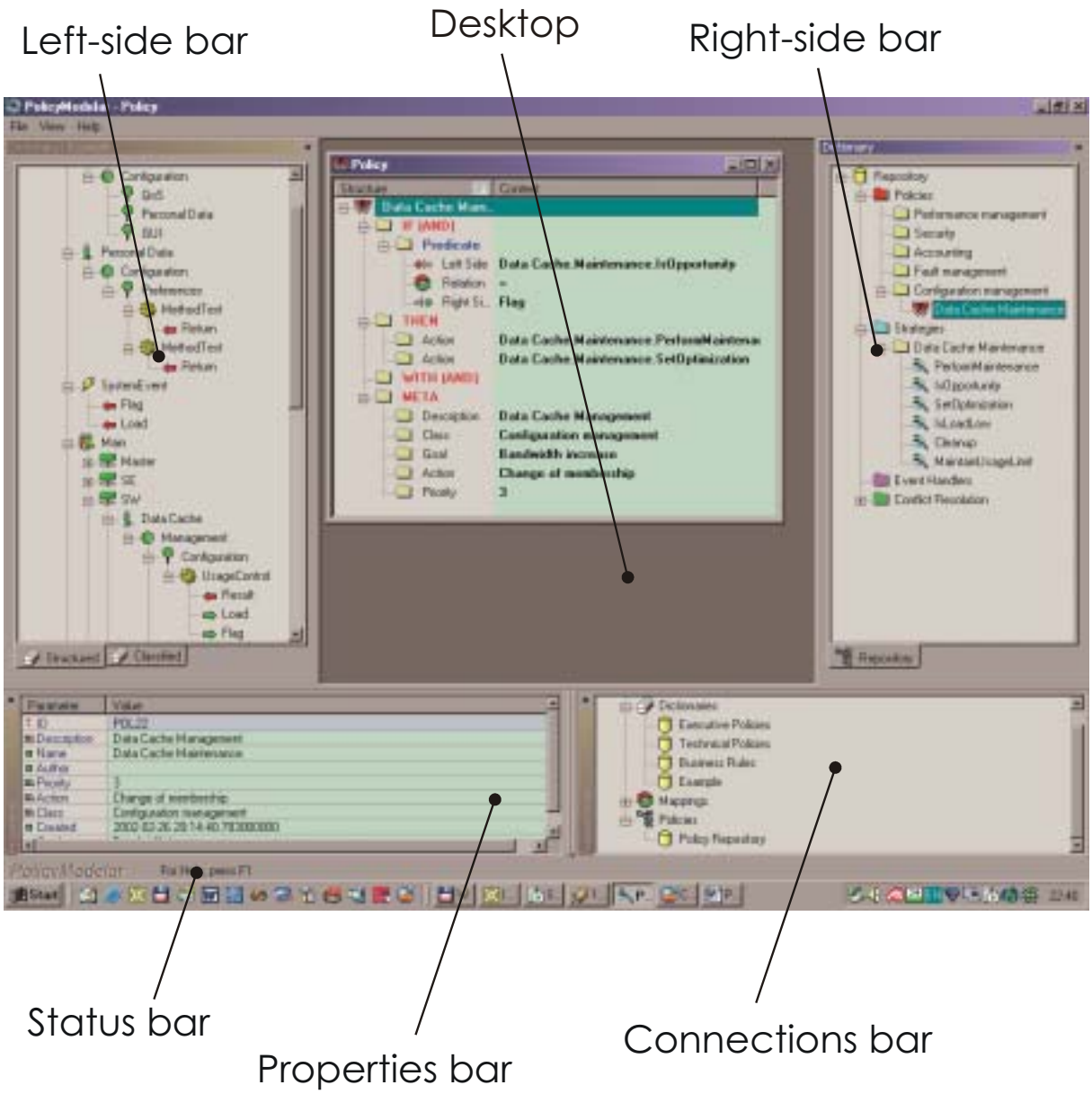


Figure 6-2 Components of the PolicyModeller GUI

PolicyModeller GUI design was based on the use cases identified in the previous section, and it was envisaged as a collection of six working areas (Figure 6-2): Left-side bar, Right-side bar, Properties bar, Connections bar, Desktop, and Status bar.

6.2.1 Left and right-side bars

Left and right-side bars are designed to host Dictionary and Policy Repository views. Dictionary and Policy Repository views are essentially database editors. They abstract and visualize the content of corresponding databases, and enable high-level data manipulation.

6.2.1.1 Dictionary view

In PolicyModeller, the content of a Dictionary is visualized from two different viewpoints:

- *Structural*. Dictionary modelling starts in the Structured View, which represents the principal visual representation of Dictionary elements.
- *Classified*. The Classified View is an optional view, which was designed to make the work with Dictionaries easier.

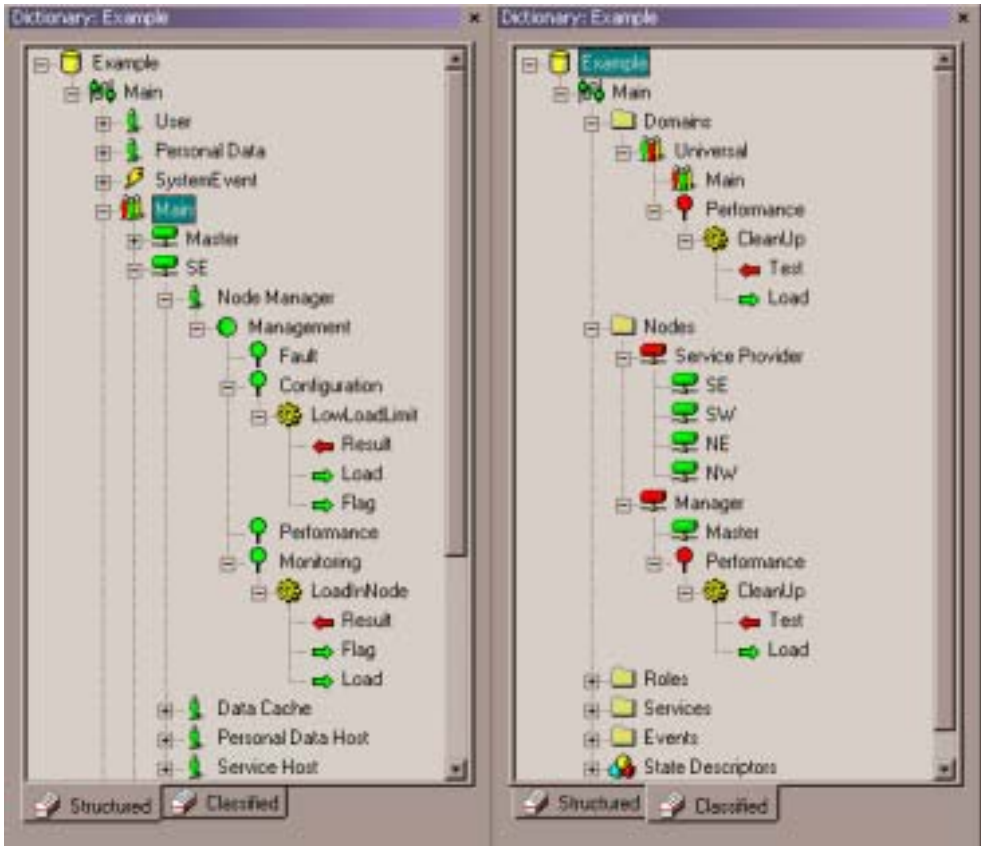


Figure 6-3 Structured (left) and classified (right) views of a Dictionary, side by side

Both the structured and the classified viewpoints are simultaneously displayed in the Right and/or Left-side bars. A Dictionary view is implemented as a tabbed pane, which may be either displaying the Structured or the Classified view. The user may toggle between the two using the tabs at the bottom of the display (Figure 6-3).

Structured view

The Structured view, as its name suggests, displays the structure of a Dictionary as a hierarchical organization of its elements. The structure of a Dictionary represented in the Structured view normally reflects the organization of a system at a certain level of abstraction. Dictionary modelling is mainly focused on management services, and their interfaces and interactions. However, when the structure of a system becomes complex, a possibility to organize Dictionary elements in logical groups becomes more important. For that reason, there is a possibility to build a structure of Systems, Domains, Nodes and Roles, and use them to logically organize basic Dictionary elements according to different criteria.

The basic functionality found in the Structured View of a Dictionary encompasses the following aspects:

- *Creating elements*: Dictionary elements can be inserted in a Dictionary hierarchy, with the respect to the containment rules (SPL syntax). Elements are created with default properties (see 6.4.1.1), which can be changed later using the Properties bar.
- *Moving elements*: It is possible to move (reposition) Domains, Nodes, Roles, Services and Events (together with their adjoin branches) in a Dictionary structure, with the respect to the containment rules. Such changes don't affect the validity of existing policies and strategies.
- *Copy/Paste elements*: It is possible to copy Dictionary elements (together with their adjoin branches), with the respect to the containment rules. The Copy/Paste operation may on may not include management operations (behaviour) from a branch. If a user wants to copy only the structure of a branch, without its behaviour, he can paste the branch without its operations. After that, a new behaviour (operations) may be specified for such branches. Alternatively, if a user wants an exact copy of the selected branch, he will paste it with the adjoin operations.
- *Insert duplicate*: This works similar to the Copy/Paste operation. A duplicate of the branch that belongs to a selected parent element will be inserted into the same parent element, with or without the adjoin operations.
- *Deleting elements*: Once created and published, Dictionary elements cannot be deleted in order to preserve referential integrity of the model.
- *Editing elements*: Instances of Interaction, Parameter and State Descriptor types in a model are inserted with the default properties and set to be "unpublished". This state is only provisional (it is not saved in a Dictionary database), and it exists only to enable changes of default element properties to the required ones. If a State Descriptor is unpublished, it is possible to change its properties. While an operation stays unpublished, it is possible to change its signature (return parameter and parameter list), and properties of all included parameters. Management operations (implemented by the Action type) and State Descriptors may be published only when they become fully and correctly specified. Once published, those elements cannot be changed (apart for names), because it could affect the existing policies and strategies (referential integrity).

Classified view

As the number of Dictionary element increases, a possibility to group similar elements into classes becomes more important. The Classified view is designed to enable abstractions of classification and generalization for Dictionary elements created in the Structured view.

The Classified view is organized in the following sections: Domain classes, Node classes, Role classes, Service classes, and Event classes. In each of these sections user can create classes of a corresponding type, and select their members among the available elements. Dictionary elements can stay unclassified as well. Classes of elements are defined by their names and class interfaces.

For every class of elements it is possible to define its class interface – a representative behaviour for the class (generalization). Interactions of such an interface can be selected among interactions provided by the class members. Every abstract interaction from a class interface must be implemented by one or more class members. For more details on classification, see section 4.2.2.3.

New Dictionary elements, created in the Structured view, will initially appear in the unclassified section for their types (for example, a new domain will appear under the Domains section). Only those elements created by Copy/Pate or Duplicate operations will be automatically classified to belong to the same class as the corresponding original element. Unclassified elements may be dragged into a class of choice. If a required class don't exist, it can be created within the section. Already classified elements may change their classes in the same way.

6.2.1.2 Policy Repository view

The Policy Repository view is designed as a visual editor for Policy Repository databases. The main purpose of the Policy Repository view is management of policies and strategies, more precisely creating, classifying, and deleting.

In a Policy Repository, policies are classified based on their Class, which is specified in the policy metadata. A new policy must be initially created in one of the existing classes (there cannot be unclassified policies). At the same time, new policy classes may be defined as required. Policies can be moved (dragged) among the classes, and their metadata information will be updated automatically.

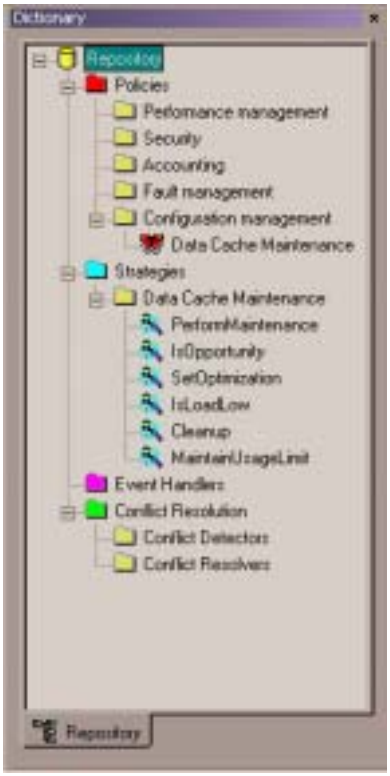


Figure 6-4 Policy Repository view

In contrast to policies, strategies don't have associated metadata, and their classification in user-defined categories is optional. Strategies are built for particular Dictionary interactions, to be their implementations (refinements). A new strategy is created after a Dictionary interaction, dragged from a Dictionary view, is dropped in the Strategies section of a Policy Repository (Figure 6-4). The selected interaction is the one we want to implement (refine) using the strategy. Similar to policies, strategies can be also moved (Drag/Drop) among categories. Strategies can be in two basic states: ready and not ready, each of them with its own icon for distinction. A strategy is ready to be used for policy refinement only when it is fully defined (for more details, see 4.2.1.2).

6.2.2 Properties bar

Properties of various model elements from different views can be viewed and, some of them, changed in the Properties bar. Every element of a model has its unique ID, which is automatically assigned to it by the PolicyModeller, and it cannot be changed. When an element is selected in a display, its properties will be retrieved from the appropriate database (based on its ID) and displayed in the Properties bar (Figure 6-8)

Property values in the Properties bar can be set either by typing a free-form text (edit box), or by selecting among predefined options (combo box). A date can be selected using a separate time/date control.

6.2.3 Connections bar

The Connections bar was designed for management of connections with databases. User can open existing Policy Repositories and Dictionaries, listed in the Connections bar, either by Drag/Drop to the Desktop or by double mouse click. Also, there is a possibility to create new Dictionary and Policy Repository databases using context-sensitive menus that open on the right mouse click.

In a more sophisticated implementation of the PolicyModeller, the user's ability to manage its connections would be limited by its access permissions. Such mechanism of access permissions was not implemented in PolicyModeller because it is not related to the goals of the implementation project.

6.2.4 Desktop

The Desktop is the central working area of the PolicyModeller GUI, designed exclusively to host policy and strategy views. In this section we will look more closely at the capacity for visual representation and modelling of policies and strategies in Policy and Strategy views respectively.

6.2.4.1 Policy view

The Policy view was designed for visualisation, modelling and enforcement of policies, stored in a Policy Repository. By using context-sensitive menus, opened on right mouse clicks, user can create predicates (in IF and WHILE parts of a policy) and actions (in THEN part of a policy). Once predicates and actions are created, they can be used as placeholders for policy building blocks – interactions and State Descriptors from Dictionaries. User can Drag/Drop interactions and State Descriptors from a Dictionary view into the placeholders (see Figure 4-20). Interactions that return "void" type cannot be dropped into Left or Right Side of predicates, because both sides of a predicate must have the same (non-void) type.

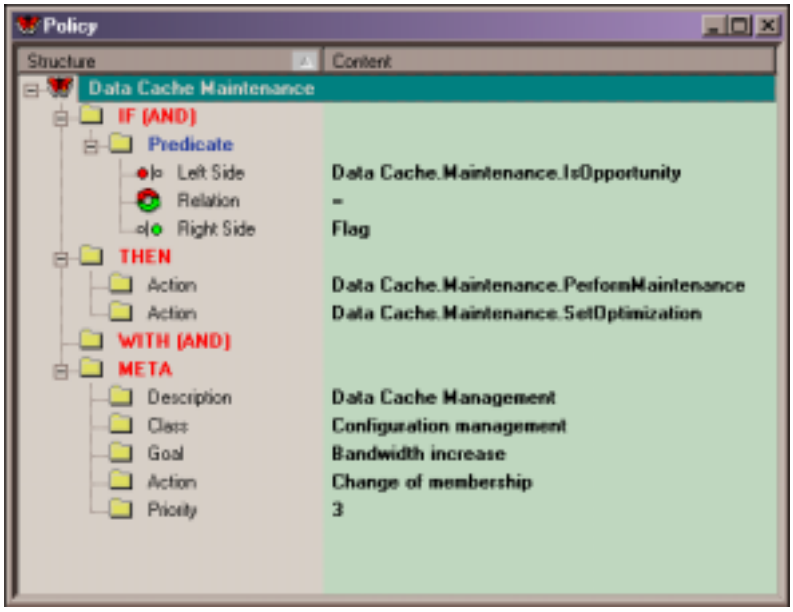


Figure 6-5 Policy view

Metadata for a policy may be set using the Properties bar:

- *Description* is a textual description of a policy in a free format.
- *Class* can be selected among existing classes, or a new policy class may be created by typing its name. Each policy must be initially created as a member of one of the existing policy classes. This can be later changed either in the Properties bar or by dragging the policy between classes in the Policy Repository view.
- *Goal* can be selected among existing policy goals, or a new goal may be created by typing its name.
- *Action* can be selected among existing courses of action, or a new one may be created by typing its name.
- *Priority* is a numeric definition of a policy priority.

A policy can be enforced using a command from the right click menu, opened on the root of the policy tree. In this implementation of PolicyModealr, enforcement of a policy will result in its instantiation, refinement and transformation into an XML representation (defined by the PolicyML, see 6.4.3). After the processing is finished, such an XML policy representation will be displayed in the (default) Web browser. In a complete implementation of the SPF, such XML policy would be sent to the Policy Parser for further transformation into the workflow engine compatible format (see 4.1.5).

6.2.4.2 Strategy view

The Strategy view is designed for visualisation and modelling of strategies from a Policy Repository. Strategy view is more complex that the Policy view because, in general, a strategy may have more complex structure than a policy.

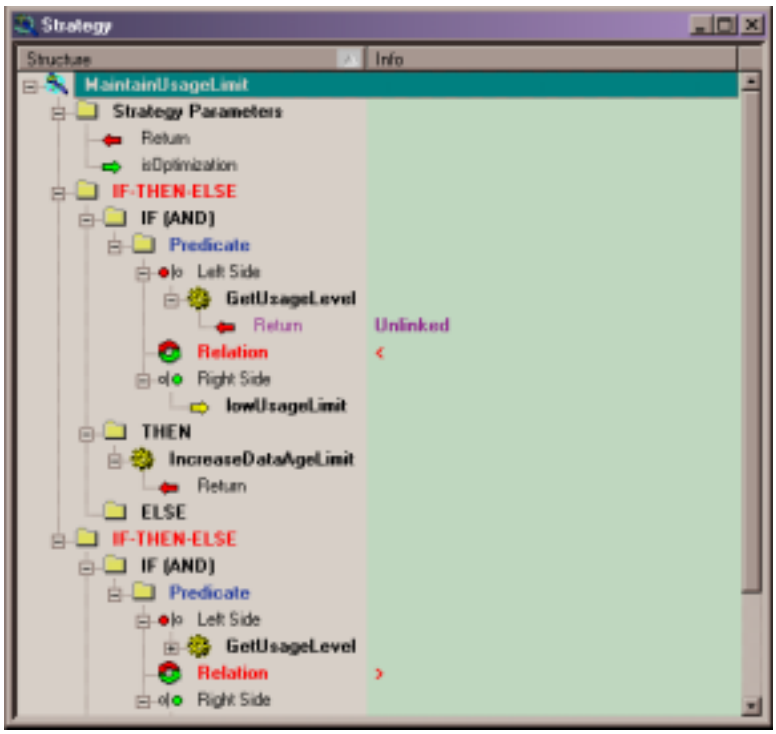


Figure 6-6 Strategy view

Just like policies, strategy components must be first created, using the context sensitive menus (right mouse clicks in the strategy body). In such a way Actions, IF-THEN-ELSE and WHILE-DO elements may be inserted into strategies (see Figure 4-17). After an element was inserted, it will play the role of a placeholder for Dictionary elements (Interactions and State Descriptors).

6.2.5 Status bar

The purpose of the Status bar is to communicate messages to users about status of the application (successful completion of tasks, or some problems) and current users' actions. Since PolicyModeller strictly enforce the SPL syntax at the modelling time, it has to validate every user's action immediately after it happens. Incorrect users' actions will be ignored by the tool, which may be frustrating for a new, inexperienced user. In order to help users to correctly use the tool and the SPL syntax, PolicyModeller provides error reporting and explanation. Since frequent popping-up of dialog boxes would be annoying, messages in the Status bar seams to be more discreet way of informing users about the incorrect actions, explaining the errors and suggesting a correct course of action.

6.2.6 Basic GUI operations

The GUI functionality is designed to revolve around three basic interaction mechanisms: commands, which are issued using context-sensitive right-click menus, Drag/Drop, which enables sharing of model components by assembling new model constructs from previously defined building blocks, and setting properties, which enables setting the attribute values for different model constructs using the Properties bar.

Context-sensitive menus, which open on the right mouse click, are very convenient way of performing actions on selected elements within the GUI. The menu commands simply show what is possible to do with a selected element. The availability of the menu commands depends on the current state of the application and on properties of the selected element.

Drag/Drop functionality is another intuitive and easy way of working with model components, particularly in a design environment that requires working with a number of different views in parallel (Figure 6-7). Drag/Drop functionality may be used in the following way:

- Within a Dictionary view:
    - State Descriptors can be moved among different groups, for the purpose of classification.
    - Branches in a Structured view may be moved around, to restructure the organization of a model.
    - Elements in a Classified view may be moved among classes as required, for the purpose of classification.

- Dictionary view ϕ Policy view: The most important use of the Drag/Drop functionality is for assembling policies from building blocks. Dictionary elements may be dragged from Dictionary views into previously created placeholders in Policy views.
- Dictionary view ϕ Strategy view: Another very important use of the Drag/Drop is for assembling strategies from building blocks. Dictionary elements may be moved from Dictionary views into previously created placeholders in Strategy views.
- Dictionary view ϕ Policy Repository view: An interaction from a Dictionary view may be dragged to a Policy Repository view and dropped in the strategy section. This will create a new strategy construct, which may then be opened in a Strategy view and specified.
- Within a Policy Repository view: Drag/Drop may be used for moving around policies and strategies. Policies may be moved among different classes, which will automatically update the Class portion of their metadata. Strategies may also be moved among different categories, as needed.
- Policy Repository view ϕ Desktop: Dropping a policy or a strategy on the Desktop will open appropriate Policy/Strategy view and the selected policy/strategy in it. The same may be performed with a double mouse click on a selected policy or strategy.
- Connections bar ϕ Left/Right side bar: Dictionaries and Policy Repositories may be opened in left and right sides bars by dragging them from the Connection bar to a side bar. The same can be performed with a double mouse click on a target database connection.

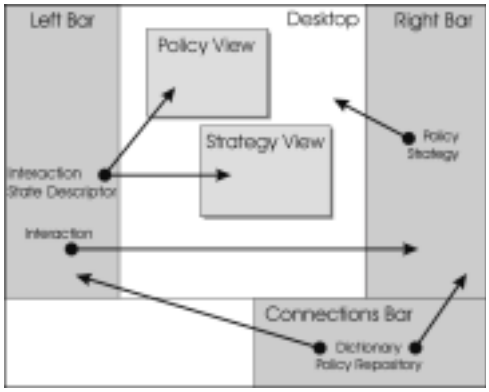


Figure 6-7 Basic Drag/Drop functionality

Attributes of various elements of a model can be directly set using the Properties bar. It is enough to select an element, and its attributes and their current values will appear in the Properties bar. Then, the values of those attributes may be simply changed either by tying the new ones, or by selecting among predefined options.

6.3 Architecture

After presenting the GUI and its basic functionality, we will look behind the interface, at the architecture of the PolicyModeller. The tool was implemented in the classic Client/Server three-tier architecture. The middle-tier (business logic focused on data transformations) is split between the Client and the Server side in order to reduce the intensity of expensive data communication between the two. As a result, the Client side is not as "tin" as it was originally planed, but such a solution is the result of a good compromise. The main task for the middle-tier in PolicyModeller is to generate structured representation

of the flat data, received from the Server side, for the purpose of visualisation on the Client side. The actual communication between the Client and the Server is handled by the DCOM RPC (a distributed object paradigm, like Java/RMI and CORBA).

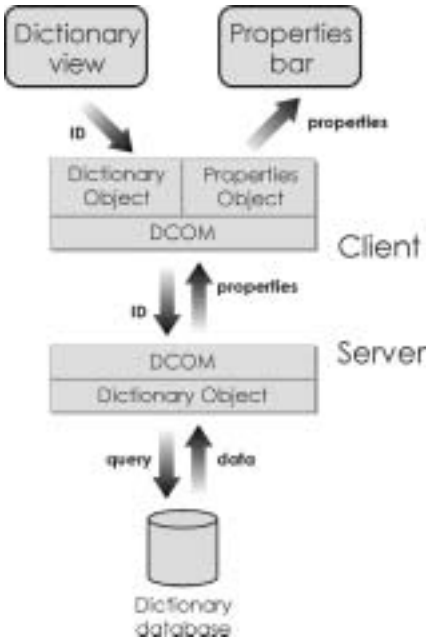


Figure 6-8 Example data flow in PolicyModeller

For example, if a user selects an element in a Dictionary view, properties for that element will be requested from the corresponding Dictionary database, returned to the Client side and displayed in the Properties bar (Figure 6-8). Every element in a model has a unique ID that may be used to generate a database query to retrieve the appropriate data and to transform it into a required format for display.

6.3.1 Client side

The Client side encompasses the following functionality:

- *GUI*, which was presented in the previous section.
- *Presentation logic* (data visualization, identification and manipulation). Every database element has its own unique ID, which is used to maintain mappings between the data in databases and its visual representation in the GUI.
- *Processing logic*. In the process of data modelling, information received from the Server side is used for assembling complex objects (Dictionary components, policies, strategies, etc.).
- *General application management* (communication, error handling, etc.).

Link between the presentation logic and the GUI was made using the standard Document/View concept, suggested by Microsoft for programming in the Windows environment. In such an approach, a data model is defined as a Document that has one or more Views attached - every logical unit of information consists of a data portion (persistent or not) and its visual representation(s). User manipulates the data using one of the Views, and those actions are reflected on the actual data contained

in the corresponding Document. PolicyModeller has six Document-View pairs: Dictionary view – Dictionary Object, Properties view – Properties Object, Policy view – Policy Object, Strategy vies – Strategy Object, Connection bar – Connections Object, and Policy Repository view – Policy Repository Object. For each of those logical units, data model is generated in the memory, based on the data from a corresponding database, and then it is visualized in the appropriate GUI view.

6.3.2 Server side component

The Server side component consists of three objects, one for each database type, which all implement similar functionality:

- *High-level data access*: Interactions with a database are based on parameterised queries. Database queries are written in TRANSACT-SQL, which is a programming language created by Microsoft as an extension to the original SQL standard. Element IDs are used to instantiate appropriate queries for every particular data request.
- *Data pre-processing*, which includes processing of data returned by queries (checking and parsing) and data exchange with the Client.

In addition to those database-specific objects, there is another shared object that implements generic, lower-level interactions with the back-end data management system (MS SQL Server) through OLE DB. OLE DB is Microsoft COM-based interface to tabular data sources, and it can be considered as a more powerful object version of ODBC (Open DataBase Connectivity). The shared object also implements database management using Microsoft SQL Server Distributed Management Objects (SQL-DMO), which is the MS SQL Server object model for programmatic administration. In PolicyModeller, it is used for programmatic creation of Dictionary and Policy Repository databases from corresponding predefined data models.

6.3.3 Back-end data management

Dictionaries, Policy Repositories, policies and strategies represent structured information, which is stored in databases in pieces, as a "flat" (tabular) information. Those constructs are regenerated (assembled) on demand, for the purpose of visualization or processing. In this section we will briefly present the data model implemented by the PolicyModeller, which was created as a mapping of the SPL information model to the Relational model.

6.3.3.1 Dictionary

The Dictionary database was designed to store complex, hierarchical structures of elements, used for Dictionary specifications. It is designed as a collection of dedicated tables for each Dictionary component (Figure 6-9). Dictionary models are disassembled for storage in database tables as a "flat" information, so they must be restored back to its hierarchical representation for display.

In the example on Figure 6-9, we have a Dictionary structure, which begins with a System element that has one Domain (Main), two Roles (User and Personal Data) and one Event (System Event). The Domain includes four Nodes (Master, SE, SW, and NE). In the NE branch, tree is one role (Data Cache), which includes one Service (Management). The Service has two Interfaces (Monitoring and Configuration). One of the interactions in the Configuration interface is the UsageControl, which has a return Parameter (Result), and two input Parameters (Load and Flag). The corresponding structure of the information in the Dictionary database is represented on the right portion of the Figure 6-9.

