A Business-Driven Approach to Policy Optimization

Issam Aib

UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE A PARIS

THESE DE DOCTORAT DE
L'UNIVERSITE PIERRE ET MARIE CURIE

Spécialité
Informatique et Réseaux

Présentée par

M. **Issam Aib**

Pour Obtenir le grade de

DOCTEUR DE l'UNIVERSITE PIERRE ET MARIE CURIE

Sujet de thèse

Une approche orientée Buts d'Affaires pour l'Optimisation des Politiques

soutenue le mercredi 11 juillet 2007.

devant le jury composé de :

Pr.	Guy	Pujolle	Directeur	Univ. Paris 6, France.
Pr.	Raouf	Boutaba	Directeur	Univ. Waterloo, Canada.
Pr.	George	Pavlou	Rapporteur	Univ. Surrey, UK.
Pr.	Olivier	Festor	Rapporteur	INRIA Nancy, France.
Pr.	Omar	Cherkaoui	Examinateur	Univ. UQAM, Canada.
Pr.	Pierre	Sens	Examinateur	Univ. Paris 6, France.
M.	Claudio	Bartolini	Examinateur	HP Labs, CA, USA.

A Business-Driven Approach to Policy Optimization

by

Issam Aib

A thesis
presented to the University Pierre & Marie Curie

in fulfillment of the
thesis requirement for the degree of

Doctor of Philosophy
in
Computer Science

Defended on 11 July 2007 before the honored committee:

Pr.	Guy	Pujolle	Advisor	Univ. Paris 6, France.
Pr.	Raouf	Boutaba	Advisor	Univ. Waterloo, Canada.
Pr.	George	Pavlou	Reporter	Univ. Surrey, UK.
Pr.	Olivier	Festor	Reporter	INRIA Nancy, France.
Pr.	Omar	Cherkaoui	Examiner	Univ. UQAM, Canada.
Pr.	Pierre	Sens	Examiner	Univ. Paris 6, France.
M.	Claudio	Bartolini	Examiner	HP Labs, CA, USA.

إلى أمي الحبيبة آمنة
أبي العزيز عبد الله
وأختي الغالية دلال

To my Mother Amina,
my Father Abdallah,
and my Sister Dellel.

A ma mère Amina,
mon père Abdallah,
et ma Soeur Dellel.

Abstrat

Le paradigme de gestion orientée buts d'affaires vise à ce que les actions de configuration de bas niveau d'un système informatique soient dérivées et actionnées d'une façon qui permet d'accomplir et optimiser les buts de haut niveau initiaux du fournisseur de service. Un ensemble de conditions est nécessaire pour permettre un tel paradigme de gestion. Celles-ci incluent la capacité de spécifier les buts d'affaire de haut niveau ainsi que des Contrats de Niveau de Service (SLAs), le raffinement de ces buts et SLAs en des actions de configuration de bas niveau, aussi bien que la validation de ces actions de bas niveau contre le comportement de haut niveau indiqué. Enfin, la boucle de surveillance aide à augmenter la performance et correction des futures exécutions du système en identifiant et en remédiant à des défauts observés lors d'exécutions antérieures.

La gestion par politiques a émergé comme mécanisme de préféré pour la mise en œuvre de la gestion orientée buts d'affaires vu la promesse qu'elle offre d'alléger le coût de configuration et d'entretien de systèmes automatisés complexes en fournissant un modèle de gestion qui sépare la fonctionnalité d'exécution de la logique comportementale.

Bien qu'il y ait eu des efforts considérables, à la fois dans l'industrie et le milieu universitaire, sur la spécifications et l'exécution des politiques et des SLAs, il y a eu peu d'efforts sur le raffinement des buts de haut niveau en des politiques de bas niveau. En plus, il y a besoin d'étudier le comportement d'exécution des politiques et de développer des modèles et des techniques qui emmènent plus loin le raffinement orienté-politiques des buts par la considération de la dynamique du comportement des politiques au temps d'exécution et comment ceux-ci peuvent se relier à l'optimisation des buts d'affaires de haut niveau.

Dans cette thèse, nous contribuons à la gestion orientée buts d'affaires des systèmes informatiques à différents niveaux d'abstraction des politiques. D'abord, nous proposons un cadre de gestion qui lie ensemble les buts d'affaires, les accords de niveau de service (SLAs), et l'exécution des politiques. Puis, nous étudions les formalismes existants de spécification des politiques et des SLAs et proposons une spécification orientée politiques des SLAs qui par conséquent s'adapte bien pour une approche de raffinement basée sur les politiques.

La contribution principale de la thèse se relie cependant aux mécanismes d'analyse des politiques. Nous contribuons à l'analyse off-line de la validité et l'uniformité des politiques de bas niveau en ce qui concerne les buts de haut niveau et les SLAs. D'une part, nous identifions un type de test des politiques qui aide à détecter un nouveau type de comportement anormal que nous appelons *boucles de politique*. Ni détecter ni résoudre ce type de comportement n'est facile à faire dans le cas général. Nous montrons cependant comment il peut être fait pour l'exemple d'application étudiée.

D'autre part, nous suggérons un nouveau type d'analyse, que nous appelons *analyse dynamique des politiques*, et qui correspond à l'investigation de la façon à influencer l'ordonnancement de l'exécution des politiques au temps d'exécution afin de réaliser une meilleure optimisation des buts d'affaires de haut niveau. En conclusion, nous identifions le besoin critique de simuler les solutions de gestion par politiques et nous présentons $\mathcal{PS}$, qui est un simulateur de politiques que nous avons développé afin de permettre l'expérimentation, l'analyse, la validation, et l'optimisation des solutions de gestion par politique avant leur déploiement dans une plateforme de gestion réelle.

Abstract

Business-driven management is a management paradigm wherewith low-level system configuration actions are derived and operated with a focus on fulfilling and optimizing the initial high-level business goals of a service provider. A set of requirements is needed for enabling such a management paradigm. These include the ability to specify high-level business goals and Service Level Agreements (SLAs), the refinement of these goals and SLAs into low-level configuration actions, as well as the validation of these low-level actions against the specified high-level behavior. A subsequent monitoring loop helps in enhancing future system performance by identifying and remedying defects learned from past operation.

Policy-based management has emerged as a preferred mechanism for implementing business-driven management due to the promise to alleviate the configuration and maintenance cost of complex computerized systems by providing a management model that separates implementation functionality from the behavioral logic.

Although there has been considerable effort in industry and academia regarding the specification and implementation of policies and SLAs, little focus has been given to the refinement of high-level goals into low-level policies. The interest in the investigation of policy refinement has gained interest only recently. Furthermore, there is a need to investigate the runtime behavior of policy and develop models and techniques that take the goal-oriented policy refinement a step further by considering the dynamics of policy behavior at runtime and how these can relate to the optimization of high-level business goals.

This thesis contributes to the business-driven management of IT systems at different levels. First, a management framework that links together business goals, Service Level Agreements (SLAs), and policy operation is proposed. Then existing policy and SLA specification formalisms are investigated and a specification for SLAs that is policy-driven is proposed.

The major contribution of the thesis relates to policy analysis mechanisms. This work contributes to the off-line analysis of low-level policies for validity and consistency with regard to high-level goals and SLAs. On one hand, a policy test type is identified that helps in detecting a new type of abnormal behavior we name *policy loops*. Neither detecting nor resolving this type of behavior is easy to do in the general case. This work however shows how it can be done for the studied example application. Moreover, a new type of analysis is proposed, that we name *policy dynamic analysis*, which corresponds to the investigation of how to influence the scheduling of policy execution at runtime so as to achieve better optimization of the high-level business goals. Finally, this work identifies the critical need for simulating policy solutions and presents $\mathcal{PS}$,

which is a policy simulator tool that we developed for the purpose of experimenting, analyzing, validating, and optimizing policy solutions before they get to their actual deployment into a real scale environment.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. First to thank is my thesis director, Professor Guy Pujolle, for allowing me to start and continue this research, his support, his patience, his vivid spirit, as well as his constant belief that I can finish this PhD.

I am deeply indebted to my advisor, Professor Raouf Boutaba, from the University of Waterloo, whose financial support, technical guidance, stimulating suggestions, strong as well as warm advices, patience, and encouragement significantly helped in enhancing the quality of my research. The time I have spent under his guidance at the University of Waterloo have certainly influenced me both at the professional as well as the personal levels.

My warm thanks to Abdel Boulmakoul, Claudio Bartolini, and all the MBO team from HP Labs. The two months that I have spent in research work with them in Bristol have been quite helpful.

I would like also to thank Professor Nazim Agoulmine for his assistance and encouragement at the beginning of my PhD.

My thanks to my previous student, now a close colleague and friend, Badis Tebbani, with whom parts of this research has been realized (appendix B).

My thanks also to Benoit Campedell for his personal support and encouragement; as well as for hosting me at ISEP during the first two years of my research.

My sincere thanks to my thesis committee members, especially Professors Olivier Festor and George Pavlou for kindly accepting to report on this thesis within a tight schedule and for their valuable feedback.

I owe my loving thanks to my sister Dellel, who saw me from day one graduating and in a way is a proxy for my mother. My gratitude naturally extends to my brother in law Djamel Rezgui who was always there when needed and also for the delicious authentic tea he has always made available to his guests.

Not forgetting to mention my close friends during the Paris times: Abdelghani Chibani, Wissam Fawaz, Mourad Belkacem, and many others.

I owe also a delightful thanks to the big team of countless quality friends at the University of Waterloo who made my two years stay in this lovely city quite enjoyable and fruitful.

Finally, I would like to give a special thanks to Gloria Ichim (yet another great friend from Waterloo!) for the scrutinized proofreading of this thesis.

Publications

Journal papers

1. Issam Aib and Raouf Boutaba, "PS: A Policy Simulator", IEEE Communications Magazine, Network Management Series, April 2007, Vol. 45, No. 4, pp. 130-137.
2. Issam Aib and Raouf Boutaba, "On Leveraging Policy-Based Management for Maximizing Business Profit", submitted to the IEEE Transactions on Network and Service Management (TNSM) in December 2006 (under review).

Conference papers

1. Issam Aib and Raouf Boutaba "*Business-driven optimization of policy-based management solutions*", the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM 2007), 21-25 May, Munich, Germany. "**Best student paper award**".
2. Issam Aib, Nazim Agoulmine, and Guy Pujolle; "*A Multi-Party Approach to SLA Modeling, Application to WLANs*", IEEE Consumer Communication and Networking Conference, CCNC, January 2005, Las Vegas, USA.
3. Issam Aib, Mathias Salle, Claudio Bartolini, Abdel Boulmakoul, and Raouf Boutaba; "Business-Aware Policy-Based Management", IEEE/IFIP Workshop on Business Driven IT Management (BDIM 2006), in conjunction with NOMS 2006, 3-7 April, Vancouver, Canada.
4. Issam Aib, Nazim Agoulmine, Mauro Sergio Fonseca, Guy Pujolle, "*Analysis of Policy Management Models and Specification Languages*", IFIP International Conference on Network control and engineering for QoS, Security and Mobility, Net-Con, October 2003, Muscat, Oman.
5. Issam Aib and Raouf Boutaba, "*Utility Driven Generation and Scheduling of Management Actions*", 13th HP OpenView University Association workshop (HP-OVUA 2006), short paper, 21-24 May, Nice, Côte d'Azur, France.
6. Issam Aib, Mathias Sallé, Claudio Bartolini, and Abdel Boulmakoul; "A *Business Driven Management Framework for Utility Computing Environments*", 9th IFIP/IEEE International Symposium on Integrated Network Management (IM 2005), short paper, 16-19 May 2005, Nice, France.
7. Badis Tebbani, Issam Aib, et Guy Pujolle, "*GXLA, a language for the Specification of Service Level Agreements*", IFIP international conference on Autonomic Networking, SMARTNET track, 2006, Sept 27-29, Paris, France.

Contents

List of Figures

Listings

Chapter 1

Introduction

Network and distributed systems management is gearing from localized device-centric management towards high-level service and business oriented management. On one hand, this evolution is being pushed by the ever increasing diversity and complexity of networked devices and applications. On the other hand, corporate and end user customers are increasingly looking for more scrutiny of the service assurance process. This need has been more accentuated by the accelerated evolution and usage of on-demand services, increased automation of business-to-business interactivity, and short-time service life cycles.

Moreover, the relative cost of Information Technology (IT) system management is ever increasing compared to the cost of the infrastructure itself. In 2005, operations labor costs accounted for over 70% of CIO budgets, with hardware, software and services costs taken together making up less than 30% [2–4].

To reduce management cost, the industry has been exploring various ways to minimize the complexity and overhead of system management. Policy-based management (PBM) has emerged as a practical alternative for achieving this goal due to the fundamental feature it offers of allowing the behavior of the system to be specified apart from the actual implementation actions required to fulfill those specifications. Hence, it promises to reduce management costs while simultaneously improving quality of service (QoS) and dynamic adaptability to change.

Policy-based management is currently present at the heart of a multitude of management architectures and paradigms with such diversified prefixes as SLA-driven, Business-driven, autonomous, adaptive, and self management. This thesis focuses on quality assurance policies and how they relate to high-level service specifications and business goals. Policies are also extensively used in the security arena although we will not focus on their analysis. The important aspect here is that being able to use policies to govern an information system brings the key advantage of uniform management by using a unified approach for both quality assurance and security enforcement.

Despite the fact that research on policy-based management has been going on for more than a decade, it is still difficult to put into practice. The challenges are attributed to the theoretical and practical difficulties in proving not only the correctness but also the efficiency of policy-based solutions when it comes to the management of real scale systems with hundreds or even millions of policies interacting in a dynamic way.

A number of policy languages and architectures have been proposed. However, effective techniques for refinement and consistency/completeness analysis remain to be developed. It is therefore reasonable that, although tempting, venturing into a full policy-based solution for managing one's enterprize infrastructure remains difficult to justify.

System performance also lies at an equal level of importance. While it is true that policy-based solutions promise dynamism and flexibility, they often come with no guarantee of high performance. Verma [5] states that policy-based management solutions should not be considered a case of expert systems because of the strong weight of the performance parameter they have to carry with them. Work on the benchmarking of PBM solutions is also marked by great scarcity. It is in fact insufficient to provide a PBM solution which works, also must it be as efficient as existing legacy solutions if not better. In this regard, the maximization of business profit should represent the crucial goal that any QoS PBM solution should aim to achieve.

1.1 Thesis Contributions

This thesis contributes to the policy-driven management of information systems in three main aspects, which are gathered under the Business Driven Management Framework (BDMF). The BDMF seeks an efficient and flexible policy and business-driven management of information systems. It suggests a way to drive the management of system resources and services from the business point of view. Most of the times, when tradeoff-kind of decisions are to be made, system managers use heuristics in order to determine which of the option available to them guarantees the minimum cost or least disruption to the service. But unless the impact of carrying out the chosen course of action onto the business layer is understood, one may run the risk of solving the wrong problem optimally. Because of this, the BDMF was designed according to the principle of making information that pertains to the business visible from the resource-level and vice versa. The BDMF is presented in section 3.4.

We present a study we conducted on the formalisms used for the specification of SLAs and policies at different levels of abstraction (chapter 3). As the ability to specify and manipulate Service Level Agreements (SLA) constitutes an important component of business-driven management, we propose a generic SLA information model, named GSLA, and a corresponding XML

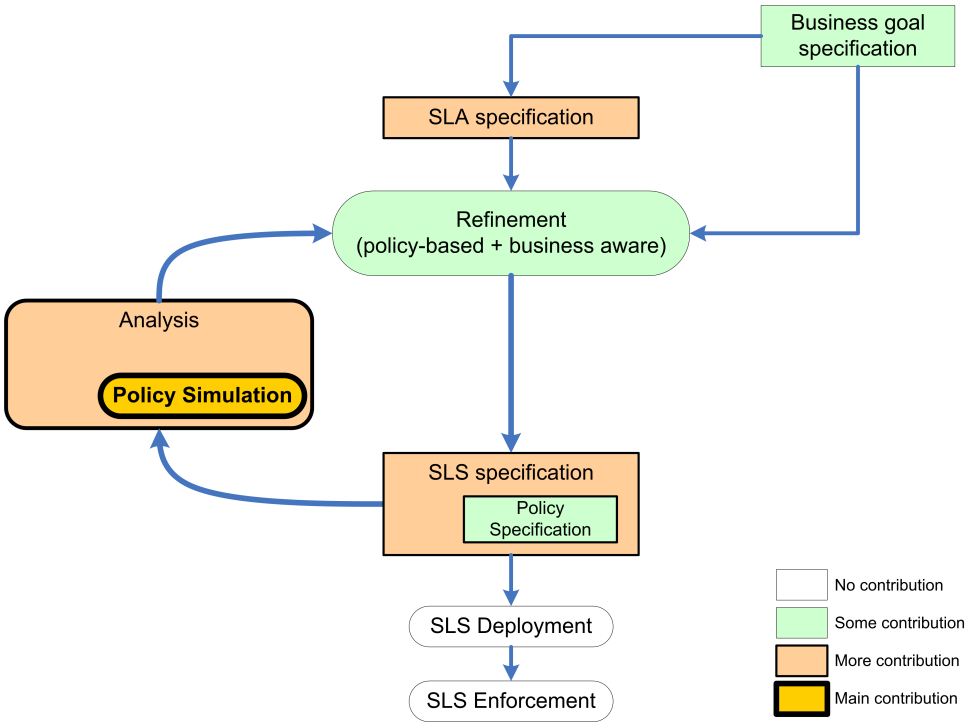


Figure 1.1 Thesis scope and contributions

schema GXLA, which allows a policy-driven specification of SLAs. The GSLA model is described in section 3.3.2 and the GXLA schema is given in appendix B.

Different policy languages are also compared and some enhancements are suggested (chapter 3). The specification we adopted is then used in a subsequent study on the policy-driven refinement of SLAs and optimization of policy-based solutions.

A major contribution of this thesis is to stress the importance of studying the *runtime dynamics of policy execution* before the actual implementation into a real-scale system. The issue is of importance as it is difficult to predict all the possible behaviors of a policy solution especially when the set of policies involved is of the order of hundreds or thousands. A primary aspect of this study concerns the determination of whether the generated policy set reflects the contracted SLAs and business goals of the service provider. Another important aspect of the study investigates the degree of efficiency and flexibility of the generated solution and how they can be enhanced (chapter 4).

In order to allow for the study of the runtime dynamics of policies, we developed $\mathcal{PS}$, a Policy Simulator tool that can be used to simulate the execution of policy solutions. $\mathcal{PS}$ is presented in chapter 4. It was developed as an alternative to expensive implementation and deployment of policy solutions that might turn out to be inefficient, or worse a disaster to the system in the case some irregular dynamic aspects where not detected beforehand. As a low cost testing facility,

$\mathcal{PS}$ provides a necessary preliminary step to testing and tuning policy-based solutions.

1.2 Example Application

In chapter 5 we present a use case of a service provider having a data center that it intends to use in hosting customer applications according to a generic SLA. The SLA has been made generic enough to span any type of application hosting and hence can be applicable with little adjustment to real life cases.