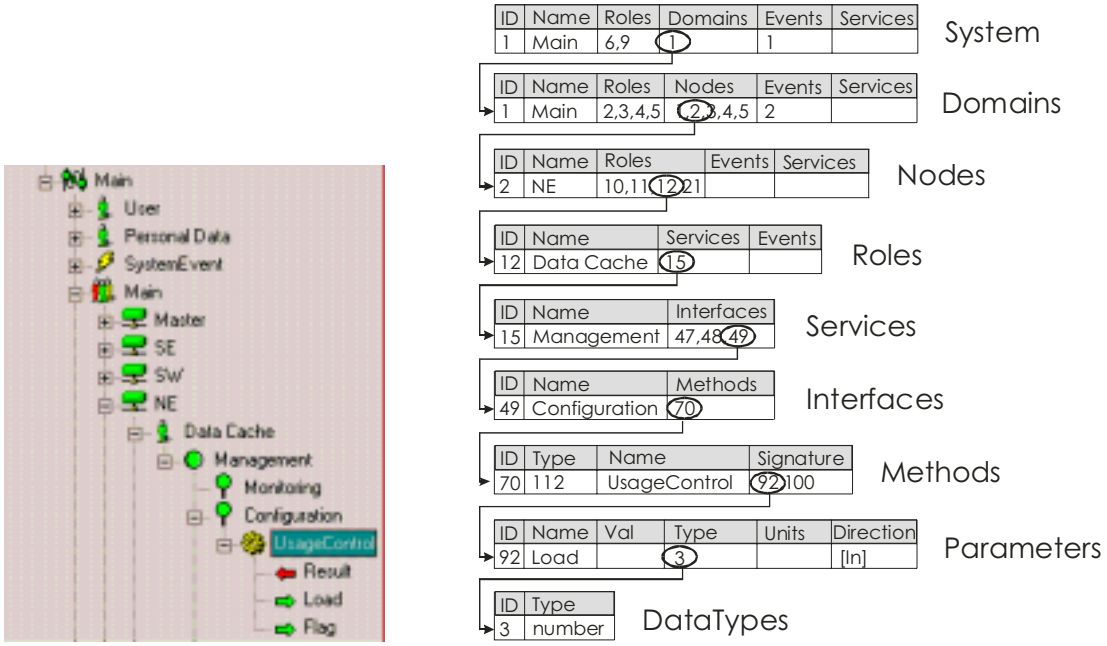


Figure 6-9 Example of a Dictionary branch and its representation in a Dictionary database

Figure 6-9 represents a Dictionary branch the way it would be displayed in the Dictionary view (left), and stored in the Dictionary database (right). In the data model, parent elements use references (IDs) to refer to their child elements (members). Each Dictionary is implemented as a separate database. Such a solution enables greater flexibility and modularity, which is important when policy modelling is performed at different locations, by different designers, and/or at different organizational levels.

PolicyModeller enforces referential integrity by restricting Delete/Edit operations on Dictionary elements. All the operations are performed in the interactive fashion and, if the rules are violated, appropriate errors will be reported. Once created and published, Dictionary elements cannot be deleted or changed (apart for names), because it might affect the validity of the existing policies and strategies. Since the PolicyModeller is only a prototype tool, such a simplified and inflexible solution is quite appropriate. However, for a more ambitious implementation of the policy modelling tool, there will be a need for greater flexibility that would come as a result of a more sophisticated enforcement of the referential integrity in the model. Such an implementation would need to keep track of Dictionary elements that are referenced from policies and strategies, so they cannot be changed. As for the rest of the Dictionary content, the full scale changes should be allowed. The implementation of such a mechanism is not complex, but it was out of scope of this implementation project.

6.3.3.2 Policy Repository

The Policy Repository database was designed to store policy and strategy specifications. Those specifications might be complex, hierarchical data types that are disassembled for storage in database tables as a "flat" information.

Example in Figure 6-10 shows a representation of a policy in a PolicyRepository. IF part of the policy is defined as one predicate list, which has only one predicate. The predicate references two actions, one for each side of the predicate relation. THEN part of the policy is defined as one action list, which has two actions. WITH part of the policy is empty. Policy Metadata and Accounting information are also defined in the appropriate tables. The actual definitions of policy actions are kept external, in a Dictionary database (with its ID equal to 200). The Subject column in the Actions table captures IDs of the appropriate action providers. The visualisation of the policy data from Figure 6-10, as it would appear in a Policy view, is shown in Figure 6-5.

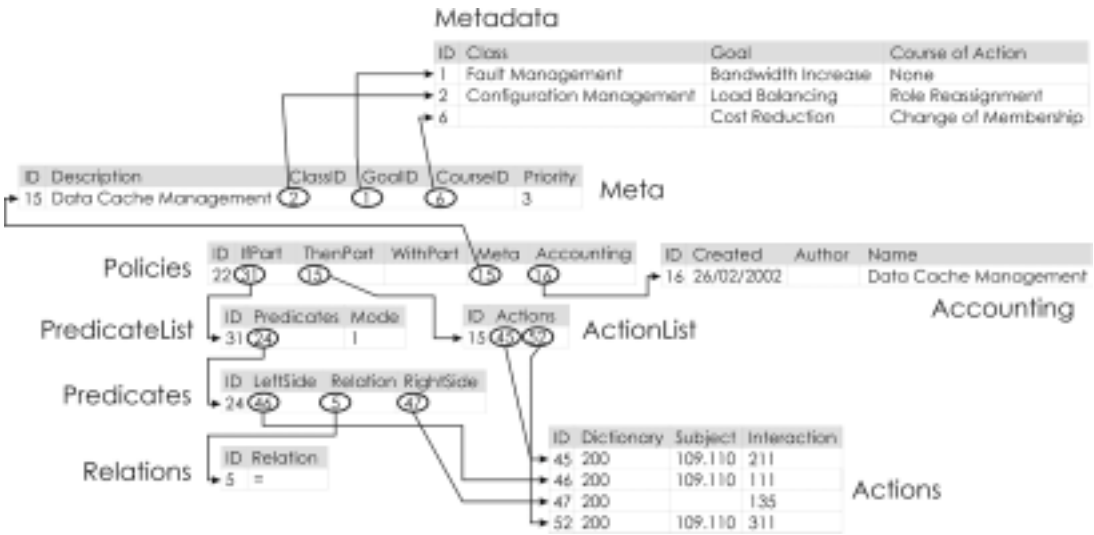


Figure 6-10 Representation of a policy in a Policy Repository database

In order to completely visualize a policy, it is necessary to load the policy definition from a Policy Repository database, and then to load its components from an appropriate Dictionary database.

Strategy components are kept in Policy Repository databases in a slightly different manner than policies (Figure 6-11). Root elements in the strategy body, which may be instances of the Action, IF-THEN-ELSE or WHILE-DO types, are stored in the RootElement column of the Strategies table, but they are defined in external tables. Name, Interaction and DictionaryHigh in the Strategies table are referring to the interaction being refined by the strategy body. The DictionaryLow column stores the low-level Dictionary ID, whose elements are used in the strategy body. The Links column keeps the list of links defined for a strategy (see 4.2.1.2) in the Links table, whereas the Unlinked column keeps track of strategy elements that are not yet linked. As long as s strategy has unlinked elements, it cannot be used for the refinement, because it is not fully defined. If a strategy refines an abstract management interaction,

ClassIDHigh stores ID of the class whose interaction is being refined, and ClassIDLow stores IDs of the class whose interactions are used for the refinement.

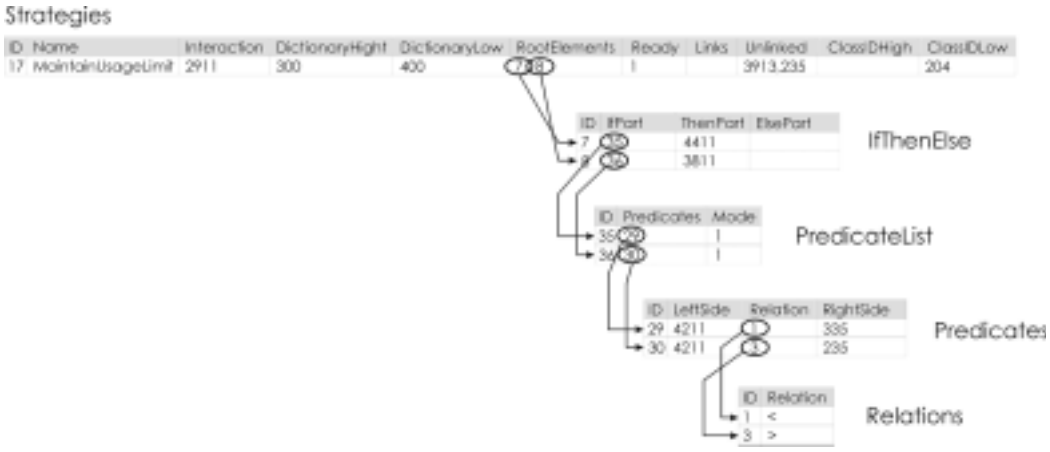


Figure 6-11 Representation of a strategy in a Policy Repository database

In order to completely visualize a strategy, information from three databases will be needed: Policy Repository, which stores the strategy definition, high-level Dictionary, which stores the strategy parameters, and low-level Dictionary, which keeps definitions of management interactions referenced in the strategy body. Strategy parameters belong to the interaction being refined.

6.3.4 Possible evolution of the implemented architecture

Because the PolicyModeller was envisaged only as a prototype, the implementation goals for the project were set to be purely pragmatic: to focus only on what must be done, and to ignore all those nice-to-have features. The biggest compromise was made in the domain of the PolicyModeller architecture. However, in order to take full advantage of the SPF management capacity, a better architectural solution should be implemented for a policy modelling tool.

The basic idea of the architecture evolution, envisaged for a future implementation of the PolicyModeller tool, was to move towards a generic and flexible solution by offering the policy management functionality as a set of Web services. An area currently receiving considerable interest is the development of Web-based management capabilities, which allow management from anywhere at any time using Web browsers and providing platform independence [Bailey 2000].

Four major steps of the architecture evolution may be envisaged:

- *Phase I* represents the actual architecture of the PolicyModeller. The Client/Server communication is based on the standard DCOM RPC, which uses a proprietary binary protocol for communication, and, therefore, offers good performance, but not so much of flexibility and interoperability.

- *Phase II* requires DCOM RPC to be upgraded to SOAP (XML-based messaging). In such a solution, the functionality of the component on the Server side would be offered as a set of Web services. The Server exposes modelling and management services, which are consumed by the Client. In this phase, the Client is still implemented as a dedicated software application.
- *Phase III* requires a major step – moving from dedicated Client application to a universal Web browser and implementation of the GUI in DHTML (Dynamic HTML). The ubiquity of Web browsers makes them an ideal platform for management workstations. According to [TechRepublic 2002], network professionals still prefer a browser-based interface, versus traditional Windows or command line interfaces. Browser-based interfaces are a bit less sophisticated, but they provide much needed mobility and flexibility.
- *Phase IV* finalizes the evolution of the architecture by reflecting the logical distribution of functionality in the three-tier Client/Server architecture in the actual physical distribution of the PolicyModeller components. It is motivated by the fact that moving databases to a dedicated location, accessed remotely, improves security of the data.

6.4 Implemented functionality

In addition to the functionality that supports GUI operations, PolicyModeller performs two basic SPL-related tasks: enforcement of the SPL syntax and data processing, which includes policy interpretation (refinement and instantiation) and policy transformation to an XML representation.

6.4.1 Language constraints

In general, systems should not exhibit constraints that are not actively enforced at the earliest possible moment. The PolicyModelat performs active real-time checks of every user's action on the model. This assures that the model is valid and consistent at any time.

6.4.1.1 Language defaults

Defaults are language implicit assumptions. Domain-specific languages offer especially good opportunities to use defaults, because it may be possible to use knowledge of the domain to infer information that is not stated explicitly [Norman Ramsey]. Good defaults can make descriptions more concise and readable. They can also reduce opportunities for errors. The formal SPL definition does not include specification of language defaults, in order no to be too restrictive. It was assumed that it would be much better to live this issue to be addressed by particular SPL implementations.

In order to actively enforce SPL syntax in the real-time, it was necessary to implement SPL defaults to some extent. Since the model must be correct and consistent at all time, insertion of a new language construct must include certain amount of default values. User can change those values afterwards, but the defaults must be initially present so that new model constructs include all the elements required by the SPL syntax.

SPL language defaults in PolicyModeller are implemented as follows:

string = “”
int = 0
number = 0
bool = TRUE
Exchange = [out]
time = now
pointer = 0
Parameter(type = string)
predicateList(type = AND)
Metadata(priority = 1)
Accounting(created = now)
Dictionary(created = now)
eventList(type = ordered)

Identity

Every instance of a SPL type will receive a unique ID, which cannot be changed. IDs are designed to be composite, so that a type information can be also included. IDs are constructed in the following way:

ID = elementID * 100 + typeID

Therefore, last two digits of an ID contain the information about the type of the corresponding element. This enables context-sensitive operation in the GUI: static type checks, right-click menus, icons, display of element properties in the Properties bar, etc.

6.4.1.2 Referential integrity

Referential integrity in PolicyModeller is enforced by restricting Delete/Edit operations on model elements. In order to make maintenance of referential integrity in a model simple, PolicyModeller does not allow deletion and changes of certain SPL type instances. In addition to the given default values, new instances of Action, Parameter and State Descriptor types are inserted in a model as “unpublished”. As such, they can be changed, but they cannot be used in (referenced from) other parts of the model until being “published”. This “unpublished” state is only provisional (it is not saved in a Dictionary database), and it exists only to enable changes of the default values. Once the changes are made, elements can be “published”, and their properties (apart from the name) become frozen.

6.4.1.3 Type checking

PolicyModeller implements strict static type checking at run-time. Every user’s action must comply with the rules of the SPL syntax, or it will be ignored. Moreover, incorrect actions will be followed by an explanation, displayed in the Status bar.

6.4.2 Policy interpretation

Policy interpretation represents one of the key mechanisms in the proposed policy-based management concept. PolicyModeller implements the policy interpretation process just like it was envisaged by the SPF design (for more details, see 4.2.2.3). The idea is to perform just-in-time interpretation of policies for enforcement. This approach guaranties that every policy will be assembled from up to date Dictionary elements and strategies. All (allowed) changes made in a model will immediately propagate to all Dictionaries and strategies, and they won't affect validity of the existing policies.

In order to be interpreted and executed, policies must be completely and correctly defined. This includes setting the values of all constants used in policies (all "[in]" type parameters and all State Descriptors). A constant value of a State Descriptor may be predefined in a Dictionary, together with the rest of the State Descriptor specification, but that is not required by SPL. When a policy interpretation is triggered, PolicyModeller will pop-up a list of all parameters and State Descriptors that need a value, so the user can fill it in.

6.4.2.1 Policy instantiation

Abstract policies, specified in terms of abstract interactions from class interfaces, need to be instantiated into a number of corresponding concrete policies, one for each member of the class that implements those abstract interactions. A simple instantiation algorithm implemented in PolicyModeller is presented in Figure 6-12.

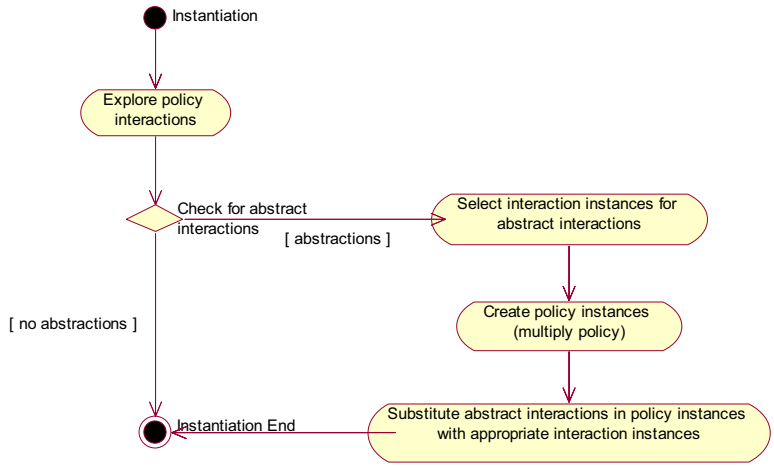


Figure 6-12 Basic steps in policy instantiation

The instantiation procedure is implemented in the PolicyModeller exactly as it was envisaged by the SPF design. For more information on policy instantiation, see section 4.2.2.3.

6.4.2.2 Policy refinement

Policy refinement is a cascade process of replacing interactions from one Dictionary (higher level) with their interpretations from another Dictionary (lower level). While the policy instantiation changes the

level of abstraction within the context of a single Dictionary, the policy refinement does the same by moving policy specification from one Dictionary context to another.

A simple refinement algorithm, implemented in by the PolicyModeller, is shown in Figure 6-13. Refinements of higher level interactions are defined as strategies. In order to be used for refinement, strategies must be completely specified as required by the SPL syntax. All strategies that are partially specified will be ignored in the refinement process. In the course of the policy refinement, every occurrence of a policy interaction that has a corresponding strategy to refine it will be replaced with that strategy, more precisely with the strategy body.

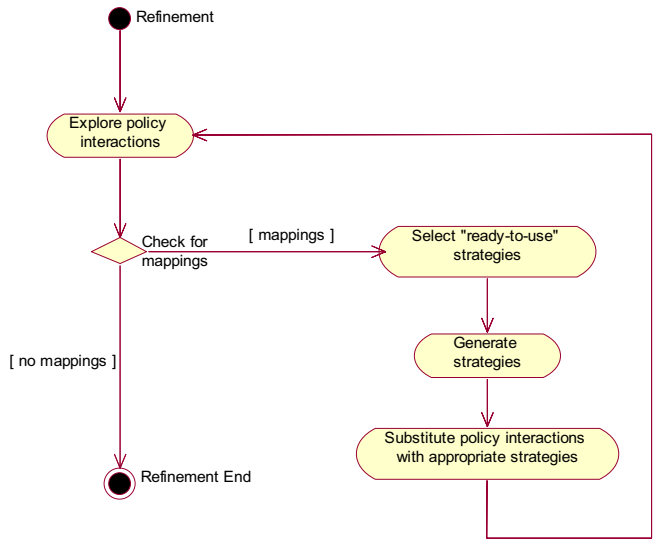


Figure 6-13 Basic steps in policy refinement

After one cycle of refinement, policy that was specified in terms of interactions from a higher level Dictionary will become defined in terms of interactions from a lower level Dictionary – it will be refined to a specification at a lower level of abstraction. The refinement cycles will continue as long as it is possible to replace policy interactions with their corresponding strategies. When no more strategies can be found, the policy refinement process is completed. For more information of policy refinement, see 4.2.2.3.

6.4.3 Policy Markup Language (PolicyML): Policy representation in XML

The final step in the process of policy interpretation is transformation of policy specifications into an XML representation. In addition to the general advantages of having information represented in XML, which are presented in section 1.4.3, the motivation for having policies in XML is that it perfectly suits the management model adopted by SPF. For more details, see section 4.1.5.2.

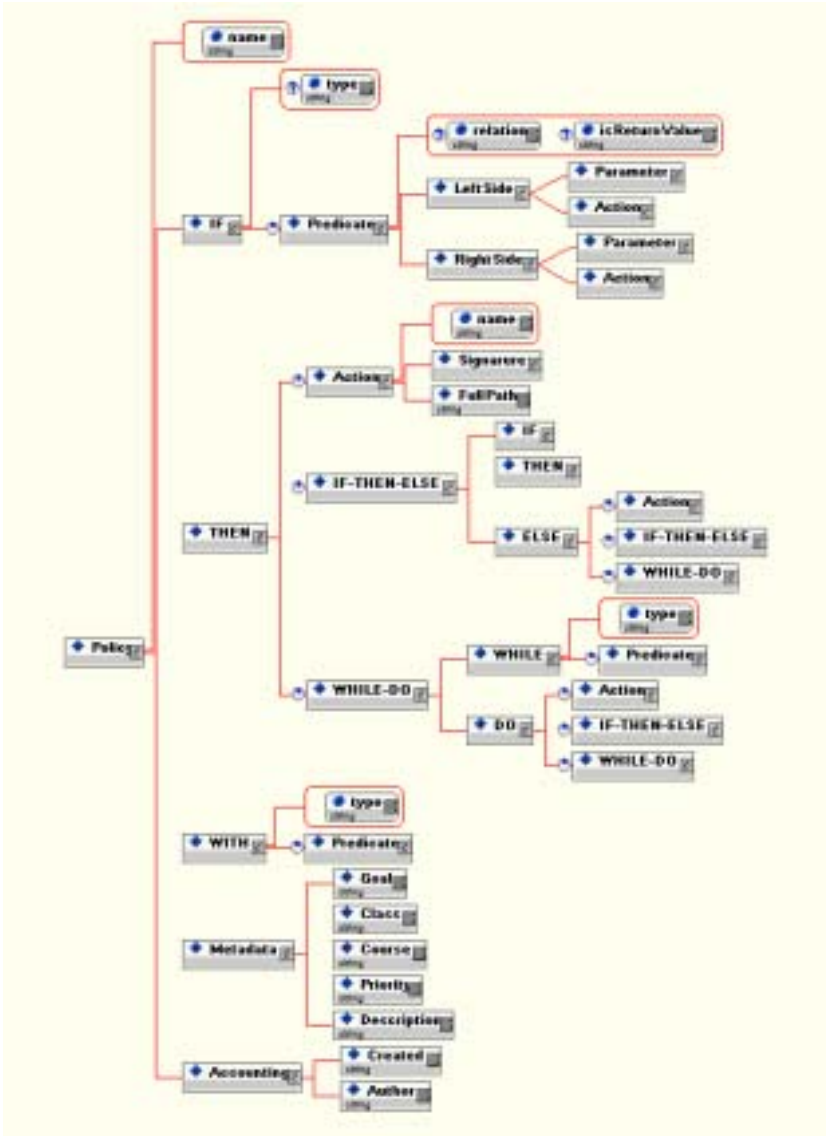


Figure 6-14 Visual representation of the SPL policy in XML

PolicyML is an XML implementation of SPLc (Figure 6-14). The goal of PolicyML is to enable exchange of policy specifications between SPF and different workflow engines. The proposed XML representation of policies is based on the formal definition of SPLc (see section 5.1.2). It is important to keep in mind that PolicyML is only one of many possible SPL implementations in XML.

PolicyML is a technology independent format for policy information, which can be transformed to workflow engine-specific formats (WSFL, ebXML, XLANG, etc.) using XSLT. The use of an intermediate language (PolicyML) and XSLT offers a greater flexibility than a direct transformation from SPL to target workflow engine-specific formats. This approach decouples SPF from the technology of managed systems, and it keeps the transformation process SPF-independent. The transformation logic will not be hard-coded, but kept in customisable XSLT files.

6.4.3.1 PolicyML Syntax and Grammar

In XML, the definition of a valid markup is handled by a DTD (Document Type Definition), which defines the grammar and vocabulary of a markup language. Since PolicyML is implemented in XML, its syntax is governed by the rules of XML syntax, and its grammar is defined by a DTD. In other words, the details of using tags, attributes, entity references, etc. are reused from the XML, while details on PolicyML elements and attributes, and their organization are specified in the PolicyML DTD. The formal definition of PolicyML is presented in Appendix B.

A DTD isn't an XML document, so its parsing requires an additional technology. Both the need for stronger data typing, as well as the desire to develop an XML-based description of constraint declarations, led to the development of an XML Schema language [Grosso 2001]. PolicyML is consciously designed to take advantage of the latest in the evolving Web technology. Thus, the grammar of PolicyML has been defined in an XML Schema as well (see Appendix B).

6.5 Conclusion

The main goal of the PolicyModeller implementation project was to demonstrate practical qualities of SPL, most of all that the proposed policy specification language is easy to implement and to visualise. Moreover, the PolicyModeller needed to demonstrate how easy it would be to work with SPL on system modelling and policy specification. PolicyModeller represents only a prototype implementation of a SPL-based modelling tool. Still, it offers some practical solution in the domain of referential integrity, modelling-time verification, abstraction and storage mechanisms, error reporting and visualisation of SPL. It also clarifies some details about the role of SPL and the Modelling Environment component within the management framework (SPF).

PolicyModeller has shown the extent to which the modularity and simplicity of SPL can be used to create a powerful policy modelling environment. Mapping of the SPL information model to the Relational model was straightforward, visualisation of the SPL constructs was effective, the concept of assembling was proven as efficient and flexible, and policy transformations were easy to implement, including the implementation of the PolicyML.

However, the PolicyModeller implementation of SPL has two limitations:

- An abstract policy can include abstract operations that belong only to one class. References to multiple classes in the same policy are not allowed for the sake of simplicity.
- Roles cannot be shared among Nodes or Domains. However, they can be replicated or moved to other Nodes/Domains.

In PolicyModellar, SPL was implemented in a way which guaranties that every step in the policy specification process is strictly controlled and that created models are valid and consistent. Validity of models is guarantied by the content of Dictionaries, which must reflect the real system management capabilities. Nothing that cannot be enforced should be allowed in a specification. This eliminates wishful thinking from the policy-based management. According to [Huber 1997], consistency includes several different classes of syntactical correctness criteria:

- *Grammatical Correctness*: The corresponding specifications obey the syntactical rules of the specification language. PolicyModeller strictly enforces SPL syntax.
- *Document Interface Correctness*: If a document is embedded into another document, those documents must have compatible interfaces to each other. In SPL, we use the rules of containment, which are defined by the SPL information model, and a strict typing system to control composition of SPL constructs.
- *Definedness*: If a document makes use of objects not defined in the document itself, those objects must be defined in a corresponding document. We implement this concept as the mechanism of referential integrity.
- *Type Correctness*: The type of an object assigned and the type of the object it is assigned to must coincide. The rules are defined by the SPL syntax and enforced by PolicyModeller.
- *Completeness*: All necessary documents of a project have to be present. Again, the rules are defined by the SPL syntax and enforced by PolicyModeller.

SPL has been designed to support all those criteria, and PolicyModeller implements them all strictly. However, in every design process there must be stages which necessitate temporary specification inconsistency. In the PolicyModeller, such a problem has been solved using publishing concept, in which some aspects of a specification may be defined but not published until they are completed. Strategy specifications are not made operational by the PolicyModeller as long as they are not completed. Also, some elements of Dictionary specifications (instances of Action, Parameter and State Descriptor types) can be partially defined and explicitly published by the user after they become complete.

In addition to a better architectural solution, discussed in 6.3.4, PolicyModeller would particularly benefit from a more sophisticated mechanism for referential integrity control, and from a support for access control in Dictionary and Policy Repository databases. A better referential integrity control would enable more flexibility in working with Dictionary models, because all Dictionary elements that are not used in policies and strategies would be fully available for Edit/Delete operations. The access control would promote security, especially for remote access to information, enable personalized views in the PolicyModeller GUI, and enforce strict separation of concerns and control in the modelling process.

In the next chapter we will give a detailed critical analysis of SPL and SPF, in order to validate the outcome of the research presented in this thesis.

7 CRITICAL ANALYSIS

In this chapter we will present a critical analysis of the policy framework (SPF) and the policy specification language (SPL) proposed by our research, which was based on the general quality factors derived from the literature. The focus of our analysis will be on SPL, because SPF is a conceptual framework whose operational characteristics may be examined in more details only after its implementation.

7.1 Policy framework (SPF)

In this section we will use a number of general software quality factors, compiled from the literature (see [Kiczales 1992], [Anderson 1999], [Keller 1990], [ANSI/IEEE 1992], [Barbacci 1997], [Barbacci 1995]), to judge what was achieved by the SPF design. However, it is impossible to talk about a conceptual framework as it was a finished software product. Therefore, our analysis can focus only on the SPF design, while its operational characteristics could be examined in more details only after an implementation.

Efficiency

Efficiency is a relationship between the level of performance and the amount of resources used under stated conditions. It includes time economy, which is a capability of software to perform specified functions under stated or implied conditions within appropriate time frames, and resource economy, which is a capability of software to perform specified functions under stated or implied conditions, using appropriate amounts of resources.

In general, the required response time from SPF will depend on particular management goals and the envisaged role of the framework in the overall management operations. We don't expect high-level management to be as intensive as lower-level when it comes to management decisions required over a certain period of a time, simply because high-level management is strategic in nature. Simple, tactical management decisions, which need to be made more often and in large numbers, and for which a fast response is essential, will be provided by the embedded, lower-level system management mechanisms.

In order to examine behaviour of the framework with regard to the time and resource economy, to detect potential sources of overhead and performance problems, and to suggest improvements, a complete implementation of SPF will be required. Some of the issues that might arise with the framework will be discussed in section 7.1.1.

Functionality

Functionality represents the existence of certain properties and functions that satisfy stated or implied needs of users. It includes completeness, correctness, security, compatibility and interoperability.

Completeness represents the degree to which software possesses necessary and sufficient functions to satisfy user needs. The completeness of SPF will be examined from four aspects, which are identified in [Strassner 1999] as required for policy control:

- *An extensible information model*. The SPF information model is defined by SPL, and it covers the definition of both data and communication primitives required by the framework. Modularity and high abstract level of SPL constructs offer lots of opportunities for extension. New abstraction primitives can be easily plugged-in, without disturbing the existing model organization.
- *A scalable framework for policy administration, management, conflict resolution, and distribution*. The SPF management process was designed to completely cover all the aspects of policy based management at a conceptual level. It includes triggering, interpretation, and analysis and control mechanisms, which are both required and sufficient for an autonomous framework operation.
- *A policy specification, written in terms of a policy definition language, which can represent business requirements and functions in a vendor- and device-independent manner*. SPL is both an information model and a policy specification language, which eliminates the problem of mappings between the two. The language is aiming at a high abstract level of policy specification, especially suitable for expressing business goals and requirements.
- *A scalable means to translate from the device- and vendor-independent policy specification to vendor- and device-specific configuration commands*. The SPF mechanism of policy interpretation was specially designed to handle the problem of policy transformation from an abstract to an executable representation. The exact level of abstraction in executable policies produced by the framework will depend only on the semantics of management operations from the target system management interface.

Correctness represents the degree to which software conforms to the requirements. The correctness may refer only to particular implementations of SPF, and it will not be discussed any further. The same is for security, which was not a problem addressed by the framework simply because the framework does not want to prescribe any implementation details. Problems related to the information and communication security are well known and appropriate solution to those problems exists in various forms.

Compatibility represents the degree to which new software can be installed without changing environments and conditions that were prepared for the replaced software. This encompasses support for the key standards as they become accepted and capability to demonstrate and capitalize on the interoperability provided by the standards, and balance between the immaturities of certain standards against the needs of customers. Interoperability represents the degree to which software can interact and integrate with existing and prevalent IT management solutions and leverage their strengths to enable a complete policy management solution.

SPF was designed to support the latest distributed system standards: XML, Web services and SOA. It may as well interact with Directory Services via LDAP, and with CORBA-based architectures. It is important to note that SPF doesn't compete with the existing policy-based management solutions, but complements them as a higher management level (see Figure 7-1).

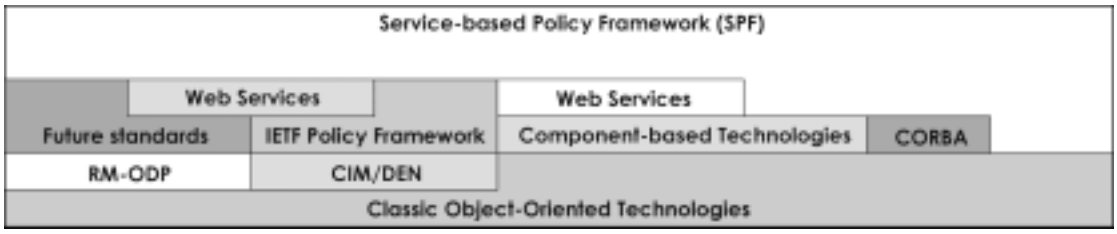


Figure 7-1 Policy-based management and related technology stack

In general, it would be possible to use SPF directly for a low-level management of an object-oriented system. With a help of Web services technology, which may be used as a wrapper for object-oriented and component-based architectures, those low-level management operations may be upgraded to a much higher level of abstraction.

DEN offers a good foundation to build upon, and it may be considered as a low-level infrastructure for the SPF policy-based management model. CIM and DEN can serve as an information base for the creation of Dictionaries, as high-level repositories of management information. Directory services in DEN support a LDAP-based discovery of system management resources. However, it is important to bear in mind that the SMS Broker must play an active role in the discovery of changes and updates in a Directory, because LDAP doesn't support the notification mechanism the way the UDDI Registries do. Mapping of SPL to the DEN information model was defined via PCIM (see Appendix A), for the sake of interoperability with the Directory services.

The RM-ODP Enterprise viewpoint supports a concept of high-level system modelling and management similar to SPF. At some level of description the ODP system is represented as an object in the community [ISO/IEC 1995a]. Because of the similar scope of the RM-ODP Enterprise viewpoint and SPF/SPL, questions about how these two concepts relate naturally arises. With regards to that, it would be particularly interesting to explore the following two issues:

- If and to what extent SPF can be seen as a refinement of the RM-ODP Enterprise viewpoint?
- If and to what extent SPL can be used to model and manage a system from the RM-ODP Enterprise viewpoint?

Making such an analysis would be a complex task, mainly because RM-ODP is largely open to the possibility of different interpretations. Even though such an analysis is far beyond the scope of this research, it could result in a number of fruitful ideas for SPF/SPL convergence towards a full support for the RM-ODP Enterprise viewpoint. This could be seen as one of the natural paths of evolution for our model because of the importance of RM-ODP.

Maintainability

Maintainability (flexibility) is the effort needed for specific modifications. It includes correctability, which is the degree of effort required to correct errors in software, scalability, which is the degree of effort

required to expand or upgrade the efficiency or performance, and verifiability, which is the degree of effort required to verify the performance.

A policy management framework must be prepared for extension into unforeseen contexts, because the management requirements cannot be determined in advance. The ability of a framework to be extended in such a case is crucial for its successful reuse [Riehle 1998]. The SPF's capacity for extension is based on its support for the latest standards, independence from managed systems, flexibility of its management process (given by the built-in rule-based control), and modularity on its architecture.

When it comes to the verifiability of the performance, various measurement and monitoring instruments can be easily introduced in the framework, because of its modular, loosely coupled design. The actual mechanisms required for the performance control are left to be designed by particular SPF implementations.

Portability

Portability is the ability of software to be transferred from one environment to another. It includes hardware and software independence, installability (the effort required to adjust software to a new environment), and reusability (the degree to which software can be reused in other applications).

Since SPF is a conceptual framework, the question of hardware and software independence is related to its particular implementations. The framework itself does not impose great requirements on managed systems, but its capacity to deliver an automated reactive management will depend on the available support for the discovery and event notifications from a managed system.

SPF was envisaged as a generic management plug-in component, designed to be technology independent to a largest possible extent. The independence comes as a result of:

- *"Back-box" management view* of the system, based on the idea of system encapsulation behind a management interface, which helps to strictly separate management activities specified in policies from their implementations in the managed system.
- *Minimal interaction* with the managed system. Essentially, there are only three points of interaction with the system: discovery of the system management operations, receipt of the system events and sending of the policies to a workflow engine.
- *Highest possible level of abstraction* in the interactions with the managed system. SPF was designed for a message-based communication with the system (XML and Web services technology), even though a RPC-style communication is also supported. SPF operates on a policy specification meta-language (SPL). The policy meta-language offers only a generic syntax, which includes rules about how to create a vocabulary and how to put together words from the vocabulary, but not the vocabulary itself. In every particular case, a system specific vocabulary will be assembled from the discovered system management operations.

Reliability

Reliability is the capability of software to maintain its level of performance under stated conditions for a stated period of time. It includes nondeficiency, which is the degree to which software does not contain undetected errors, error tolerance, which is the degree to which software will continue to work without a system failure and to which software includes degraded operation and recovery functions, and availability, which is the degree to which software remains operable in the presence of system failures.

All these issues, although very important, were not addressed by the SPF conceptual design, because the question of reliability is very much technology-dependent. Those issues can be easily addressed by a framework implementation.

Usability

Usability is the effort needed for use. It includes understandability, which is the amount of user effort required to understand software, ease of learning, which is the degree to which user effort required to understand software is minimized, operability, which is the degree to which the operation of software matches the purpose, environment, and physiological characteristics of users, and communicativeness, which is the degree to which the software is designed in accordance with the psychological characteristics of users.

The SPF design was based on the best practices in the distributed systems modelling and commonly agreed requirements for the policy-based management. The quality of interaction between SPF and its human users depends to a large extent on the design of the framework GUI. The major goal of the PolicyModeller project, presented in Chapter 6, was to demonstrate to what extent policy modelling and management may become intuitive, simple and user-friendly.

7.1.1 Potential issues with SPF

The "use-and-waste" policy life-cycle (see 4.2), implemented in SPF, may be a source of management overhead. The adopted concept gives the flexibility to the SPF management process, but the price of that flexibility must be analysed in the context of particular management goals. The overhead would be easy to measure and it would stay constant in time, because it does not depend on the size of a managed system.

It is important to bear in mind that the only alternative to this approach would be to maintain the correctness of all deployed policies in time, by implementing an update mechanism within the framework (solution used for Ponder, see 2.4.2). We believe that such a solution would make the framework much more complex to implement. Also, as a result of such an approach, the SPF management process would become more fragile. Correctness on the management would depend on correctness and the speed of updates, and it is likely that the management process would constantly experience temporary inconsistencies because of an inevitable delay in propagation of changes. This opens a question of scalability of the solution, because its efficiency will depend on the number of policies, on the size of the management model and on the nature of changes.

Another source of management overhead may be located in the process of transformation between SPL specifications and an execution-ready format (see 4.1.5.2). We envisage an intermediate step between SPL policies and a target XML format in order to maintain independence between the framework and the managed system and in order to standardize the process of transformation itself.

Finally, the policy triggering model (based on the Event Handlers mechanism, see 4.2.2.1) may also become a source of an overhead in the management process. The adopted triggering model gives lots of flexibility to the management process by enabling independence of policies from system-specific events, by enabling filtering of the events, and by enabling reuse of policies for multiple events. However, the price is more complicated triggering mechanism and an overhead in management response.

There will be a need for a complete implementation of SPF to judge about the efficiency of its management process. Also, a proper balance between speed and flexibility will always depend on particular management goals. It is a general rule that flexibility usually comes at a price of responsiveness, and the Web services technology is an obvious example. However, benefits of the flexibility are usually much more important and the processing power of modern systems is able to address the additional work quite easily. Moreover, the additional processing required to address all mentioned overheads is concentrated only in one framework component, which will be installed only on one physical node, and it can be easily addressed by installing additional processing power.

7.2 Policy specification language (SPL)

SPL was created to be in the line of the major standardization efforts in the domain of distributed systems (Figure 7-2). It practically refines the RM-ODP Enterprise viewpoint towards the IETF policy management framework. The formal mappings to the PCIM are defined in Appendix A, while its likely convergence towards a full compatibility with the RM-ODP Enterprise language is left for future work (see more details in 8.3.2). Although Ponder was not originally designed as a RM-ODP Enterprise language, it does provide a concrete representation to implement most of its essential concepts. An analysis of how Ponder relates to RM-ODP is given in [Lupu 2000], while our analysis of the same problem is given in 3.1. On the other hand, the question of compatibility between Ponder and IETF standardization efforts remains unanswered. It is questionable to what extent a fairly complex policy model implemented in Ponder can be mapped to a relatively simple, events-free policy representation envisaged by the IETF.

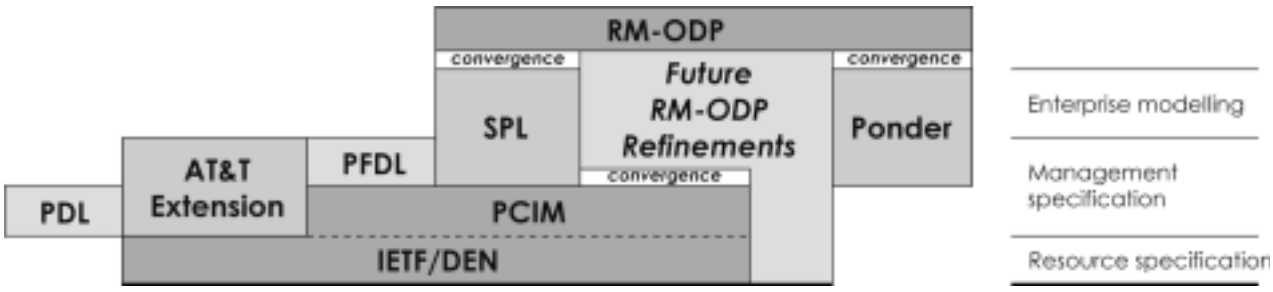


Figure 7-2 Policy specification concepts

SPL targets both enterprise modelling and policy specification. Just like in RM-ODP, an enterprise model of a system in SPF consists of multiple viewpoints. Unlike the RM-ODP, viewpoints in SPL have the same concern and the same organizational units and rules, and are tied together by the explicitly defined relations of abstraction/refinement. A system model in SPL has a precisely defined structure and abstract state domains, enriched with additional relationships among its organizational units. Relationships among the organizational units can be different in nature:

- *Aggregation* of behaviour. Just like Ponder and RM-ODP, a SPL system model supports aggregation of behaviour in named organizational units. However, every organizational unit in SPL is stateless and encapsulated behind its representative (collective) behaviour, while RM-ODP and Ponder have typical object-oriented (state + behaviour + identity) approach to aggregation and don't require encapsulation. The behaviour of an organizational unit in SPL defines both the requirements of the unit membership (what is required to join the unit), and the collective behaviour of the unit. The state in SPL is modelled only at the system level as an abstract state domain, defined in terms of a structured hierarchy of parameters (State Descriptors).
- *Classification* of behaviour is supported in SPL for the purpose of addressing the problem of complexity within viewpoints (Dictionaries). The general concept of classification, as a relationship of inheritance between types, exists in RM-ODP and Ponder as well. However, instead of classification (parameterization) that enables reuse of structure (RM-ODP and Ponder), SPL gives preference to the classification of organizational units (reuse of building blocks), which seams to be more suitable for a high-level system modelling (fewer structures, variety of building blocks) and easier to implement.
- *Generalization* of behaviour is an important abstraction mechanism in SPL, used to create relationships between different viewpoints. The relationships of generalization/specialization are defined in terms of strategies, which specify relationships of behavioural compatibility among management operations at different levels of abstraction (from different viewpoints).
- *Delegation and authorization* of behaviour. With the extension proposed in 7.2.3.2, SPL is additionally capable of expressing the relationships of delegation and authorization. Unlike the relationships in Ponder, which are regarded as management policies, the relationships in SPL can be seen as policy interpretation rules. The SPL relationships dynamically reconfigure the management model, but they do not act on the managed system itself.

A policy specification in SPL is always made in the context of an enterprise model, and it relies on the following key concepts:

- *Operational semantics*. SPL extends the classic "If-Then" policy semantics with the policy post conditions, which makes it somehow similar to the AT&T policy model. Such a policy form gives lots of expressive power to SPL specifications and it enables the SPF management process that consists of cyclic management workflows of the type: Event ⇓ State ⇓ Actions ⇓ State, which is essential for the implementation of the Management Control Model (see 1.4.1).
- *Metadata information* consists of policy metadata and policy attributes. The idea of policy metadata comes from the need to relatively easily extract the semantics of a policy specification for the purpose of policy understanding and management. All policy specification concepts include some sort of metadata, but SPL puts it in an explicit form to enable easy access and manipulation. Policy attributes, similar to the concept of policy properties in AT&T extension of the IETF policy model,

extend the policy metadata mechanism with custom-designed meta information. SPL also supports external specifications, similar to Ponder. Unlike Ponder, which attaches external specifications to policies, the external specifications in SPL are attached to management operations only, in order to provide binding information and/or additional semantic data.

- *Policy grouping*. The concept of policy grouping is implemented in SPL as a dynamic selection mechanism, based on the policy metadata. The members of policy groups are not predefined, like in Ponder and IETF model, but dynamically selected at run-time, which offers a greater flexibility.
- *Policy triggering*. Just like the IETF policy model, SPL policy specifications don't include the notion of events. Unlike the IETF policy model, however, SPL directly supports reactive management by means of Event Handlers. The mechanism of Event Handlers supports ordered/unordered combination of triggering events (like in PDL and IETF models), passing event parameters to policies, and conditional, state-based policy triggering.
- *Policy interpretation*. Policy interpretation is essential for a dynamic policy model: storage format (static, persistent information) is combined with the current state of a managed system (dynamic information) in order to assemble the operational policy format (transient information). In the process of policy interpretation, static policy structure is merged with its dynamic content from Dictionaries in order to produce valid and executable policies. Without the policy interpretation there is no dynamic policy specification. The importance of the policy interpretation has been also acknowledged in the IETF management framework.
- *Policy refinement*. Multi step policy refinement is essential for specification of large-scale management operations. As a result of the refinement, simple abstract policies may expand to more complex management workflows. PDL also supports the specification of management workflows, but it does that by interrelating predefined sequences of policy triggering using events.
- *Semantic analysis* is used for general policy management and conflict resolution. Ponder applies a similar concept of meta-rules, but we believe that the approach in SPL is simpler and much better integrated in the overall information model and the supporting policy framework., and therefore much easier to support and use. The semantic analysis in SPL is based on the analysis of both policy metadata and policy operational semantics, and therefore more sophisticated than the meta-policies in Ponder.

7.2.1 Formality of SPL

We will start our analysis of SPL with regard to the so called seven deadly sins of formal specifications – common deficiencies of informal specifications, which were identified in [Meyer 1985] as noise, silence, overspecification, contradiction, ambiguity, forward referencing and wishful thinking.

Noise is the presence of irrelevancy and unnecessary duplication which masks the basic intent of the specification. The presence of elements that do not carry information relevant to any feature of the problem is addressed in SPL by a careful choice of the language constructs, which was made to include only what is essential for specification of a system management behaviour and constraints on it. The problem of redundancy SPL addresses with its modularity and a mechanism of reuse of specifications it produces. Every component of a specification consists of a predefined skeleton and a number of references to externally defined building blocks, which are shared.

Silence is the (unintentional) omission of parts of the intention. The existence of a feature of the problem that is not covered by any element of the specification is addressed by the language syntax, especially in the part that defines the information model. The SPL formal grammar, which shall be strictly enforced at design time by a modelling tool, does not allow any presence of partially specified constructs in specifications. Moreover, for a specification to become operational, all the related components of a model must be completely defined.

Overspecification is providing details of how the specification may be realized, thereby suggesting we employ a particular implementation which may or may not be appropriate. The presence of an element that corresponds not to a feature of the problem but to features of a possible solution is actually addressed by design of the SPF management model. In SPF, policies are delegated to the system for execution, without any restriction on how it may be done internally. Encapsulation of a system from the management point of view hides all the details of the system internals, including the details on how policies will be executed within the system.

Contradiction is the case when two or more elements define a feature of the system in an incompatible way. The possibility of contradiction in a SPL specification is eliminated by the modular design of its information model and by the formal definition of the language syntax, which enable strict static typing and elimination of redundancies. At the semantic level, this issue is addressed by the SPF management process, and by means of semantic and conflict analysis of policy specifications.

Ambiguity comes as a result of a possibility to interpret an aspect of the problem in at least two different ways. This problem was one of the most important concerns during the language design. The choice of the language constructs and their names were made to closely match elements of the problem domain and commonly agreed terminology in the field. Moreover, a precisely defined language vocabulary is the only source of building blocks for SPL specification constructs. Together with the previously presented language mechanisms that eliminate the possibility of contradiction, SPL is able to guaranty unambiguous specifications.

Forward referencing is appealing to concepts that are defined later yet are used to make an important point early in the specification. In SPL, language constructs must be defined before other specification constructs can reference them.

Wishful thinking is including some feature(s) which, with all the goodwill in the world, cannot be realistically implemented. Policy specifications are built upon a management model of a system. The model is created as an abstraction of the real system management capabilities, which guaranties policy executability. The system management interface is a promise that the system must keep, but the actual implementation of the system management interface is a strictly internal problem of the system, and it is considered to be encapsulated behind the system management interface.

7.2.2 Cognitive dimensions of SPL

The cognitive dimensions framework is a technique for evaluation of programming environments. It uses a small vocabulary of terms designed to capture the cognitively-relevant aspects of structure, and shows how they can be traded off against each other. These terms can be used to analyse the usability of a notation or its supporting modelling tool.

The framework is focusing on the processes and activities a user of a notation must perform, rather than on the specification itself. There are thirteen terms currently defined, each of which represents a concept that is orthogonal to each of the other terms. The following term descriptions are taken from [Green 1996]:

- *Abstraction* What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
- *Closeness of mapping*: What 'programming games' need to be learned?
- *Consistency*: When some of the language has been learnt, how much of the rest can be inferred?
- *Diffuseness*: How many symbols or graphic entities are required to express a meaning?
- *Error-proneness*: Does the design of the notation induce 'careless mistakes'?
- *Hard mental operations*: Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what is happening?
- *Hidden dependencies*: Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
- *Premature commitment*: Does the user have to make decisions before they have the information they need?
- *Progressive evaluation*: Can a partially-complete specification be examined to obtain feed back on "How am I doing"?
- *Role-expressiveness*: Can the reader see how each component of a specification relates to the whole?
- *Secondary notation*: Can the user make use of layout, colour, or other cues to convey extra meaning, beyond the 'official' semantics of the language?
- *Viscosity*: How much effort is required to perform a single change?
- *Visibility*: Is every part of the specification simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the specification is dispersed, is it at least possible to know in what order to read it?

An analysis along each of these dimensions may give a number of defined points related to usability that can be discussed in relation to any particular language. To evaluate the usability of SPL, we will examine it under each of the cognitive dimensions presented above.

Abstraction

Information hiding is a technique for handling complexity. By hiding implementation details, programs can be understood and developed in distinct modules and the effects of a change can be localized. One technique for information hiding is data abstraction [Larochelle 2002]. According to [Green 1996], languages can be grouped as abstraction-hating, abstraction-tolerant, or abstraction-hungry, based on their minimum starting level of abstractions and their readiness or desire to accept further abstraction.

The SPL abstraction mechanism was introduced to handle potential complexity of the language vocabulary, which reflects complexity of the system management interface. Specification of a large, complex system requires a model construction and property description. The collection of properties being specified is often fairly large; the language should thus allow the specification to be organized into units linked through structuring relationships [van Lamsweerde 2000].

SPL may be seen as an abstraction-hungry language, because it requires a vocabulary (Dictionary) at an appropriate level of abstraction to be specified prior to policy specification. Of course, policy modelling may start at a low level of abstraction, and such an approach would be quite appropriate for simple system models. However, for a complex distributed system management, a hierarchy of Dictionaries will be needed in order to successfully face the problem of complexity.

Syntax extensions provide syntactic sugar for problem-specific abstraction [Cardelli 1994]. Syntactic sugar is a term introduced by Peter Landin for additions to the syntax of a language that do not affect its expressiveness, but make it "sweeter" for humans to use. Syntactic sugar enables an alternative way of coding that is more concise or more like some familiar notation. It can be easily translated ("desugared") to produce a program in some simpler "core" syntax. In SPL, abstract policies, specified in terms of abstract management operations, are “sugared” specifications that must be “desugared” for execution via the process of refinement.

The abstraction mechanism in SPL is one of its greatest strengths, because an abundance of low-level primitives is one of the great cognitive barriers to policy specification. Simplified system models enable easy understanding and using of system management operations in policies. An initial investment in defining powerful policy building blocks pays-off in a straightforward management model, which consists of a simplified view of the system and small number of simple policies.

Closeness of mapping

Closeness of mapping refers to the distance between a mental model of the problem and its equivalent specified using a language. It was derived from the fact that the closer the language world is to the problem world, the easier the problem solving will be. Ideally, elements of the mental model should be mapped onto constructs in the language domain.

In SPL, appropriate abstractions are defined for a clear and concise representation of management activities and their relations, so that elements in a specification can closely and expressively describe the concepts from the problem domain. Terminology for the language constructs and the SPL data model have been derived from the existing work on policy-based management and distributed systems, so it may be considered to be standard in the field. Abstractions enable easy representation of management operations and their elements, as well as policy specification. Dictionary represents a manager’s mental model of a system, assembled from management operations dynamically discovered in the system management interface. SPL provides all the main abstractions required for specification of such a mental model: actions, predicates, policies, policy metadata and attributes, triggering and control rules. Moreover, the language supports structural and classified representations of a system from the management point of

view, by organizing its management operations logically into services, and assigning it to an abstract organization of roles, nodes domains and systems.

Consistency

Consistency is concerned with the relationship between different language components, and it was based on the fact that, in a consistent language syntax and semantics, conventions propagate on all language components. As a result, a person that knows some of the language can guess lots about the rest of the language constructs and rules.

Consistent (re)use of the language elements was one of the main concerns through the SPL design, in order to come up with a small set of simple language constructs. The language has been designed in a modular fashion, where a small number of auxiliary language constructs was used to build more complex ones. Moreover, a significant number of the language elements are connected by the relation of inheritance, which also improves consistency.

In order to simplify language syntax, all rules in SPL have the same policy form. This includes policies, general semantic rules, and rules that handle conflicts. However, users should be aware of the fact that these rules essentially have different targets. Policies are targeting a managed system in order to constraint is behaviour, while semantic rules are targeting the SPF management process in order to control its operation.

Diffuseness

Diffuseness or terseness of a language refers to the number of symbols required to express a unit of information. Very diffuse languages may be hard to understand, because they are over-full of different language constructs. Visual languages generally have a lower diffuseness, because pictures have more expressive power than text.

SPL was designed with brevity in mind. Closeness of mapping achieved in SPL, which was discussed earlier, reflects in a small number of abstract language constructs that are able to represent all the necessary symbols from the problem domain. However, the amount of details introduced in policy specifications depends largely on the level of abstraction that a user wants to achieve. For example, management actions are essential component of every policy, and, as such, they largely influence the size of every policy specification. Actions may be specified to include subjects and targets of interactions, but that sort of information is not required. Moreover, the amount of information specified in policy metadata as well as the number and content of policy attributes depend on the user.

For the impression of the size of a specification, important is the support from a modelling tool too. SPL is simple enough to be easily visualised, and various parts of its structured specifications may be expanded/collapsed as needed in order to adjust the level of details on the display. By such hiding of irrelevant parts of specifications, a modelling tool may present SPL as a very simple language to use.

Error-proneness

Error-proneness is concerned with mistakes that are the result of the difficulties to correctly use the language, introduced by the language design.

SPL was designed to be modular, with specifications that consist of skeletons and shared building blocks. Correctly specified building blocks are used and reused across a specification for the creation of bigger components. Language constructs are assembled, not written, so the probability of making syntax errors is minimal. SPL implements strict static typing, which makes design time error checks easy for a modelling tool. Strict syntax checks should be supported by a modelling tool at design time, together with error explanations. Such functionality was easily implemented by PolicyModeller (see Chapter 6).

Semantic errors are much harder to detect, especially at design time. In SPL, semantic errors may arise from misunderstandings of semantics attached to building blocks that are used to construct larger specification elements. For example, semantics of management operations, which are the building blocks given by a managed system, may be sometimes vague. Because they are not designed by the user himself, the probability of misinterpretation is considerable, especially when it comes to the possible side-effects of their execution within the system. However, this problem is related to the design of management interfaces, and it cannot be addressed by the policy language itself. The use of Web services for management operations largely reduces this type of problem, because Web services essentially represent explained behaviour.

Hard mental operations

In most of the languages it is possible to write ambiguous specifications that are grammatically and semantically correct. To understand what exactly is meant by a specification represents a 'hard mental operation'. Two issues can be responsible for hard mental operations in a language: bad notations, which express concepts inappropriately, and combinations of bad notations, which increase the difficulty of understanding.

SPL language constructs are chosen to have a straightforward semantics, and their combinations are strictly controlled by the language syntax. The only aspect of the language that may become a source of interpretation problems is the universal problem of predicate combination and nesting. Different combinations of AND and OR predicates may be hard to understand, but a language shouldn't be too restrictive regarding this matter because it could affect its expressive power. It is on the user to carefully design policy predicates to be simple and clear to understand.

In general, structuring of the Dictionary information, especially in large models, may be seen as the hardest part of the SPL modelling process. However, it greatly depends on the user's experience with general modelling techniques, object-oriented methodology in particular. Also, planning for a multi-step abstraction in Dictionaries may be a hard task for an inexperienced user. In order to avoid plan composition problems (difficulties in putting the pieces of a specification together) and to suggest an order of specification development, we are proposing an informal design methodology for modelling in

SPL (see 4.2.1). This can be very helpful for novice users of SPL, especially if they are without software modelling experience.

Hidden dependencies

A hidden dependency is a relationship between two language elements that exists, but is not fully visible.

All dependencies and relations within a SPL specification are explicitly created by the user. The only dependencies that may be considered as hidden are those that come as a result of reuse. All the elements and their relations will be uniformly reused across a specification, because it is an essential property of any consistent specification. For example, a strategy specified for a management operation will refine every occurrence of that management operation in all policies, and the user must be aware of that. As a result of reuse, the rules of referential integrity must strictly apply to a specification at design time, which sometimes may be annoying (for example, when trying to delete constructs referenced from other parts of a specification).

Premature commitment

The premature commitment refers to the amount of planning necessary for using the language, and the cost of making errors, which may range from a simple change to a complete repeat of several specification steps. Some languages may force the user to make decisions before appropriate information is available. It usually comes as a result of many internal dependencies in the language, a supporting tool that constrains the order of doing things, and an inappropriate order of required steps envisaged by the language.

In the case of SPL, the problem of planning ahead depends only on the language implementation. The language itself doesn't impose any commitment, apart from the rule that only fully specified language constructs may be used as building blocks in other constructs. In general, the user will need to understand the management capabilities of the system through Dictionary modelling before being able to create policies to constraint them.

Progressive evaluation

Progressive evaluation is the ability to use a partially written specification, mainly for the purpose of testing.

A general rule is that every SPL construct must be syntactically complete in order to be legal. In SPL, types provide a way of controlling progressive evaluation, by partially verifying specifications at each stage. Since type checking is mechanical, one can guarantee, for a well designed language, that certain classes of errors cannot arise during execution, hence giving a minimal degree of confidence after each change [Cardelli 1991].

Dictionaries may be progressively specified, as well as mappings among them. Parts of Dictionaries which are specified may be immediately used for policy and strategy specifications. The refinement and triggering of policies require that all corresponding strategise are fully specified, to enable policy refinement all the way down to the management operations implemented in a real system. An activate/deactivate mechanism may be implemented within a modelling tool to make parts of a specification operational or not, like it was done in PolicyModeller (see section 6.2.1.1). However, only syntactically correct language constructs should become active.

Role-expressiveness

Role-expressiveness addresses the issue of readability. It refers to the ability of someone who did not write a specification to understand its meaning. Semantics should follow directly from the syntax - form of a statement should strongly suggest what the statement is meant to accomplish. Domain experts should be able to read and understand existing descriptions and to use the description framework to create new descriptions, while those who are not experts in the domain should be able to understand at least parts of existing descriptions, although they may not be able to write new ones.

In SPL names exist only for the users' convenience, and they can be freely used to additionally explain the semantics of corresponding language constructs. For example, names for management operations may be selected to describe the nature of those interactions in a natural language. This enables creation of mental models of a managed system using a terminology of users' choice, while still preserving the formality and executability of policy specifications. Regardless of the choice of names, policies will be always refined to the appropriate operations, which can be triggered against the system management interface.