The use case shows in detail the steps required to go from SLA and business goal specifications down to implementable policy solutions. It spans the entire business-driven SLA management loop. Particularly, we show the details of how the refinement process is conducted to produce a policy-based Service Level Specification (SLS). The two analysis phases are then applied to that generated SLS. The *Static Analysis* phase checks the SLS policy set for consistency and stability. It helps detect and correct anomalies in the SLS that are not easy to detect at first glance. The subsequent *Dynamic Analysis* phase addresses the business-driven analysis of policy runtime dynamics. This phase shows the need for incorporating business (and SLA) related data, encoded mainly within metrics generated during the SLA refinement process, to handle the orchestration of policies at runtime. This analysis proves crucial in making the same set of generated policies (SLS) achieve better performance at runtime. The policy simulator $\mathcal{PS}$ is used for policy simulation and tuning.

1.3 Thesis organization

The thesis is organized into six chapters. First, chapter 2 sets the requirements for a management solution to obtain the business-driven label. It shows how business-driven management and policies are naturally related to each other and pays a closer look at the different steps required to achieve such a management model.

Chapter 3 looks at the different specification formalisms for the specification of policies, Service Level Agreements (SLAs), as well as generic business-driven management models. Section 3.3 analyzes efforts in the specification of SLAs. It then follows by presenting the SLA information model GSLA, which is expected to offer a better support for SLA modeling that fits well with a policy-based management approach as well as with the broader requirements of SLA specification in pervasive service-driven environments. The GXLA XML schema GXLA is given in appendix B. Section 3.4 presents the BDMF, which is a business aware framework for the policy-based management of information systems and features a high-level business and service-driven layer on top of a policy-based resource control layer.

Chapter 4 considers the state-of-art research in policy refinement and analysis and presents two contributions. Section 4.2 identifies a set of new policy inconsistencies and describes how they need to be solved in the general case. Section 4.3 identifies the need to consider runtime policy dynamics for the purpose of policy inconsistency detection, resolution, as well as for policy optimization in terms of maximizing the original high-level business goals. The Policy Simulator tool $\mathcal{PS}$, that we have developed for the purpose to serve for policy consistency analysis and performance evaluation, represents a prototype implementation of the BDMF and is presented in section 4.4.

Chapter 5 develops a use case for the business-driven SLA refinement into low-level management policies, in addition to an implementation which maximizes the business profit of the service provider. The case spans the entire business-driven SLA management loop. We show the details of how the refinement process is conducted to produce a policy-based SLS. The SLS is taken through the static and dynamic analysis phases and the result simulated onto the $\mathcal{PS}$ simulator. The use case shows how the different theoretical tools presented in the previous chapters come together to be used in a practical use case.

Finally, chapter 6 concludes by summarizing the thesis contributions, pointing the limitations within and the lessons learned, and finally identifying future research directions that are worth considering.

Chapter 2

Requirements for Business-Driven Management

The continuous evolution of the Information Technology market is driving world-wide business towards a universal business interactivity model. The advantages of this are numerous and equally are the driving motivations. Thus, the models and tools that form the building blocks of such an infrastructure are of significant importance.

This chapter looks at the requirements for a management solution to be business-driven. It demonstrates how business-driven management and policies are naturally related to each other, often differing only in name. A closer look will be given to the different steps required to achieve such a management model.

2.1 Business-Level Goals

Linguistically, a goal is a purpose toward which an endeavor is directed. Business goals can be understood initially by considering concrete examples, as is shown in figure 2.1. For an enterprize manager, a network operator, or the manager of any business, a business goal defines an abstract aim. Achieving this aim can be done in a variety of ways and often requires the coordination of different tasks and activities.

2.2 Goal or Policy?

Business goals have been given different names and/or adjectives in the literature. Moffet [7] defines a hierarchy of abstraction levels for policies in which an organizational or corporate goal is equal to a *Directional Policy*.

- Achieve at least 85% customer satisfaction for this business term [6]
- The management department is to protect the assets of company X [7]
- Effectively control cost/performance [8]
- Quality should be high [9]
- More importance should be given to service $\mathcal{A}$ during summer time.

Figure 2.1 Sample business-level goals / policies

1. **Morris Sloman**
   - Rule governing choices in behavior of a system [10]
   - The plans of an organization to meet its objectives [7]
   - A *persistent specification* of an objective to be achieved or a set of actions to be performed in the future or as an on-going regular activity [7]
2. **Seraphin Calo**
   - A set of considerations designed to guide decisions or courses of actions [8, 9]
3. **Raouf Boutaba**
   - A management policy represents an intermediate step between *management goals* and *management plans*. Management policies are general statements about how the information system will achieve the management goals, and are used to ease decision making by restricting the set of solutions to a problem from the range given by the management goals to a more easily handled size [11].
   - Management plans derived from management policies may be executed immediately to achieve some of the management goals (e.g., "turn on accounting"), or may be stored for execution in response to some situation as part of a persistent management goal (e.g., "if response time exceeds 50 ms, reroute") [11].
4. **IETF**
   - A definite goal, course or method of action to guide and determine present and future decisions [12]
   - Set of rules to administer, manage, and control access to network resources [13, 14]
5. **Mark Burgess**
   - Constraints on the behavior of objects and agents in a system [15]

Figure 2.2 Some common definitions of Policy

In the literature different definitions of what a policy is have been given. Figure 2.2 shows some of the most prevalent definitions. By considering these definitions of what a policy is, it can be inferred why different formalisms for policy specification and modeling have been proposed depending on where in the management plane the policy is expected to act. This is better expressed through the term *policy continuum* [16] which is defined next.

Figure 2.3 Policy Continuum

2.3 Policy Continuum

The term policy continuum [16, 17] refers to the fact that policy can be expressed at different abstraction levels of the management plane of a system. The highest level being the *Business View* which defines policy in business terms only. This is because business users do not need (nor want) to understand low-level system terminology, but still must be involved in the definition of policy. Policies of this type help translate SLAs as well as business procedures into a form that can be used to build device configurations and control their deployment. The *System View* translates the business policy into system terminology without regard to the particular devices that make up the infrastructure or the technologies that they are using. The *Administrator View* maps this into the specific technologies available in the system. The *Device View* defines the specific types of devices used to implement the information system, while the *Instance View* takes into account particular software releases, device configuration language, and other vendor- and device-specific characteristics.

Maullo and Calo [8, 9] defined six levels in the policy hierarchy. The same hierarchy is also recognized by Moffet and Sloman [7]. However, the actual number of layers is not important in itself, as one can imagine many levels in between. The importance lies in linking between the different levels and creating more automation in moving from one level to the other. The higher levels of policy (*principles* and *goals*) tend to be more abstract and more applicable to human than automated interpretation [7]; whereas *guidelines*and *executable policy* may be automated and more formally specified. The highest level of policy, the *Societal principles*, goes beyond the scope of a single business and is only included for completeness.

In the management of computing systems, only the two lowest levels are typically considered, since they are the most amenable to automation. These two levels are also given different names in different contexts [9].

Societal policy (principles) such as ethics or laws
e.g. Quality is a primary success criterion

Directional policy (goals) such as organizational or corporate goals
e.g. Quality should be high

Organizational policy (practices) such as contractual agreements or quality programs
e.g. Considerations for organizational elements and procedures that affect quality

Functional policy (targets) such as workload targets or quality measures
e.g. Considerations for functional areas that determine quality

Low-level goal (guidelines) such as automated quality tracking, which may be partly encoded in computer programs
e.g. General quality criteria against identified processes

Executable policy (rules) which are fully encoded in an executable computer language.
e.g. Specific actions that must be taken under given conditions in the manufacturing process to maintain quality

Figure 2.4 Policy Hierarchy

2.4 Requirements for business-driven management

It is now possible to state the requirements for a management solution to be business-driven. This implies the ability to model and specify policies at the different levels of abstraction of the policy continuum, as well as developing the mechanisms that help in the smooth transition between adjacent policy levels. The downward transition is referred to as *Policy Refinement* whereas the upward transition is referred to as *Policy Induction*.

2.4.1 Policy Specification

The effort in this area is to develop models, specification languages as well as tools that allow the specification of policies at different levels of abstraction. Chapter 3 presents an analysis of the work conducted in this regard.

As a general guideline, the specification of policies needs to abide by a set of criteria [5]. It should be *precise* regarding its interpretation, *consistent* in its deployment across system elements and *compatible* with their respective capabilities. In addition, and depending on the targeted level of abstraction, the specification formalism should offer a level of easiness and intuitiveness that simplifies policy expression by a manager, system administrator, or technician.

2.4.2 Service Level Agreements

In order to capture the business relationship between a customer and a service provider, there is a need for a formal way to express this contract so as to determine the benefits and responsibilities

as well as the objectives of each contractor. This contract is what is commonly referred to as a *Service Level Agreement* or SLA.

The rapid evolution of the information system market is creating an environment where the introduction of new services and new networking technologies is occurring in ever-shorter time scales. SLAs can help encourage customers to use such technologies and services as they provide a commitment from the service provider to guarantee specified performance levels. The increasing dependency on the availability of networks and communication and information services for an increasing number of critical business activities means that greater numbers of customers are looking for SLA guarantees to enable them to carry out their business.

In addition, many information system departments are being measured by the service levels they provide to other business units within their organization and must demonstrate their ability to deliver on these internal SLAs. Customers can therefore use the SLA commitments from the service provider when planning their own systems and future growth. SLAs are also advantageous for smaller organizations that do not have their own information technology department and networking specialists as SLAs can give them the performance guarantees they need [18].

The business relationships between the various roles in a management value chain illustrate the principal points of contact between the roles. These roles comprise customers, service providers/operators, other providers/operators, third party applications vendors, and suppliers to service providers. A variety of business relationships can exist between these roles in a competitive market with multiple providers. These relationships can be formalized at the interfaces between the roles, as these are the points at which information needs to be exchanged.

The overall relationship is important in this context as customers and service providers jointly negotiate an SLA covering the services being provided by the service provider to the customer and in so doing develop a greater understanding of the other's approach, requirements and responsibilities.

Traditionally, an SLA is defined as a contract between a service provider and a customer that defines all aspects of the service to be provided. The SLA covers availability, performance, customer service details, as well as the measures to be taken in case of deviation and failure to meet the asserted service guarantees, for example, a notification of the service customer.

When an SLA is specified in a form that can be directly implemented by the different parties into their respective information infrastructures, it is referred to as a *Service Level Specification* or SLS. Since an SLA is generally specified using high-level expressions that are often in natural language, the task of refinement of an SLA into an SLS is similar to that of refining a high-level policy into a low-level policy. In the use case chapter (5) an abstract SLA is refined into a concrete SLS using a policy-driven refinement process.

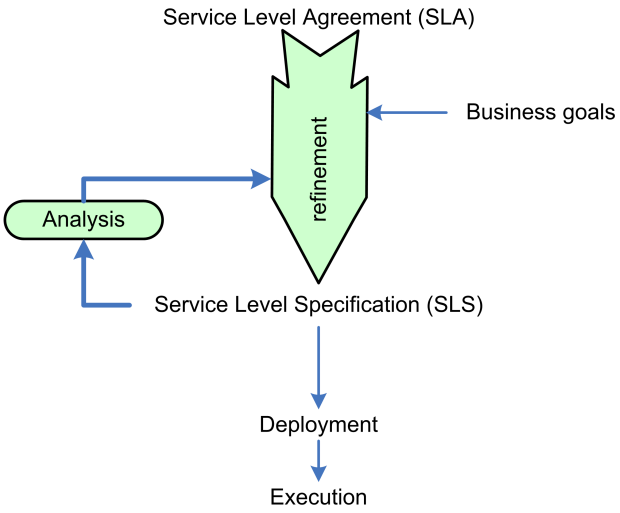


Figure 2.5 SLA to SLS refinement process

Section 3.3 presents the different models and languages that have been proposed in the literature in order to specify SLAs and SLSs. In section 3.3.2, we present an information model that we developed for modeling SLAs along with an XML schema for it in appendix B.

2.4.3 Policy Refinement

For an SLA or high-level goal specification to be deployed and executed, there is a need for it to be refined into a form that is directly enforceable by the underlying resource infrastructure. As a result the final output of the refinement process is hence system dependent while intermediate phases can be system independent. The refinement of an SLA produces an SLS, while the refinement of a high-level policy produces a low-level *policy rule*.

Policy refinement follows a number of phases which take high-level goals down through the policy continuum until reaching the policy rule level. Although recent advances have been made to solve this open problem [19], the holistic implications confronting systematic policy refinement have not been explicitly addressed [20]. It is currently believed that there cannot be a universal refinement method that fits with all refinement scenarios. This however does not hinder the investigation of generic and application-specific refinement patterns which can contribute to a growing knowledge base about the refinement process.

2.4.4 Policy Analysis

As is the case for any system implementation, the set of policies generated from the refinement process needs to be checked for actually implementing the original high-level requirements. This

process is referred to as *policy analysis* or *ratification*. It is tightly linked with the refinement process and can be performed at any of its phases. By employing various policy analysis technologies [21], one can validate the correctness and consistency of policies automatically, thus increasing the reliability and predictability of the system's behavior.

The analysis of the final set of policy rules needs to be checked against not only the original high-level policy, but also the existing system policy rules. Policies can in fact interact with each other, often with undesirable effects, and a system administrator needs to be aware of such interactions among policies [21]. The problem of policy interaction is particularly acute in a distributed system where it is likely that a policy author would have only a partial view of the entire system and where multiple authors may write policies applicable to the same set of resources [22].

2.5 Conclusion

This chapter showed the natural linkage between business-driven and policy-based management as well as at the different formalisms, tools, and techniques that are needed for achieving business-driven management. Based on that, the next chapter looks at the first step in business-driven management by studying the different models, languages, and frameworks for the specification and management of policies and service level agreements.

Chapter 3

Specification formalisms for BDM

This chapter looks at the different specification formalisms that have been proposed for policies, Service Level Agreements (SLAs), as well as generic policy, SLA, and business driven management models.

The first part presents the major efforts that have been undertaken in the specification of policies, the different notations developed, and how policies are deployed within a managed system. A classification of these policy notations will be given based on several criteria and some enhancements will be proposed. The focus is on quality assurance policies rather than security enforcement policies.

The second part will briefly analyze efforts in the specification of service level agreements. It then follows by presenting the SLA information model GSLA[1] which is expected to offer a better support for SLA modeling that fits well with a policy-based management approach as well as with the broader requirements of SLA specification in pervasive service-driven environments. The GXLA XML schema GXLA is given in appendix B.

Finally, section 3.4 presents a framework for the business and policy-based management of information Systems[2]. The framework features a high-level business and service-driven layer on top of a policy-based resource control layer. The linkage between the two layers is assured in two ways. First, a policy-based refinement engine, supported by an appropriate SLA information model, ensures the off-line derivation of low-level policy rules from high-level requirements that are expressed in terms of SLAs and business objectives. Second, a business profit maximization engine provides decision support that seeks to maximize the business goals at system runtime whenever it is deemed appropriate.

[1] Published in [23]
[2] Published in [24]

3.1 Policy specification

Policy discipline is gaining increasing interest in industry and academia leading to a continuous evolution towards a service-driven model of information systems management. Policy is being studied for usage in service quality assurance as well to enforce security constraints.

For the policy approach to systems management to become effective, policy needs to be specified in a *language* that is *easily understandable* by human administrators and policy-makers but also *easily enforceable* on the system resources. The language must be applicable to a *uniform* representation of system elements as well as their properties, operations, and relationships expressed in a *device-independent information model*. To be executable, the policy specification must be *free of conflicts* and must *match the capabilities* of the resource it is intended to be enforced into.

As a means of management, a policy definition must conform to a number of requirements [5]. It should be *precise* regarding its interpretation, *consistent* in its deployment across system elements and *compatible* with their respective capabilities. The specification should be at a level of easiness and intuitiveness that simplifies policy expression using an abstract day-to-day activities language. This more user-friendly language helps in generating policies optimized for specific devices.

The following sub-sections provide an account and critics of related work in policy specification. Different approaches are considered and classified according to their level of *abstraction* and *expressiveness*. The question of whether there is a best policy language and that of a best policy architecture will also be considered.

First, we will briefly review the main paradigms that have been introduced in the last fifteen years in the representation and management of network policies. The bosom of this discipline could be found in the early works on routing (such as BGP [25], IDPR [26], RPSL [27], IDRP [28]) and network management protocols, which were followed by more elaborate frameworks.

3.1.1 IETF/DMTF Policy Framework

Instead of defining a policy notation using a specification language and its underlying grammar, the IETF policy working group [29] used a UML-like object oriented modeling for policy and Network elements. This is a good choice from the point of view of low-level policy distribution among the management system components and their work serves also in the definition of an overall policy framework in which policy notation is only one single part of a whole system. However, there is still a need for an easy to use notation for policy at the high-level entry in the system management tool and all other related operations of compilation, validation, and refinement of high-level policies, as well as their exchange among different components.

The IETF defined a set of information models that capture all aspects of a managed environment. At the top of the hierarchy is the Core Information Model (CIM) [30], which is composed of a small set of classes and associations that establish a conceptual framework for the schema of the rest of the managed environment. Second, comes the Policy Core Information model (PCIM [14], extended recently to PCIMe [13]) which defines a set of classes and relationships that provide an extensible means for defining policy control of managed objects.

PCIM enables administrators to represent policy in a vendor-independent and device independent way. Thus, service-level abstractions can be supported at a higher level, and be translated at a lower level to device-specific configuration parameters, across an aggregate of heterogeneous managed entities.

In addition to the ability to structure policy rules into groups and groups of groups, PCIM introduces the notion of *policy Role* and *component Role*. The concept of role is central to the design of the entire IETF policy framework. A role is a type of property that is used to select one or more policies for a set of entities and/or components from among a much larger set of available policies [14]. Components are assigned specific roles and this helps the management system or the system administrator in selecting the appropriate policies that apply to these roles. If ever it is desired to assign a different functionality to that same element, a simple role exchange is done and the appropriate set of policies is applied accordingly. In addition, the use of roles enables a policy definition to be targeted to the required function of a system element, rather than to the element's type and capabilities [31].

The QoS Policy Information Model QPIM, introduced by the IETF in [32], builds on PCIM (and PCIMe) a set of constructs for specifying and representing policies that administer, manage, and control access to network resources. It deals specifically with QoS enforcement for differentiated and integrated services using policy. High-level business needs, available network topology, and the desired network QoS methodology such as DiffServ [33] or IntServ [34] drive the process of QoS policy definition in QPIM. Listing 1 illustrates the abstraction level of policies defined using QPIM. In 2003, QPIM has been enhanced to an IETF standard and renamed to Policy QoS Information Model [31].