The IF-THEN-WHILE policy form was carefully selected, because of intuitiveness of its classical rule structure. Also, semantics of policy specifications is additionally explained via policy metadata and attributes. With policy metadata and attributes, policies in SPL become completely self-described. Strategies appear to have the appearance of a pseudo-code, which may include well-known IF-THEN-ELSE and WHILE-DO control structures. Dictionaries organize information in tree-like hierarchies, which is a common way of structuring system models. Moreover, logically similar Dictionary elements may be grouped in classes for easier understanding.

Secondary notation

The readability of programs is immeasurably more important than their writeability [Hoare 1989]. Writability describes how easily a language can be used to create specifications for a chosen problem domain. The syntactic features that make a program easy to write are often in conflict with those features that make it easy to read. Writability is enhanced by use of concise and regular syntactic structures, whereas for readability a variety of more verbose constructs is helpful [Pratt 2001]. In order to improve writability, a secondary notation can be supported by a language to enable adding information that does not belong to the actual language semantics. The most common forms are the use of comments, indentation and white space.

The use of comments and visual layout of SPL specifications are purely implementation-dependent. The language itself doesn't restrict the layout of specifications in any way, and it is up the implementation to make distinction between the language syntax and additional comments. The language itself is simple enough to be visually represented in a modelling tool, which was proven by the implementation of PolicyModeller. Moreover, every construct in a model may have associated an informal textual description as one of its properties.

Viscosity

Viscosity is the resistance of the language to local changes. It refers to the amount of work required to make a small change in a specification. In general, the viscosity may be improved by a good modelling tool support, but it also largely depends on the nature of changes being made.

In the case of SPL, a user's impression of viscosity is very dependent on the actual language implementation and a modelling tool support. We imagine that an implementation of the Copy/Paste operation for the major elements of the specification, like it was done in PolicModelar (see section 6.2), should be enough to release the full power of the language. The language itself is designed to be modular and with a high potential for reuse. Specification building blocks are centrally defined and shared through the specification. Components of a specification are assembled from predefined building blocks, not written. As a result, the effect of changes, which must respect general referential and integrity rules defined by the SPL information model, propagates to all parts of the specification immediately.

Visibility

Visibility refers to ability to browse a specification without significant amounts of cognitive work. In particular, the need to see different parts of the specification simultaneously, for comparison. This feature is also largely implementation-dependent, and it becomes more important as the specification grows.

In the case of SPL, this dimension is heavily affected by a tool support. We imagine that a Find operation with multiple search criteria, implemented for Dictionary and Policy Repository trees, should be enough to enable fast access to specification elements. SPL supports presentation of a Dictionary content from two viewpoints (structured and classified), for the purpose of an easy navigation. In PolicyModeller, for example, the design environment has a possibility to simultaneously display two Dictionaries (same or different), and each of the displays has tabs for fast switching between structured and classified views. Moreover, policies and strategies in a Policy Repository may be classified for faster access. Finally, a user can see properties (name, ID, attribute values, etc.) for every element of a specification in a separate window display.

7.2.3 Potential issues with SPL

In this section we will point out some issues that might arise in practical use of the language.

7.2.3.1 Instantiation control

In low-level models, which try to capture internal structure of systems, abstract policies that are specified for multiple classes of elements may produce a combinatory explosion and an illegal outcome. We will examine this issue using the following sample policy:

Printers that are out of toner should be ignored by printer servers.

IF
Printers.IsOutOfToner() = TRUE
THEN
PrinterServers.Ignore(Printers)
WITH

If we have a Dictionary with classes Printers and PrinterServers, defined as shown on Figure 7-3, the previous abstract policy will be instantiated for every possible combination of class members from the two classes it references. This will result in 3 x 8 = 24 concrete policies.

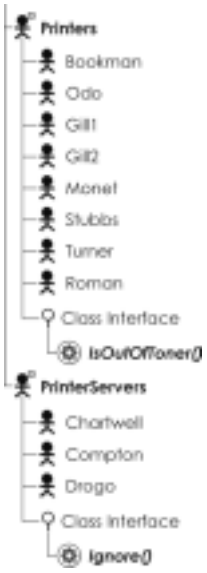


Figure 7-3 Example Printers and PrinterServers classes

The problem is, however, that a printer may be simultaneously managed only by one printer server. Therefore, 16 out of 24 instantiated policies would be illegal, because they would constraint a behaviour that does not exist in the real system. For example:

In this example, we will assume that Bookman printer is managed by Chartwell printer server, so the following instantiated policy will be legal:

When Bookman is out of toner, Chartwell shall ignore it.

IF
Bookman.IsOutOfToner() = TRUE
THEN
Chartwell.Ignore(Bookman)
WITH

The following policy, also produced by the instantiation of the abstract policy, will be illegal, because Compton does not have any authority over Bookman:

When Bookman is out of toner, Compton shall ignore it.

IF
Bookman.IsOutOfToner() = TRUE
THEN
Compton.Ignore(Bookman)
WITH

In order to avoid such type of problems, a great deal of knowledge about the system internal structure and behaviour would be needed. In our example, it would be necessary for the management framework to know which associations between printers and printer servers do really exist. Solution to this type of problem may be:

- Allow references only to members of one class per policy. This simple solution was implemented in PolicyModeller. As a result, the abstract policy from our example would be illegal, and a user would need to separately specify 8 concrete policies to handle all the 8 printer-server combinations.
- Implement an additional mechanism of relations among class members. In such a case, user will use his knowledge of the real system to specify legal combination among class members – constraints on policy instantiations. However, the question is whether or not it would be easier to create such relationships or to simply specify all the required policies manually.

7.2.3.2 Delegation

Another issue with SPL is related to the concept of delegation of authority and tasks. Currently, SPL supports only delegation of tasks on a per-action basis. As a result, delegation specified for one action may not be reused in other actions. The importance of delegation is closely related to the importance of structural modelling and to envisaged level of abstraction within a management model. As the model becomes less abstract (more detailed), the need to specify delegation will increase.

SPL was designed especially for a high-level policy-based management, with management models that represent a simplified management view of systems. In such simple models, the importance of delegation is minimal, because models will encompass a relatively small number of elements. However, SPL may be slightly extended to support delegation in order to fully support lower-level management scenarios. In such cases, user may want to model a real system organization, and to specify delegation of authority and tasks among system components.

For the purpose of low-level management, the following SPL type may be introduced:

```
Assignment ::= nameID,
               [pointer("Actor") ⇓ ] pointer("Actor"),
               pointerList("Action"),
               [predicateList],
               delegation | authorization
```

The Assignment type would enable specification of delegations of tasks and authorities in a management model. A delegation happens between two actors, and it could include a number of actions. Moreover, delegation is conditional, subject to satisfaction of certain conditions.

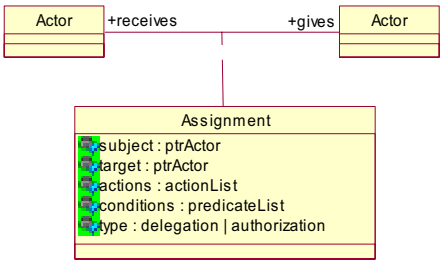


Figure 7-4 Abstract definition of the Assignment type

Assignment has two alternative modalities:

- *Delegation*, which represents delegation of tasks between actors. If conditions are satisfied, the target of a delegation will perform delegated tasks for the subject of delegation. If conditions are not satisfied, delegation will be ignored.
- *Authorization*, which represents a delegation of authority (between actors) to perform tasks. If conditions are satisfied, authorization will be granted. Otherwise, the authorization will be revoked.

All such assignments in a model are global, and they must be enforced in all policies. When a policy is triggered, the management process will need to check all specified assignments against the policy. If any of the authorizations is disrespected, the policy will be ignored. When it comes to the delegation, all the actions that should be initiated by a subject of delegation will be reassigned to the target of the delegation. For example, if actor A has delegated action X() to actor B, every occurrence of action A ⇓ X() in a policy will be replaced with B ⇓ X().

The question is, however, if all this is really needed, and what is the price of implementing this additional feature in SPF. This is related to the question if the problem of delegation is a policy concern at all, or if it would be better to leave this problem to some other form of control (configuration, for example). If a delegation of tasks and authorities is envisaged to be conditional, it seams reasonable to believe that it might be a policy problem. However, unconditional delegation is not really something that policy-based management should be worried about.

7.2.3.3 Parameterization

Parameterization of the policy and the strategy types in SPL may be useful to empower the reuse within a specification, and to simplify management models. For example, it is likely that different abstract operations might have same refinement patterns, only with different values for some constant parameters in the strategy body. In such cases, parameterization of the strategy type will introduce additional simplification in the refinement process.

For instance, we might have two abstract operations that trigger load balancing based on different load limit. For each of these operations, a separate strategy will be needed, in order to enable their refinement.

```
        SensitiveResponse()
EQUIVALENT
        IF
                GetLoad() > 40%
        THEN
                LoadBalancing()
USING
```

```
        ImmediateResponse()
EQUIVALENT
        IF
                GetLoad() > 80%
        THEN
                LoadBalancing()
USING
```

Those two strategies may be generalized in only one, by parameterization.

```
        Response [p] ()
EQUIVALENT
        IF
                GetLoad() > p
        THEN
                LoadBalancing()
USING
```

In the course of policy refinement, the same parameterized strategy may be used to refine each of these two abstract operations:

```
SensitiveResponse()

        ⊥ refinement  Response[40%]()

IF
        GetLoad() > 40%
THEN
        LoadBalancing()
```

```
ImmediateResponse()

        ⊥ refinement  Response[80%]()

IF
        GetLoad() > 80%
THEN
        LoadBalancing()
```

When it comes to policies, it is less likely that additional reuse mechanisms will be needed, especially for high-level management models. It is expected that high-level policies will be few in numbers and mainly with exclusive semantics, so the reuse may not be possible at all. In low-level management models, policy reuse may be important, but the concept of classification offers sufficient opportunities for reuse through the mechanism of policy instantiation.

Policy and strategy parameterization may be the features that can be added to the language in the course of its evolution. However, those features may come only at the price of additional complexity in both SPL and SPF.

7.2.3.4 Conflict analysis

The conflict analysis mechanism that uses only the policy metadata is not absolutely precise, and it might happen that some policies that are not in conflict will be discarded. However, if the need for those policies remains, they will be triggered in the next triggering cycle as well. Therefore, in the worst case scenario, the system may experience a slight delay in the management response.

A more precise conflict resolution mechanism would need to consider the semantics of all policy management operations, and to be able to compare their intentions.

To check for potential conflicts between a policy with n management operations and a policy with m management operations, up to m x n comparisons of the following type will be needed (pseudo-code):

```
for(1<i<n)
        for(1<j<m)
            if(Action_i.Target = Action_j.Target & Action_i.Intention = ! Action_j.Intention)
                ConflictDetected();
```

Interpreted policies in SPL may include IF-THEN-ELSE and WHILE-DO structures, so the number of management operations that will be actually executed depends on the current system state. Only the management operations that will be executed need to be compared.

The SPL syntax enables an additional external specification to be attached to policy management operations (using the path member of the Action type, see 5.1.2). Such specifications will need to specify intentions of all policy management operations in the system in a sufficiently general way to enable their comparison. A simple comparison of modalities wouldn't be enough, since it does not necessary means that both operations are referring to the same behaviour.

For example, a metadata would be required to specify that *GrantAccess()* and *DenyAccess()* are operations with the opposite intentions. We could describe the semantics of the management operations in the following way:

Name = GrantAccess
FunctionalScope = Access Control/Access Authorization
Modality = A+

Name = DenyAccess
FunctionalScope = Access Control/Access Authorization
Modality = A-

The SPF conflict resolution mechanism may implement the following rule to determine the compatibility of the intentions of policy operations (pseudo-code):

```
if(Action1.FunctionalScope = Action2.FunctionalScope & Action1.Modality = ! Action2.Modality)
        IncopatibilityDetected();
```

7.3 Conclusion

This chapter has presented a discursive evaluation of the conceptual policy framework and the policy specification language proposed by our research. SPF has been evaluated with respect to several criteria that usually apply to software systems. SPL has been evaluated with respect to a number of cognitive dimensions, each of which addresses different aspects of the language usability. Potential issues that may arise in the practical use of the proposed concepts were identified, together with suggestions about how they might be dealt with.

8 CONTRIBUTION AND CONCLUSIONS

In this chapter we will summarize our work on policy-based management, describing the contribution made and suggesting areas for future work. We will briefly review the initial goals of this research and identify what has been achieved. We will conclude by identifying possible avenues of research that would be of clear benefit for the policy framework and the language proposed by the research.

8.1 Summary of research goals

Requirements that management goals put on the behaviour of distributed systems today tend to be complex, sophisticated and to change rapidly. As a result, lots of the behaviour required at run-time is unknown at design time. The more complex distributed systems get the bigger part of decisions about their behaviour needs to be postponed for later. The role of the development process is to identify the need and to envisage mechanisms that enable such dynamic decisions at run-time. We see management policies as the best way to seamlessly integrate dynamic management goals into the behaviour of distributed systems. The ambition of this thesis is to demonstrate the new application possibilities that policy-based management presents by providing the necessary supporting framework and better understanding of the background itself.

At the beginning of this research, our impression was that the existing policy specification languages lacked the power to express complex management models in a sufficiently abstract way that enables high-level system management. Moreover, we came to a conclusion that the policy-based management needs much more than just a way to specify policies, and that existing policy-based management solutions lack completeness. This research was particularly motivated by the need for a high-level solution for the problem of management of large distributed systems. Goals of the project were derived directly from deficiencies of the existing solutions, and with regard to the general requirements for the policy-based management. These research goals were identified as follows:

- Address the problem of management complexity in large distributed systems.
- Provide enough flexibility to support ongoing system changes.
- Enable automation of the management process.
- Preserve independence of management from the managed system.
- Deliver concise but expressive policy specification.
- Assure formality of policy specifications.

8.2 Achievements

As a result of achieving research goals, the following contribution was made to the field of policy-based management:

- *A complete conceptual framework for policy-based management.* Architectural solutions proposed by the framework address the problems of flexibility and independence of high-level policy-based management.

- *A complete policy-based management process*. Operational solutions proposed by the framework address the problems of flexibility, adaptability and automation of high-level policy-based management.
- *A formal, but flexible and expressive policy specification language*. The language design addresses the problem of complexity of the framework information model, and problems of conciseness, expressivity and formality of high-level policy specifications.

In the rest of this section we will look more closely at each of these achievements and present their unique features.

8.2.1 Management framework

Comparing to the existing policy-based management frameworks, presented in the Chapter 2 of this thesis, unique qualities of SPF may be identified as completeness, compatibility, interoperability, adaptability, and system independence. Moreover, as a result of the system independence, the framework has gained the qualities of portability, reusability and scalability.

Completeness is the most important quality of SPF. It comes from the fact that the framework components and processes are built to support the entire policy life-cycle. The conceptual framework design has included definition of the following elements:

- *Basic roles* of the management process, whose scope was adjusted to the task of high, system-level management of distributed systems.
- *Information model*, defined by SPL, which includes support for all mechanisms required to support the management process.
- *Flow of control and data* within the management process.

Compatibility with the latest standards comes from the design of SPF. The management framework was designed to fit perfectly into the latest trends and standards in the domain of distributed systems modelling, design and management. Links to available implementation technologies are well established, in order to avoid the problem of isolation from other products and technology trends.

Interoperability with the existing distributed systems architectures and policy-based management concepts is based on the framework compatibility, but it depends on the extent to which those concept respect standards. SPF was specially designed to complement the existing policy-based management solutions, and to be able to interface a vide range of distributed systems architectures, both emerging and legacy ones.

Adaptability of the framework is based on its capability to discover management behaviour. SPF is capable of detecting and adjusting to the particular system management capabilities, and to dynamically adapt as they evolve. Moreover, modularity of the SPF design enables its easy extension by plugging-in additional components to support new steps in the management process.

System independence comes from the fact that SPF was designed for loose coupling with the managed system. In effect, the framework does not directly manage a system – the system manages itself, while the

framework only gives (delegates) management instructions (policies). Such an approach protects the privacy and promotes the security of the system by empowering its role in the management process.

The SPF management model recognizes the need to implement control at enterprise level by modelling and managing the system as a whole. The framework stands back from the kinds of detail that make real distributed systems so diverse, to focus on the high-level aspects that make them similar. The major goal of the SPF design was to address the problem of complexity by managing a system as a whole, strictly from the outside, and in a generic way. The only knowledge about the system comes from the discovery of its management interface. Management systems built on top of a management interface benefit from three main qualities:

- *Portability* comes from isolating the framework from the system implementation details. The basic vocabulary of SPL is defined by the system management interface.
- *Scalability*, just like portability, comes as a result of independence from the system internals. Moreover, the capacity for vocabulary abstraction enables handling of the management models at literally any level of complexity. Finally, the lightweight nature of the SPL notation causes fewer problems with scale up than more detailed notations.
- *Reusability* comes from making the abstraction general-purpose. In SPF/SPL, policy-based control is based on a general-purpose concept of interrelating management operations in order to create behavioural constraints.

8.2.2 SPF management process

The fact is that existing policy-based management frameworks do not envisage policy-based management as a process. Rather, they mainly focus on policy triggering in isolation. However, since the idea of management is based on continuity of management operations in time, the SPF management process has been defined with properties of completeness, flexibility, autonomy, and continuity.

Completeness of the management process comes from the fact that it includes full support for policy triggering, both proactive (human-based) and reactive (event-based), policy refinement, policy conflict and semantic analysis, and control, which is based on policy post conditions and mechanism of exceptions (management events). The control implements a run-time monitoring (goal checking) on both system and policy performance, and performs policy-based corrective actions.

Flexibility of the overall management process comes from the fact that all processes within the management framework are controlled and guided by rules, and are based on the actual system state.

Autonomy of the management process is based on the built-in control mechanism, which enables automation and adaptability of the framework management operations. Autonomy of the SPF management process is based on automation of the policy refinement, automation of the management response, executability of policy specifications, and presence of a built-in control mechanism.

Continuity of the management process is one of its most important qualities. Management processes will continue until the management goals are achieved, by using ad-hoc flows of policy executions. The

enrolment of the management process is guided by the actual system state and the outcome (success or failure) of executed policies. Policy may be seen as specification of a semi-structured management process (workflow), with a predefined macro structure and management operations as their building blocks, which fit into the overall management process. This is a high-level, process-oriented approach to distributed system management, which is in contrast to the existing low-level, resource-oriented solutions. The separation from the underlying system guaranties simplicity (as a result of a high-level approach) and flexibility (as a result of loose coupling) of management operations.

8.2.3 Policy specification language

Within the context of existing policy specification solutions, unique features of SPL may be identified as completeness, formality, separation of concerns, design time verifiability, adaptability, scalability, and expressiveness.

Completeness. The language defines the entire information model for SPF and supports all steps of the policy life-cycle: triggering, instantiation, refinement, conflict and semantic analysis, and exception handling. Moreover, an informal specification methodology was defined for building complex system specifications to avoid the problem of poor guidance, often presented in formal specification techniques.

Formality. SPL has a formally defined grammar and employs a strict static typing. The language is modular, with strictly defined form for all its constructs, and with precisely defined vocabulary of shared building blocks. Policy specifications are assembled from predefined building blocks, not written, which enables extensive semantic analysis, simpler language syntax, and reuse of building blocks.

Separation of concerns. SPL emphasizes strict separation of concerns by enforcing a clear distinction between modelling of managed system (descriptive properties) and specification of constraints (prescriptive properties). Modelling of the system behaviour defines the language vocabulary, and it is logically separated from specification of constraints on that behaviour (policies). Policies only refer to, but not include, externally defined and shared vocabulary of available management operations. As a result, changes made to the vocabulary immediately propagate to all parts of the specification. Such an approach empowers the consistency and formality of SPL specifications. This explicit distinction between modelling of the system behaviour and the policy specifications results in system-independent policy specifications, which are:

- *Abstract*, because the system behaviour is represented symbolically.
- *Dynamic*, because policy specifications are decoupled from the actual behavioural specifications. Policy specifications are constraints on the symbolic system behaviour. Thus, the actual system behaviour, which is defined externally to policy specifications, can be changed without affecting the existing policies.

Design time verifiability. The presence of a vocabulary, which reflects the real system management capabilities, enables design time verifiability of policy specifications. Policy specifications are formalized in both form and content. The form of policies is strictly prescribed by the language syntax, while the policy content is firmly restricted to the precisely defined vocabulary. Policies are assembled by putting

elements from a predefined vocabulary into a predefined form. As a result of the discovery and formal abstraction mechanism, SPL vocabulary guaranties compatibility of specified policies with the capabilities of a managed system - anything defined in the vocabulary can be used in policy specification, but nothing else. Therefore, the validity of a policy specification may be guarantied even at design time.

Adaptability. The language vocabulary is system-specific, and it evolves together with the system management capabilities. SPL management vocabulary is based on a dynamic model of the system. In [Maullo 1993], Maullo and Calo emphasize the need for real-time evolution of the management vocabulary – vocabulary based on a dynamic discovery of the system management capabilities. The syntax of SPL is fixed, but its vocabulary is open and unlimited. The rules on how to create vocabularies are defined, but not the content itself. The vocabulary of SPL consists of discovered management operations. Just like natural languages, which evolve on-the-fly, SPL grows with its vocabulary and together with the managed system. The vocabulary of the policy specification language dynamically reflects the management capabilities of a system, and it may grow both quantitatively (introduction of new management operations) and qualitatively (introduction of better management operations).

Scalability. Complexity presents a strong challenge for policy notations. Policy languages that require commitment to a lot of detail will produce specifications that hide the big picture behind the clouds of details as the scale of a system increases. The lightweight nature of SPL specifications causes fewer problems with scale up than more detailed notations. SPL is capable of facing the problems of any complexity, thanks to its abstraction mechanism. It offers a rich abstraction/refinement support, in order to enable modularization and incremental development, as well as creation and use of high-level, problem-oriented data structures. The abstraction mechanism relies on two instruments:

- *Hierarchy of interrelated viewpoints.* A management model can include multiple levels of abstraction, each of which is specified as a Dictionary. The lowest-level (basic) Dictionary models the real management behaviour of a system, while other Dictionaries abstract that behaviour. Policies can be specified using the elements from any Dictionary in the hierarchy, even using the elements from different Dictionaries. After triggering, every policy specification will be refined into a specification in terms of the real management behaviour of the system.
- *Abstraction mechanisms within viewpoints.* Every Dictionary itself supports abstractions of aggregation/decomposition, classification/instantiation, and generalization/specialization.

Expressiveness. The expressive power of SPL comes from the close mapping between mental model of the problem and its specification, as well as from intuitive language constructs and naming convention that reflects the commonly agreed terminology from the problem domain.

Properties of SPL policy specifications may be summarized as follows:

- *Simple*, as a result of both expressive power of SPL and modularity and simplicity of its syntax.
- *Modular*, as a result of specification assembled from predefined building blocks.
- *Self-explained*, as a result of intuitive policy form and associated rich metadata and policy attributes.
- *Executable*, as a result of dynamic mappings of specification building blocks to the real system behaviour.
- *Unambiguous*, as a result of the language formality.

∉ *Declarative*, as a result of implementation-independent nature of policy building blocks.

8.3 Future work

There are several aspects where SPF and SPL would benefit from further investigation. Some of these aspects have already been discussed in previous chapters. The reminder of this section will revisit these and try to point at some other areas where further work may be beneficial.

8.3.1 Management framework

SPF was designed as a conceptual framework for policy-based management. In order not to be too restrictive, implementation details, together with some design decisions, were intentionally left out of the picture. This allows framework independence from the actual software technologies, which evolve rapidly, and helps avoiding premature design solutions. However, in order to be implemented, SPF must be refined to meet particular management goals.

8.3.1.1 Pilot implementation of SPF

A complete SPF implementation would enable operational testing and reverse engineering of the framework. The most important outcome of a performance testing would be an estimation of the price paid for the flexibility and system independence of SPF. Such an analysis would be valuable for the future applications of the management framework, because it would offer guidelines about how to make good compromises between flexibility and system independence on one side, and performances on the other side.

Reverse engineering would be another benefit from a SPF implementation. Based on the performance analysis of the pilot implementation, it would be possible to detect bottlenecks and sources of overhead, to find solution to those problems and to abstract it back to the conceptual framework.

Apart from the functionality and performance, we expect that the key to a practical success of SPF will be its GUI, because it will be directly responsible for users' impression of both the framework and the policy language. Although the PolicyModeller project did address some important problems of SPF/SPL visualization, more can be done to realize potentials of the management framework and the policy language at their full power. For example, GUI could be successfully implemented to be DHTML-based, which would make it lightweight and portable, and which would enable remote access to the framework literarily from everywhere.

8.3.1.2 Refinement of SPF

Since the conceptual framework was targeting the essential aspects of the policy-based management in distributed systems, some of the general design issues, which would normally be part of an application

design, were left out of the picture intentionally. As a result, further refinement of SPF may be useful, to make the framework more attractive and easier for implementation.

In general, the framework refinement process should be split in two steps, in order to make a distinction between the nature of introduced details:

- *Technology-independent refinement*, which should include instruments to support monitoring, security and performance measurements. The new instruments should be introduced at a conceptual level, in order not to compromise implementation independence of SPF.
- *Technology-specific refinement*, which should produce more detailed design that relies on the full potential of Web services and SOA on one side, and DEN on the other. These refinement efforts would target the framework interface with the managed system, more precisely functionality of the SMS Broker and Event Listener.

8.3.2 Extension of SPL

One of the very important features of every good concept is its relations to existing solutions and well established standards in the field. In addition to already defined mappings to CIM/DEN via PCIM (see Appendix A), SPF would particularly benefit from a precisely defined relation to the RM-ODP Enterprise viewpoint. It would be particularly interesting to explore if and to what extent SPF can be seen as a refinement of the RM-ODP Enterprise viewpoint, and if and to what extent SPL can be used for system modelling from the RM-ODP Enterprise viewpoint. A convergence towards a full support for the RM-ODP Enterprise viewpoint could be seen as one of natural paths of evolution for SPL.

Design of a full graphical notation for SPL would be an important step towards further improvement of its expressive power. This work would need to include two aspects:

- *Representation in UML*. Based on our experience with examples presented in Appendix C, we envisage that UML class diagrams would be particularly useful in the initial stage of a system modelling. Therefore, we believe that a set of stereotypes for the UML class diagrams would be very helpful. Such a set of stereotypes already exists (see Figure 9-10), and it was used to represent Dictionaries in our examples. However, we expect that a closer look at this problem would be beneficial.
- *Representation in a dedicated policy modelling tool*. The PolicyModeller project has made some efforts in this direction, but there are opportunities for improvement. A successfully created visual representation of SPL would definitely help in making the entire concept more useful and attractive.

In section 7.2.3 we have investigated potential issues that may arise while working with SPL. As a result, two of the possible extensions of the SPL syntax were indicated:

- Support for the relation of generalization/specialization via parameterization of the Policy and Strategy types. This could help reducing the number of policies and strategies, as well as the number of their building blocks in a model. However, we believe that benefits from such an extension are questionable, especially for the high-level management, and that a detailed analysis of this problem is needed.
- Support for authorization and delegation of tasks. This extension is more likely to be useful, because some of the experts in the field believe that problems of delegation and authorization are an

important issue in policy-based distributed systems management. Although we agree on the importance of authorization and delegation in general, the question is to what extent these are policy-related problems. Also, an extension of SPL with a support for delegation would require alterations of the SPF management framework.

One should bear in mind that extensions to SPL syntax come at a price of complexity. We didn't include such extensions in our policy language simply because one of the most important goals of this research was to come up with a simple, minimal solution. Any extension of SPL should be very carefully investigated and its effect on SPF and the management process vigilantly measured in order to assure that benefits are worth the additional complexity.

8.4 Closing remarks

Through this thesis we have explored the concept of management policies and explained its application to distributed systems management. We have investigated the need for high-level, system-wide management operations in large and complex systems, and explore various techniques for their representation, simplification and use.

Our work on the problem of high-level management in distributed systems has shown that there are lots of complexities and challenges in this field. The need for a powerful set of instruments for high-level distributed systems management is evident, and it will grow with the size and complexity of new, emerging systems and technologies. We believe that our work has demonstrated a number of new application possibilities which policy-based management presents. We have defined a necessary supporting management framework and offer a better understanding of the background itself, which will encourage and enable the application of policy-based management in large-scale distributed systems. We expect that our research will lead to a better understanding of policies and provide means with which policy-based management can be more widely and easily utilised.