It can be noticed that PCIM(e) and QPIM have a high degree of articulation in policy definition [35]. In fact, nearly every single component of a policy is made up to be a full class. If the policy is say of the form $(C1 \wedge C2 \wedge C3) \vee (C4 \wedge C5) \Rightarrow A1, A2, A3, A4$ then this needs to be modeled in at least ten classes! Five for the conditions, four for actions, and one for the policy itself let alone objects resulting from the instantiation of the different relationship classes. This puts the risk of having an implementation that is too slow and overly verbose. There is no real need of this fragmentation in order to represent usual "conditions⇒actions" situations. Such a fragmentation in policy representation makes it difficult for a policy administrator to modify

Listing 1 Level of abstraction of a QPIM policy

High-level policy:
In the human resources department, applications should have better QoS than simple web applications unless it is an executive's web application

QPIM policy (a possible formulation):

```
if packet IP@(source ∨ destination) ∈ human resources department then
    if packet source/destination IP@ ∈ execSet then
        packet.priority ← High
    else if packet.protocol == HTTP then
        packet.priority ← Low
    else
        packet.priority ← Default
    end if
end if
```

or maintain a policy repository and makes it much harder for a policy manager to manipulate policies defined by another policy manager or even to update polices which he defined in the past. Though this is useful at the QPIM level due to the extensive reusability nature of network-level policies, this is not desirable at the human administrator specification level. A language based policy notation would help in improving policy readability and hence supports team based policy specification for the management of large systems.

3.1.2 Traffic flow policy languages

These are relatively low-level policy languages that make use of pattern matching in the selection of the network traffic on which a policy can be applied. Their efficiency in policy-based management is limited since they do not provide a view of how the overall policy managed network system will work. A policy is simply an abstract device level representation of the functionality that a node is to hold. Falls in this category PAX-PDL [36], SRL [37], and PPL [38]. PAX-PDL and SRL codes look much like ANSI C code. The basic concept is the *pattern*, with simple patterns combined to form more complex patterns. SRL adds on pattern recognition the ability to act on the identified packet/flow by saving some statistics about it. The Path-based Policy Language PPL [38] was intended to bring a satisfactory solution to the integration of the IntServ and DiffServ IETF models by having a policy extended to act on the full data path of a flow. This avoids many NP-complete calculations that try to establish a coherent path based on policies spread over different nodes. Paths are analogous to static routes when they are completely instantiated. However, pre-computing paths may also incur excessive overhead if not designed

appropriately. Yet, it is not always practical to a priori freeze paths by which a known traffic should flow.

Listing 2 Priority management in PPL

Assign a very low priority for all data traffic during working hours except if this is the managers own data flow

Policy6 <net_manager> {∗}
{traffic_class = {data}} {∗}
{time≥0800, time≤1600 : priority := 10}

The use of userID in the grammar specification is not a needed feature and unnecessarily complicates the grammar. As also noticed by Nicodemos [1], it would better be specified as part of some meta-information that could be used for the analysis of policies. Current traffic flow policy languages lack mechanisms for grouping policies and do not tend to take advantage of Object Oriented concepts. The search for a good policy specification language should better be done within the network wide policy specification languages, following.

Higher-Level policy languages have also been proposed, starting with the elementary Clark Policy Terms [38]. A policy definition is generally viewed as an "**IF** *Condition* **Then** *Action*" statement or further refined to an "**On** *Event* **Then IF** *Condition* **Then** *Action*".

3.1.3 Clark's Policy Term (1989)

The Clark's policy term [38] constitutes one of the earliest works in network policy specification. Clark proposed a template to represent network policies called *policy term* (figure 3). Scalability is achieved through the use of *Administrative Regions* (ARs), which may include networks, links, routers, and gateways. A term is made up of four fields: source, destination, user class, and other global conditions. The source field is made up of the source host IP address, the source AR, and the entry AR. The destination field is made up of the destination IP address, the destination AR, and the exit AR. Wild cards along with user classes allow for better flexibility and scalability.

Listing 3 A Clark Policy Term

```
((132.227.60.13,1,-),(194.2.232.170,3,-), University, Unauthenticated UCI)
```

Traffic flow marked of class "University" can pass without authentication between the host with IP address 132.227.60.13 in AR1 and the host with IP address 194.2.232.170 in AR3.

The notation used by Clark is a good starting point for abstract network policy representation.

However, it lacks the ability to represent explicit paths formed by a sequence of ARs as part of the terms [38], which does not make it a good candidate for IntServ technology support. Also, the language syntax is not natural for the policy manager since it requires him to memorize the semantics of each parameter position in a term, while it would have been easier to introduce names of fields into it.

3.1.4 Policy Description Language PDL (1999)

PDL [39] is a declarative policy definition language for information systems management. It defines a policy as a *collection of general principles specifying the desired behavior of a system.* Policies are formulated using the Event-Condition-Action (ECA) rule paradigm of active databases and supported by a rich event sub-language allowing only uninterpreted concurrent actions.

A PDL policy is a function that maps a series of events into a set of actions. The semantics in PDL is founded on recent results in formal descriptions of action theories based on automata and their application to active databases [40, 41].

PDL envisions policy specification in two steps. First the policy manager collects from the system events, actions, and functions (status information) that are supported. Then he writes the policy and send it to the policy server where it is implemented. This is depicted in figure 3.1. Nothing is said about the structure of the policy server or that of the entities manipulated by the policies. In addition, it is only said that the policy server will take the policy defined by the manager and implement it in the system without specifying how this would be done at the conceptual level.

PDL distinguishes between two types of policies, *policy rule propositions* and *policy defined event propositions* which follow intuitive semantics.

A noteworthy feature of PDL concerns its rich event representation. It represents primitive (system or policy defined) events, which can be composed into complex events using boolean and temporal operators such as sequencing and alternation. An event happens in an *epoch* and a series

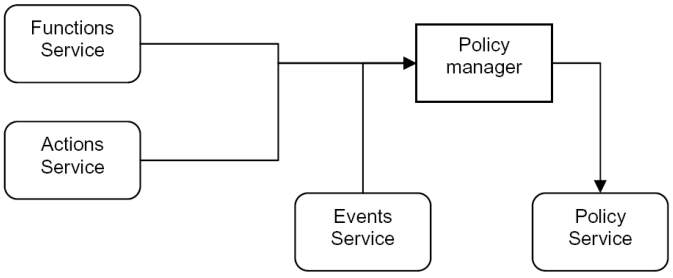


Figure 3.1 PDL Policy Framework

Listing 4 PDL policy types

Policy rule proposition: Event **causes** action **if** condition

Fax management example: Deliver received faxes to home when user is not at work.

```
FaxArrive CAUSES DeliverFaxHome IF FaxArrive.Time>EndWorkTime
```

Policy defined event proposition:
Event **triggers** PolicyDefinedEvent(m1=t1,...,mk=tk) **if** condition

Soft switch overload control example: Threshold Overload status is considered when an excessive number of signalling network time outs over calls made is reached.

```
normal_mode, group((callmade|time_out))
Triggers overload_mode
IF Count(time_out) / Count(call_made)>
```

of epochs constitute an *event history*. PDL has been implemented and used in Lucent switching products. The use of events for policy triggering is desirable as a feature in a policy specification language. The rationale behind this concerns information that can be carried out by current status information of a system. Events are used to transport useful local status information to portions of the system that need it. Moreover, in order to trigger a policy action there is sometimes a need to know not only about the current status of the system but also about specific evolution patterns that might have occurred before and which may provide valuable information to correct policy decisions. Events allow to mark those history states/patterns that are needed and hence provide a better understanding of "state" information.

3.1.5 The Ponder Framework (1992-2006)

Ponder is a declarative, object-oriented language for specifying security policies with role-based access control as well as general-purpose management policies. Security policies are related to managing which object has the right to do what in the system. Management policies specify actions to carry out when specific events and conditions occur within the system. Unlike many other policy specification notations, Ponder supports *typed policy* specifications. Policies can be written as parameterized types, which can be instantiated multiple times with different parameters in order to create new policies. New policy types can be derived from existing policy types, supporting policy extension through inheritance [42].

Ponder views policy as a *rule that defines a choice in the behavior of a system* [42]. This behavior is intended to reflect objectives of the system managers. At a more formal level, a Ponder policy defines a relationship between objects (*Subjects* and *Targets*) of a managed system. It supports four *basic policy* types: *authorizations*, *obligations*, *refrains*, and *delegations* and three *composite policy* types: *roles*, *relationships*, and *management structures* that are used to compose policies

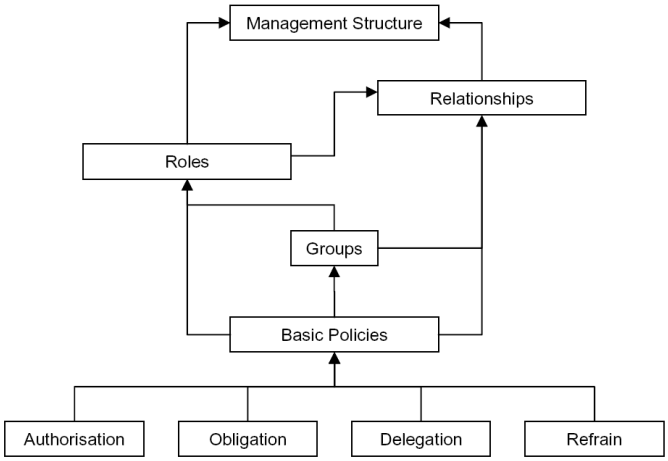


Figure 3.2 Hierarchy of policy types in Ponder

(figure 3.2). It also has a number of supporting abstractions that are used to define policies: *domains* for hierarchically grouping managed objects, *events* for triggering obligation policies, and *constraints* for controlling the enforcement of policies at runtime. All constructs can be specified as single instances or as types that can be instantiated as many times as needed. Types in ponder can be parameterized which adds to the generality of the language.

Listing 5 Application-level video conference authorization [1]

Members of Agroup plus Bgroup can set up a video conference with USA staff except the New York group. The constraint of the policy is composite. The time-based constraint limits the policy to apply between 4:00pm and 6:00pm and the action constraint specifies that the policy is valid only if the priority parameter (the 2nd parameter of the action) is greater than 2.

```
inst auth+ VideoConf1 {
  subject   /Agroup + /Bgroup;
  target    USAStaff - NYgroup;
  action    VideoConf(BW, Priority);
  when      Time.between("1600", "1800") and (Priority > 2);
}
```

Domains provide a means for grouping objects to which policies apply, or partition objects in a large system according to some criteria. An element can belong to more than one domain at a time. The main advantage of specifying policy scope in terms of domains is that domains may freely evolve in time with new objects added and others removed. In addition, *domain scope expressions* offer better flexibility to the scope of a policy action. Listing 5 shows how they are used to define subjects and targets of a policy. A comparison of Ponder domains with IETF roles is given in section 3.2.3.

Obligation policies represent the key to quality assurance policy specification in Ponder. An obligation specifies the actions that must be performed by manager objects within the system

Listing 6 Scripted actions in obligation policies [1]

```
inst auth+ domainManagement1 {
  subject  /domainAdmin;
  action   execute;
  target   /scripts/domainMove(A,B,x);
  when     A="/users/gold" and B="/users/silver";
}

inst oblig domainMove {
  on offensiveRequest(user);
  subject  /domainAdmin;
  do       /scripts/domainMove("/users/gold","/users/silver",user);
}
```

The authorization policy permits domain administrators to execute the script object 'domainMove' when A and B are "/users/gold" and "/users/silver". This script moves an object x from domain A to domain B. Because of security considerations only domain administrators are permitted to execute it, and only to downgrade gold service users to silver service users. The obligation policy specifies that domain administrators must move a user from the gold service domain ("/users/gold") to the silver service domain ("/users/silver") when that user has requested access to a web-page which is considered offensive.

when specific events occur and a set of conditions are met. That is, obligations are event triggered and they follow the ECA paradigm. Ponder supports a rich event composition mechanism which is much similar to that found in PDL. Ponder obligations support exception management in a way similar to that provided by object oriented languages.

Actions in ponder are generally object method invocations, but can also be externally defined scripts. Scripts in Ponder can be implemented as objects and stored in domains where they can be accessed through authorizations.

Scripts in Ponder can be useful in certain cases. However, they can introduce security vulnerabilities and cut-off the uniform design of the language. In the same spirit of keeping language design uniform domains are modeled as objects in the system. A further step towards uniformity is to provide them with methods to manage objects they contain. In this way a domain could filter, using some criteria, objects it can accept. For example, a domain that is intended to reference only teachers would refuse to include a student into its referenced objects set. This feature could enhance the uniformity of the language. The example given in listing 6 could be enhanced using the following new notation:

To cope with scalability, Ponder offers tools for grouping policies. Four types of composite policies are provided: groups, roles, relationships, and management structures. Roles are a major concept in Ponder. They provide a semantic grouping of policies having a common subject such as a DiffServ edge router. Within the scope of a composite policy structure or a domain sub

Listing 7 Enhancing Domain Representation

```
inst auth+ domainManagement1 {
  subject  /domainAdmin;
  action   MoveTo();
  target   <Domain> A;
  when     (A.path ="/users/gold") and (B.path="/users/silver");}

inst oblig domainMove {
  on offensiveRequest(user);
  subject  /domainAdmin;
  do       "/users/gold".MoveTo("/users/silver",user);
  when     user.path = "/users/gold";}
```

Methods concerning domain management can be defined in the Domain class. This example specifies the same functionality as that of listing 6 but with enhanced uniformity and security.

tree, there is a need for rules regarding permitted policy types and authorized concurrent action sequencing. For this, *Meta policies* have been introduced. They are expressed using the Object Constraint Language (OCL). This provides a useful way to avoid policy anomalies during run time. The ponder framework is self managed in that policies and other constructs such as roles and relationships are implemented as objects stored within domains. Hence, they can be managed by policies stored in those domains.

3.1.6 Policy Management for Autonomic Computing (PMAC)

Policy Management for Autonomic Computing (PMAC) [43] is a generic middleware platform developed by IBM to provide software components that can be embedded in software applications to reduce the cost of writing applications capable of taking input from a policy-based management system.

The PMAC platform supports the system model adopted by the IBM Autonomic Computing (AC) architecture, which defines a framework for self-managing information systems [44]. The AC architecture given in fugure 3.3 presents two key abstractions:

- An autonomic manager (AM), which monitors computing resources, analyzes the status of the resources, plans action for the resources, and executes the planned actions
- A managed resource (MR), which is a computing system controlled and managed by the AM1

In PMAC, the AM is a policy-based manager, which monitors, analyzes, and plans according to the policies that have been defined for the resources managed by the AM. In this respect, the role of the AM is similar to that of the policy decision point (PDP) as defined in RFC 3198 [12]. An AM

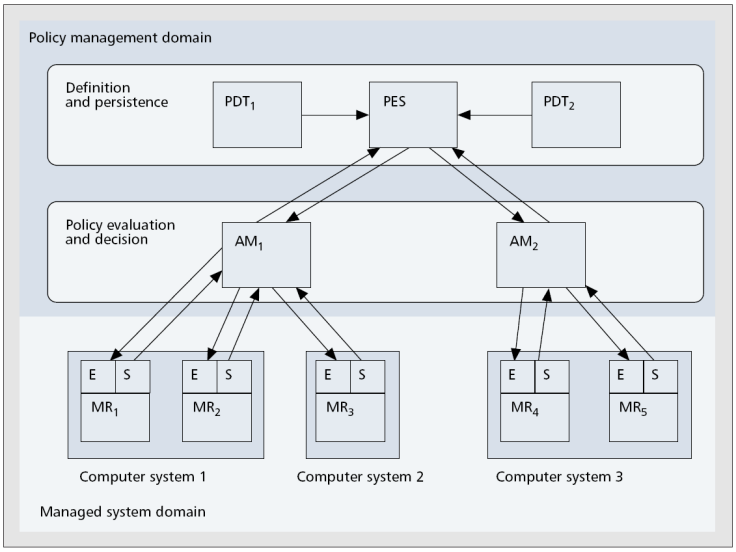


Figure 3.3 PMAC architecture

includes the functionality of a PDP and supports additional features, such as state monitoring, event correlation, and notification, that many traditional PDPs do not provide. Likewise, there is a similarity between the managed resources and the traditional policy enforcement point (PEP) component.

PMAC is implemented in JAVA and offers a balanced specification of policies by providing two different policy languages: ACPL which is based on XML, hence verbose, and SPL which is concise and human friendly, making it easily editable in a text editor. In PMAC policies are written and stored in the Autonomic Computing Policy Language (ACPL). ACPL is an XML-based policy language whose syntax closely mirrors the policy information model of PMAC. PMAC features also a Policy Definition Tool (PDT).

The AM in PMAC exposes a set of Java application programming interfaces (APIs) that are useful when the AM and MR are running in the same Java virtual machine (i.e., the policy module is embedded as a library in applications). Alternatively, the AM can be run inside an application server and be offered as a stateless session Enterprise Java Bean (EJB) or as a Web service to remotely located managed resources. Thus, a managed resource can request policy guidance to the AM through RMI (in the EJB case) or SOAP (in the Web service case) protocols.

Autonomic Computing Expression Language (ACEL, 2005)

At the core of ACPL is a rich expression language, the Autonomic Computing Expression Language (ACEL), that facilitates writing policy rules [45]. ACEL has been designed so that it can express most common policy conditions while closely following standard XML conventions. It is

Listing 8 Boolean expression in ACPL Vs. SPL

Boolean expression in ACPL

```
<And>
 <Not>
  <Equal>
   <PropertySensor propertyName= "NumberOfPorts"/>
    <IntConstant>
     <Value>16</Value>
    </IntConstant>
  </Equal>
 </Not>
 <Equal>
   <PropertySensor propertyName= "VendorId"/>
    <IntConstant>
     <Value>5</Value>
    </IntConstant>
 </Equal>
 <Equal>
  <PropertySensor propertyName= "Type"/>
   <StringConstant>
    <Value>Core Switch</Value>
   </StringConstant>
 </Equal>
</And>
```

The same expression in SPL

```
 (Sensor(NumberOfPorts) != 16) and (Sensor(VendorId) = 5) and
 (Sensor(Type) = \Core Switch")
```

a strongly typed language that can be parsed and type checked almost entirely by XML parsers, thereby making it attractive to applications that can consume XML format (e.g. Web services policies).

A condition in a PMAC policy can be any ACEL expression of type Boolean with variables corresponding to sensor names. The value of the variable is the same for all occurrences of the variable in the AM policies. Variables in ACEL are represented by an XML element type called PropertySensor with an attribute Property- Name identifying the name of the variable.

It is interesting to note that the XML representation of ACPL is primarily for internal processing, policy persistence, and deployment. In such cases, it is expected that policies will be created and updated using a PDT. However, for cases when a PDT is not used, PMAC also supports a simple policy language called SPL. SPL is more human friendly, and policies in SPL can easily be written using a text editor. Listing 9 from [45] explains the difference between ACPL and SPL:

In PMAC, all policies written in SPL are internally translated into ACPL, and both versions, the SPL and translated ACPL, are stored in the PES. Parsing and evaluation is then done in the same manner as for policies originally written in ACPL. The simplicity and convenience of SPL, however, comes at a cost. Theoretically, it is possible to define and implement SPL so that it provides functionality equivalent to that of ACPL. However, such an implementation will be

a potentially problematical undertaking since it cannot rely on standardized tools and libraries similar to those available for XML. Therefore, the PMAC implementation of SPL provides a subset of ACPL functionality.