9 BIBLIOGRAPHY

[Agrawal 2001] Agrawal R., Bayardo R.J., Gruhl D., Papadimitriou S., *Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications*, In Proceedings of the 10th International World Wide Web Conference WWW10, pp. 355–365, Hong Kong (2001)

[Aiken 2000] Aiken B., Strassner J., Carpenter B., Foster I., Lynch C., Mambretti J., Moore R., Teitelbaum B., *Network Policy and Services: A Report of a Workshop on Middleware*, IETF Network Working Group Request for Comments, RFC 2768 (2000)

[Anderson 1999] Anderson B. B., *Identification and Estimation of Software Quality Factors' Effects on the Evaluation of New Computing Architectures*, Heinz School of Public Policy and Management, Carnegie Mellon University, Pittsburgh, Pennsylvania (1999)

[ANSI/IEEE 1992] Institute of Electrical and Electronics Engineers (IEEE), *Standard for a Software Quality Metrics Methodology*, ANSI/IEEE Standard 1061-1992 (1992)

[Bailey 2000] Bailey A., Inverso J. F., *Network Management Systems: Perspective*, Gartner Research Report (2000)

[Bakker 2000] Bakker J. L., McGoogan J. R., Opdyke W. F., Panken F., *Rapid development and delivery of converged services using APIs*, Bell Labs Technical Journal, Vol. 5, No. 3, pp. 12-29 (2000)

[Balcer 2002] Balcer M., Mellor S., *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley (2002)

[Barbacci 1995] Barbacci M. R., Klein M. H., Longstaff T. H., Weinstock C. B., *Quality Attributes*, Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania (1995)

[Barbacci 1997] Barbacci M. R., Klein M. H., Weinstock C. B., *Principles for Evaluating the Quality Attributes of a Software Architecture*, Technical Report CMU/SEI-96-TR-036, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania (1997)

[Barros 1997] Barros A.P., ter Hofstede A.H.M., Proper H.A., *Essential Principles for Workflow Modelling Effectiveness*, In Proceedings of the Third Pacific Asia Conference on Information Systems (PACIS'97), pp. 137-147, Brisbane, Australia, (1997)

[Bauer 1993] Bauer M.A., Finnigan P. J., Hong J. W., Rolia J. A., Teorey T. J., Winters G., *CORDS Distributed Management*, In Proceedings of the Third Centre for Advanced Studies Conference (CASCON'93), pp. 27-40, IBM Centre for Advanced Studies, Toronto, Canada (1993)

[Bauer 1994] Bauer, M.A., Coburn, N., Erickson D. L., Finnigan P. J., Hong J. W., Larson P.-A., Pachl J., Slonim J., Taylor D. J., Teorey T. J., *A distributed system architecture for a distributed application environment*, IBM Systems Journal , Vol. 33, Issue 3, pp. 399–426 (1994)

[Bauer 1994a] Bauer M. A., Bochman G., Coburn N., Erickson D. L., Finnigan P. J., Hong J. W., Larson P.-A., Martin T. P., Mendelzon A., Neufeld G., Silberschatz A., Slonim J., Taylor D., Teorey T. J., Yemini Y., *The CORDS Architecture: Version 2*, IBM Centre for Advanced Studies, Toronto, Canada (1994)

[Bauer 1994b] Bauer M., Finnigan P., Hong J., Rolia J., Teorey T., Winters G., *Reference Architecture for Distributed Systems Management*, IBM Systems Journal, Vol. 33, No. 3, pp. 426-444 (1994)

[Bauer 1997] Bauer M. A., Bunt R. B., Rayess A. E., Finnigan P. J., Kunz T., Lutfiyya H., Marshall A. D., Martin P., Oster G. M., Powley W., Rolia J. A., Taylor D., Woodside C. M., *Services supporting management of distributed applications and systems*, IBM Systems Journal, Vol. 36, No. 4, pp. 508-526 (1997)

[Bearden 2001] Bearden M., Garg S., Lee W., *Integrating goal specification in policy-based management*, In Proceedings of the IEEE Workshop on Policies for Distributed Systems and Networks (POLICY 2001), Bristol, UK, Sloman M., Lobo J., Lupu E. (editors), Lecture Notes in Computer Science, Vol. 1995, pp. 153-170, Springer-Verlag (2001)

[Bench-Capon 1990] Bench-Capon T., *Knowledge Representation: An Approach to Artificial Intelligence*, Academic Press (1990)

[Bergin 1994] Bergin J., *Data Abstraction: the object-oriented approach using C++*, McGraw-Hill (1994)

[Biddle 1966] Biddle B. J., Thomas E. J. (editors), *Role theory: Concepts and research*, John Wiley and Sons (1966)

[Booch 1992] Booch G., *The End of Objects and the Last Programmer*, Addendum to the Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92), pp. 3-8, Vancouver, Canada, (1992)

[Burbeck 2000] Burbeck S., Graham S., *Creating target-rich environments in a service-oriented architecture*, IBM Whitepaper (2000)

[Cardelli 1990] Cardelli L., Leroy X., *Abstract types and the dot notation*, SRC Research Report 56, DEC Systems Research Center (1990)

[Cardelli 1991] Cardelli L., *Typeful Programming*, In Neuhold E. J., Paul M., (editors), *Formal Description of Programming Concepts*, Springer-Verlag (1991)

[Cardelli 1994] Cardelli L., Matthes F., Abadi M., *Extensible Syntax with Lexical Scoping*, SRC Research Report 121, DEC Systems Research Center (1994)

[Carzaniga 1997] Carzaniga A., Picco G. P., Vigna G., *Designing distributed applications with mobile code paradigms*, In Proceedings of the 19th International Conference of Software Engineering (ICSE'97), Taylor R. (editor), pp. 22-32, Boston, Massachusetts, ACM Press (1997)

[Casassa Mont 1999] Casassa Mont M., Baldwin A., Goh C., *Role of Policies in a Distributed Trust Framework*, Technical Report HPL-1999-104, HP Laboratories Bristol (1999)

[Casassa Mont 2000] Casassa Mont M., Baldwin A., Goh C., *POWER Prototype: Towards Integrated Policy-Based Management*, In Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'2000), Hong J. (editor), pp. 789-802, Honolulu, Hawaii (2000)

[Clark 1999] Clark E., *Directory-Enabled Networking*, Network Magazine, August 1999, http://www.networkmagazine.com/article/NMG20000508S0031

[Cockburn 2000] Cockburn A., *Writing Effective Use Cases*, Addison-Wesley (2000)

[Damianou 2000] Damianou N., Dulay N., Lupu E., Sloman M., *Ponder: A Language for Specifying Security and Management Policies for Distributed Systems*, Imperial College Research Report DoC 2000/1, Imperial College, Department of Computing (2000)

[Damianou 2001] Damianou N., Dulay N., Lupu E., Sloman M., *The Ponder Policy Specification Language*, In Proceedings of the IEEE Workshop on Policies for Distributed Systems and Networks (POLICY 2001), Bristol, UK, Sloman M., Lobo J., Lupu E. (editors), Lecture Notes in Computer Science, Vol. 1995, pp. 18-38, Springer-Verlag (2001)

[Damianou 2002] Damianou N., Bandara A. K., Sloman M., Lupu E. C., *A Survey of Policy Specification Approaches*, Submitted for publication, Imperial College of Science, Technology and Medicine, University of London, Department of Computing (2002)

[Damianou 2002a] Damianou N., Dulay N., Lupu E., Sloman M., Tonouchi T., *Tools for Domain-based Policy Management of Distributed Systems*, IEEE/IFIP Network Operations and Management Symposium (NOMS'2002), pp. 213-218, Florence, Italy (2002)

[Damianou 2002b] Damianou N. C., *A Policy Framework for Management of Distributed Systems*, PhD thesis, Imperial College of Science, Technology and Medicine, University of London, Department of Computing (2002)

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

[Dini 1999] Dini P., Mihai N., *Policy Framework: Specifying Policies Types and Active Policy Relationships*, Policy Workshop 1999, HP-Laboratories, Bristol, UK (1999)

[Farrell 2002] Farrell J. A., Kreger H., *Web services management approaches*, IBM System Journal, Vol. 41, No. 2, pp. 212-227 (2002)

[Fischer 1993] Fischer A. E., Grodzinsky F. S., *The anatomy of programming languages*, Prentice-Hall (1993)

[Follows 1999] Follows J., Application-driven networking: Concepts and architecture for policy-based systems, IBM Redbook (1999)

[Goh 1997] Goh C., *A Generic Approach to Policy Description in System Management*, In Proceedings of the 8th IFIP/IEEE international workshop on Distributed Systems Operations & Management (DSOM '97), pp. 1-12, Sydney, Australia (1997)

[Goh 1998] Goh C., *Policy Management Requirements*, Technical Report HPL-98-64, HP Laboratories Bristol, UK (1998)

[Gottesdiener 2002] Gottesdiener E., *Top Ten Ways Project Teams Misuse Use Cases, and How to Correct Them*, Rational Edge e-zine, Jun 2002, http://www.therationaledge.com/splashpage_jun_02.html

[Green 1996] Green T. R. G., Petre M., *Usability Analysis of Visual Programming Environments: A 'cognitive dimensions' framework*, Journal of Visual Languages and Computing, Vol. 7, No. 2, pp. 131-174 (1996)

[Grosso 2001] Grosso P., *XML Scehmas and DTDs Working Together*, XML Journal, Vol. 2, No. 5 (2001)

[Halpin 1998] Halpin T., *UML data models from an ORM perspective: Par 1*, Journal of Conceptual Modelling (JCM), No. 1, April 1998

[Haneef 2002] Haneef A. M., *Web Services – Beyond the Hype*, Multimedia Networks Laboratory, University of Massachusetts (2002)

[Hoare 1981] Hoare C. A. R., *The emperor's old clothes*, Communications of the ACM, Vol. 24, No. 2, pp. 75-83 (1981)

[Hoare 1989] Hoare C. A. R., *Hints on programming language design*, In Hoare C. A. R., Jones C. B. (editors), *Essays in Computing Science*, Prentice-Hall (1989)

[Hoare 2000] Hoare C. A. R., Jifeng H., Sampaio A., *Algebraic Derivation of an Operational Semantics*, In Plotkin G., Stirling C., Tofte M. (editors), Proof, Language, and Interaction, MIT press (2000)

[Hofstadter 1999] Hofstadter D., *Gödel, Escher, Bach: An Eternal Golden Braid*, HarperCollins (1999)

[Huber 1997] Huber F., Schätz B., Einert G., *Consistent Graphical Specification of Distributed Systems*, In Proceedings of the 4th International Symposium of Formal Methods Europe (FME '97), Lecture Notes in Computer Science 1313, pp. 122–141, Graz, Austria (1997)

[ISO/IEC 1995] ITU Recommendation X.902 | ISO/IEC 10746-2, *Open Distributed Processing – Reference Model – Part 2: Foundations* (1995)

[ISO/IEC 1995a] ITU Recommendation X.903 | ISO/IEC 10746-3, *Open Distributed Processing – Reference Model – Part 3: Architecture* (1995)

[ISO/IEC 1996] ITU Recommendation X.901 | ISO/IEC 10746-1, *Open Distributed Processing – Reference Model – Part 1: Overview* (1996)

[ISO/IEC 1997] ITU Recommendation X.904 | ISO/IEC 10746-4, *Open Distributed Processing – Reference Model – Part 4: Architectural Semantics* (1997)

[ITU/ISO 1995] ITU/ISO, *Reference Model of Open Distributed Processing – Trading Function*, Committee Draft ISO/IEC DIS 13235 | Draft Rec. X.950 (1995)

[Jagadish 1998] Jagadish H. V., Mendelzon A. O., Mumick I. S, *Managing conflicts between rules*, Journal of Computer and System Sciences, Vol. 58, No. 1, pp. 13-28 (1998)

[Judd 1998] Judd S., Strassner J. (editors), *The Directory-enabled Networks - Information Model and Base Schema Specification*, Draft v3.0c5, DEN Ad Hoc Working Group (1998)

[Keller 1990] Keller S.E., Kahn L.G., Panara R.B., *Specifying Software Quality Requirements with Metrics*, In Thayer R.H., Dorfman M. (editors)., Tutorial: System and Software Requirements Enginering, pp. 145-163, IEEE Computer Society Press (1990)

[Kiczales 1992] Kiczales G., *Towards a new model of abstraction in software engineering*, In Proceedings of the International Workshop on New Models for Software Architecture (IMSA'92), Workshop on Reflection and Meta-level Architectures, pp. 1-11, Tokyo, Japan (1992)

[Kiczales 1996] Kiczales, G., *Beyond the Black Box: Open Implementation*, IEEE Software, Vol. 13, No. 1, pp. 8-11 (1996)

[Koch 1996] Koch T., Kramer B., *Rules and agents for automated management of distributed systems*, Distributed Systems Engineering Journal, Special issue on management, Vol. 3, No. 2, pp. 104-114 (1996)

[Kohli 1999] Kohli M., Lobo J., *Policy Based Management of Telecommunication Networks*, Policy Workshop 1999, HP-Laboratories, Bristol, UK (1999)

[Langsford 1993] Langsford A., Moffett J., *Distributed Systems Management*, Addison-Wesley (1993)

[Larochelle 2002] Larochelle D., Evans D., *Splint Manual*, Version 3.0.6, Secure Programming Group, Department of Computer Science, University of Virginia (2002)

[Letier 2002] Letier E., van Lamsweerde A., *Deriving Operational Software Specifications from System Goals*, In the Proceedings of the FSE'10 - 10th ACM S1GSOFT Symposium on the Foundations of Software Engineering, pp. 119-128, Charleston, South Carolina, (2002)

[Lewis 1996] Lewis L., *Implementing Policy in Enterprise Networks*, IEEE Communications Magazine, Vol. 34, No. 1, pp. 50-55 (1996)

[Lewis 2000] Lewis D., *A Review of Approaches to Developing Service Management Systems*, Journal of Network and Systems Management (JNSM), Vol. 8, No. 2, pp. 141-156 (2000)

[Linington 1995] Linington P., *RM-ODP: The Architecture*, In Raymond K., Armstrong E. (editors), Open Distributed Processing: Experience with Distributed Environments, pp. 15-33, IFIP, Chapman and Hall (1995)

[Linington 1998] Linington P., Milosevic Z., Raymond K., *Policies in Communities: Extending the ODP Enterprise Viewpoint*, In Proceedings of the 2nd International Workshop on Enterprise Distributed Object Computing (EDOC'98), San Diego, California, pp. 11-22 (1998)

[Linington 1999] Linington P., *Options for Expressing ODP Enterprise Communities and Their Policies by Using UML*, In Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC99), University of Mannheim, Germany, pp. 72-82, IEEE Publishing (1999)

[Lobo 1999] Lobo J., Bhatia R., Naqvi S., *A Policy Description Languag*e, In Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99), pp. 291-298, Orlando, Florida (1999)

[Lupu 1996] Lupu E., Marriott D., Sloman M., Yialelis N., *A Policy Based Role Framework for Access Control*, In Proceedings of the first ACM Workshop on Role-based access control, Article No. 11, Gaithersburg, Maryland (1996)

[Lupu 1997] Lupu E., Sloman M., *A Policy Based Role Object Model*, In Proceedings of the First International Enterprise Distributed Object Computing Workshop (EDOC'97), pp 36-47, Queensland, Australia, (1997)

[Lupu 1997a] Lupu E., Sloman M., *Towards a Role-based Framework for Distributed Systems Management*, Journal of Network and Systems Management (JNSM), Vol. 5, No. 1, pp. 5-30, Plenum Press Publishing (1997)

[Lupu 1997b] Lupu E., Sloman M., Yialelis N., *Policy Based Roles for Distributed Systems Security*, In Proceedings of the 4th Plenary Workshop of the HP OpenView University Association (HP-OVUA'97), pp. 1-12, Madrid, Spain (1997)

[Lupu 1999] Lupu E., Sloman M., *Conflicts in Policy-based Distributed Systems Management*, IEEE Transactions on Software Engineering, Special Issue on Inconsistency Management, Vol. 25, No. 6, pp. 852-869 (1999)

[Lupu 1999a] Lupu E., Milosevic Z., Sloman M., *Use of Roles and Policies for Specifying, and Managing a Virtual Enterprise*, In Proceedings of the 9th IEEE International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises (RIDE-VE'99), pp. 72-79, Sydney, Australia (1999)

[Lupu 2000] Lupu E., Sloman M., Dulay N., Damianou N., *Ponder: Realising Enterprise Viewpoint Concepts*, In Proceedings of the 4th Int. Enterprise Distributed Object Computing (EDOC2000), pp. 66-75, Mukahari, Japan, IEEE Computer Society (2000)

[Maeda 1997] Maeda C., Lee A., Murphy G., Kiczales G., *Open Implementation Analysis and Design*, In Proceedings of the 1997 Symposium on Software Reusability, pp. 44 - 52 , Boston, Massachusetts, ACM Press (1997)

[Mahon 2000] Mahon H., Bernet Y., Herzog S., Schnizlein J., *Requirements for a Policy Management System*, IETF Internet Draft, <draft-ietf-policy-req-02.txt> (2000)

[Marriott 1993] Marriott D., *Management Policy Specification*, Imperial College Research Report DoC 1994/1, Imperial College, Department of Computing (1993)

[Marriott 1996] Marriott D., Sloman M., *Management Policy Service for Distributed Systems*, In Proceedings of the IEEE Third International Workshop on Services in Distributed and Networked Environments (SDNE'96), pp. 2-9, Macau, China, IEEE Publishing (1996)

[Maullo 1993] Maullo M. J., Calo S. B., *Policy Management: An Architecture and Approach*, In Proceedings of the First IEEE International Workshop on System Management, UCLA, California (1993)

[Meyer 1985] Meyer B., *On formalism in specifications*, IEEE Software, Vol. 2, No. 1, pp. 6-26 (1985)

[Meyer 1990] Meyer B., *Introduction to the theory of programming languages*, Prentice-Hall (1990)

[Meyer 1992] Meyer B., *Eiffel – the Language*, Prentice-Hall, (1992)

[Minsky 1963] Minsky M., *Steps toward artificial intelligence*, Computers and Thought, McGraw-Hill (1963)

[Moffett 1993] Moffett J., Sloman M., *Policy Hierarchies for Distributed Systems Management*, IEEE Journal on Selected Areas in Communication, Vol. 11, No. 9, pp. 1404-1414 (1993)

[Moffett 1994] Moffett J., Sloman M., *Policy Conflict Analysis in Distributed System Management*, Journal of Organizatonal Computing, Vol. 4, No. 1, pp. 1–22 (1994)

[Moore 2001] Moore B., Ellesson E., Strassner J., Westerinen A., *Policy Core Information Model - Version 1 Specification*, IETF Network Working Group Request for Comments, RFC 3060 (2001)

[Moore 2003] Moore B. (editor), *Policy Core Information Model (PCIM) Extensions*, IETF Network Working Group Request for Comments, RFC 3460 (2003)

[Mosses 1992] Mosses P. D., *Action Semantics*, Cambridge University Press (1992)

[Mylopoulos 1999] Mylopoulos J., Chung L., Yu E., *From Object-Oriented to Goal-Oriented Requirements Analysis*, Communications of the ACM, Vol. 42 , No. 1, pp. 31-37 (1999)

[Norman Ramsey] Norman Ramsey N., Davidson J. W., Fernandez M. F., *Design Principles for Machine-Description Languages*, Electronic Engineering and Computer Science, Harvard University (unpublished)

[Oliva 1998] Oliva A., Garcia I.C., Buzato L.E., *The Reflective Architecture of Guaraná*, Technical Report IC-98-14, Institute of Computing, State University of Campinas, São Paulo, Brazil (1998)

[OMG 1995] Object Management Group, *CORBAfacilities: Common Facilities Architecture*, V4.0, Object Management Group (1995)

[Paige 2000] Paige R. F., Ostroff J. S., Brooke P. J., *Principles for modelling languages design*, Information & Software Technology, Vol. 42, No. 10, pp. 665-675 (2000)

[Pratt 2001] Pratt T. W., Zelkowitz M. V., *Programming Languages: Design and implementation*, Prentice-Hall (2001)

[Pratten 1995] Pratten A. W., Hong J. W., Bauer M. A., Bennett J. M., Lutfiyya H., *A Resource Management System Based on the ODP Trader Concepts and X.500*, In Proceedings of the Fourth International Symposium on Integrated Network Management, pp. 118-130, Santa Barbara, CA (1995)

[Ramanathan 1999] Ramanathan S., Caswell D., Neal S., *Auto-Discovery Capabilities for Service Management: An ISP Case Study*, Technical Report HPL-1999-68, HP Laboratories Bristol (1999)

[Raymond 2001] Raymond E. S., *The Cathedral and the Bazaar*, O'Reilly (2001)

[Raymond 1995] Raymond K., *Reference Model for Open Distributed Processing: Introduction*, Tutorial, IFIP International Conference on Open Distributed Processing (ICODP'95), Brisbane, Australia (1995)

[Richard 1999] Richard T. Simon R. T., Zurko M. E., *Separation of Duty in Role-Based Environments*, in Proceedings of the 4th ACM workshop on Role-based access control, pp. 43-54, ACM Press (1999)

[Riehle 1998] Riehle D., Gross T., *Role Model Based Framework Design and Integration*, In Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98), pp. 117-133, ACM Press (1998)

[Robins 2002] Robins B., *Understanding Web Services Management*, Analysis memo, Stencil Group (2002)

[Ross 2001] Ross R. G., Lam G. S. W., *The Do's and Don'ts of Expressing Business Rules*, in The BRS RuleSpeak™ Practitioner Kit (2001), www.BRSolutions.com

[Sandhu 1996] Sandhu R. S., Coyne E. J., Feinstein H. L. Youman C. E., *Role-based access control models*, IEEE Computer, Vol. 29, No. 2, pp. 38-47, (1996)

[Sebesta 2002] Sebesta R. W., *Concepts of programming languages*, Addison-Wesley (2002)

[Sleeper 2002] Sleeper B., Robins B., *The Laws of Evolution: A Pragmatic Analysis of the Emerging Web Services Market*, Analysis memo, Stencil Group (2002)

[Sloman 1994] Sloman M. (editor), *Network and Distributed Systems Management*, Addison-Wesley (1994)

[Sloman 1997] Sloman M., Lupu E., *Conflict Analysis for management Policies*, In Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM formerly known as ISINM 97), pp. 430-443, San Diego, California, Chapman and Hall (1997)

[Stansifer 1995] Stansifer R., *The Study of Programming Languages*, Prentice-Hall (1995)

[Steen 1999] Steen M.W.A., Derrick J., *Formalising ODP Enterprise Policies*, In Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC99), pp. 84-94, University of Mannheim, Germany, IEEE Publishing (1999)

[Steen 2000] Steen M.W.A., Derrick J., *ODP Enterprise Viewpoint Specification*, Computer Standards and Interfaces, Vol. 22, No. 3, pp. 165-189 (2000)

[Stefani 1995] Stefani J. B., *Open distributed processing: An architectural basis for information networks*, Computer Communications, Vol. 18, No. 11, pp. 849-862, (1995)

[Stevens 1999] Stevens M. L., Weiss W. J., *Policy-Based Management for IP Networks*, Bell Labs Technical Journal, Vol. 4, No. 4 (1999)

[Stevens 1999a] Stevens M., Weiss W., Mahon H., Moore B., Strassner J., Waters G., Westerinen A., Wheeler J., *Policy Framework*, IETF Internet Draft, <draft-ietf-policy-framework-00.txt> (1999)

[Stone 2001] Stone G., Lundy B., Xie G., *Network Policy Languages: A Survey and a New Approach*, IEEE Network, pp. 10-21, Jan/Feb 2001

[Strassner 1998] Strassner J., Schleimer S., *Policy Framework Definition Language*, IETF Internet Draft, <draft-ietf-policy-framework-pfdl-00.txt> (1998)

[Strassner 1999] Strassner J., *Directory Enabled Networks*, New Riders Publishing (1999)

[Strong 1968] Strong E.P., Smith R.D., *Management Control Models*, Holt, Rinehart and Winston (1968)

[Sutton 1997] Sutton S. M., Osterweil L. J., *The design of a next-generation process language*, In Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE 97), Jazayeri M., Schauer H. (editors), pp. 142-158, Springer-Verlag (1997)

[TechRepublic 2002] TechRepublic, *What Systems and Network Professionals Need in Automated Monitoring Solutions*, TechRepublic Survey (2002)

[van Lamsweerde 1995] van Lamsweerde A., Darimont R., Massonet P., *Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt*, In Proceedings of the 2nd IEEE Symposium on Requirements Engineering (RE '95), pp. 194-203, York, UK, IEEE Computer Society Press (1995)

[van Lamsweerde 1996] van Lamsweerde A., *Divergent Views in Goal-Driven Requirements Engineering*, In Proceedings of the ACM SIGSOFT Workshop on Viewpoints in Software Development, pp. 252-256, San Francisco, California (1996)

[van Lamsweerde 1999] van Lamsweerde A., *Goal-Oriented Requirements Analysis with KAOS*, Policy Workshop 1999, HP-Laboratories, Bristol, UK (1999)

[van Lamsweerde 2000] van Lamsweerde A., *Formal Specification: a Roadmap*, In Proceedings of the International Conference on the Future of Software Engineering, pp. 147-159, Limerick, Ireland, ACM Press (2000)

[van Lamsweerde 2001] van Lamsweerde A., *Building Formal Requirements Models for Reliable Software*, In Proceedings the 6th International Conference on Reliable Software Technologies (Ada-Europe'2001), Leuven, Belgium, Lecture Notes in Computer Science, Vol. 2043, Springer-Verlag (2001)

[Virmani 2000] Virmani A., Lobo J., Kohli M., *Netmon: Network management for the SARAS softswitch*, In Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'2000), Hong J. (editor), pp. 803-816, Weihmayer, Hawaii (2000)

[Westerinen 2000] Westerinen A., Strassner J. (editors), *Common Information Model (CIM) Core Model*, Whitepaper, Version 2.4, DMTF Technical Committee and System/Devices Working Group (2000)

[Wies 1995] Wies R., *Using a Classification of Management Policies for Policy Specification and Policy Transformation*, In Proceedings of the Fourth International Symposium on Integrated Network Management (ISINM '95), pp. 44-56, Santa Barbara, California, Chapman and Hall (1995)

[Wies 1997] Wies R., Mountzia M. A., Steenekamp P, *A practical approach towards a distributed and flexible realization of policies using intelligent agents*, In Proceedings of the 8th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management, pp. 292-308, Sydney, Australia (1997)

[Wirth 1971] Wirth N., *Program Development by Stepwise Refinement*, Communications of the ACM, Vol. 14, No. 4, pp. 221-227 (1971)

Appendix A

SPL MAPPING TO PCIM

Business policies expressed in high-level languages may be refined to the PCIM, which can then be mapped to a number of different network device configurations. Classes comprising the Policy Core Information Model are intended to serve as an extensible class hierarchy. Because PCIM and PCIMe (PCIM extension) provide the core classes for modelling policies, they are not in general sufficient by themselves for representing actual policy rules. PCIM and PCIMe themselves simply define elements that may be of use to submodels [Moore 2003]. Domain-specific submodels should provide the means for expressing policy rules, by defining subclasses of the classes defined in PCIM and PCIMe. A particular submodel will not, in general, need to use every element defined in PCIM and PCIMe.

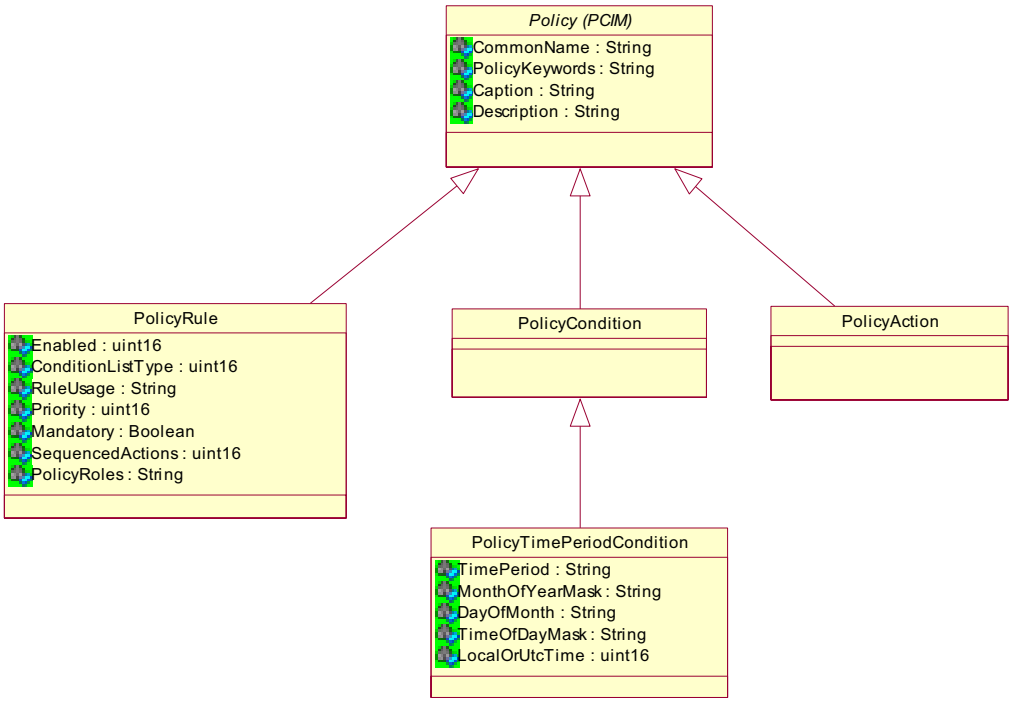


Figure 9-1 Class definitions in PCIM

PCIM is closely aligned with the CIM Core Policy model. Since there is no separately documented set of rules for specifying IETF information models such as PCIM, it is reasonable to look at the CIM specifications for guidance on how to modify and extend the model. Some of the CIM rules for changing an information model are [Moore 2003]:

- Properties may be added to existing classes.
- Classes may be inserted into the inheritance hierarchy above existing classes, and properties from the existing classes may then be "pulled-up" into the new classes. The net effect is that the existing classes have exactly the same properties they had before, but the properties are inherited rather than defined explicitly in the classes.
- New subclasses may be defined below existing classes.