3.1.7 Cfengine (1993)

Cfengine [46–48] is one of the earliest policy-based configuration management solutions. It is fully specialized in the configuration of Unix-like and Windows networked computers. In Cfengine, a policy specifies what a *healthy* system state is using a declarative language. Cfengine then takes care of keeping the system always near to the healthy state by taking appropriate actions each time the system drifts to a *sick* state. Policies in Cfengine are similar to service-level objectives or high-level goals in the sense that they only specify the desired objective but not how to achieve it. However, Cfengine is not common purpose and does not provide a generic policy-based platform as is the case, for example for PMAC.

Cfengine grew out of the need to replace complex shell scripts used for the automation of administration tasks on Unix systems and allows the creation of single, central configuration files which describe how every host on the network should be configured. It uses the idea of classes to group hosts and dissect a distributed environment into overlapping sets. Host-classes are essentially labels which document the attributes of different systems. The following classes are meaningful in the context of a specific host: (i) the identity of the machine, including hostname, address, network, (ii) the operating system and architecture of the host (iii) an abstract user-defined group to which the host belongs (iv) the result of any proposition about the system, including the time or date. Policies are specified for classes of hosts and define a sequence of actions regarding the configuration of a host.

The main components of cfengine are:

- A central repository of policy files, which is accessible to every host in a domain.
- An active agent which executes intermittently on each host in a domain.
- A secure server which can assist with peer-level file sharing, and remote invocation, if desired.
- A passive information-gathering agent which runs on each host, assisting in the classification of host state over persistent times.
- Various supporting tools.

The following example demonstrates the use of the language for configuration management [49]:

What makes Cfengine different from similar approaches to configuration management is that it embraces a stochastic model of system evolution. Rather than assuming that transitions between

Listing 9 Cfengine script

```
files:
 (linux|solaris).Hr12.OnTheHour.!exception_host::
      /etc/passwd mode=0644 action=fixall inform=true
```

The first line defines the name files for the action. The second line identifies the class of hosts for which the action is to be executed, followed by the actual command. The command-line specifies that the Cfengine agent, which is always the subject of the policy, must search for all password files with an invalid mode, fix them, and inform the administrator. The class membership expression specifies all hosts which are of type linux or solaris, during the time interval from 12:00am to 12:59am, apart from a host labelled with the class exception_host. The second line identifies the target of the policy, i.e. all the hosts falling within the classification, the condition for execution of the policy, which is a time interval, and a trigger which specifies that the action must be executed on the hour.

states of its model occur only at the instigation of an operator, or at the behest of a protocol, Cfengine imagines that changes of state occur unpredictably at any time [50].

Although specific to configuration management of mainly Unix-like systems, the interesting feature of cfengine in terms of policy specification that is not found in other languages is that it actually defines policies as goals to achieve [**?**] and elaborates strategies for that at the runtime.

3.2 Analysis of policy specifications

Based on the previous discussion, we identify in the following a set of criteria we think are important for classifying the different policy specification formalisms thus far described:

3.2.1 Supported Abstraction level

High-level policy specifications are application or business driven. They tend to be declarative. On the contrary, network level policy specification is more procedural and is closer to the logic supported in device actions even if it is specified in a device independent way. Policy refinement cannot be automated unless the high-level abstract policy specification is done using formalisms with precise semantics. Logic-based approaches to policy specification allow formal reasoning about the specified policies, and enable properties of the specification to be proved, but they are not aimed at human interpretation and do not directly map to an implementation [1]. There have also been efforts to describe policy using natural language like expressions as is the case with the HP Power prototype [51] and the work done in [52] where is described a suite of tools that serves as an expert database management system to automate the process of mapping natural language policy statements into equivalent first-order predicate calculus.

3.2.2 Formalism used to represent policies

The formalism used for policy specification has a direct impact on the expressiveness, scalability, and usage flexibility of the policy language. Flat formalisms for example are easy to implement but are not scalable. Other tradeoffs make it difficult to support the flexibility of Object Orientation along with formal verification. Languages

3.2.3 A note on Roles and Domains

In Ponder, a Role is a semantic structure that gathers policies with a common subject, generally pertaining to a position within an organization such as a department manager. It can also specify policies that apply to an automated component acting as a subject in the system or to a network device such as a router [1]. More formally, a role is a set of authorization, obligation, refrain, and delegation policies having the same subject domain. The idea behind a role is to attach to one semantic structure all policies related to a known position in the managed system. Hence, it is possible to assign a different human or automated agent to a role without having to change the set of activities related to it since the activities are related to the role rather than to the agent assigned to it at a given time interval.

Listing 10 A Ponder Role for Paris DiffServ Core Routers

```
inst role Roles/DiffServ/Core/Paris {
 inst oblig Classifier {...}
 inst oblig Meter {...}
 ...}
```

For the IETF, a Role is a simple property. PCIM introduces the notion of *policy Role* and *component Role*. A role is a type of property that is used to select one or more policies for a set of entities and/or components from among a much larger set of available policies [14]. QPIM uses the concept of roles defined in PICM(e) to help the administrator map a given set of devices and interfaces to a given set of policy constructs. The use of roles enables a policy definition to be targeted to the network function of a network element rather than to the element's type and capabilities. This helps support model scalability where a QPIM policy can be mapped to large-scale policy domains by assigning a single role to a group of network elements [31]. When collective (e.g. network) behavior must be changed, the policy administrator can perform a single update to a policy for a role, and the elements noted above will ensure that the necessary configuration updates are performed on all the resources playing that role [14].

The domain concept in IETF represents a *set of network components playing the same role*, where a role is simply a *string attribute* within an object. Role combination is specified using a "+" sign included in the Role property as is the case in figure 3.4. This is very primitive compared

Figure 3.4 Policy dissemination in the IETF policy framework

to domain scope expressions offered in Ponder. In addition, the domain (role) concept in IETF is scattered throughout the different role strings that exist in the managed system elements and it would be difficult if not impossible to do domain maintenance operations on them.

In Ponder, Domains provide a means of hierarchically grouping objects to which policies apply and can be used to partition objects in a large system according to geographical boundaries, object type, responsibility and authority or for the convenience of human managers. A domain merely holds references to object interfaces. It can hold references to any object type, including a person. It can also reference sub-domains. Policies normally propagate to members of sub domains and domain scope expressions can be used to combine domains to form a set of objects for applying a policy.

The advantage in specifying policy scope in terms of domains is that domains may freely evolve in time with some objects being added and others removed. However, a design choice in Ponder was that objects have to be explicitly included in domains since it is not viewed as practical, for performance concerns, to define domain membership in terms of a predicate based on object attributes. On the other hand, a policy can select a subset of members of a domain and its sub domains, to which it applies, by means of a constraint in terms of object attributes.

Though the domain model defined in Ponder is a better choice for our analysis, there are two features that we think are desirable to be added. These are *domain membership constraints* and *domain compatibility rules* [35]. Building on listing 10, the examples in listing 11 illustrate these two concepts.

Domain compatibility rules can be viewed as a special type of a domain membership constraint. The above mentioned examples identify the need for providing domain objects with a membership constraint (predicate) that each of its members must verify before being accepted within. This predicate serves, in a sense, as an authorization agent which controls membership to a domain. Membership constraints serve in *conflict avoidance* since they forbid potential misbehavior of system elements due to their being included into a wrong domain.

Conversely, *role compatibility rules* can also be defined between roles (IETF semantics). There is a real need to determine whether a device can support a given role or not as it would be equally disastrous if a device is assigned a wrong role or if it were assigned a role it cannot fulfill. Testing a device for a role to check that it is really what it purports to be is important to the correct operation of the system. In this way, it is possible to have a component which rejects

Listing 11 Sample domain membership compatibility rules

1. A DiffServ Role cannot be assigned to a non DiffServ Enabled router. ⇒ a domain of DiffServ Routers should define the domain membership constraints through the use of which only DiffServ enabled routers can be included. In practice, this constraint can be in the form of a script that checks the router configuration and factory capabilities in order to make sure it is DiffServ enabled.
2. Router R can behave as a DiffServ Edge Router, DiffServ Core Router, or an IntServ Router, but cannot behave in two modes simultaneously. By stating that DSE=Routers/DiffServ/Edge is incompatible with DSC=Routers/DiffServ/Core it can ensure that R cannot belong simultaneously to DSE and DSC. If such rules are not defined then a situation could occur where R receives the deployment of both a Core and Edge DiffServ router, thus leading to unpredictable behavior.
3. Service S_1 is deployed on domain D_1. A new service S_2 is getting deployed on domain D_2. Knowing that S_2 is incompatible with S_1 (either through a property in S_2 or through system manager knowledge), D_2 is provided with a domain compatibility rule that forbids the inclusion of objects already belonging to D_1. The system can thus raise an exception when an attempt is made to violate this rule.

from being included into a domain because it finds it is not compatible with the roles that this domain supports.

It is also possible to think about compatibility rules at the policy rule level. However, this provides the ability to do the same thing in many different ways, which is not good for a uniform specification language. This issue needs to be further investigated as compatibility rules can provide an additional dimension to intelligent policy conflict management.

3.2.4 Policy Release mechanism

Some languages base their policy triggering on special system states specified by the condition part of an ***if***-*condition*-***then***-*action* policy. These languages are referred to as *proactive* languages in [53]. In this case, [54] states that potential problems are detected and handled before they actually happen. This implicitly implies that in *reactive* languages, i.e. event based, policies are triggered only when anomalies occur. However, this is not always the case. Events are used for more than mere anomaly reporting. Events are needed in order to summarize particular epochs in the development of the managed system which the system manager judges of significance. These epochs are characterized by a set of conditions that help identify them. For example, the manager may specify a policy that launches backup activities at specific time epochs:

Listing 12 Daily backup policy in PDL

```
event BackupTime = WorkHours.End
event BackupTime triggers SystemBackup
```

This same policy can be extended to take into account security considerations in the presence of potential system threats:

Listing 13 Augmented Daily backup policy in PDL

```
event BackupTime = WorkHours.End | SystemFailureAlarm
event BackupTime triggers SystemBackup
```

When events are supported in a management system, more complex events can be defined based on system event history. This feature is supported by PDL and is interesting since it enables the characterization of specific evolution patterns of the managed system. For example, in order to increase the security of a managed system it is useful to register previous *evolution patterns* that lead to system failure or some specific anomaly detection. This can be used in preventing the future occurrence of previously encountered malicious patterns.

3.2.5 Summary

A policy specification language is designed to help a system administrator easily enter the required decisions or rules of behavior that are desired for the proper functioning of the information system. Hence, this language should be easy enough to be used by a human administrator and at the same time strong enough to be able to specify any kind of policy requirement as well as possibilities for future extensions of those requirements. Through this study, it can be noticed that the domain of policy-based management matured in the last decade and moved from simple languages centered over individual system components to higher-level languages offering more structures and targeting the management of the whole system.

However, there is still much to do in policy-based management and policy notation. We summarize the following set of useful constructs that are desired to be present in a practical high-level policy specification language:

ECA rules for event triggered management actions. The support of event histories, as in PDL for example, helps in identifying specific evolution patterns which can bring valuable information to correct policy decisions. Ponder and PDL offer ECA rules in a more natural way (as predicates).

Structuring techniques to promote scalability of policy specification in large systems. This requires the ability to apply policies to large collections of objects. In this context, the work undertaken in DMTF for the representation of the different managed entities is very useful. In additioon, the use of domains entity grouping facilitates the specification of policy subjects and targets. Specifying policy scope in terms of subjects and targets is useful in identifying the context of policy and helps in policy analysis.

Policy Composition structures to group together policies relating to some criteria. Composite policies are essential to manage policy in large enterprise information systems. PCIM provides the PolicySet class for policy grouping and allows the composition of conditions and actions into complex hierarchies. Ponder provides a more elaborate way for policy grouping in terms of Groups, Roles, Relationships, and Management Structures. The two structuring techniques are not incompatible and could be combined into one specification language.

Reusability of existing definitions that can be accomplished via the use of object-oriented concepts in policy definition. Both the DMTF and Ponder use Object Oriented concepts. Ponder also offers policy class definitions, named policy types, but does not offer the specialization of policy classes while it allows the specialization of composite policy structures. Modeling a policy as a class is useful for deriving more specific policies and reusing existing design. Allowing policy definitions to contain attributes and methods is also important. For example, complex actions can be specified separately in a method and then called in the action part of the policy. Attributes can also serve to characterize specific values to be used in the condition part.

Extensibility to cater for new policy types that may arise in the future. This is implicitly offered by the object-oriented design.

Ease of use of the policy language. It must be comprehensible and easy to use by policy users. For all the studied languages, the policy manager needs particular skills in order to specify the policy correctly. Ease of use can be provided by developing graphical interfaces for policy definition and/or building a natural language processing layer which translates naturally specified expressions into structured policy terms. Work on POWER [51] serves in this area. However, more important is the notation we use for technical policy specification before going up to natural language policy specification.

Uniform management An final desirable feature concerns uniform management introduced in Ponder [1] where policies are employed to manage not only other system components but also system policies them selves. This allows the uniform automation of activation, deactivation, and redefinition of policy objects via policy objects.

The next section deals with the current state of knowledge on the modeling of Service Level Agreements and how they can relate to policies both at the high and low-level of management.

3.3 SLA specification

Traditionally, an SLA is a contract between a service provider and a customer wherein all aspects of the service to be provided are specified. The SLA covers availability, performance, customer service details, as well as the measures to be taken in case of deviation and failure to meet the asserted service guarantees.

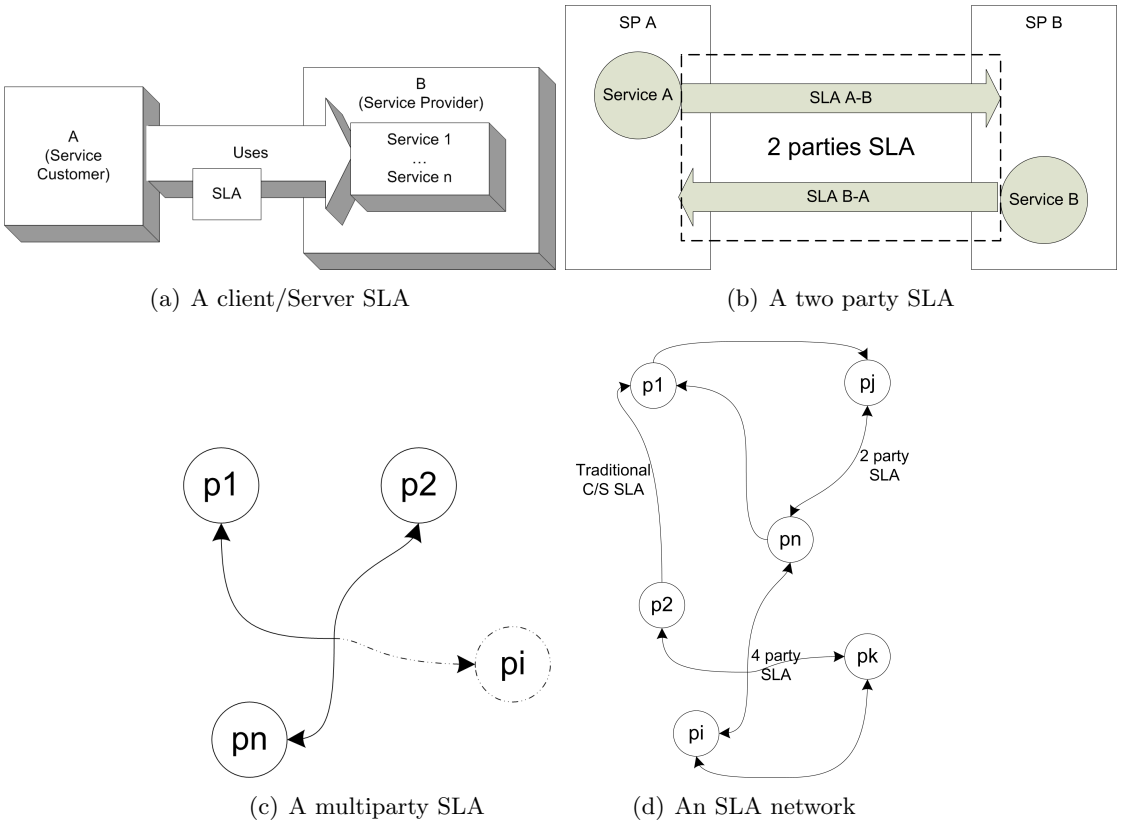
This section will first present the requirements for a good SLA modeling. It then summarizes current efforts in SLA modeling. After that, it presents a UML-based SLA model, called GSLA [55], which would capture the identified conceptual components that have been previously identified. An example of usage of the GSLA is given at the end.

3.3.1 Requirements for SLA Specification

SLA models proposed in the literature [18, 56–59] reveal a similar overall structural components. Adapted from [18, 60], an SLA is defined as a contract between a Service Provider and a Service Customer which sets a *clear measurable common understanding of the minimal expectations and obligations about what the customer is requesting and what the provider has committed to provide and at which constraints*. The constraints may be of any type and normally include contract scope (temporal, geographical, etc.), the agreed upon billing policy, as well as the behavior in case of abnormal service operation. Hence, an SLA constitutes a legal foundation that both parties can rely on in order to plan their relative businesses and future growth.

However, this view of SLAs concerns only single client-server relationships. As such, it is unable to capture many of real life situations where there exist complex business relationships involving more than a simple client and a simple server. If an SLA is to be used to model relations in an applicative peer to peer network or business wireless community setting where the obligations and services involved extend beyond simple file sharing.

The simple form of an SLA that is just above a simple client/server SLA occurs when two parties, say A and B, agree upon a given exchange of services. For example, A delivers some service(s) to B and B delivers other service(s) to A according to specified constraints. A typical example is that of two physically neighbor Network Operators where the exchanged service is the bandwidth contained in the links that join them. In this case, it is both useful and more uniform to capture the service relationship between the two operators into one semantic and structural unit. In this case, the SLA between A and B can contain rules that might specify some actions related to flows from A to B if ever the B-to-A service experiences unexpected irregular behavior. Using two client/server SLAs would not be enough as the rules that specify what actions to do in case the service of the other party downgrades below acceptable quality cannot belong to any of the



(a) A client/Server SLA (b) A two party SLA

(c) A multiparty SLA (d) An SLA network

Figure 3.5 Different types of SLA settings

two SLAs and need to be specified outside the normal SLA specification as stand alone policies.

A more complex SLA type can also exist where more than two parties participate in the contract with complex dependencies between all the involved parties and services. This would be the case, for example, where multiple network operators tightly cooperate in order to deliver strong end-to-end QoS assurances for certain well-known services. A similar example but with less stringent requirements can be found in a MANET where all parties (Ad Hoc nodes) participate in the overall QoS assurance policy of specific services.