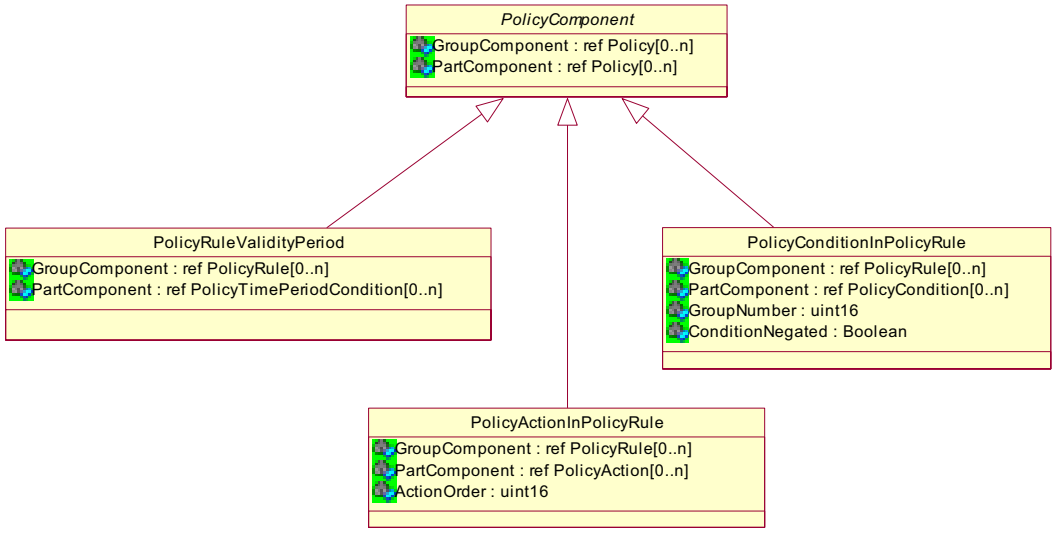


Figure 9-2 Associations in PCIM

The purpose of a policy condition is to determine whether or not the set of actions (aggregated in the *PolicyRule* that the condition applies to) should be executed or not. For the purposes of the PCIM, all that matters about an individual *PolicyCondition* is that it evaluates to TRUE or FALSE. A logical structure within an individual *PolicyCondition* may also be introduced, but this would have to be done in a subclass of the *PolicyCondition*.

The purpose of a policy action is to execute one or more operations. The class *PolicyAction* exists in PCIM as an abstract superclass for domain-specific policy actions, defined in subclasses.

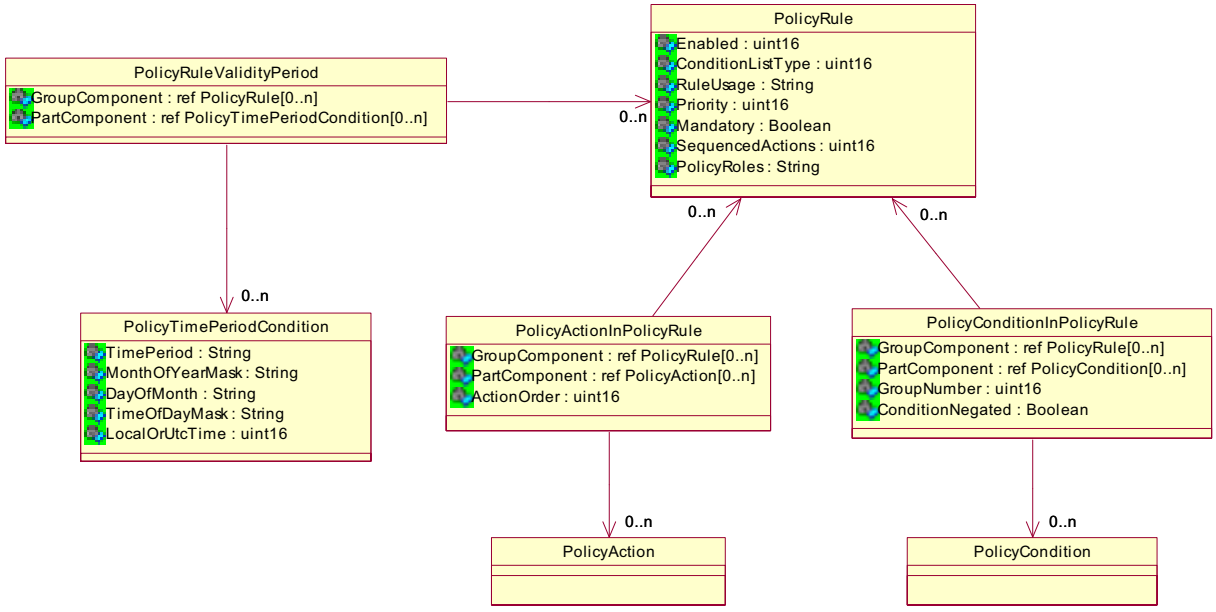


Figure 9-3 Policy definition in PCIM

PCIM is an information model that describes the basic concepts of policy groups, rules, conditions, actions, repositories and their relationships. This model is described as a "core" model since it cannot be applied without domain-specific extensions. Because it is general, the *PolicyCondition* class does not itself contain any "real" conditions. These will be represented by properties of domain-specific subclasses of the *PolicyCondition*. Similarly,

the *PolicyAction* class exists as an abstract superclass for domain-specific policy actions, defined in subclasses. Therefore, in order to define mappings from SPL to PCIM, we must define two domain-specific classes as subclasses of the *PolicyAction* and *PolicyCondition*.

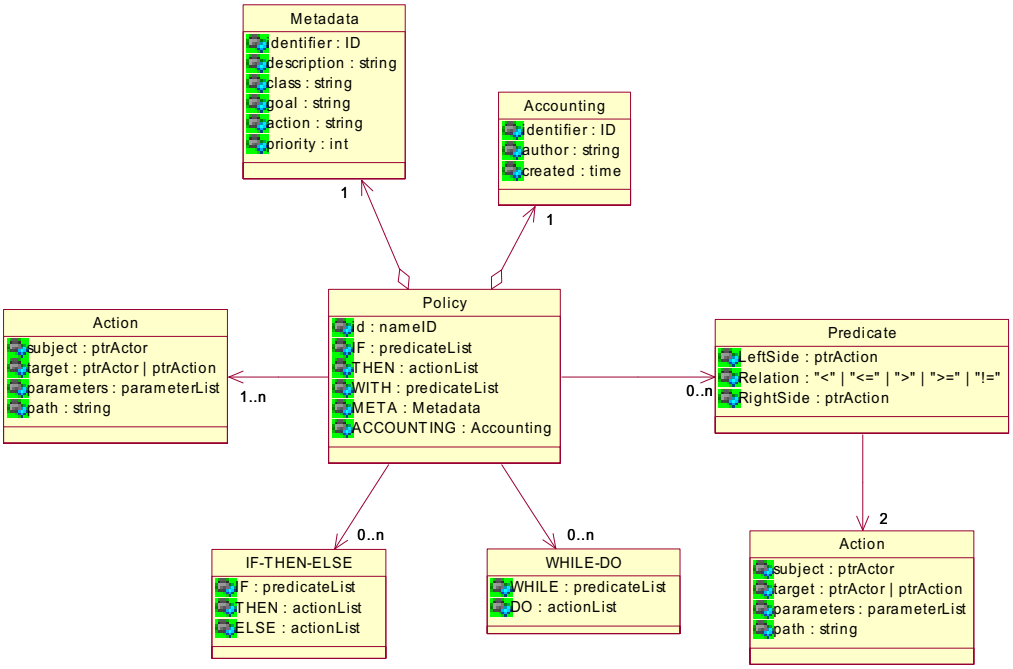


Figure 9-4 Policy definition in SPLc

Policy models for application-specific areas may extend the CIM Policy Model in several ways. We chose to extend the *PolicyRule* class, as well as the *PolicyCondition* and *PolicyAction* classes.

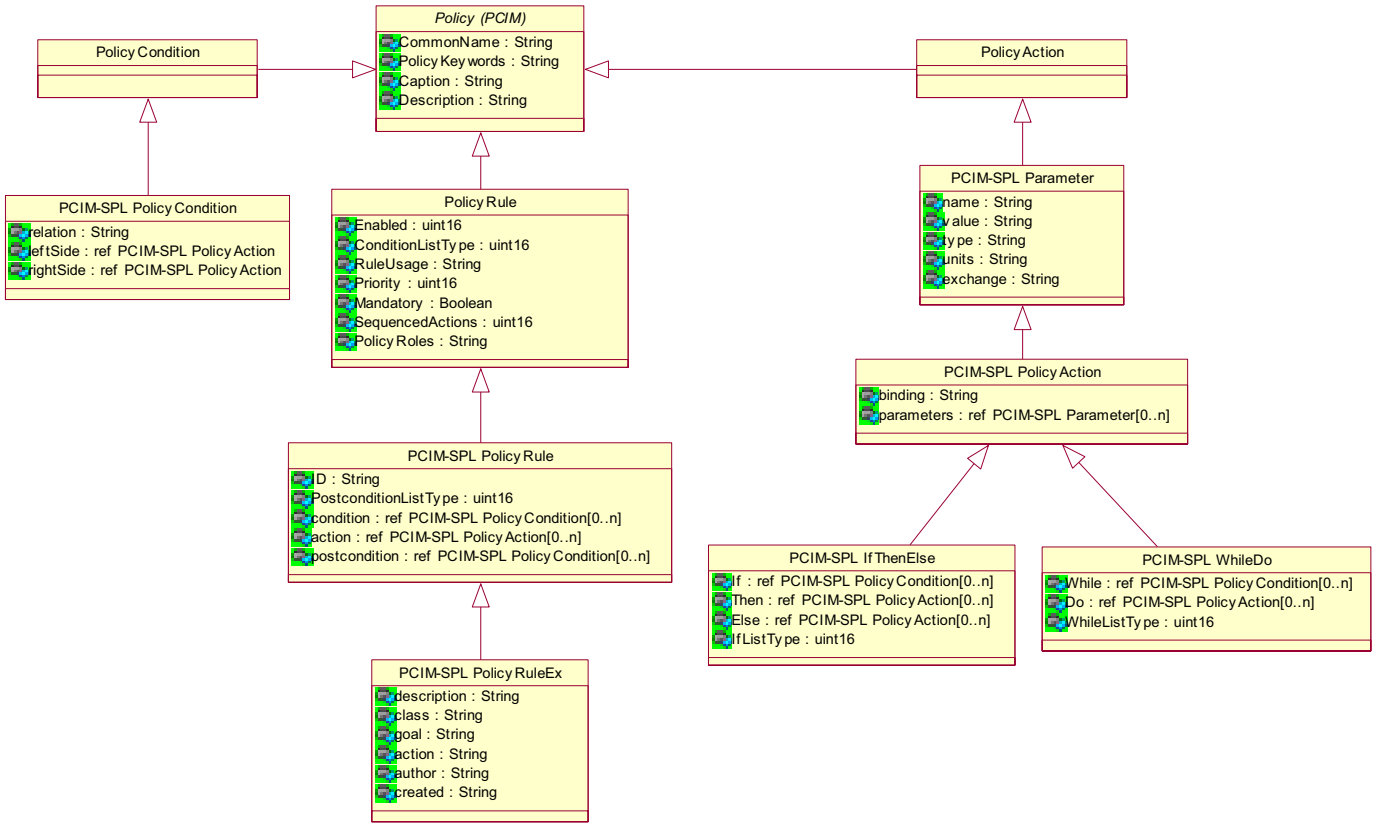


Figure 9-5 SPL-specific extension of PCIM

An action is represented as a special type of parameter that gets its value as a result of an interaction. IF-THEN-ELSE and WHILE-DO types are simply defined as special types of actions that return TRUE (bool). CIM extension guidelines suggest that declaring a parameter class makes the action much easier to understand. It also permits to specify default values and other aspects of the parameters. Furthermore, this approach allows the action to be extended by the addition of properties or subtypes to the parameter class without impacting existing applications.

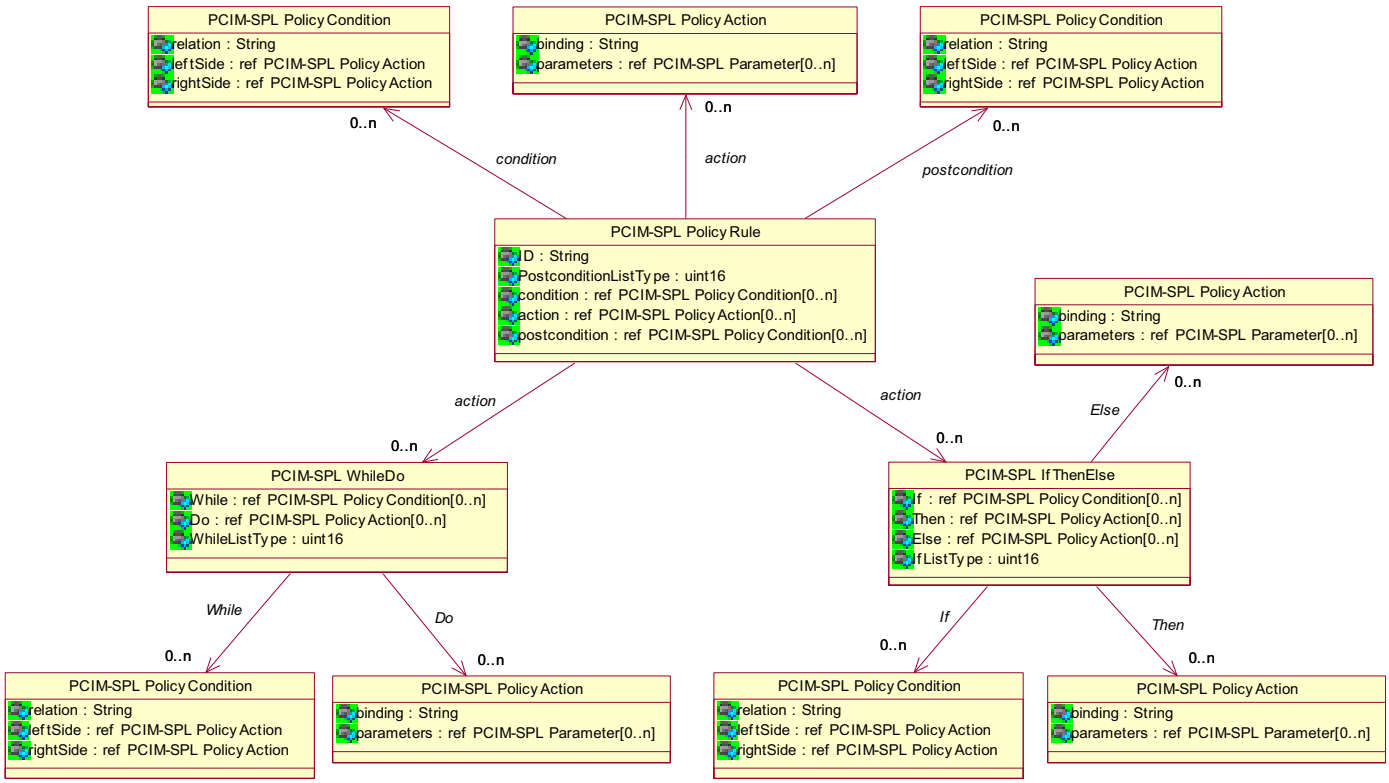


Figure 9-6 SPLc policy representation in PCIM

In PCIM, time is simply represented as a string in a special format. PCIM default values will be used for the parameters not provided via mappings from SPL.

SPLc	***PCIM***
ID	string
pointer	<classname> ref
time	string
Parameter	PCIM-SPL Parameter
Action	PCIM-SPL PolicyAction
Predicate	PCIM-SPL PolicyCondition
Element	PCIM-SPL PolicyAction
IF-THEN-ELSE	PCIM-SPL IfThenElse
WHILE-DO	PCIM-SPL WhileDo
Policy	PCIM-SPL PolicyRule, PCIM-SPL PolicyRuleEx

Table 9-1 SPLc to PCIM type mappings

POLICYML DEFINITION

PolicyML DTD

```
<?xml version='1.0' encoding='UTF-8' ?>

<!ELEMENT Accounting (Created , Author)>
<!ELEMENT Course (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT Class (#PCDATA)>
<!ELEMENT Created (#PCDATA)>
<!ELEMENT DO (Action* , IF-THEN-ELSE* , WHILE-DO*)>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT Direction (#PCDATA)>
<!ELEMENT ELSE (Action* , IF-THEN-ELSE* , WHILE-DO*)>
<!ELEMENT FullPath (#PCDATA)>
<!ELEMENT Goal (#PCDATA)>
<!ELEMENT IF (Predicate*)>
<!ATTLIST IF  type CDATA  #IMPLIED >
<!ELEMENT IF-THEN-ELSE (IF , THEN , ELSE)>
<!ELEMENT Action (Signarure , FullPath)>
<!ATTLIST Action  name CDATA  #REQUIRED >
<!ELEMENT LeftSide (Parameter | Action)>
<!ELEMENT Metadata (Goal , Class , Course , Priority , Description)>
<!ELEMENT Parameter (Type , Value , Direction)>
<!ATTLIST Parameter  name         CDATA  #REQUIRED isReturnValue CDATA  #REQUIRED >
<!ELEMENT Policy (IF , THEN , WITH , Metadata , Accounting)>
<!ATTLIST Policy  name CDATA  #REQUIRED >
<!ELEMENT Predicate (LeftSide , RightSide)>
<!ATTLIST Predicate  relation     CDATA  #IMPLIED isReturnValue CDATA  #IMPLIED >
<!ELEMENT Priority (#PCDATA)>
<!ELEMENT RightSide (Parameter | Action)>
<!ELEMENT Signarure (Parameter+)>
<!ELEMENT THEN (Action* , IF-THEN-ELSE* , WHILE-DO*)>
<!ELEMENT Type (#PCDATA)>
<!ELEMENT Value (#PCDATA)>
<!ELEMENT WHILE (Predicate*)>
<!ATTLIST WHILE  type CDATA  #REQUIRED >
<!ELEMENT WHILE-DO (WHILE , DO)>
<!ELEMENT WITH (Predicate*)>
<!ATTLIST WITH  type CDATA  #IMPLIED >
```

PolicyML Schema

```
<?xml version = "1.0" encoding = "UTF-8"?>

<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
        <xsd:element name = "Accounting">
            <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref = "Created"/>
                        <xsd:element ref = "Author"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name = "Course" type = "xsd:string"/>
```

```
<xsd:element name = "Author" type = "xsd:string"/>
<xsd:element name = "Class" type = "xsd:string"/>
<xsd:element name = "Created" type = "xsd:string"/>
<xsd:element name = "DO">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "Action" minOccurs = "0" maxOccurs = "unbounded"/>
                        <xsd:element ref = "IF-THEN-ELSE" minOccurs = "0" maxOccurs = "unbounded"/>
                        <xsd:element ref = "WHILE-DO" minOccurs = "0" maxOccurs = "unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "Description" type = "xsd:string"/>
<xsd:element name = "Direction" type = "xsd:string"/>
<xsd:element name = "ELSE">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "Action" minOccurs = "0" maxOccurs = "unbounded"/>
                        <xsd:element ref = "IF-THEN-ELSE" minOccurs = "0" maxOccurs = "unbounded"/>
                        <xsd:element ref = "WHILE-DO" minOccurs = "0" maxOccurs = "unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "FullPath" type = "xsd:string"/>
<xsd:element name = "Goal" type = "xsd:string"/>
<xsd:element name = "IF">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "Predicate" minOccurs = "0" maxOccurs = "unbounded"/>
            </xsd:sequence>
            <xsd:attribute name = "type" type = "xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "IF-THEN-ELSE">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "IF"/>
                        <xsd:element ref = "THEN"/>
                        <xsd:element ref = "ELSE"/>
            </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "Action">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "Signarure"/>
                        <xsd:element ref = "FullPath"/>
            </xsd:sequence>
            <xsd:attribute name = "name" use = "required" type = "xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "LeftSide">
        <xsd:complexType>
            <xsd:choice>
                        <xsd:element ref = "Parameter"/>
                        <xsd:element ref = "Action"/>
            </xsd:choice>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "Metadata">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "Goal"/>
                        <xsd:element ref = "Class"/>
                        <xsd:element ref = "Course"/>
                        <xsd:element ref = "Priority"/>
                        <xsd:element ref = "Description"/>
            </xsd:sequence>
```

```
        </xsd:complexType>
</xsd:element>
<xsd:element name = "Parameter">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "Type"/>
                        <xsd:element ref = "Value"/>
                        <xsd:element ref = "Direction"/>
            </xsd:sequence>
            <xsd:attribute name = "name" use = "required" type = "xsd:string"/>
            <xsd:attribute name = "isReturnValue" use = "required" type = "xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "Policy">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "IF"/>
                        <xsd:element ref = "THEN"/>
                        <xsd:element ref = "WITH"/>
                        <xsd:element ref = "Metadata"/>
                        <xsd:element ref = "Accounting"/>
            </xsd:sequence>
            <xsd:attribute name = "name" use = "required" type = "xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "Predicate">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "LeftSide"/>
                        <xsd:element ref = "RightSide"/>
            </xsd:sequence>
            <xsd:attribute name = "relation" type = "xsd:string"/>
            <xsd:attribute name = "isReturnValue" type = "xsd:string"/>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "Priority" type = "xsd:string"/>
<xsd:element name = "RightSide">
        <xsd:complexType>
            <xsd:choice>
                        <xsd:element ref = "Parameter"/>
                        <xsd:element ref = "Action"/>
            </xsd:choice>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "Signarure">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "Parameter" maxOccurs = "unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "THEN">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "Action" minOccurs = "0" maxOccurs = "unbounded"/>
                        <xsd:element ref = "IF-THEN-ELSE" minOccurs = "0" maxOccurs = "unbounded"/>
                        <xsd:element ref = "WHILE-DO" minOccurs = "0" maxOccurs = "unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name = "Type" type = "xsd:string"/>
<xsd:element name = "Value" type = "xsd:string"/>
<xsd:element name = "WHILE">
        <xsd:complexType>
            <xsd:sequence>
                        <xsd:element ref = "Predicate" minOccurs = "0" maxOccurs = "unbounded"/>
            </xsd:sequence>
            <xsd:attribute name = "type" use = "required" type = "xsd:string"/>
```

```
            </xsd:complexType>
        </xsd:element>
        <xsd:element name = "WHILE-DO">
            <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref = "WHILE"/>
                        <xsd:element ref = "DO"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name = "WITH">
            <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref = "Predicate" minOccurs = "0" maxOccurs = "unbounded"/>
                </xsd:sequence>
                <xsd:attribute name = "type" use = "required" type = "xsd:string"/>
            </xsd:complexType>
        </xsd:element>
</xsd:schema>
```

EXAMPLES: MANAGEMENT POLICIES IN SPL

The following examples will be used as a context for a discussion about some practical challenges related to the specification of management policies, and how the expressive power of SPL faces those challenges.

Toaster

A toaster makes a good example to start with, because it represents a system that is physically encapsulated behind its interface, and its behaviour is straightforward to model and understand. The main goal of this example is to demonstrate how to use SPL and management policies to implement new functionality on top of the existing behaviour – how to program the behaviour using a management interface. Even with this very simple example it will be possible to show the essence of policy-based management using SPL.

The primary objective of a toaster is to toast bread, or, in other words, to offer a toasting service. Toasters can be different, more or less complex to use. From a management point of view, it would be nice to have a flexible control over a toaster, so we can make it easier and cheaper to use. For that purpose, let us imagine a semi-automatic toaster, which has a bit unorthodox functional and management interfaces. The toasting service it offers is implemented as a synergy of:

- *Functionality*, given by design.
- *Management policies*, which are based on the management interface provided by the design.

Management model of the system

In this example, modelling of our toaster will start from the lowest (least abstract) level. Our system is relatively simple, so such a bottom-up modelling strategy will be perfectly suitable. Basic components of out toaster are presented in Figure 9-7.

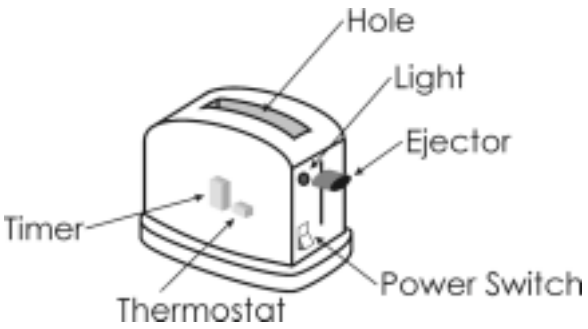


Figure 9-7 The toaster system

Corresponding management operations, implemented by the Toaster components, are presented in Figure 9-8.

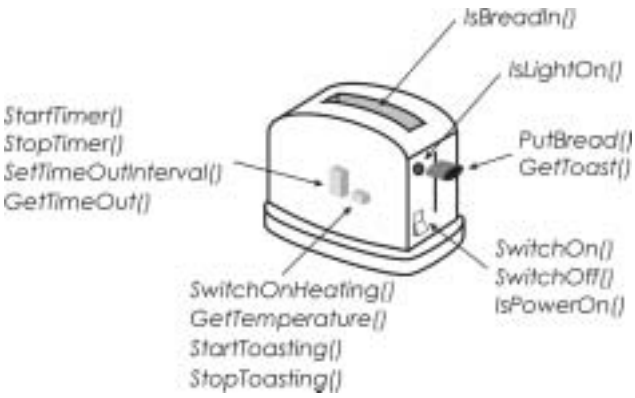


Figure 9-8 Toaster management interface

The first, lowest-level model of the system is a simple collection of management operations. It defines the Toaster management interface (Figure 9-9).

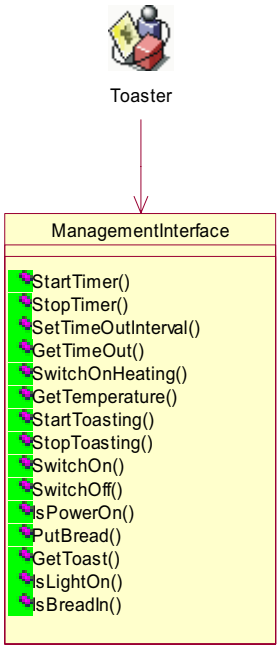


Figure 9-9 Low-level model of the Toaster

Further modelling of the system from the management point of view may be performed in various ways. For example, we could try to logically organize the available management operations as a first step towards an abstract, mental model of the system. Abstract management roles in the system, dedicated to provide management operations, may be structured as following:

- **Controllers** domain, which enable control over the properties of the toasting service, may include the following abstract roles:
  - *Temperature Controller*.
  - *Power Controller*.
  - *Release Controller*.
  - *Operation Controller*, which has a general control over the toasting service. It also implements the Timer service.
- **Indicators** domain, dedicated to monitoring, include only one abstract role:
  - *Indicator*.
- **Suppliers** domain, which actually delivers the service to a user, includes the following role:

∉ *Hole.*

Conceptual system modelling may be performed in UML, using a set of custom designed stereotypes. The following diagrams are created in Rational Rose, by customizing the UML class diagram with special stereotypes to represent SPL types (Figure 9-10). Such conceptual models may be further refined in a dedicated policy modelling tool, such as PolicyModeller (see Chapter 6).

Figure 9-10 SPL stereotypes for UML

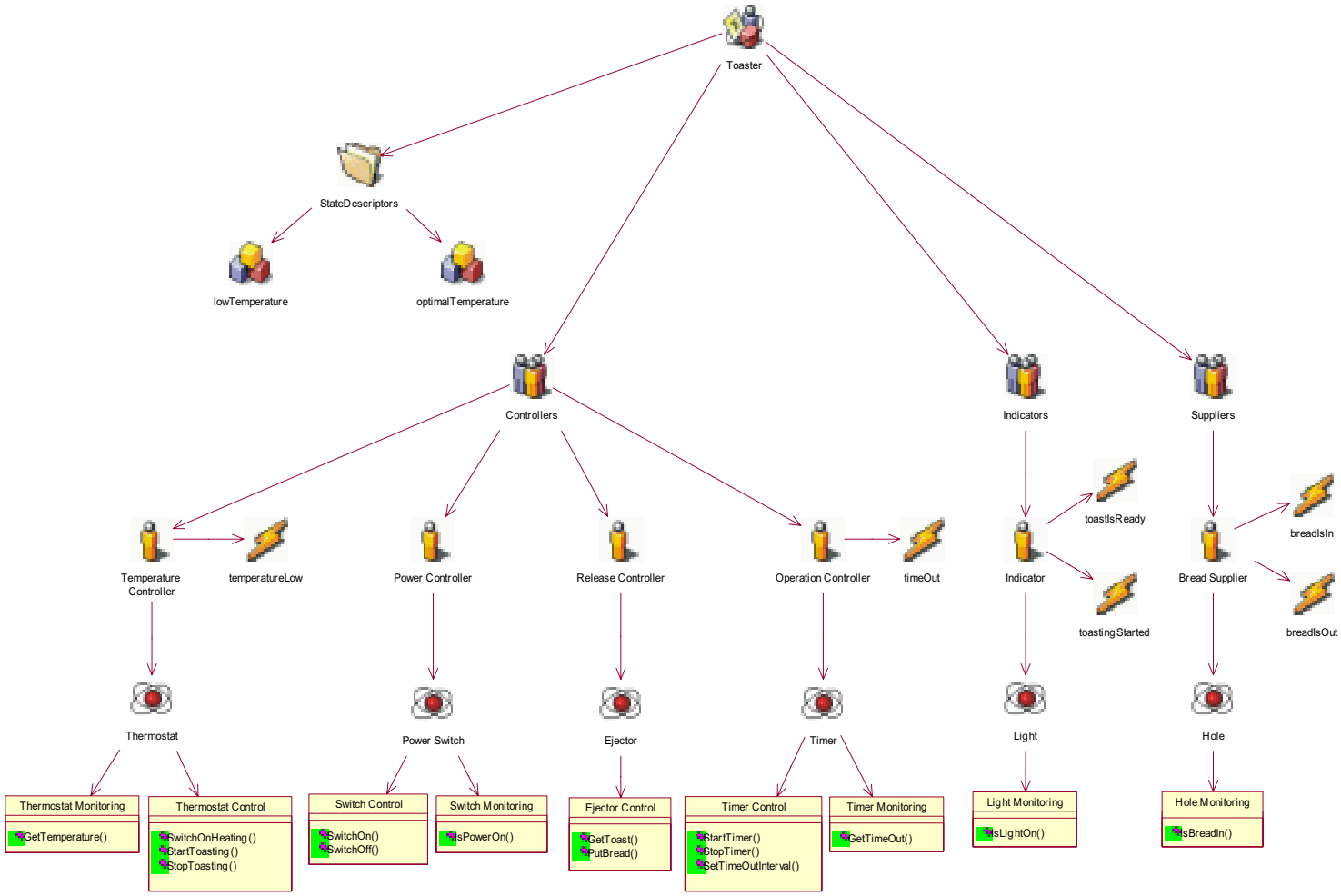


Figure 9-11 Management model of the Toaster in UML, with SPL stereotypes

Our management model of the system (Toaster) is obviously event–based, made for reactive management that uses notification events (Figure 9-11). Event Handlers will be specified to capture system events and trigger management policies.

Management policies and Event Handlers

Policy-based modelling usually begins with a definition and analysis of the system state domain. The state domain we derive from the management interface, as a portion of the system state of interest that can be acquired from the system using management operations. We must analyse the potential influence of the system state on his behaviour, and identify the unacceptable states that must be avoided using management policies.

	Power	***Content***	***Light (toasting)***	***Standby***	***Description***
A	On	Empty	Off	Off	Service ready
B	Off	Empty	Off	On	Service hibernation
C	On	Toast	Off	Off	Toast is ready for delivery
D	On	Bread	On	Off	Toasting in progress
E	On	Bread	Off	Off	Bread is ready for toasting

Table 9-2 Toaster states

Functional specification is a precise abstract model of the system behaviour as seen by an external observer, created as the formal response to the design objectives. It includes preconditions, which define properties required of the input, and post conditions, which define relationships required between the input and the output. The goal of a functional specification is to define the functional capabilities of both functional and management interface of a system, while the management specification is focusing on the particular usage scenarios for the system management interface. Functional specifications can be made in terms of design time policies, which are used to specify system invariants that need to be frozen by the design. The management policies are focusing on run-time behaviour, with a goal of narrowing design time behaviour defined by the functional specification.

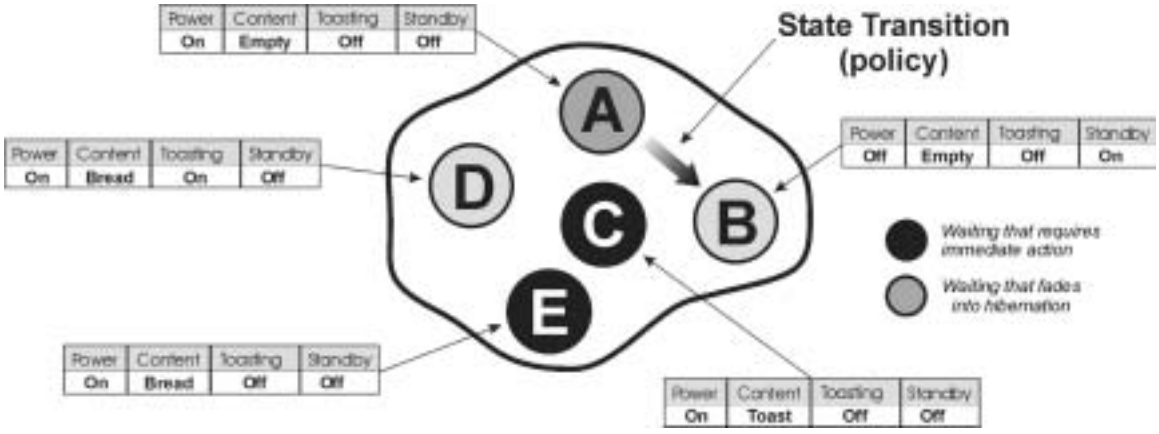


Figure 9-12 Toaster state domain

In case of our Toaster, the functional and the management interfaces overlap. The functional interface, which initially encompasses the functionality of the *Hole*, the *Light*, the *Ejector* and the *Power Switch*, is a subset of the management interface, which includes the complete behaviour of the toaster. Our goal is to use the management policies for the simplification of the toaster's functional interface. As a result, the toaster's functional interface will be reduced only to the behaviour of the *Hole*, namely *PutBread()* and *GefToast()*.

States A, C, and E (Figure 9-12) we want to avoid using management policies, because those states are unproductive. Our management strategy may be defined as follows:

- ***Objective***: To offer a quality toasting service.
- ***Goals***: Automation (improves usability) and optimization (saves energy without affecting QoS).
- ***Policies*** (implement goals). The design of the Toaster opens an opportunity for implementation of additional behaviour using management policies. In our example, this will include programming of the following behaviour:
    - *Automation* (eject-when-ready) of the toasting service, which simplifies the functional interface and makes the service easier to use. This includes the following state transitions:

E ⇓ D: *If the bread is ready, start toasting.*
D ⇓ C ⇓ A: *If the toast is ready, stop the toasting and deliver it.*

- *Standby functionality*, which optimizes the system functionality by making a compromise between the response time and energy savings. This includes the following state transition:

  A ⇓ B: *If the service is not required for a while, go to standby mode.*

By building management policies to respond to events that occur in the system, we will try to control the state transitions, and therefore the behaviour of our Toaster.

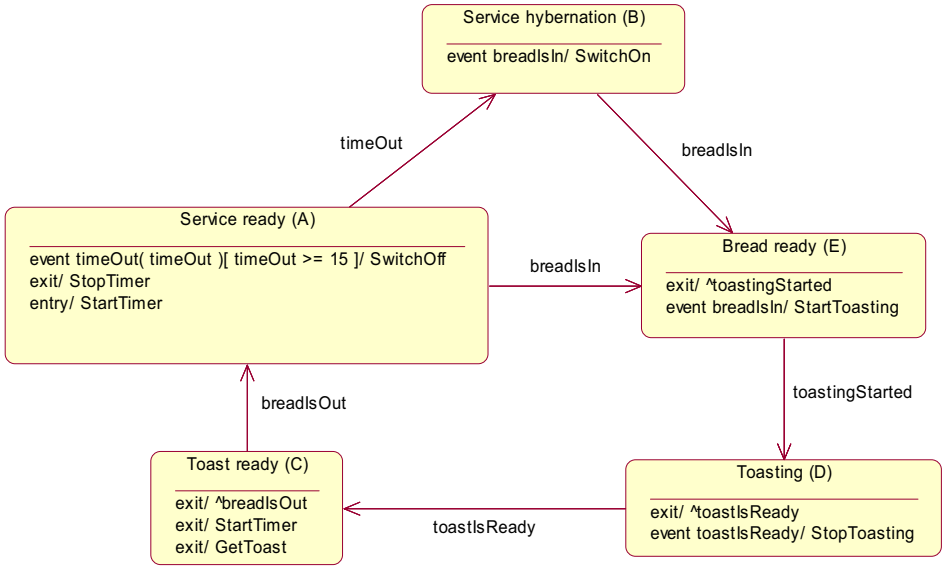


Figure 9-13 UML State diagram: The behaviour we want to implement using toaster management interface and policies

Figure 9-13 is an UML State diagram, which specifies the behaviour of the Toaster that we want to implement using management policies.

Automation

Many real toasters implement a simple functionality that enables releasing (ejecting) of the toast as soon as it is ready. In our model, we'll look at a semi-automatic toaster that doesn't have such built-in functionality, but its management interface enables its automation using management policies. Such automation is not a new behaviour, but just a combination of the existing one. The advantage of this approach compared to the built-in one is that it enables control and fine tuning of implemented automation.

Our semi-automatic toaster has an indicator light that switches on when the toast is ready, so the user must release (eject) it.

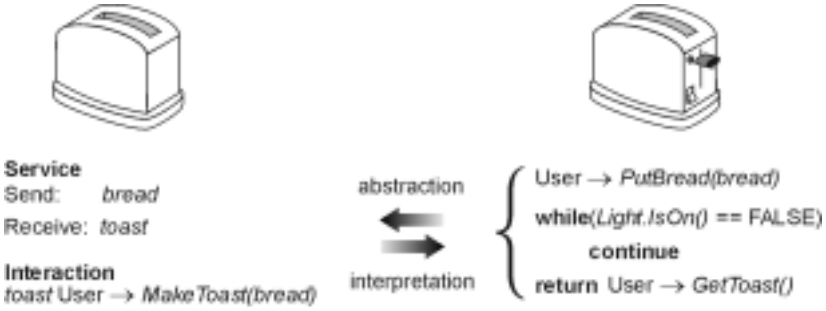


Figure 9-14 Automation of the Toasting service

We will use the toaster management interface and policies to implement automation of the service, so that user intervention will no longer be needed (Figure 9-14). As the result, our service will hide the complexity of the toasting operation from the user and become as simple as this:

toast MakeToast(bread) // if you give me a slice of bread I'll return a toast

the automation for the toaster service essentially implements the following state transition:

D ⇓ C ⇓ A

The meaning of this state transition is:

If the toast is ready, release it while keeping the service on.

To implement the state transition, we must express its semantics in the predefined IF-THEN-WHILE form, al follows:

IF
Toast is ready
THEN
Release the bread
WITH
Keep the power on

The next step is to formalize the policy content by referring to the operations that define the system management behaviour – operations from the Dictionary (Figure 9-11). The policy *Deliver toast* will be accompanied with an appropriate Event Handler, which essentially links the *toastIsReady* event and the policy.

ON
toastIsReady
IF
TRIGGER
Deliver toast
USING

Deliver toast (D ⇓ C ⇓ A)

IF
Operation Controller ⇓ Indicator.IsLightOn() = TRUE
THEN
Operation Controller ⇓ Temperature Controller.StopToasting(),
Operation Controller ⇓ Release Controller.GetToast()
WITH
AND
(Operation Controller ⇓ Power Controller.IsPowerOn() = TRUE,
Operation Controller ⇓ Bread Supplier.IsBreadIn() = FALSE)
METADATA
Description: "If the toast is ready, release it while keeping the service on",
Class: Performance management,
Goal: Automation,
Action: Change of state,
Priority: 1

The policy may be specified to include more details (rich semantics), and consequently it will be at a lower level of abstraction. However, when it comes to the policy enforcement, the abstract portion of an action will be simply ignored in the process of interpretation, because it doesn't make any sense in the context of the real toaster management interface. Regardless of the level of details used in policy specification, after triggering policies will be interpreted to specifications in terms of management operations only, and executed. In our example:

Operation controller ⇓ Temperature Controller.StopToasting() ◊ StopToasting()

because roles that we have in our model are abstract – they do not necessary correspond to the real system objects. We imagine that those objects, even if they exist, are hidden behind the system management interface. The purpose of additional details is to enrich semantics of policy specifications and make working with policies easier. However, the internals of the system are unknown (system is a "black box"), and those details are only abstractions that most likely don't correspond to the real system organization. Therefore, since it is not real, it cannot be used for the actual policy enforcement in the real system. In light of that, previous policy would be interpreted for execution simply as:

IF
IsLightOn() = TRUE
THEN
StopToasting(),
GetToast()
WITH
AND
(IsPowerOn() = TRUE,
IsBreadIn() = FALSE)

Alternatively, if the details of a system internals are known (low-level management scenario), such detailed management specifications may actually correspond to the real system structure. As a result, those specifications will be used in full details to trigger management interactions among particular objects/roles in the real system.

When it comes to the policy metadata, it is up to the manager to decide about the importance and the extent of its use. Manager may start policy specification by specifying the policy metadata first, and then extend it with the operational specification. Alternatively, manager may specify the operational part first, and then enrich it with the metadata as needed.

Self management

Self management of the Toaster has a goal of saving energy. After being idle for 15 minutes, the service will be switched to a stand-by (hibernation) mode. On service request, which is made by putting bread into the Toaster, the system will:

- ∉ Re-establish the service from the hibernation to the operational state.
- ∉ Start the toasting service.

The *breadIsIn* event is raised by the *Bread Supplier* role, but we want that our toaster does not blandly respond to that event. Therefore, we will have a policy (*Switch on toasting*), which will require that the *Operation Controller* checks the state of the system and then decide if the toasting service should be started. In response to this event, some other policies may be triggered too, to perform additional checks and management actions. In this example such behaviour seams to be trivial, but in complex systems, where events are raised by the monitoring roles, state of the system must be checked by controlling roles before a decision to make an action can be made.

Start service (B ⇓ C)

IF
AND
(Operation Controller ⇓ Power Controller.IsPowerOn() = FALSE,
Operation Controller ⇓ Bread Supplier.IsBreadIn() = TRUE)
THEN
Operation Controller ⇓ Power Controller.SwitchOn(),
Operation Controller ⇓ Operation Controller.StopTimer()
WITH
METADATA
Description: "Establish the service operational mode on request",
Class: Operation management,
Goal: Respond to service request,
Action: Change of state,
Priority: 1

Switch on toasting (C ⇓ D)

IF
Operation Controller ⇓ Bread Supplier.IsBreadIn() = TRUE
THEN
Operation Controller ⇓ Temperature Controller.StartToasting()
WITH
METADATA
Description: "Start the toasting service of request",
Class: Operation management,
Goal: Deliver service,
Action: Change of state,
Priority: 1

ON
breadIsIn
IF
TRIGGER
Start service
Switch on toasting
USING

Even though a simplified policy like this is not very useful in reality, it opens an opportunity to control conditions under which a service should become operational. This may include: user authentication, QoS, timing, price etc.

Timeout service

IF
Operation Controller ⇓ Bread Supplier.IsBreadIn() = FALSE
THEN
Operation Controller ⇓ Operation Controller.StartTimer()
WITH
METADATA
Description: "Start the timeout time on switching to idle mode",
Class: Performance management,
Goal: Cost control,
Action: Change of state,
Priority: 2

ON
breadIsOut
IF
TRIGGER
Timeout service
USING

The *timeOut* event is designed to include the delay in service usage. Notifications are sent by the *Timer*, with a certain frequency. For example:

timeOut(5)
timeOut(10)
timeOut(15)…

An Event Handler will be used to pass the delay to a policy, which will make a management decision based on the parameter value. The USING portion of the Event Handler links the event and policy parameters, by stating that event parameter shall pass its value to the policy parameter when policy is triggered.

ON
timeOut(delay)
IF
TRIGGER
Stop service(timeOut)
USING
delay ⇓ timeOut

Stop service(timeOut) (A ⇓ B)

IF
AND
(Operation Controller ⇓ Power Controller.IsPowerOn() = TRUE,
Operation Controller ⇓ Bread Supplier.IsBreadIn() = FALSE,
timeOut >= 15)
THEN
Operation Controller ⇓ Power Controller.SwitchOff()
WITH
METADATA
Description: "If the service is idle for more than 15 minutes, put it in standby mode",
Class: Performance management,
Goal: Cost control,
Action: Change of state,
Priority: 2

Alternatively, we could check the value of the event parameter within the Event Handler itself, and avoid the instantiation of corresponding management policy if the parameter value is not within the acceptable range. In this case, there will be no need to pass the parameter to the management policy, which would simplify the instantiation. The filtering performed by the Event Handler would eliminate unnecessary policy instantiation, and therefore sped up the management process. The management policy would become simpler and more general in nature, so it may be reused in other management scenarios. On the other hand, shifting the management focus towards the Event Handler would make the triggering more complex and less flexible, because the Event Handler would loose in its generality.

ON
timeOut(delay)
IF
timeOut >= 15
TRIGGER
Stop service
USING

Stop service (A ⇓ B)

IF
AND
(Operation Controller ⇓ Power Controller.IsPowerOn() = TRUE,
Operation Controller ⇓ Bread Supplier.IsBreadIn() = FALSE)
THEN
Operation Controller ⇓ Power Controller.SwitchOff()
WITH
METADATA
Description: "If the service is idle for more than 15 minutes, put it in standby mode",
Class: Performance management,
Goal: Cost control,
Action: Change of state,
Priority: 2

Thermostat

Self management of the toaster may be extended even further with a thermostat functionality, which will be responsible to maintain an optimal toasting temperature, and offer even better management control over the service behaviour.

The toasting temperature shall be maintained within acceptable limits, as close as possible to an optimal value. The goal is to maintain the optimal temperature, but, because of inertia of the heating process, we must tolerate certain offset. Using the policy preconditions and post conditions, we can set an interval of the acceptable temperature values. It will define a "grey zone" around the target state, as an optimal value and an acceptable offset. Built-in functionality of the *Thermostat* will switch off the heating as soon as the temperature reaches the optimal value (upper

temperature limit), so we only need to define a lower temperature limit. The boundaries of the state domain are modelled in the Dictionary as State Descriptors (*lowTemperature* and *optimalTemperature*).

ON
temperatureLow
IF
TRIGGER
Maintain temperature
USING

Maintain temperature

IF
Operation Controller ⇓ Thermostat.GetTemperature() = lowTemperature
THEN
Operation Controller ⇓ Thermostat.SwitchOnHeating()
WITH
Operation Controller ⇓ Thermostat.GetTemperature() = optimalTemperature
METADATA
Description: "Keep the temperature within the limits",
Class: Performance management,
Goal: Optimization,
Action: Change of state,
Priority: 1

Instead of using such a complex management policy, we may model the temperature maintenance functionality as an abstract management operation, provided by the *Thermostat* role. This operation, called *MaintainTemperature()*, will be refined to the existing (real) operations from the toaster management interface during the process of policy interpretation. The purpose of such an abstract management operation is only to simplify the system management interface, and, therefore, the Toaster management.

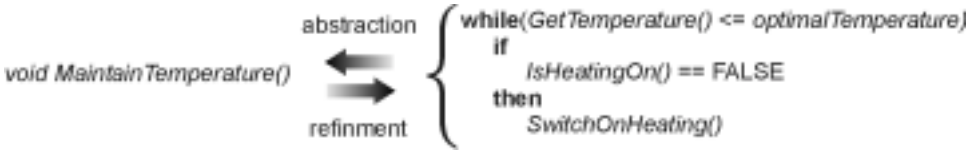


Figure 9-15 Example refinement

In SPL, such a refinement would be specified using the Strategy type, as follows:

MaintainTemperature()
EQUIVALENT
WHILE
GetTemperature() <= optimalTemperature
IF
IsHeatingOn() = FALSE
THEN
SwitchOnHeating()
USING

Using this new management operation, our management policy may be simply specified as follows:

ON
temperatureLow
IF
TRIGGER
Maintain temperature
USING

Maintain temperature

IF
THEN
Operation Controller ⇓ Thermostat.MaintainTemperature()
WITH
METADATA
Description: "Keep the temperature within the limits",
Class: Performance management,
Goal: Optimization,
Action: Change of state,
Priority: 1

When triggered, this policy will be first refined to refer only to the real operations from the Toaster management interface, and then it will be sent to the Toaster for execution:

```
IF
THEN
        WHILE
            GetTemperature() <= optimalTemperature
            IF
               IsHeatingOn() = FALSE
            THEN
               SwitchOnHeating()
WITH
```

Conflict detection and resolution

It is possible that *Stop service* and *Switch on toasting* policies will be triggered at the same time. In that case, they will be in conflict of interests, which needs to be resolved. To handle such a potential conflict, we may specify a conflict resolution rule in the policy form (semantic rule). This rule implements the fact that saving energy and responding to a service request represents a conflict of interests, and that the priority shall be given to serving users.

Resolve conflict of interests

IF
AND
(Policy.Metadata.Goal = "Respond to service request",
TriggeringPool.Policy.Metadata.Goal = "Cost control")
THEN
Discard(TriggeringPool.Policy)
WITH
METADATA
Description: "Saving energy and responding to a service request is a conflict of interests",
Class: Semantic analysis,
Goal: Conflict resolution,
Action: Discard,
Priority: 1

The logic of this rule is completely equivalent to the following rule:

Resolve conflict of interests

IF
AND
(Policy.Metadata.Goal = "Cost control",
TriggeringPool.Policy.Metadata.Goal = "Respond to service request")
THEN
Discard(Policy)
WITH
METADATA
Description: "Saving energy and responding to a service request is a conflict of interests",
Class: Semantic analysis,
Goal: Conflict resolution,
Action: Discard,
Priority: 1

However, there will be cases where it is important which policy was triggered first (which one is already in the TriggeringPool) for the decision which policy shall be discarded. For example, we may decide that every request for service that comes after the *Stop service* policy was triggered must be delayed for certain period of time:

Resolve conflict of interests

IF
AND
(Policy.Metadata.Goal = "Respond to service request",
TriggeringPool.Policy.Metadata.Goal = "Cost control")
THEN
Delay(Policy)
WITH
METADATA
Description: "Every request for service that comes after a call for hibernation must be delayed",
Class: Semantic analysis,
Goal: Conflict resolution,
Action: Delay,
Priority: 1

For more examples of conflict detection and resolution, see Appendix D.

Library

A library is a much more complex system that a toaster, and its modelling is better to perform in a top-down fashion, from a simplified (abstract) representation towards a more detailed one. With the library example we will be focusing on the policy refinement concept, as it was envisaged in SPF/SPL.

Management model of the system

In contrast to the Toaster example, where the service provision lasts in time and requires a special state of the system, the Library service is just an instantaneous transaction between a user and the system. Even though a record of the transaction will be created, during the transaction itself the system doesn't maintain a particular state.

User of the Library service (borrower) is not a system role (it is not a part of the Library). For the Library service, a user may be represented by its Library card. Library card can be considered to be a part of the Library system, more precisely of its interface, because it is used as a point of interaction between a user and the Library service.

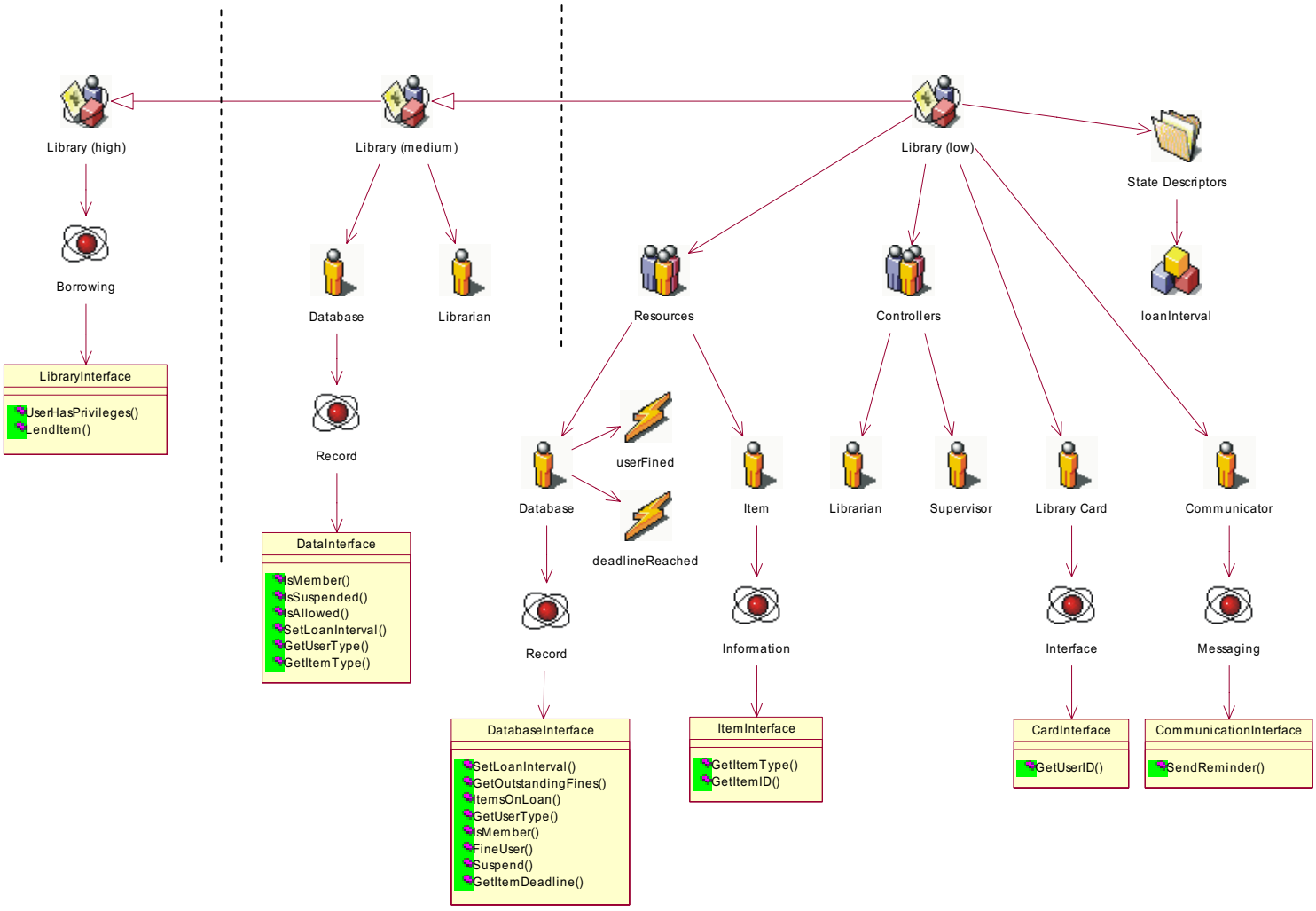


Figure 9-16 Management model of the Library in UML, with SPL stereotypes

The Library system is modelled at three levels of abstraction (three Dictionaries), which are connected with the relation of refinement/abstraction (Figure 9-16). A high-level management interface, which includes only two management operations, offers a simplified view of the Library service. This abstract view of the system was refined into more detailed models. The Library system actually implements the management interface defined in the lowest level Dictionary. Simple policy (*Lend item*), specified in terms of the abstract management operations from the high-level Dictionary, will be refined into a more complex, lower-level policy in terms of the real management operations implemented by the Library system. Such refined policy may be then executed against the system management interface.

Policies

The main goal of management operations in a library is to establish controlled access to its resources. Just like in a real library, we will not deal with the situation in which resources are not available – in order to borrow an item a user needs to get it from the shelves. A situation in which a user wants to reserve an item is not considered, because it doesn't trigger any particularly interesting management scenario.

In the Library example, our management strategy will be defined as follows:

- ***Objective***: To offer a quality library service.
- ***Goals***: Access control (privileged usage of the resources) and usage control (fair share of the resources).
- ***Policies*** (implement goals). In the Library example, the design of management policies will include enforcing of the following behaviour:

- *Access to the Library resources*. Access to the Library resources is conditional, and it requires several verifications. Policy may be used to implement a control of the access rights in a flexible manner. Even though in real libraries borrowing aspect of the Library service is human-controlled, it may be as well successfully automated.
- *Usage of the Library resources*. Not only the access, but also the usage of the Library resources is regulated by rules. Usage of the Library resources is limited in time, and fines are introduced for those who break the rules. This portion of the Library behaviour should be fully automated.

Access to the Library resources

From a user's point of view, the Library is just a lending service. It may be simply specified as follows:

item GetItem(userID, itemID) // if I say who I am and specify what I want, I'll get it

By the design, Library service is available to users, but we use a management policy to restrict access to the service only to the qualified users:

Qualified users may borrow items from the library.

IF
UserHasPrivileges(userID) = TRUE
THEN
LendItem(userID, itemID)

If we include abstract roles responsible for performing the actions, the detailed policy specification will look as follows:

Lend item

IF
Librarian ⇓ Supervisor ⇓ Database.UserHasPrivileges(userID) = TRUE
THEN
Supervisor ⇓ Librarian ⇓ Database.LendItem(userID, itemID)
WITH
METADATA
Description: "Qualified users may borrow items from the library",
Class: General management,
Goal: Operations control,
Action: Change of state,
Priority: 1

The predicate action is initiated by the *Librarian*, who asks the *Supervisor* to confirm user's privileges. *Supervisor* will check the user's privileges with the *Database*. If the privileges are confirmed, *Supervisor* will ask the *Librarian* to lend the item to the user by making a record of it in the *Database*.

In a more detailed specification, we may want to specify details of interactions that identify users and Library items, *userID* and *itemID* respectively, as follows:

Lend item

IF
AND
(userID = Librarian ⇓ LibraryCard.GetUserID(),
Librarian ⇓ Supervisor ⇓ Database.UserHasPrivileges(userID) = TRUE)
THEN
itemID = Librarian ⇓ Item.GetItemID(),
Supervisor ⇓ Librarian ⇓ Database.LendItem(userID, itemID)
WITH
METADATA
Description: "Qualified users may borrow items from the library",
Class: General management,
Goal: Operations control,
Action: Change of state,
Priority: 1

Usage of the Library resources

The Library management must establish an instrument of control over the way resources are used. In case when a misuse is detected, there will be a corrective action. The following two policies will be triggered in response to the *deadlineReached* event, in order to fine and notify a user.