3.3.2 The Generalized SLA Information Model (GSLA)

This section presents a generic SLA model, named GSLA [23], that responds to the requirements presented in the previous section in addition to the native support of policies as identified in the requirements for business-driven management identified earlier in section 2.4.

The GSLA is defined as a

> *contract between two or more parties linked through (a) service relationship(s) and that sets clear measurable common understanding of the role each party agrees to adhere*

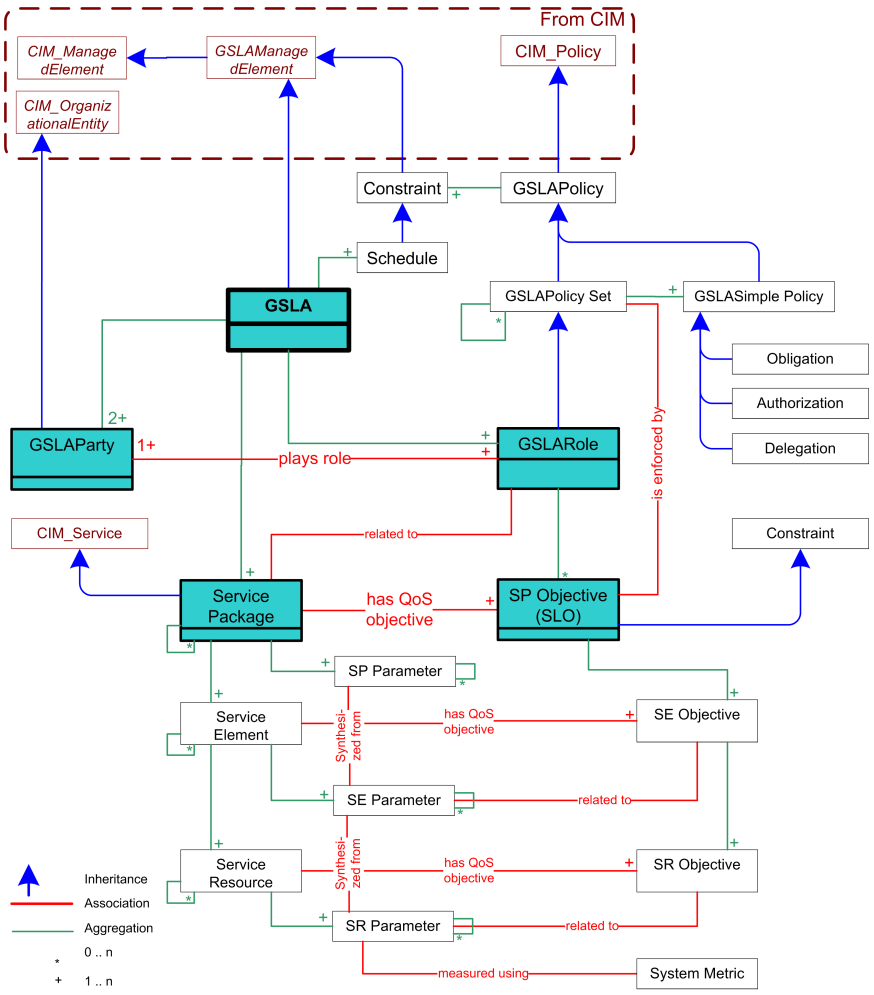


Figure 3.6 The GSLA Information Model

to.

Figure 3.6 shows the GSLA information model and is explained in the following:

- A *party role* represents a set of objectives and rules which define the minimal service related expectations, constraints, and obligations it has with other roles. During a GSLA life cycle, a required behavior or constraint related to a GSLA role is captured in the model through the abstract *GSLAPolicy* class. A role is modeled at first approximation by a policy that a party follows. Hence, the *GSLARole* class inherits indirectly from *GSLAPolicy*.
- A *Schedule* class is a specialization of a Constraint.
- A *Constraint* is an abstract class intended to capture any type of logical predicates over parameters of GSLA components.
- A GSLA is related to one or more Service Packages (bundle of one or more several services) to each of which is associated an *objective* that some GSLA party is required to guarantee

as is specified in the role(s) it is related to. A Service Package represents a group of related *Service Elements* that are instantiated and managed as a whole and/or are offered altogether to customers.

- To each offered service is associated an expected run-time quality as is promised by the service provider role and as should be experienced by the service customer role.
- *Service quality* is captured through a set of Service Level Objectives (SLOs). The modeling of SLOs is always faced with the tradeoff between customer facing QoS parameters and provider facing technical QoS parameters that are spread within technical details related to service resources. We propose a model that bridges both QoS levels.
- In the GSLA information model, multiple party service relationships are supported and each party has a set of SLOs to assure, if any, and/or a behavior to observe with respect to the other parties.
- Also, with each SLO there are normally associated policies that specify actions to take in case the SLO has not been respected or some warning-level has been reached. Policies are also generated by a role for enforcing its SLOs. Such enforcement policies are normally only viewed by the party related to that role and need not be specified at the common SLA unless explicitly requested by the concerned service customer party.
- The behavior of a party is ultimately modeled through policies. A *GSLARole* is modeled through a set of policies as well as the set of SLOs it is required to ensure as part of its responsibilities in the GSLA.
- A role contains two different types of policies: *role intrinsic policies* and *SLO enforcement policies* (figure 3.7). Role intrinsic policies are not linked to a specific SLO and are not the result of an SLO mapping.
- A GSLA *Service Package* is composed of a set of *Service Elements* each of which is related to one or more *Service Resources*.
  - A Service Element [18] typically represents a single technology-specific service capability, such as an IP connectivity, or an operational capability, such as a help desk support.
  - A *service package* groups together a set of related service elements that need to be instantiated and managed as a whole. A simple example concerns an Internet web access service, which requires at least an 'HTTP protocol' and an 'IP connectivity' service elements. A service package concerns a group of services that are generally offered altogether, such as a web browsing and a mail service and/or a web hosting service. In this case, service elements can have "requires" relationships that are needed

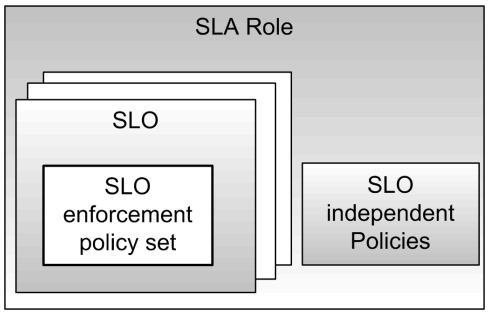


Figure 3.7 SLO enforcement Policies and Role intrinsic policies
In addition to SLO enforcement policies, a GSLA role can contain policy definitions that are intrinsically related to the role itself or to the set of SLOs it is required to fulfill.

for their operation. A service element can be directly related to a physical service resource.

– A *service resource* is intended to be transparent to the customer and represents a basic provider resource, such as an email server, a network element, a processing server, a database, or a stockpile.

Modeling Service Level Objectives

Overall Service Quality is a broad concept covering many performance aspects and numerous measures. It may be defined as [18]

> *the collective effect of service performances that determine the degree of satisfaction of the user of the service*

The GSLA information model captures all aspects related to service quality starting from the Service Package Objective class. A *Service Package Objective*, as its name suggests, defines Service Level Objectives for one or more service packages. It is essentially a constraint and it can be defined in two different ways:

- First, it can be defined as a set of predicates or logical expressions over one or more service package Parameters. This represents a high level way of defining QoS objectives based on direct calculus made over high-level service parameters that are synthesized up (Figure 3) from the basic System Metrics up through System Resource (SR) parameters and System Element (SE) Parameters.
- The other possibleapproach is to calculate the objectives based on QoS appreciations coming from subordinate Service Element Objectives. This represents the high-level compilation of low-level QoS appreciations. This second approach to evaluating the overall Service

Listing 14 Example of the computation of a service package objective

```
Suppose for example that the offered service package is composed up of a web
browsing, email, FTP and VoD service elements.

A service package parameter SPgMeanPerf may be defined equal to
Mean(SPgPerfTimeSeries). SPgPerfTimeSeries being a series of SPgPerfElement
values. A SPgPerfElement parameter is calculated through a function f(MailSE.Perf,
FTPSE.Perf, WBSE.Perf, VoDSE.Perf) of the performance of each constituent service
element of the service package. The value of SPgMeanPerf may then be used by certain
service package objectives, say SPgPerfO, to appreciate whether the overall service
package performance is acceptable or not based on some agreed upon thresholds.

The other possibility is to define a service package objective that constructs the
overall performance appreciation based on the results of the individual service
element objectives of each service element of the package. In this case, the
SPgPerfO might contain a set of logical expressions such as "if (at least three of
(WBSEPerfO, WBSEPerfO, WBSEPerfO, WBSEPerfO)) == good then SPgPerfO←good)".
```

Quality reflects more accurately the way users appreciates a given service infrastructure, i.e, by giving a final appreciation based on separate 'sub' appreciations over the different service components.

Role Based SLA modeling and the policy-based approach

The final building block of the GSLA model considers the behavior each party of the GSLA is required to observe during the GSLA life cycle. As each party adheres to at least one a role in the GSLA, role modeling should be able to capture all aspects of behavior a party can have. From the study conducted in section 3.2 on policy specification [35].

A system component behavior at the lowest view is modeled as a policy. This covers any component that can be concerned during the overall GSLA life cycle, be it a person, a software component, a network element, or an organization. Research made in Role Based Access Control (RBAC) systems and security management systems shows that policies are mainly of two kinds: *action policies* and *authorization policies*. Conceptually, an authorization policy defines conditions for limiting access to or actions over some system components or operation. Authorization policies are subdivided into permissions and prohibitions. An action policy on the other hand, defines conditions that need to be met in order to execute some system operations.

Conceptually, an *action policy* is made up of two main components: a (set of) condition(s) implying the execution of a (set of) action(s)

$$\text{ACTION POLICY} = \text{CONDITION(s)} \Rightarrow \text{ACTION(s)}$$

A Condition is a generic term. It can be a temporal condition, a condition over existing system parameters, or a condition concerning specific system states. Because of the special importance temporal conditions and system state conditions possess, further decompositions of a Condition

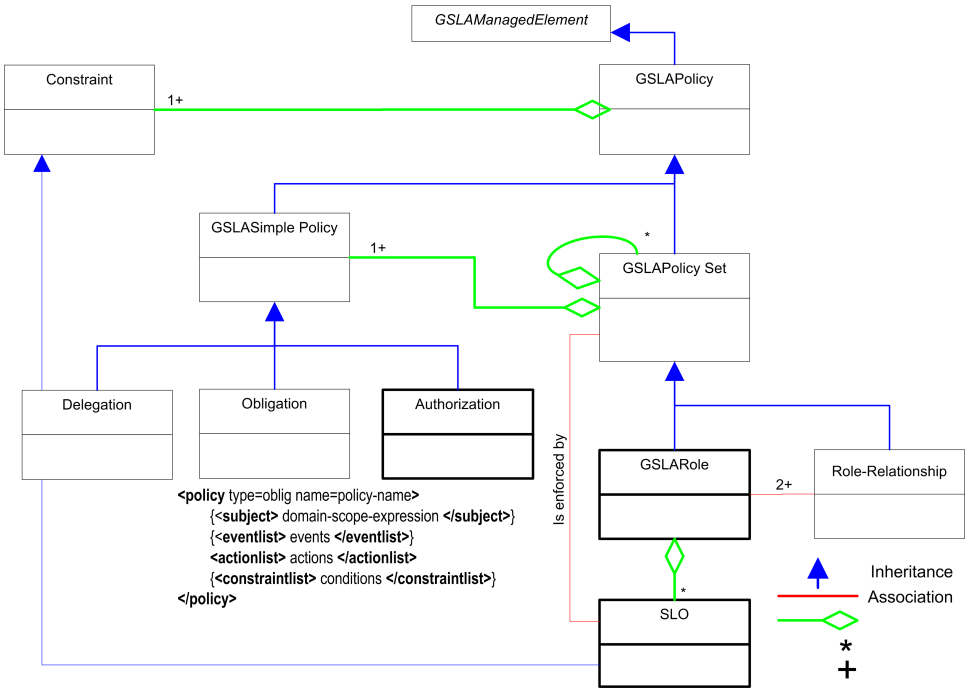


Figure 3.8 Modeling of roles and policies in the GSLA

into temporal condition, event condition, and other conditions has been largely accepted within both the networking management and the security management communities.

Borrowed from Ponder [42] notation, the GSLA recognizes three policy types: permissions, prohibitions, and obligations. A *Permission* specifies an authorization to execute a certain action, such as accessing a customer' profile data. It can also be a delegation to another role to execute an action. For example, a service provider can delegate the monitoring of some SLA parameters to a third party that the customer is unaware of. In this case the delegation policy represents a formal way to capture the role of that third party. A policy can also represent a *prohibition*, that is, a negative authorization to access some system components or execute some specified action types. Finally, an *obligation policy* represents actions that a role is required to take during system run time. Obligations represent the key to QoS policy specification in the GSLA. An obligation specifies the actions that a role object agrees to execute within the system when specific events occur and a set of conditions are met.

Finally, a *role-relationship* is a type of policy set which contains rules defining the rights and duties of roles towards each other. For example, if a service provider is required to send a monthly report about a given service quality parameter to the customer, a role-relationship object will contain this specification. A role-relationship can also include policies related to resources that are shared by the roles. It thus provides an abstraction for defining policies that are not part of the role specifications, but are part of the interaction between the roles.

3.3.3 Related work on SLA modeling

In the literature, several SLA information models are proposed. The main feature of those SLAs is that they are all built on the client/server model, with an emphasis on isolated individual view of SLAs, in which the network wide view of service and SLA interactions is nearly absent. There are two main works consider the network-wide view of SLAs [57,61]. They consider the modeling of SLAs with particular emphasis on SLA parameter monitoring.

[62] considers Client/Server SLAs and introduces the notions of client responsibilities, server responsibilities, and mutual responsibilities with respect to non-functional properties. The GSLA-Role seamlessly captures all those parameters. Moreover, it extends it by allowing roles to be attached to more than one party, hence bringing forth a multi-party responsibilities object.

WSLA [57,63] from IBM research and WSMN [59,61,64] from HP Labs analyze and define SLAs for Web Services by building new constructs over existing Web Services formalisms (WSDL, WSFL, XLANG or BTP/ebXML, etc.). Sahai [64] states: " ...as these web services interact and delegate jobs to each other they would need to create and manage SLAs amongst each other. SLAs are signed between two parties for satisfying clients, managing expectations, regulating resources and controlling costs". It specifies SLOs within SLAs and relates each SLO to a set of Clauses. Clauses provide the exact details on the expected service performance. They are used to specify service level objectives. A clause is based on measured data. Each clause represents an event-triggered function over a measured item which evaluates an SLO and triggers an action in the case the SLO has not been respected. In a recent work, [65] defines an FSA (Finite State Automaton) for SLA state management in which each state specifies the set of SLA clauses that are active. Transitions between states can be either an event generated by an SLA monitoring layer or an action taken by parties in the SLA. This represents a step towards the implementation of the declarative nature of SLOs.

In contrast to the multi-party approach to service relationships we proposed, the Web Services Management Network (WSMN) framework [61] considers the Web Services business environment as a network of individual "Client-Server" relationships. WSMN is a middleware architecture representing a logical overlay network constituted of communicating intermediaries, each such intermediary implemented as a proxy situated between a service and the outside world. It assumes a service-centric model for application usage, and focuses on managing the service offering (as opposed to the internals of applications). ý

[57, 63] define the Web Service Level Agreement (WSLA) Language for the Specification and Monitoring of Service Level Agreements for Web Services. The framework provides differentiated levels of Web services to different customers on the basis of SLAs. In their work, an SLA is defined as a bilateral contract made up of two signatory parties, a Customer and a Provider.

Service provider and service customer are ultimately responsible for all obligations, mainly in the case of the service provider, and the ultimate beneficiary of obligations. Supporting parties are sponsored by one of the two signatory parties to perform one or more roles according to the monitoring model. In this view, our model extends to multi-signatory-party service relationships with the supporting party role captured through delegation rules. WSLA defines an SLO as a commitment to maintain a particular state of the service in a given period. An action guarantee performs a particular activity if a given precondition is met. Any party can be the obliged of this kind of guarantee. Action guarantees are used as a means to meet SLOs. In our model, we consider a modular design in which SLOs are first specified in a declarative manner. Then, special enforcement policies are generated to meet the SLOs. These policies need not be specified in contract sign time, they can change according to run-time circumstances. [65] considers a business goals oriented view, in which an SLO might be deliberately left down if it happens that this would help the responsible party maximize his local business objectives cost functions. [66] proposes an approach of using CIM for the SLA-driven management of distributed systems. It proposes a mapping of SLAs, defined using the WSLA framework onto the CIM information model.

Finally, [67] introduces within a client/server SLA model, the notion of service-centric, client centric, and server centric views of an SLA. It proposes an SLA model that supports policies in a basic form. Views represent an important concept and we consider them as a second step in the refinement of the GSLA model. Hence, each GSLA party would have its own local view of the overall logical GSLA. At a transversal plan, refinement of SLOs and high-level behavioral rules represents a second way in defining views over the GSLA components.

In the following, two examples are presented to illustrate how service relationships can be captured within the GSLA model.

3.3.4 GSLA example I: Service delegation for VoD delivery

Assume a set of VoD (Video on Demand) service providers which collaborate to offer as much content as possible to their customers and at the same time seek to maximize their individual profits. It is to note that similar cases hold for example for internet flight reservation agencies, hotel reservation agencies.

Suppose that there are have five VoD service providers, which are located respectively in Paris, London, Berlin, Rome, and Madrid. Each provider serves customers worldwide (over the Internet) and offers a variety of VoD content but specifically specializes in serving local language content. Since no VoD can contain all the existing Video contents with all existing languages, the VoD service providers would agree to delegate customer requests, whenever deemed appropriate, to the

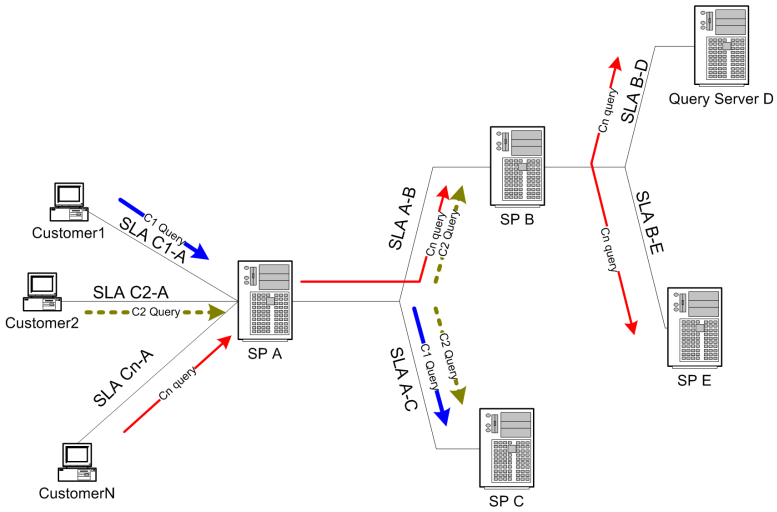


Figure 3.9 Service delegation between a set of geographically distant VoD-service providers

provider which is better at providing the service. The agreements are not only required for just content shortage, they are also required to meet QoS requirements. The inability to meet a QoS requirement can have multiple sources. If a provider is congested or the network path linking it to a new customer session is congested, it might delegate the task of serving the customer session to another provider that is able to meet the customer requirements. The delegation is operated on the fly and is managed through specific delegation policies that each provider incorporates within the service provider Role it plays in the GSLA it contracted with the service customer.

3.3.5 GSLA example II: Wireless Management Communities

This section proposes a service-driven model for structuring WLANs into overlay networks of interacting Wireless Management Communities. A *Wireless Management Community* (WMC) is composed of a set of *parties* and is governed by a charter named the WMC-SLA (Service Level Agreement). A WMC constitutes the basic unit of management upon which will be installed any form of service interaction between parties belonging to the wireless community. The WMC-SLA model is presented as a use case of the GSLA.

Concept of a Wireless Management Community (WMC)

We define a Wireless Management Community 3.10, or WMC [3], as a set of physically close wireless nodes that agree to collaborate for the sake of exchanging one or more services. The WMC is governed by a set of (either high-level or low-level) objectives and rules of behavior that are

[3] Published in [23]

gathered within a WMC-SLA. The WMC-SLA represents the behavioral charter to which each member of the WMC is entitled to adhere to for the proper conduct of communication services within the WMC. It is conceptually perceived as an SLA, since having a WMC membership is viewed as a service in itself in the same way Network connectivity is considered a service upon which other services, such as Internet connectivity, depend upon.

In the near future of ubiquitous wireless connectivity where approximately all first hop communications will be wireless, the need to regulate the parties with which we communicate is of particular importance. A WMC can relate to as many situations as is the diversity of our personal communication environments, be it a corporate WLAN, a set of researchers or business associates holding a meeting, a teacher giving a lecture in a lecture-hall or at an open air field, a set of people doing some peer to peer file sharing, an open air interactive game, and even to some stringent situations such as military or disaster settings. The goal is to capture the structure necessary to the WMC to regulate service interactivity and communication medium usage inter and intra WMCs.

A WMC can have a void WMC-SLA and hence be completely open to any new member. This case can represent for example normal MANETS that offer best effort services. On the other hand, it can be strictly closed either in terms of its members or in terms of the membership constraints, such as the case for QoS enabled WMCs, contained within its WMC-SLA.

The WMC concept is intended to offer a structuring mechanism that would bring stronger flexibility, security, reliability, and scalability to service interactions in a conglomeration of WLANs. These criteria are assured by the following properties:

1. The first property concerns *loose containment*, which means that members of A WMC are not forbidden to belong to other WMCs unless explicitly stated in the WMC-SLA. This type of membership is not inclusive and is nearer to the notion of a directory or domain [68] with the added feature of having rules that govern the membership to the WMC instead of explicit affectation.

2. The second property is that of *scalable composition*. This means that a WMC can also be defined in terms of a WMC-Expression representing a combination of union, intersection, difference, and complement operations applied to other WMCs. The difficulty here concerns the possible conflicts that might exist between the WMC-SLAs of the involved WMCs.

The environment that a WMC offers is by its very nature a multi-party environment in which each member has at least one active role to play. In the case of a MANET, the basic role which at least any WMC member must play concerns the correct routing of information within the WMC according to the WMC-SLA. WMC members are free to have other service relationships and interactions as long as they still respect the role assigned to them within the WMC-SLA.

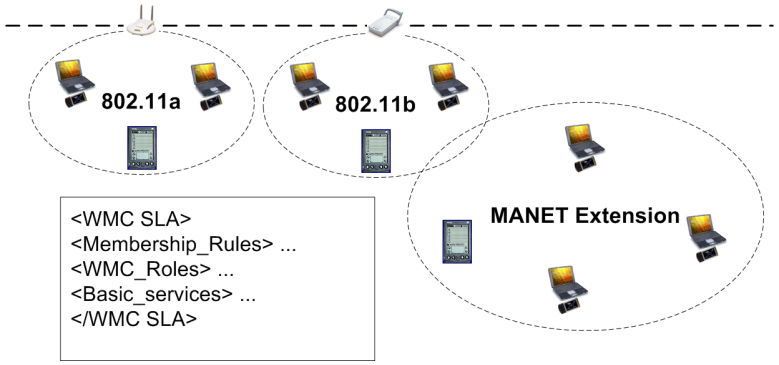


Figure 3.10 A corporate wireless management community (WMC)

Using the GSLA to model the WMC-SLA

A Wireless Management Community SLA (WMC-SLA) exists within a WMC and contains all necessary information relevant to the profile of the WMC Parties and the basic services that exist, or are supported within the WMC. All types of *WMC management policies* also exist within the WMC-SLA. We call these *Membership-Rules* and they constitute a set of rules that are needed to manage events that concern the WMC parties and the basic services offered by the WMC. Among these, there are find authentication rules for new members and constraints limiting the maximum number of parties and the temporal interval within which the WMC is available.

Required Roles are also specified in the WMC. For example, a WMC for an ad hoc network might contain constraints concerning the ad hoc routing capabilities of the communication device used by each ad hoc party. Such constraints are collected within a *compulsory role* 'AdHocRouter' that each party should support. A new party cannot be accepted if it is unable to fulfill a compulsory role.

The SubjectParty attribute of the AdHocRouter WMC-Role indicates that it is a compulsory role for every WMC member. A set of services are required for this role such as the support of QOLSR routing and a specific (fictive) QoS mechanism called AdHocDiffServ. Failing to support the AdHocRouter WMC-Role denies the possibility to associate to the WMC. On the other hand, being accepted in the WMC requires the execution of some configurations in order to adhere to the AdHocRouter profile.

After the different roles and party profiles have been specified, there remains the issue of how users are actually accepted or rejected to join a specific WMC. For this, there must be some WMC party with the prerogative to decide that such a user can be accepted to the WMC or that such party should be disconnected from the WMC. A specific role is assigned to this party called the *WMC-Controller*. At least one party must assume the WMC-Controller role. Because of the nature of WMCs, more than one party can play the "WMC-Controller" role; and depending on

Listing 15 An AdHocRouter WMC-Role

```
<WMC-AdHoc>
 <scope start='04-06-21 8:30' duration='4 hours'/>
 <Membership-Rules> ... </Membership-Rules>
 <SPG>
 <Service name="Printing" IpAddress="" PortNb=""/>
 <Service name="VoIP">..service details..</service>
 <service name="VideoConferencing">....</service>
 </SPG>
 <WMC-Role name="AdHocRouter" SubjectParty=all>
 <constraint>
   <routing protocol=QOLSR>
 </constraint>
 <constraint>
   <QoSSupport technology=AdHocDiffServ>
 </constraint>
 <constraint>
   <MAC protocol=802.11e MIN=11Mb/s encryption="EAP">
 </constraint>
 <SLO>
   <Service>routing</Service>
   <constraint name="Classes_of_Service">
   <class name="Gold" bdwidth="5%" forwarding="EF"/>
   <class name="Silver" bdwidth="10%" forwarding="AF"/>
   <class name="BE"/>
 </constraint>
 </SLO>
 <WMC-Role>
</WMC-AdHoc>
```

the WMC type, the decision process can be based on either one WMC-Controller or the set of all WMC-Controllers of the WMC.

A conference use case

The last section gave an example of an ad hoc WMC. This section considers the case of a conference meeting in which a wireless access infrastructure is installed to get access to the Internet and other networked facilities such as VoIP, p2p file sharing, Video Conferencing, local news Broadcasting, etc. In this regard, we would like to structure the different participants of the conference within a set of different but overlapping WMCs. The number of involved WMCs depends essentially on the importance of the conference and the number of participants. Major conference meetings, such as IEEE ComSoc ICC or Globecom in the computing field, involve a considerable number of participants each of which may be involved in multiple activities during the conference time span. For example, ICC 2004 had 864 accepted papers and the number of attendees exceeded 1500, with a large number of sub-meetings including sessions, tutorials, executive committee meetings, technical committee meetings, VIP meetings, and business meetings. Some meetings are open to all participants such as technical symposia sessions; while others are restricted to specific members such Executive committees or VIPs.

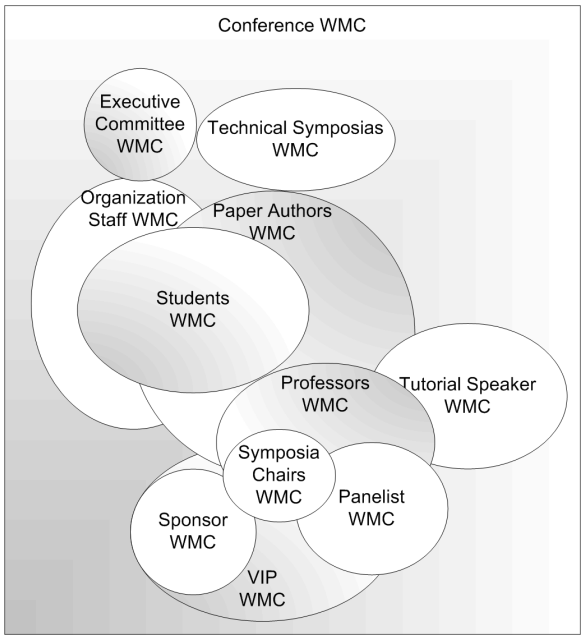


Figure 3.11 Example of a Conference WMC

Figure 3.11 shows some intersection properties among all the possible WMCs that might exist within a conference setting. When circles (or ellipses) intersect then the two corresponding WMCs have members in common. However, non-overlapping WMCs are not necessarily independent. For instance, an executive committee member can also belong to the Professors WMC and a panelist can be a student.

In the context of the global Conference WMC, which we call here WMC-CONF, each sub-WMC has a specific associated WMC-SLA which subsumes (includes) the global WMC-SLA. In addition, association to a specific WMC may be based on an authentication mechanism and the WMC membership rules.

First, each member of the WMC-CONF at his subscription is attributed an ID and a password which identifies the type of member he represents (student, author, sponsor, etc) and with which he will be accepted to join a set of authorized WMCs. The authentication mechanism can also be more sophisticated and included within a CIM or JAVA-card that enables wireless devices to automatically connect and authenticate to the appropriate WMCs and take advantage of the offered services.

For a conference member to join a specific WMC, he would normally need to have access to some services that this WMC offers. The WMC-conf takes care of continuously presenting to its members (or parties) the availability of the different sub-WMCs. When a user enters a technical session room, he would be invited to connect to that session WMC and start using available services.

Listing 16 Some of the WMCs required within a large conference meeting

```
<WMC-SLA name="ICC04"> <Scope> <Temporal>
    Start = "20-06-2004 08:00"
    End = "24-06-2004 18:00"
 </Temporal> <location Area="Disney Land, Paris">
     <site>"New Port Bay Hotel"</site>
     <site>"New York Hotel"</site>
 </location> </Scope> <cardinality max=2000> <Authentication>...</Authentication>
 <SPG>
   <Service name="Printing" IpAddress="" PortNb=""/>
   <Service name="ColorPrinting" IpAddress="" PortNb=""/>
   <Service name="VoIP">..service details..</service>
   <service name="VideoConferencing">...</service>
 </SPG>
</WMC-SLA>
<WMC-SLA name="ICC04-symp01-Session06">
    <Scope>
     <Temporal>
      Start = "21-06-2004 14:15" End = "24-06-2004 17:45"
     </Temporal>
     <location>
       <site>"New York Hotel" <room>"Montparnasse"</room></site>
     </location> </Scope>
   <cardinality max='30'>
   ...
</WMC-SLA>
```

Among the available services can be direct streaming (or broadcast) of the talks; a white board for exchanging questions and answers concerning the presented ideas; a file sharing service for exchanging some demos or related work; and when members intervene for some questions they may appear on the screens of the other members belonging to the same WMC, etc. Privileged in the talk streaming service are prioritized members such as VIP, organizing committee members, the session chair, the session papers authors, but also those members who for some reason happen to be outside the session room but still want to be informed about what's happening around in it. Being outside the session room is understood by the impossibility to be directly accessible by the room's access point; that is, accessibility is provided either through another access point or indirectly via other members (multi-hop). This last case is common and many members although physically present in one session might want to have glances about what's going around in other parallel talks of interest to them and, if possible, they might intervene within the session' white board or even debate if he is located outside session rooms.

In order to manage the services offered by the different WMCs, special QoS support needs to be provided by the wireless infrastructure; and a special platform is required to enforce prioritization policies and access control policies for the authentication of members; as well as special routing policies to manage the available bandwidth. Policies are required for the appropriate set up of talk(s) streaming or other videos, VoIP support, and the prioritization of traffic based on the importance of the conference members.

3.4 The Business-Driven Management Framework

Regardless of how successful an enterprize might be with its adoption of a policy-based management solution, it must be remembered that its information infrastructure is aimed at the provision of a service which is exchanged for an economic value. Therefore, it is of significant importance to make policy-based management aware of business-level considerations. This observation is central to our approach which defines a management stack including a business-driven management layer and an underpinning policy-based resource control layer, with the first providing timely business context to the latter.

This section presents a framework for the business and policy-based management of information systems [4]. The framework features a high-level business and service-driven layer on top of a policy-based resource control layer. The linkage between the two layers is assured in two ways. First, a policy-based refinement engine, supported by an appropriate SLA information model, ensures the off-line derivation of low-level policy rules from high-level requirements that are expressed in terms of SLAs and business objectives. Second, a business profit maximization engine provides decision support that seeks to maximize the business goals at system runtime whenever it is deemed appropriate.

The current approach to policy-based systems management does not provide mechanisms to drive policy-related decisions at system runtime based on the business and service-level context. We explain how this is remedied in our framework through the interaction mechanism between the *reactive* policy-based resource control layer and the *more proactive* business profit maximization engine.

3.5 The Business-Driven Management Framework

The objective of the business-driven management framework (BDMF) is to drive the management of system resources and services from the business perspective rather than from the low-level technician's point of view. Most of the times, when tradeoff-kind of decisions are to be made, system managers use heuristics in order to determine which of the option available to them guarantees the minimum cost or least disruption to the service. But unless the impact of carrying out the chosen course of action onto the business layer is understood, one may run the risk of solving the wrong problem optimally. Because of this, the BDMF was designed according to the principle of making information that pertains to the business visible from the resource-level and vice versa.

[4] published in [24]

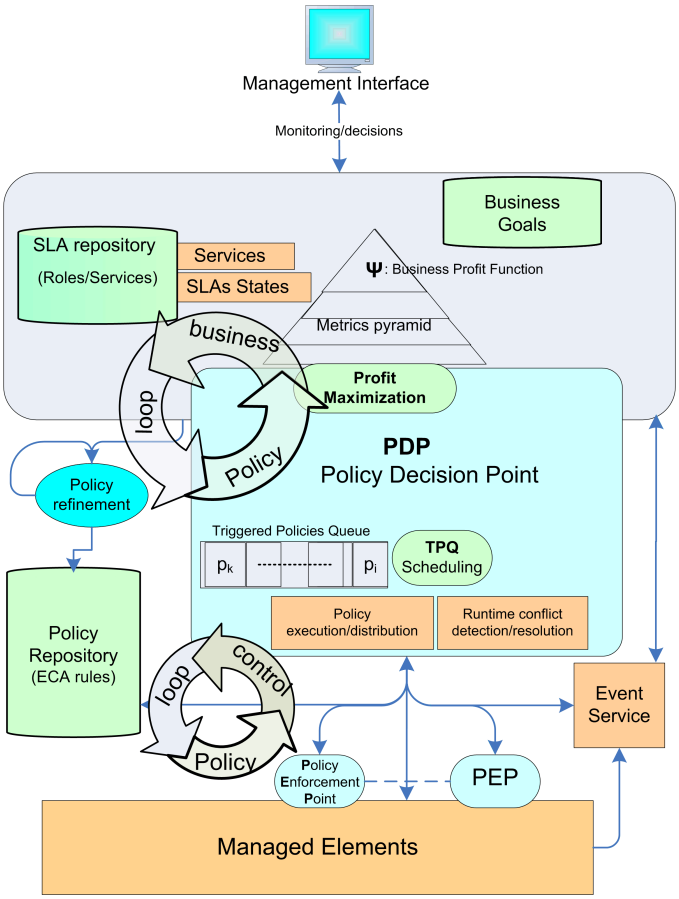


Figure 3.12 The Business-Driven Management Framework

- The *policy-control loop* identifies the traditional view of the dynamics of policy execution where policy is determined beforehand prior to system execution through a refinement process.
- The *policy business loop* above represents our claim of the additional need of controlling policy dynamics at runtime not through a default treatment but rather through a more elaborate engine that seeks to maximize business profit based on runtime context.

As presented in figure 3.12, the BDMF is divided into two main layers. The high-level business layer is intended to host the *long term* control of the information system based on the business objectives and market strategy of the service provider. Beneath it is a resource control layer that hosts the real time logic for the reactive short term control of the resource infrastructure. The Business Management Layer is responsible for optimizing the alignment of the usage of system resources with the objectives of the service provider based on a set of business objectives defined and audited over relatively long periods of time (monthly, quarterly, etc.).

Business goals are the reflection of the service provider' business strategy and range over diverse key performance indicators related to service operations and SLAs. Business relationships con-

tracted by the service provider are formalized by SLAs and modeled using the GSLA information model (section 3.3.2). Using the GSLA, each contracted service relationship is modeled as a set of parties with each party assigned to one or more roles that work together to achieve the SLA objectives. Each role in the SLA is associated with a set of Service Level Objectives (SLOs) to be achieved; as well as a set of *intrinsic policies* related to the role behavior per see. The Role-to-policies mapping engine, translates Roles, SLOs and high-level policies into a set of low-level policy rules. Low-level policies enclose all the logic required to correctly run the system resources.

Business objectives affect the way SLAs are defined and managed at the resource control layer. So whenever a business objective is changed, added, or removed, important impact takes place at the long term time scale on the SLA database. Low-level policies are dealt with by the Policy Decision Point (PDP) module [14, 69] of the resource control layer. Part of the PDP's task is to monitor and respond to system events and notifications by selecting, activating, and scheduling the enforcement of the appropriate policies at the appropriate resources. The PDP contains also sub-components for policy runtime conflict detection, root cause analysis, generation of the set of options available in the presence of some incident or problem, as well as a the generation of appropriate configuration flows in order to enforce active policies.

As it is impossible to define policies upfront to cover all runtime events, it will happen that low-level policies may not be sufficient to deal with certain conditions. In those cases, the PDP passes up the control to the MBO engine of the business layer. Given the different available options, the MBO will select the one that will maximize the value to the service provider. That is, the option that will result in the closest alignment to the business objectives. Such interactions offer also the opportunity for the architecture to learn and refine the policy repository.