Fine user

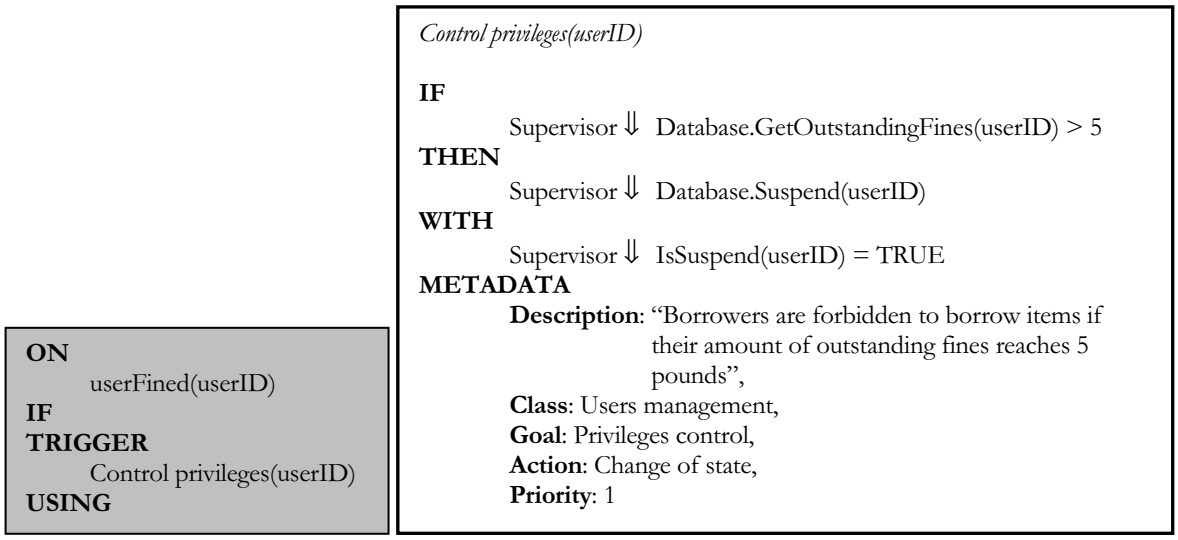
IF
Supervisor ⇓ Database.GetItemDeadline(itemID) < 00/00/0000
THEN
Supervisor ⇓ Database.FineUser(userID, itemID)
WITH
METADATA
Description: "If a borrower didn't return an item on time, fined him/her",
Class: Users management,
Goal: Privileges control,
Action: Change of state,
Priority: 2

Inform user

IF
Supervisor ⇓ Database.GetItemDeadline(itemID) = now, - 01/00/0000
THEN
Supervisor ⇓ Communicator.SendReminder(userID, itemID)
WITH
METADATA
Description: "Reminders shall be sent for overdue items one day after the item was due",
Class: Users management,
Goal: Privileges control,
Action: Change of state,
Priority: 2

ON
deadlineReached(userID)
IF
TRIGGER
Fine user(userID)
Inform user(userID)
USING

In order to control users' borrowing privileges, their status must be kept up-to-date. A user must pay his fines in order to preserve his status with the Library service. The following policy will be triggered in response to the *userFined* event, in order to check if the user can preserve his privileges with the Library.



Since both the event and policy use the same parameter, an explicit link between event and policy parameters need not to be specified.

Policy refinement

The SPL policy refinement concept we will explain on the example on the *Lend item* policy. The policy is specified in terms of the interactions from the high-level Dictionary (see Figure 9-16).

Lend item

IF
UserHasPrivileges(userID) = TRUE
THEN
LendItem(userID, itemID)
WITH
METADATA
Description: "Qualified users may borrow items from the library",
Class: General management,
Goal: Operations control,
Action: Change of state,
Priority: 1

In the Toaster example we had a number of small, relatively independent management activities, each of which was specified as a separate policy and triggered by events. In the Library, the lending procedure has a number of interrelated steps, which are not suitable to be specified as separate policies. For example, we could specify a policy to set the loan interval as follows:

Books lending interval for undergraduates

IF
AND
(GetUserType(userID) = "undergraduate",
GetItemType(itemID) = "book")
THEN
SetLoanInterval(userID, itemID, (now, +00/01/0000))
WITH
METADATA
Description: "Undergraduates can borrow books for one month",
Class: General management,
Goal: Operations control,
Action: Change of state,
Priority: 1

However, the policy above is just a part of the overall lending procedure, and it is not very useful in isolation. Therefore, the lending procedure is best to be described as an integral behavioural specification. It will come as a result of the refinement of interactions from the Lend item policy.

The *UserHasPrivileges()* interaction, from the high-level Dictionary, may be refined using the interactions from the medium-level Dictionary (see Figure 9-16). For that purpose, we will define the following strategy:

status UserHasPrivileges(userID)
EQUIVALENT
IF
AND
(IsMember(userID) = TRUE,
IsSuspended(userID) = FALSE,
IsAllowed(userID) = TRUE)
THEN
return = TRUE
ELSE
return = FALSE
USING
status ⇓ return

The *IsSuspended()* interaction may be further refined in terms of the interactions from the low-level Dictionary (see Figure 9-16):

status IsSuspended(userID)
EQUIVALENT
return = GetOutstandingFines(userID) > 5
USING
status ⇓ return

Borrowers can borrow up to their allowance (limit):

- Undergraduates may borrow 8 books. Books may be borrowed for 4 weeks. Undergraduates are forbidden to borrow periodicals.
- Postgraduates may borrow 16 books or periodicals. Periodicals may be borrowed for one week. Books may be borrowed for one month.
- Teaching staff may borrow 24 books or periodicals. Periodicals may be borrowed for one week. Books may be borrowed for one year.

To implement this rule, we will refine the *IsAllowed()* interaction by specifying the following strategy:

```
        status IsAllowed(userID, itemID)
EQUIVALENT
        IF
            OR
            (           AND
                        (GetUserType(userID) = "undergraduate",
                        ItemsOnLoan(userID) < 8,
                        GetItemType(itemID) = "book"),
                        AND
                        (GetUserType(userID) = "postgraduate",
                        ItemsOnLoan(userID) < 16),
                        AND
                        (GetUserType(userID) = "staff",
                        ItemsOnLoan(userID) < 24)
            )
        THEN
            return = TRUE
        ELSE
            return = FALSE
USING
        status ⇓  return
```

The *LendItem()* interaction, from the high-level Dictionary, may be directly refined using the interactions from the low-level Dictionary as follows:

```
        LendItem(userID, itemID)
EQUIVALENT
        IF
            GetUserType(userID) = "undergraduate"
        THEN
            IF
               GetItemType(itemID) = "book"
            THEN
               loanInterval = now, +00/01/0000
            ELSE
               loanInterval = now, now

        IF
            GetUserType(userID) = "postgraduate"
        THEN
            IF
               GetItemType(itemID) = "book"
            THEN
               loanInterval = now, +00/01/0000
            ELSE
               loanInterval = now, +07/00/0000

        IF
            GetUserType(userID) = "staff"
        THEN
            IF
               GetItemType(itemID) = "book"
            THEN
               loanInterval = now, +00/00/0001
            ELSE
               loanInterval = now, +07/00/0000

        SetLoanInterval(userID, itemID, loanInterval)
USING
```

The complete pattern of refinement for our policy may be seen in Figure 9-17.

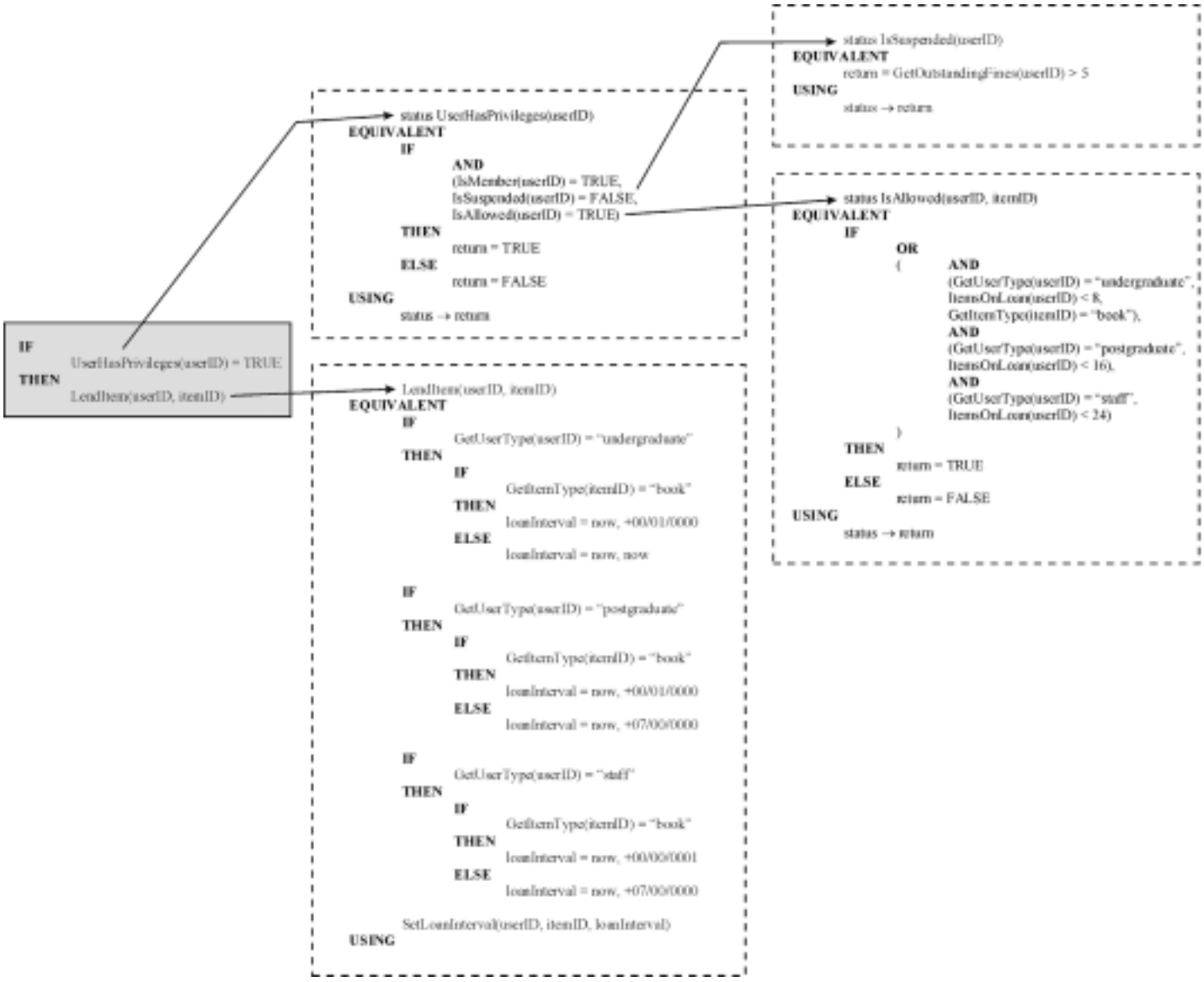


Figure 9-17 Pattern of refinement for the Library example

Using the pattern of refinement, the *Lend item* policy, specified in terms of the abstract management operations, may be refined in two steps into a low-level policy that is executable against the Library management interface.

First step refinement (medium level policy)

Lend item

IF
    AND
    (IsMember(userID) = TRUE,
    IsSuspended(userID) = FALSE,
    IsAllowed(userID) = TRUE)
THEN
    IF
        GetUserType(userID) = "undergraduate"
    THEN
        IF
            GetItemType(itemID) = "book"
        THEN
            loanInterval = now, +00/01/0000
        ELSE
            loanInterval = now, now

    IF
        GetUserType(userID) = "postgraduate"
    THEN
        IF
            GetItemType(itemID) = "book"
        THEN
            loanInterval = now, +00/01/0000
        ELSE
            loanInterval = now, +07/00/0000

    IF
        GetUserType(userID) = "staff"
    THEN
        IF
            GetItemType(itemID) = "book"
        THEN
            loanInterval = now, +00/00/0001
        ELSE
            loanInterval = now, +07/00/0000

    SetOnLoan(userID, itemID, loanInterval)
WITH
METADATA
    Description: "Qualified users may borrow items from the library",
    Class: General management,
    Goal: Operations control,
    Action: Change of state,
    Priority: 1

Second step refinement (low level policy)

Lend item

```
IF
        AND
        (
                IsMember(userID) = TRUE,
                GetOutstandingFines(userID) > 5 = FALSE,
                OR
                (               AND
                                (GetUserType(userID) = "undergraduate",
                                ItemsOnLoan(userID) < 8,
                                GetItemType(itemID) = "book"),
                                AND
                                (GetUserType(userID) = "postgraduate",
                                ItemsOnLoan(userID) < 16),
                                AND
                                (GetUserType(userID) = "staff",
                                ItemsOnLoan(userID) < 24)
                )
        )
THEN
        IF
                GetUserType(userID) = "undergraduate"
        THEN
                IF
                   GetItemType(itemID) = "book"
                THEN
                   loanInterval = now, +00/01/0000
                ELSE
                   loanInterval = now, now

        IF
                GetUserType(userID) = "postgraduate"
        THEN
                IF
                   GetItemType(itemID) = "book"
                THEN
                   loanInterval = now, +00/01/0000
                ELSE
                   loanInterval = now, +07/00/0000

        IF
                GetUserType(userID) = "staff"
        THEN
                IF
                   GetItemType(itemID) = "book"
                THEN
                   loanInterval = now, +00/00/0001
                ELSE
                   loanInterval = now, +07/00/0000

        SetOnLoan(userID, itemID, loanInterval)
WITH
METADATA
        Description: "Qualified users may borrow items from the library",
        Class: General management,
        Goal: Operations control,
        Action: Change of state,
        Priority: 1
```

Appendix D

EXAMPLES: SEMANTIC ANALYSIS AND CONFLICT HANDLING IN SPL

Even though the primary goal of SPL design was to create a specification language for management policies, SPL also supports the entire SPF management process. Thus, SPL must address the issues related to control of the policy-based management process, in order to enable its automation:

- *Exception handling.* In SPF, exceptions are events that may be raised by the management framework or by the managed system. Examples of event handling were presented in the Toaster and the Library examples (see Appendix C).
- *Semantic analysis of management steps*. Policies are management specifications that address particular management issues, so they are focused. However, there is also a need for certain general rules, concerned with the management process as a whole. Policies will be triggered every time their preconditions are satisfied, but we may want also to fine tune such behaviour by analysing policy semantics before triggering. This is very useful for the implementation of policy versioning and scheduling, as well as for enabling/disabling policies.
- *Conflict resolution.* Conflict analysis and handling must guarantee that individual policy goals make sense in the context of the overall management activities and current system state. In other words, individual policies should be in harmony with other management policies that are triggered at the same time.

Semantic analysis

There can be different styles in implementing versioning control for management policies (see also the example in section 4.2.2.2):

- The applicability of policies may be limited to a particular time period, or made valid after a particular date. In the following example a rule checks the date and discards policies if appropriate.

The following semantic rule will ignore policies created before 12/02/2001:

IF
    Policy.Accounting.Created < 12/02/2001
THEN
    Discard(Policy)
WITH
METADATA
    Description: "Ignore policies created before 12/02/2001",
    Class: Semantic analysis,
    Goal: Version control,
    Action: Elimination,
    Priority: 1

- Policy Accounting information can be extended to support the time component of management – planning and scheduling. Semantic rules can access the Accounting information to determine if a policy is valid at that particular moment in time (the mechanism of restrictions implements the time component). We can specify policies for different time of the day, week, month and/or year. Policies will be triggered only if they are valid at the triggering time. For example:

```
IF
        Policy.Accounting.ValidityPeriod != now
THEN
        Dicard(Policy)
WITH
METADATA
        Description: "Ignore invalid policies",
        Class: Semantic analysis,
        Goal: Validity control,
        Action: Elimination,
        Priority: 1
```

- If there is a need in the system to explicitly implement policy enabling/disabling, an attribute (for example Validity) may be added to the policy Accounting information for that purpose. In such a case, semantic rules can check if a policy is enabled in the following way:

```
IF
        Policy.Accounting.Validity = FALSE
THEN
        Discard(Policy)
WITH
METADATA
        Description: "Ignore invalid policies",
        Class: Semantic analysis,
        Goal: Validity control,
        Action: Elimination,
        Priority: 1
```

Conflict handling

The key to a successful conflict management is in the policy metadata, because richer metadata specifications will enable more sophisticated and more general analysis of policy semantics. The policy metadata may encompass various semantic data:

- *Subject*, which may be used to define the policy authority. For example, a subject may be defined as: *System/Access Control*. A subject with less authority might be specifies as: *System/Databases/Personal Data/Access Control*. So, the authorities of two policies could be simply compared as *TriggeringPool.Policy.Accounting.Subject* > *Policy.Accounting.Subject*.
- *Target*, which defines the organizational scope of a policy. For example, a policy target may be specified as follows: *System/Servers/Cache/Maintenance*. The length of the target path defines the generality of policies – the longer the path the more specific policy will be. A policy target defined in this way may be used to implement the concept of hierarchical management domains.
- *Modality* may be used to specify the general semantics of policy actions. It may include positive or negative authorization (A+/A-), positive or negative obligation (O+/O-), etc.

In addition, a policy class may be specified at various levels of details. For example, policy class information may look like this: *Load Balancing/Code Mobility/Triggering*.

Main issues related to policy conflicts were identified and discussed in [Moffett 1994], and they will be briefly addressed in the following SPL examples.

Positive-Negative Conflict of Modalities

Direct positive-negative conflict occurs when a subject is both authorised and forbidden for the same goal on an object, or both required and deterred: for example *'The Accounts Supervisor is permitted to sign payment cheques'* and also *'The Accounts Supervisor is forbidden to sign payment cheques'* (from [Moffett 1994]). In SPL, such a conflict may be handled in the following way:

IF
AND
(Policy.Accounting.Subject = TriggeringPool.Policy.Accounting.Subject,
Policy.Accounting.Target = "Payment cheque",
Policy.Accounting.Modality = A+,
Policy.Metadata.Action = "Sign"
TriggeringPool.Policy.Accounting.Target = "Payment cheque",
TriggeringPool.Policy.Policy.Accounting.Modality = A-,
TriggeringPool.Policy.Metadata.Action = "Sign")
THEN
Discard(Policy)
WITH
METADATA
Description: "If same person is both authorized and forbidden to sign payment cheques, ignore the authorization",
Class: Colflict Resolution,
Goal: Resolution of positive-negative conflict of modalities,
Action: Elimination,
Priority: 1

The previous rule may become more useful if we make it more general:

IF
AND
(Policy.Accounting.Subject = TriggeringPool.Policy.Accounting.Subject,
Policy.Accounting.Target = TriggeringPool.Policy.Accounting.Target,
Policy.Accounting.Modality = A+,
TriggeringPool.Policy.Policy.Accounting.Modality = A-)
THEN
Discard(Policy)
WITH
METADATA
Description: "When two policies have same targets and subjects, give priority to Negative Authorization over Positive Authorization",
Class: Conflict Resolution,
Goal: Resolution of positive-negative conflict of modalities,
Action: Elimination,
Priority: 1

The role of the metadata and policy attributes may be significant in the process of conflict analysis. For example, there can be a pair of system security policies specified as follows:

Check security clearance

IF
User.GetSecurityClearance() > highSecurityLimit
THEN
AuthorizeAccess(User, TRUE)
WITH
METADATA
Description: "Grant the access to users with height security clearance",
Class: Access Control,
Goal: Security,
Action: Filtering,
Priority: 2
ACCOUNTING
Author: John Smith
Created: 14/09/03
Modality: A+
Target: System

Check location

IF
User.GetLocation() != internal
THEN
AuthorizeAccess(User, FALSE)
WITH
METADATA
Description: "Only the internal users can access the system resources",
Class: Access Control,
Goal: Security,
Action: Filtering,
Priority: 1
ACCOUNTING
Author: Peter Brown
Created: 14/01/03
Modality: A-
Target: System/Resources

To resolve the A+/A- conflict, we will need to apply one of the following principles for establishing precedence:

- *Negative policies always have priority*. It is quite common for negative authorization policies to always override positive ones so that a forbidden action will never be permitted.

IF
AND
(Policy.Metadata.Class == TriggeringPool.Policy.Metadata.Class
Policy.Accounting.Modality = "A+",
TriggeringPool.Policy.Accounting.Modality = "A-",
Policy.Metadata.Action == TriggeringPool.Policy.Metadata.Action)
THEN
Discard(TriggeringPool.Policy)
WITH
METADATA
Description: "If two authorization policies have the same class and course of action but opposite sign, discard the positive one",
Class: Conflict Resolution,
Goal: Management consistence,
Action: Elimination,
Priority: 1

∉ *Assigning explicit priorities.*

IF
AND
(Policy.Metadata.Class == TriggeringPool.Policy.Metadata.Class,
Policy.Metadata.Action == TriggeringPool.Policy.Metadata.Action)
THEN
Discard(min(Policy.Metadata.Priority, TriggeringPool.Policy.Metadata.Priority))
WITH
METADATA
Description: "If two policies of the same class have the same course of action, discard the one with a lower priority",
Class: Conflict Resolution,
Goal: Management consistence,
Action: Elimination,
Priority: 1

∉ *Distance between a policy and the managed objects.* In the concept of distance between a rule (policy) and a class of objects it refers to, the distance indicates the relevance of the policy to the objects. Priority is given to policies applying to the class closer (more specific) in the inheritance hierarchy.

IF
AND
(Policy.Metadata.Class == TriggeringPool.Policy.Metadata.Class,
Policy.Metadata.Action == TriggeringPool.Policy.Metadata.Action)
THEN
Discard(min(Policy.Metadata.Target, TriggeringPool.Policy.Metadata.Target))
WITH
METADATA
Description: "If two policies of the same class have the same course of action, discard the more general one",
Class: Conflict Resolution,
Goal: Management consistence,
Action: Elimination,
Priority: 1

Conflict between Imperatival and Authority Policies

Conflict between imperatival and authority policies (O+/A-) occurs when a subject is both required to initiate and forbidden to carry out an action on an object: for example *'The Accounts Supervisor is obliged to sign all payment cheques which have been approved by the Accounting Manager'* and also *'The Accounts Supervisor is forbidden to sign payment cheques'*.

IF
AND
(Policy.Accounting.Subject = TriggeringPool.Policy.Accounting.Subject,
Policy.Accounting.Target = "Payment cheque",
Policy.Accounting.Modality = O+,
Policy.Metadata.Action = "Sign"
TriggeringPool.Policy.Accounting.Target = "Payment cheque",
TriggeringPool.Policy.Policy.Accounting.Modality = A-,
TriggeringPool.Policy.Metadata.Action = "Sign")
THEN
Discard(Policy)
WITH
METADATA
Description: "If the same person is both obligated and forbidden to sign payment cheques, ignore the obligation",
Class: Colflict Resolution,
Goal: Resolution of conflict between imperatival and authority policies,
Action: Elimination,
Priority: 1

The previous rule may become more general in the following way:

IF
AND
(Policy.Accounting.Subject = TriggeringPool.Policy.Accounting.Subject,
Policy.Accounting.Target = TriggeringPool.Policy.Accounting.Target,
Policy.Accounting.Modality = O+,
TriggeringPool.Policy.Policy.Accounting.Modality = A-)
THEN
Discard(Policy)
WITH
METADATA
Description: "When two policies have same targets and subjects, give priority to negative authorization over obligation",
Class: Conflict Resolution,
Goal: Resolution of conflict between imperatival and authority policies,
Action: Elimination,
Priority: 1

Conflict of Priorities for Resources

Clearly when two or more policies between them require the use of more resource than is available, there is a conflict for resources [Moffett 1994]. Such conflicts may be resolved in the following manner:

When the load in the Records Database is high, restrict and prioritize access.

IF
AND
(Policy.Accounting.Target = “Records Database”,
TriggeringPool.Policy.Accounting.Target = “Records Database”,
Records Database.GetLoad() = “high”)
THEN
Discard(min(Policy.Metadata.Priority, TriggeringPool.Policy.Metadata.Priority))
WITH
METADATA
Description: “When the load in the Records Database is high, restrict and prioritize access”,
Class: Conflict Resolution,
Goal: Resolution of conflict of priorities for resources,
Action: Elimination,
Priority: 1

Using the semantic analysis, we may restrict access to resources:

Don’t perform maintenance when load in the Records Database is high.

IF
AND
(Policy.Accounting.Target = “Records Database”,
Policy.Accounting.Goal = “Maintenance”,
Records Database.GetLoad() = “high”)
THEN
Discard(Policy)
WITH
METADATA
Description: “Don’t perform maintenance when load in the Records Database is high”,
Class: Semantic analysis,
Goal: Performance management,
Action: Elimination,
Priority: 1

Conflict of Duties

An example of the principle is *'The same person must not be allowed both to enter a payment and sign the payment cheque'*, which would be violated by two such policies as *'The Accounts Supervisor is authorised to enter payment information'* and *'The Accounts Supervisor is authorised to sign payment cheques'*. In SPL, such a conflict of duties may be resolved in the following way:

IF
AND
(Policy.Accounting.Subject = "Accounts Supervisor",
Policy.Accounting.Target = "Payment cheque",
Policy.Accounting.Modality = A+,
Policy.Metadata.Action = "Sign"
TriggeringPool.Policy.Accounting.Subject = "Accounts Supervisor",
TriggeringPool.Policy.Accounting.Target = "Payment cheque",
TriggeringPool.Policy.Policy.Accounting.Modality = A+,
TriggeringPool.Policy.Metadata.Action = "Enter payment information")
THEN
Discard(TriggeringPool.Policy)
WITH
METADATA
Description: "Accounts Supervisor must not be allowed to enter payment information if it is authorized to sign payment cheques",
Class: Conflict Resolution,
Goal: Resolution of conflict of duties,
Action: Elimination,
Priority: 1

The previous policy may be made more general in the following way:

IF
AND
(Policy.Accounting.Subject = TriggeringPool.Policy.Accounting.Subject,
Policy.Accounting.Target = "Payment cheque",
Policy.Accounting.Modality = A+,
Policy.Metadata.Action = "Sign"
TriggeringPool.Policy.Accounting.Target = "Payment cheque",
TriggeringPool.Policy.Policy.Accounting.Modality = A+,
TriggeringPool.Policy.Metadata.Action = "Enter payment information")
THEN
Discard(TriggeringPool.Policy)
WITH
METADATA
Description: "Same person must not be allowed to enter payment information and to sign payment cheques",
Class: Conflict Resolution,
Goal: Resolution of conflict of duties,
Action: Elimination,
Priority: 1

Conflict of Interests

When the subjects of two authority policies overlap, this implies that the same subject can perform management tasks on two different sets of targets. In some application-defined circumstances there is a conflict, known as a conflict of interests [Moffett 1994]. The best example of this is when a merchant bank is acting as adviser to two different organisations, e.g. on a takeover bid for one client while advising other clients on investment decisions which would be influenced by knowledge of the takeover.

Lots of information would be required to successfully detect and resolve conflicts like this. However, some preventive measures (such as to postpone the management activity) may be taken in doubtful situations. For example:

First class clients should not be advised on delicate matters at the same time.

IF
AND
(Policy.Accounting.Subject = TriggeringPool.Policy.Accounting.Subject,
Policy.Accounting.Target = "First class client",
Policy.Metadata.Action = "Delicate advise"
TriggeringPool.Policy.Accounting.Target = "First class client",
TriggeringPool.Policy.Metadata.Action = "Delicate advise")
THEN
Delay(Policy)
WITH
METADATA
Description: "First class clients should not be advised on delicate matters simultaneously",
Class: Conflict Resolution,
Goal: Resolution of conflict of duties,
Action: Delay,
Priority: 1

Multiple management

A potential conflict may arise from multiple managers having authority over the same object. In some cases multiple managers of an object are forbidden on the grounds of potential conflict, e.g. generally each worker has one line manager [Moffett 1994]. For example, the following is a conflict resolution rule based on policy modality:

Authorizations given to branch employees by branch managers override authorizations given by central managers.

IF
AND
(Policy.Accounting.Subject = "Branch Manager",
Policy.Accounting.Target = "Branch Employee",
Policy.Accounting.Modality = A,
TriggeringPool.Policy.Accounting.Subject = "Central Manager",
TriggeringPool.Policy.Accounting.Target = "Branch Employee",
TriggeringPool.Policy.Policy.Accounting.Modality = A)
THEN
Discard(TriggeringPool.Policy)
WITH
METADATA
Description: "Authorizations given to branch employees by branch managers override authorizations given by central managers",
Class: Conflict Resolution,
Goal: Resolution of conflict of multiple management,
Action: Elimination,
Priority: 1

The previous conflict may be also prevented using semantic rules, as follows:

Authorizations given to branch employees by central managers are not allowed.

IF
AND
(Policy.Accounting.Subject = "Central Manager",
Policy.Accounting.Target = "Branch Employee",
Policy.Accounting.Modality = A)
THEN
Discard(Policy)
WITH
METADATA
Description: "Authorizations given to branch employees by central managers are not allowed",
Class: Conflict Resolution,
Goal: Resolution of conflict of multiple management,
Action: Elimination,
Priority: 1

Self management

Self management is when a manager is managing himself. Such conflicts may be solved using semantic rules. For example:

Branch employees may not authorize themselves.

IF
AND
(Policy.Accounting.Subject = Policy.Accounting.Target,
Policy.Accounting.Modality = A,
Policy.Accounting.Target = "Branch Employee")
THEN
Discard(Policy)
WITH
METADATA
Description: "Branch employees may not authorize themselves",
Class: Conflict Resolution,
Goal: Resolution of conflict of self management,
Action: Elimination,
Priority: 1