3.6 Business Management Layer

The Business Management Layer is responsible for

- managing the life cycle of the contracted SLAs.
- managing unexpected events by maximizing the alignment to the business goals of the service provider
- deriving low-level policies that will ensure compliance to the contracted SLAs and business goals.

Business goals are used by the roles-to-policies refinement engine to drive low-level policies. When changes in business goals occur, the refinement engine update existing policies accordingly.

3.7 Business Profit Maximization Engine

The business profit maximization engine computes the alignment to high-level goals that is expected for each of the possible given options (or course of action) aimed at managing the information resources. The engine needs to be able to monetize the measure of alignment thus derived and use the monetization value together with other information on the cost of carrying out the respective course of action to rank the available options. Upon ranking the options, it returns a suggestion on what course of action to take, substantiated by the evidence that it has for assessing the alignment with respect to the business objectives.

Although it can vary for different information system management domains, the timescale at which the profit maximization engine works tends to be of the same order of magnitude as the system decisions that require humans in the loop. Depending on the domain, that can be of seconds, minutes or even hours. The timescale is therefore much longer than the one at which the PDP works. As the PDP is required to quickly and reactively deal with situations that do not require human intervention.

Figure 3.13 shows an example of a dependency that the profit maximization engine needs to process in order to calculate the business impact of an "SLO compliance" indicator at the occurrence of a single incident in the system.

A formulation of the profit maximization engine functionality for incident impact minimization is developed in [70], where incident prioritization for the sake of impact minimization is approximated to an instance of an integer assignment problem. The impact of an incident i on business indicator Ij when i is supposed to be dealt with within a time of at most ti of its occurrence is represented by $Ij(i, ti)$.

Business indicators can be of various types, such as overall customer satisfaction and overall SLA compliance. The calculation of the impact of an incident on a given business indicator is an inherently complex process and is to be assured by the profit maximization engine. Besides, at system run time, multiple incidents can occur concurrently and there is a need to prioritize

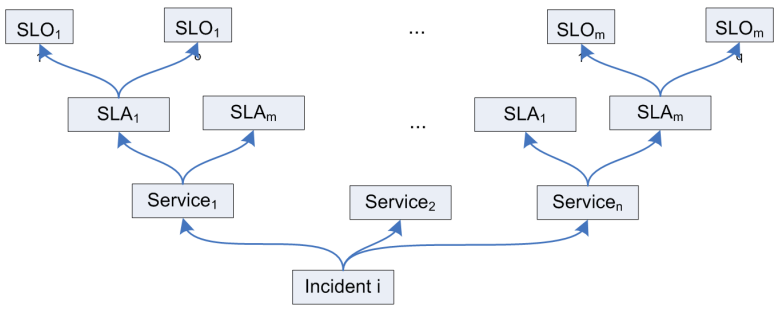


Figure 3.13 SLO and SLA related Impact tree of an incident

between them in order to determine which incident needs to be dealt with first. Mathematically, this summarizes to the determination of:

$$\text{Minimum (Impact (incident-set))} = Min\left(Sum_{I(i,t_i)}\right) | i \in \text{incident-set}$$
$$\text{where,} I(i,t_i) = \sum_j \omega_j Ij(i,t_i) \text{and} \sum_j \omega_j = 1$$

The I_j's represent business indicators. An ω_j represents the "importance" that the service provider gives to business indicator j. Hence, depending on what is currently most important to the service provider (as the priorities might change by time) different optimization choices could be taken. Such high-level driven decisions are by no means at the grasp of a traditional PDP and this is why we advocate that it might often end-up optimizing a local function (a specific SLO for example) if it blindly applies a set of policies based on a default incident prioritization scheme.

3.8 Conclusion

This chapter discussed the state-of-art formalisms related to business-driven management. It initially analyzed different policy specification languages and models and presented a set of criteria to identify what can be regarded as a good policy specification language. It is noteworthy to mention that some existing languages, such as Ponder, are already at an acceptable level of maturity in terms of the specification. Thus, there is no need for yet another policy language. What is required is some few enhancements and much practical experimentation with these languages in order to mature in our understanding of policy as a programming paradigm.

At the SLA specification level, the GSLA information model has been proposed as a model that fits well with a business and policy-driven management paradigm. Three practical cases have been presented to show how the GSLA can be used in a variety of situations to capture complex real-scale contractual relationships.

The Business Driven Management Framework we presented shows how the GSLA, policies, and business goals can be brought together into a single framework for business-driven management. The BDMF is a simple framework for information systems management which extends traditional policy-based management with a wider decision ability that is informed and driven by the business goals and contractual obligations of the service provider. The main idea that the BDMF framework stresses is the need of a business-level policy management loop to back the traditional low-level policy control loop in order to achieve a maximization of business goals.

The next chapter provides backing for this claim through a theoretical study of the problems of business-driven policy refinement and analysis. It will also present the $\mathcal{PS}$ prototype policy simulation tool which has been implemented following the BDMF architecture with the intent of offering an environment for policy simulation. The following chapter presents a practical use case

where the business-level policy loop of the BDMF is used to maximize business profit at system runtime.

Chapter 4

Policy refinement and optimization

This chapter considers the state-of-art research in policy refinement and analysis and presents two contributions. First, we identify a set of new policy inconsistencies and describe how they need to be solved in the general case. Second, we identify the need to consider runtime policy dynamics for the purpose of policy inconsistency detection/resolution as well as for policy optimization in terms of maximizing the original high-level business goals. The Policy Simulator tool $\mathcal{PS}$, that we have developed for the purpose to serve at the dynamic analysis phase and represents a prototype implementation of the BDMF framework (section 3.4), will be presented at the end of the chapter.

4.1 Approaches to policy refinement

Policy refinement is meant to derive low-level management policies from high-level requirements. It finds its roots in requirement engineering [71, 72], which is a branch of engineering concerned with the real-world goals for functions and constraints on software systems and the way they are related to precise specifications of software behavior as well as to their evolution over time and across software families [73].

Policy analysis always goes hand-in-hand with policy refinement and relates to analyzing a set of policies for consistency and correctness vis-a-vis of the high-level policy. This phase may be required several times during a refinement process whenever the policy specification attains a new level in the policy continuum until the lowest level of directly enforceable policies is obtained.

Research in policy refinement has recently gained increased attention from the policy research community [19–21, 51, 74–78]. This chapter summarizes efforts conducted in policy refinement and analysis and presents two contributions to it. The first relates to a new type of test on policy consistency, we call the *policy-loop anomaly test*, which adds to the existing set of policy

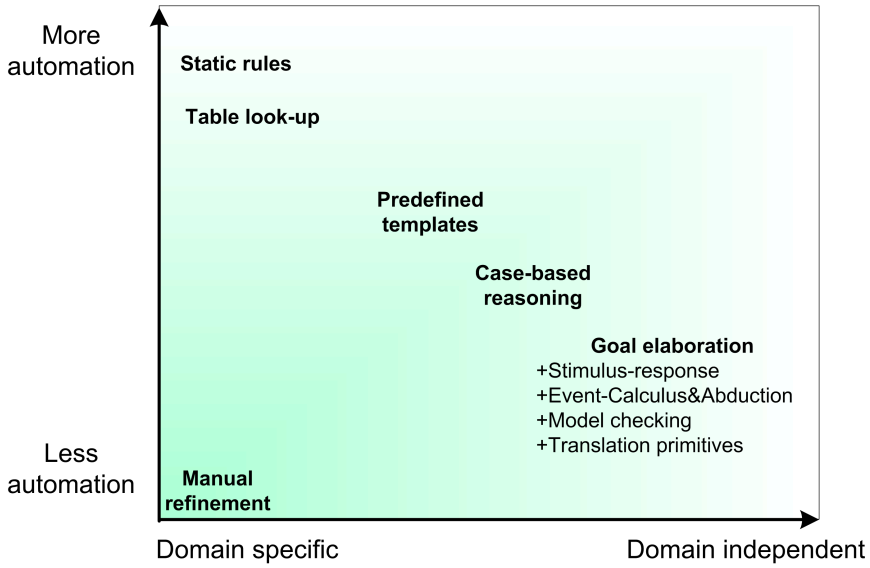


Figure 4.1 Spectrum of policy refinement methodologies

conflict detection and resolution techniques. The second and most important contribution is related to the $\mathcal{PS}$ policy simulator which is intended for use in the off-line dynamic analysis of policy solutions for anomaly detection/resolution and policy efficiency enhancement.

Policy refinement techniques can be classified based on the formalism employed, the degree of automation offered, as well as the level of dependence to domain-specific solutions. Figure 4.1 shows how these are related to each other.

4.1.1 Manual refinement

Policy-driven goal refinement is a new systems engineering paradigm. Similar to any paradigm of elaborating solutions from requirements, there is a need to acquire some degree of maturity in understanding the policy derivation process.

Listing 17 summarizes an example adapted from [77] and shows a simple manual refinement case. The next chapter also shows a consistent manual refinement process of an application hosting SLA.

4.1.2 Static rules

Transformation using static rules is the simplest type of transformation [76]. It assumes the pre-existence of a set of static transformation rules for converting high-level policies into low-level operational policies. These rules are expected to have been defined by an expert user who knows the details of the system and the definitions of the various objectives, such as what it means to

Listing 17 A Manual policy derivation example

1. **Requirement**:
   provide an IP telephone to a user who logs in from a location outside their home office

2. The requirement looks like:
   provide *service* ***to*** *user* ***on*** *event* ***where*** *condition*
with,
   a- service=IP telephone
   b- event=user logs in
   c- condition=from a location outside their home office

3. The policy to consider is then an obligation policy (section 3.1.5) of the form
   on event **do** action **where** condition

4. Subsequent steps will then require the translation of the expressions in a,b, and c into their system-level equivalents.

provide gold service in terms of performance and security. The transformation module simply transforms the objectives to low-level configuration parameters using the definitions specified by the transformation rules. Listing 18, adapted from [76], provides an example of the usage of such rules.

Listing 18 Refinement through static rules example

High Level policy

if the user is from Schwab domain **then** provide Gold level service

Low Level Policy

if *user* $\in$ subnet 9.10.3.0/24 **then**
reserveBandwidth(*user*, 20 Mbps) $\wedge$ provideEncryption(*user*, 128 bits)
end if

Transformation Rules

- Schwab users $\in$ 9.10.3.0/24 subnet
- Gold service is to provide a bandwidth of 20 Mbps and an encryption of 128 bits

4.1.3 Table lookup

This technique [5, 76, 79] has been proposed for use with network policies of the form:

```
policy p = (SrcIP, DstIP, SrcPort, DstPort, Protocol, ServiceLevel (kbps))
```

SrcIP, DstIP, SrcPort, DstPort, and Protocol can be a single values as well as a range of values. A policy represents a goal to achieve by the network.

Example policies include:

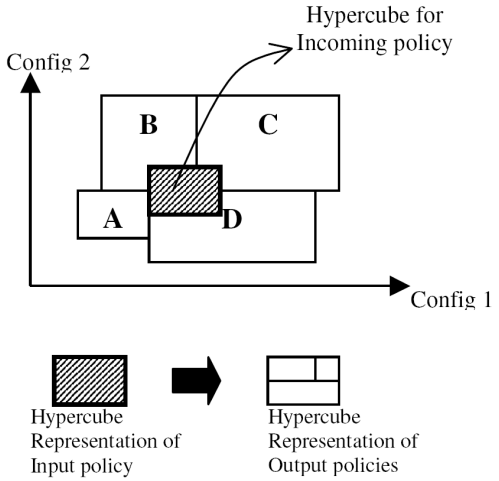


Figure 4.2 Table lookup refinement

policy p = (1.4.0.0, 1.3.0.0, ANY, ANY, TCP, 20kbps)

Guarantee 20kbps for all TCP traffic between 1.4.0.0 and 1.3.0.0

policy p = (ANY, ANY, ANY, ANY, ANY, 10kbps)

Guarantee a minimal bandwidth of 10kbps for all links

This technique assumes that the policy refinement module holds a table of policies that are appropriate for the system. The system administrator queries the module with a set of configuration parameters in order to obtain a set of goals that can be achieved given the input. In order to perform the refinement, each policy in the table needs to be first mapped into an N-dimensional hyperspace object, where N or the dimension of the space is the number of configuration parameters in the policy. Each policy is considered to be a region in the hyperspace, which may be connected or disconnected. Each hyper-cube points to a set of goals or actions. The system needs to match the hypercube corresponding to the policy being queried against all the hyper-cubes representing all the policies in the table. It finds the hyper-cube from the table that fully contains the specified hyper-cube.

The specified hyper-cube might not be fully contained in any single hyper-cube from the table. In other words, it might overlap several hyper-cubes, in which case the incoming hyper-cube needs to be split into smaller hyper-cubes where each new hyper-cube now corresponds to a different set of goals. In order to make sure we find a match for all segments of the incoming hyper-cube, we need to perform a coverage check on the policy table.

The coverage checker is based on the ability to subtract two regions (i.e. perform group difference): A - B = x — x in A but not in B where A and B are two regions and A-B denotes the difference between A and B. Coverage checking is performed by subtracting from the region of interest all the regions defined by the group of policies. Once the region of interest becomes empty, then

coverage is deemed complete. If, after iterating through all the policies the region of interest is not empty, then the coverage is deemed incomplete.

Figure 4.2 shows a simple 2-dimensional hyperspace with 4 policies shown as A, B, C, and D. The policy being queried is shown with a dashed pattern and is shown to overlap policies B, C and D.

4.1.4 Predefined templates

The POWER prototype [51] from HP Labs defines a policy refinement GUI environment that uses templates expressed in PROLOG. Policy templates, such as the one in listing 19, are manually created by a policy expert. In addition, an interpreter is provided to manipulate policy templates according to the embedded information, provides support to the POWER graphical user interface, and guides users through the refinement process.

The use of PROLOG to specify policy templates helps in borrowing from the inference capabilities inherent to PROLOG. However, the policy template definition as described in the POWER prototype is still at a preliminary level, only a manual use case is provided and there is a need to investigate in more depth the template selection and refinement process.

4.1.5 Case-based reasoning

Case-based reasoning (CBR) [80], broadly construed, is the process of solving new problems based on the solutions of similar past problems. An auto mechanic who fixes an engine by recalling another car that exhibited similar symptoms is using case-based reasoning. Case-based reasoning has been formalized for purposes of computer reasoning as a four-step process [81]:

Retrieve Given a target problem, retrieve cases from memory that are relevant to solving it.

Reuse Map the solution from the previous case to the target problem.

Revise Test the new solution in the real world (or a simulation) and, if necessary, revise.

Retain After the solution has been successfully adapted to the target problem, store the resulting experience as a new case in memory.

The case-based reasoning approach to policy management has been introduced in [76]. In this approach, the system learns experientially from the operational behavior it has seen in the past. The system maintains a database of past cases, where each case is a combination of the system configuration parameters and the business objectives that are achieved by the specific combination of the system configuration parameters. When the configuration parameters needed for a new business objective are required, the case database is consulted to find the closest matching case, or

Listing 19 A POWER Policy template

```
template(t3,
 [
 [ c0, keywords, [$engineer$, $information$, $organisation$ ]],
 [ c1, category, $Access to Information$],
 [ c2, abstract, $All engineers can perform operations
       on information within their organisation$],
 [ c3, description, $Users that are Engineers can perform operations
       on information that relate to the same organisation they belong to].$],
 [ c4, expiration-date, $01/01/1999$],
 [ c5, deployable, $deployable$],
 [ c6, start, c7],
 [ c7, sequence, [c8, c12, c13, c16, c18]],
 [ c8, context, [internal: [and([belongsTo(information,orgUnit(U)),
       isMember(user(Un,UId), engineer),
       isMember(user(Un,UId), orgUnit(U))])],
       refinementBy: [[information,c10],
       [orgUnit(U),c10]]]],
 [ c10, refinementDetails, [category: ism,
        condition: [],
        refinementBy: [class]]],
 [ c12, policyStatement, [category: deployable,
        internal: [and([canAccess(user(Un,UId), operation, information)])],
        condition: [],
        refinementBy: [[user(Un,UId),c10],
        [information,c10]]]],
 [ c13, classRefinementChoice, [class: [orgUnit(U),c10]]],
 [ c16, constraintChoice, [constraint:[and([about(information,user(Un,UId))])],
choices:[accept:c18, ignore:c18]]],
 [ c18, end, []] ]).
```

an interpolation is performed between the configuration parameters of a set of cases to determine the appropriate configuration parameters that will result in the desired business objective.

Table 4.1 [76] shows an example of a case table in a 2-tiered web server measuring the user response times as a function of the number of disks and nodes in tiers. The table shows a one dimensional mapping of four configuration parameters to one service objective, which is the response time. Based on this information, it is possible to predict the minimal system configuration is capable of assuring some input response time objective.

In general, the case database contains multi-dimensional relationships between configuration parameters and performance (goal) values. Several steps are then required in order to derive the best rules for deciding on which configuration set best fulfills a given set of system performance objectives. There is a need to employ several techniques in order to enhance the consistency of the case database, reducing the dimensionality of data sets and normalizing them, in addition to determining an appropriate search algorithm that helps in finding the required system configuration that corresponds to some input performance objectives.

Table 4.1 Sample of a case database of the performance of a 2-tiered web site

Tier configuration				User Response Time
Number of Disks		Number of Nodes		
Tier 0	Tier 1	Tier 0	Tier 1	
1	4	1	2	0.039 sec
3	2	2	4	0.029 sec
2	3	2	4	0.082 sec
4	1	1	2	0.015 sec
2	1	3	3	0.042 sec
2	4	1	2	0.053 sec
1	2	2	4	0.032 sec
3	2	3	4	0.098 sec

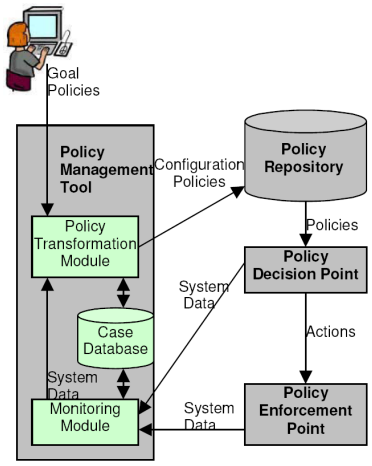


Figure 4.3 Policy based management architecture supported by CBR and real-time policy adaptation

The case-based approach to policy refinement suffers from the traditional weaknesses of CBR systems. These systems have a difficulty at bootstrap time where there is a need to populate the case database. Accepting error is also part of this approach as the generalizations made out of a set of cases can be wrong.

The main advantage of a CBR system is that it becomes increasingly effective as its case database grows to an acceptable size. For this, Beigi et al. [76] argue that a policy-based management architecture that uses CBR needs to employ a real-time policy transformation mechanism. In this architecture (figure 4.3), the policy generation module uses new knowledge gained at the case database in order to update the policy repository with more accurate policies.

4.1.6 Goal elaboration and the KAOS methodology

The goal elaboration approach to policy refinement has received more interest recently [19, 20, 74, 82]. The approach is based on the KOAS goal refinement methodology proposed by Darimont et al. [83–87]. This section describes the KAOS refinement methodology. The next two sections

Table 4.2 Temporal logic operators

P	P holds in *the current* state		
$\circ$ P	P holds in *the next* state	$\bullet$ P	P holds in *the previous* state
$\diamond$ P	P holds in *current or some future* state	$\blacklozenge$ P	P holds in *current or some previous* state
$\Box$ P	P holds in *current and all future* states	$\blacksquare$ P	P holds in *current and all previous* states
P $\mathcal{W}$ Q	P holds unless Q holds	P $\mathcal{U}$ Q	P holds unless Q holds

describe the recent efforts on goal-based policy refinement.

KAOS (named after the the project project "Knowledge Acquisition in autOmated Specification" [87]) is a formal approach to goal refinement and operationalization which is aimed at providing constructive formal support while hiding the underlying mathematics. The principle is to reuse generic refinement patterns from a library structured according to strengthening/weakening relationships among patterns. Once the patterns are proved correct and complete, they can be used for guiding the refinement process or for pointing out missing elements in a refinement. Tactics are proposed to the requirements engineer for grounding pattern selection on semantic criteria [84].

KAOS provides a multi-paradigm specification language and a goal-directed elaboration method. The language combines semantic nets [88] for the conceptual modeling of goals, constraints, agents, objects and operations in the system; temporal logic [89] (**??** 4.2) for the specification of goals, constraints and objects; and state-based specifications [90] for the specification of operations.

The KAOS Language

The specification language provides constructs for capturing various kinds of concepts that appear during requirements elaboration, namely, goals, constraints, agents, entities, relationships, events, actions, views, and scenarios. There is one construct for each type of concept.

Listing 20 provides an example of a goal specification in KAOS. The FormalDef structure defines the goal using temporal logic. The other informal structures are provided for user-friendly assistance.

Listing 20 Sample KAOS *Achieve goal*

```
Goal Achieve [ParticipantsConstraintsKnown]
  InstanceOf InformationGoal
  Concerns Meeting, Participant, Scheduler, ...
  RefinedTo ConstraintRequested, ConstraintProvided
  InformalDef A meeting scheduler should know the constraints of the various
participants invited to the meeting within C days after appointment
  FormalDef
     ∀ m:  Meeting, p:  Participant, s:  Scheduler
     Invited (p, m) ∧ Scheduling (s, m) ⇒ ◇≤Cdays Knows (s, p.Constraints)
```

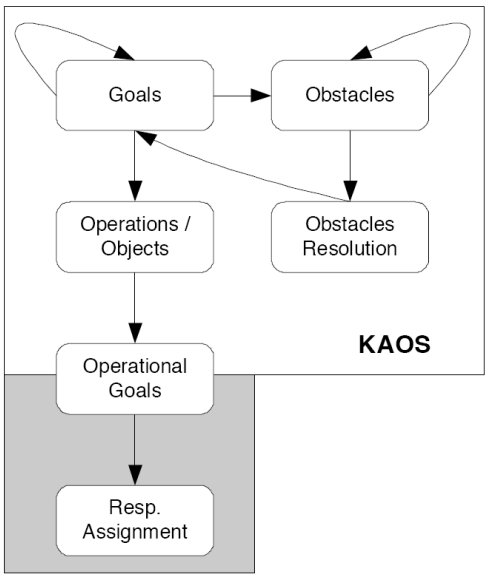


Figure 4.4 KAOS goal elaboration process

Goals can be of the form *Achieve* ($P \Rightarrow \Diamond Q$), *Maintain* ($P \Rightarrow \Box Q$), *Cease* ($P \Rightarrow \Diamond \neg Q$), or *Avoid* ($P \Rightarrow \Box \neg Q$). *Achieve* and *Cease* goals obey to system behaviors that require some target property to be eventually satisfied or denied respectively. *Maintain* and *Avoid* goals, on the other hand, restrict behaviors in that they require some target property to be permanently satisfied or denied respectively.

Goal elaboration

Goal elaboration in KAOS (figure 4.4 [91]) is based on the refinement of AND/OR structures by defining goals and their refinement/conflict links until implementable constraints are reached; offspring goals are identified by asking HOW questions whereas parent goals are identified by asking WHY questions. In addition, *obstacles*, which are negated goals, may also be used in the goal elaboration process as it is sometimes useful to reason about "what not to do goals".

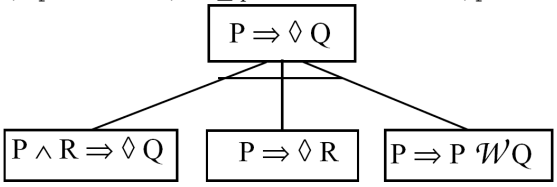
Refinement patterns

The idea behind the definition of refinement patterns is to provide formal support for building goal *refinement graphs* that are *complete*, proved *correct*, and integrate *alternatives*.

Formally, a refinement pattern is a one-level AND-tree of abstract goal assertions such that the set of leaf assertions is a *complete refinement* of the root assertion. A set of goal assertions G1, G2,..., Gn (n>1) is a complete refinement of a goal assertion G iff it is *consistent*, *minimal*, and *logically* entails the original goal G.

Listing 21 Refinement of an Achieve sequential installation progress goal

Goal Achieve [InstallationProgress]
FormalDef
(∀ i: installation, p: Phase, $0 \leq p < N$) [∀ At(i,p) ⇒ At(i,p+1)]



The refinement derives a progress mode wherein an OK signal has to be triggered each time there is a need to continue to the next installation phase. This can be implemented in a variety of ways, such as using a pop-up dialog box which invites the user to confirm the beginning of the next installation phase. Another refinement possibility can be that the dialog box offers a limited time, say of one minute, before making a default move to the next phase.

Goal Achieve [ProgressAfterOk]
FormalDef
(∀ i: installation, p: Phase, $0 \leq p < N$) [$At(i,p) \wedge Ok(i,p+1) \Rightarrow At(i,p+1)$]

Goal Achieve [StartNextPhaseOk]
FormalDef
(∀ i: installation, p: Phase, $0 \leq p < N$) [$At(i,p) \Rightarrow \Diamond Ok(i,p+1)$]

Goal Achieve [ProgressAfterOk]
FormalDef
(∀ i: installation, p: Phase, $0 \leq p < N$) [$At(i,p) \Rightarrow At(i,p+1) \mathcal{W} At(i,p+1)$

Domain-independent refinement patterns have been developed and included as part of the KAOS library of refinement patterns. Temporal logical proofs are generally complex even for simple proofs. However, whenever a formal domain-independent refinement pattern has been proven true, it is stored in the library. When needed, that pattern is reused without requiring the user to prove the correctness of the refinement. Another equally useful way of using the library of refinement patterns is to assist the user in completing manual refinements by comparing them to the templates, detecting what is missing, and suggesting more complete refinements. This latter case is particularly useful as practice shows that manual refinement patterns tend to be incomplete [85].

Table 4.3 Some propositional refinement patterns for *Achieve* goals ($P \Rightarrow \Diamond Q$)

	Subgoal	**Subgoal**	**Subgoal**
RP3	P ⇒ ◊R	R ⇒ ◊Q	
RP4	P ∧ P1⇒ ◊ Q1	P ∧ P2 ⇒ ◊ Q2	❑ (P1 ∨ P2) Q1 ∨ Q2 ⇒ Q
RP5	P∧¬ R ⇒ ◊R	P ∧ R ⇒ ◊Q	P ⇒ ❑ P
RP6	¬ R ⇒ ◊R	P ∧ R ⇒ ◊Q	P ⇒ ❑ P
RP7	P ⇒ ◊R	R ⇒ R *U* Q	

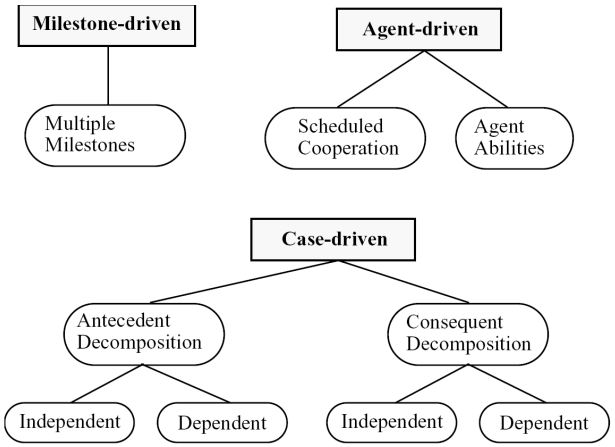


Figure 4.5 Some well-known domain-independent strategies

Table 4.3 shows a set of proven domain-independent propositional refinement patterns for Achieve goals. Listing 21 provides an example of the refinement related to achieving progress in the installation of a software package.

Refinement tactics

In KAOS, refinement *tactics* are proposed to the requirements engineer for grounding pattern selection on semantic criteria. Tactics capture heuristics to drive the elaboration or select among alternatives. As is the case for refinement patterns, there are domain-independent as well as domain-dependent strategies. Figure 4.5 [84] shows a set of domain-independent strategies.

During the refinement process, it often happens that the initial goals are discovered to be too ideal to be realized by the underlying information system. Such goals need to be de-idealized in order to make them implementable. This step is difficult to automate as it requires changing the original goal.

Goal operationalization

The refinement process can be supported at a first stage by the reuse of non-operational patterns; that is, patterns which do not refer to operational notions such as agents, events, actions, etc. In a second stage, *operational patterns* must be considered.

Given a goal and a set of possible agents, there is a need to determine whether the goal could be enforced by one of them through appropriate control over state transitions. If this is the case the goal becomes an *assignable constraint*; otherwise it must be reduced further.

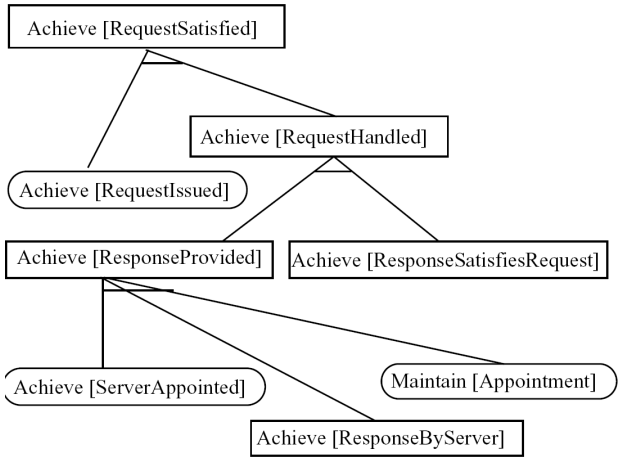


Figure 4.6 A Stimulus-response refinement pattern for a Satisfaction goal

Goal operationalization often requires a change in level of abstraction. Abstract concepts involved in goal formulations need to be mapped to concrete ones to make it possible to formulate constraints/actions assignable to agents [85]. This mapping is often hard to do as it corresponds to the choice of good representation functions to map abstract objects to concrete variables.

In order to complete goal operationalization, [84] suggests the use of stimulus-response patterns. Recent efforts have investigated other solutions to goal operationalization. These are described in sections 4.1.7–4.1.9.

Stimulus response Patterns

Stimulus-response patterns have been proposed as one way to derive operational refinement patterns. A *stimulus* is an event perceived by some agent which requires some action to be performed by the agent. A *response* is an agent reaction to some stimulus. Figure 4.6 shows an AND-tree of goals and constraints proposed by stimulus-response patterns for a Satisfaction goal. In this figure, a stimulus represents a request for service while a response indicates that the requested service has been provided. The pattern is formally proven [92] and hence can be directly reused.

The KAOS approach, however, is limited in the assignment decision step. It provides little support for formal reasoning about alternative assignments, which is quite important issue. The next three sections elaborate on this issue through the use of abduction, model checking techniques, and translation primitives respectively.

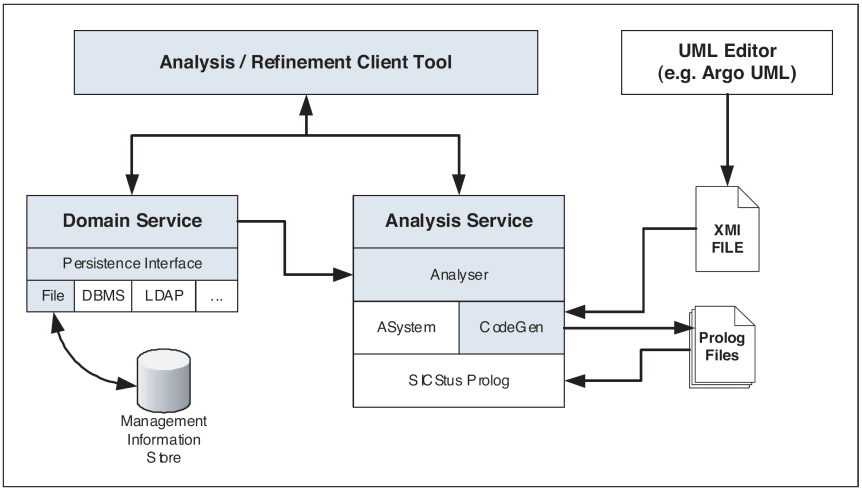


Figure 4.7 Policy refinement tool

4.1.7 Goal elaboration using Event-calculus and abductive reasoning

Bandara et al. presented in [19, 82] an approach to policy refinement by which a formal representation of a system, based on the Event Calculus [93], is used in conjunction with *abductive reasoning* [94] techniques to derive the sequence of operations that will allow a given system to achieve a desired goal. The refinement relies on using the KAOS methodology first for the process of refining the abstract goals into lower-level ones.

At a given level of abstraction there will be some description of the system (SD) and the goals (G) to be achieved. The relationship between the system description and the goals is called a *Strategy* (S) [19]. It describes the mechanism by which the system represented by SD achieves the goals denoted by G. Formally this would be stated as: (SD, S $\vdash$) G.

Statecharts are used to describe system behavior. A State transition indicates the invocation of an operation and/or the occurrence of a system event. Guards are specified for transitions with pre-conditions for invoking the operation.

Abductive reasoning is a form of reasoning that naturally arises in the context of Declarative Problem Solving. In Declarative Problem Solving, the expert formulates his domain knowledge of the problem as a logic theory. A solution of the problem is then the outcome of reasoning process that uses this logic theory as input. For many problems, the reasoning task requires to compute an interpretation of a relation so that the query is entailed by the logic theory augmented with this interpretation. This inference is called abduction [95].

Given the rules describing a system (SD) and the definition of some desired system state (i.e., the goal — G), abductive reasoning allows to derive the *facts* that must be true for the desired system state to be achieved. As the goal is represented by a desired system state the abductive

reasoning process is essentially *deriving a path in the statechart from some initial state* to the desired one. Whether a strategy should be encoded as policy, or as system functionality, will depend on the particular application domain.

Event Calculus is expressed in First Order Logic and as such supports the deductive, inductive, and abductive modes of reasoning. *Deduction* helps in deriving *fluents* from the system description (SD) along with a given history of events. If we think of the system states as a state chart, a fluent represents a system state, that is a property that can hold at a particular point during the system lifetime. *Induction* is related to the reverse process of deriving the high-level system behavior SD from a set of system traces (event history combined with fluents). Finally, *Abduction* is related to finding the sequence of events that need to occur in order for a given set of fluents to hold. The set of fluents represent the goal to achieve.

Listing 22 Syntax of a goal elaboration strategy

```
Strategy AchievedGoal
   OnEvent Events derived from transitions with system events.
   DerivedActions Actions derived from transitions with operations.
   Constraints Constraints derived from guards.
```

Users first provide the high-level policy they are interested in the form "**on** event, **if** condition **then** *achieve* goal". The KAOS approach is applied to elaborate the high-level policy, making use of both application-independent and application- specific refinement patterns. At each stage of elaboration, the system description and the goals are used to attempt to *abduce* a strategy for achieving the goal. If no strategy can be derived, then the preferred course of action is to further elaborate the goals. However, if the existing low-level goals are already expressed at the lowest level of abstraction in the system, it is not possible to elaborate the goals further. In this situation the system description must be augmented with more detail. This involves specifying additional management operations for the system, either as custom-written scripts or using functionality of commercial management platforms. The post-conditions of these new operations should match the goals for which a strategy is required.

Abductive reasoning is provided by the $\mathcal{A}$system abductive proof engine [94] which runs within the SICStus Prolog environment. The A-System is an abductive constraint logic solver for the knowledge representation language ID-Logic. Procedurally, the $\mathcal{A}$system is a mixture of the SLDNFA, IFF and ACLP abductive logic procedures. It uses existing sub-solvers to perform a part of the reasoning. The procedure is sound and complete with regard to the three-valued completion semantics [95].

The main contribution of Bandara et al. [19] is the development of a prototype policy refinement tool that can handle the specification of policies, goals, domain hierarchy and the generation of analysis results. The tool features:

- The possibility to import XMI (XML Meta-data Interchange (XMI) format of UML) specifications into the policy analysis and refinement tool.
- A *domain service* [42] that provides functionality for storing and retrieving information that describes the entire managed system.
- An *analysis service* deals with the requirements of translating the high-level representations of the policy-based management system into the underlying formalism and generating the analysis and refinement results. To do this the analysis service is integrated with the SICStus Prolog system which provides deductive reasoning capabilities and the $\mathcal{A}$system abductive proof engine.
- Ability to translate the high-level representations of the domain hierarchy, managed object behaviors, policies and goals through the use of XML style sheet transformations (XSLT).
- An *Analysis and Refinement Client* which implements the user interface and provides a view that is based on the Ponder hyperbolic domain browser [68]. It also provides a context sensitive properties pane to view the detailed information regarding any particular entity in the domain hierarchy.
- language support that extends the Ponder language with constructs for the specification of goals, strategies, refinement patterns.

Example: Adapting to traffic increase

The following example [19] shows how this approach is used to derive policies that adapt to traffic increases in a DiffServ network. The goal (high-level policy) is informally stated as

> *When network utilization exceeds 85 percent of the maximum allocation, and the time is between 11am and 1pm, the bandwidth allocation should be increased by 10 percent and spare capacity should be equally split amongst the PHBs.*

Formally, the goal is specified as:

```
G6:  goal ConfigAdaptedForBWUtilIncrease
   FormalDef alarmRaised(bwUtilIncr, [utilValue, PHB]) ⇒ à configAdapted.
```

Abducting G6 yields no strategy, which implies the need to refine it further using the KAOS methodology. The refinement process generates the graph of figure 4.8. Abduction on the leaf goals of this figure leads to the strategy S5 which seeks to fulfill the original goal by fulfilling sub-goals g11 and g15 (listing 23). Finally, a *manual encoding* of this strategy into the Ponder obligation P3 (listing 23) is done. Note that the policy includes the original time constraint time.between("11:00", "13:00") which was not considered in the refinement process.

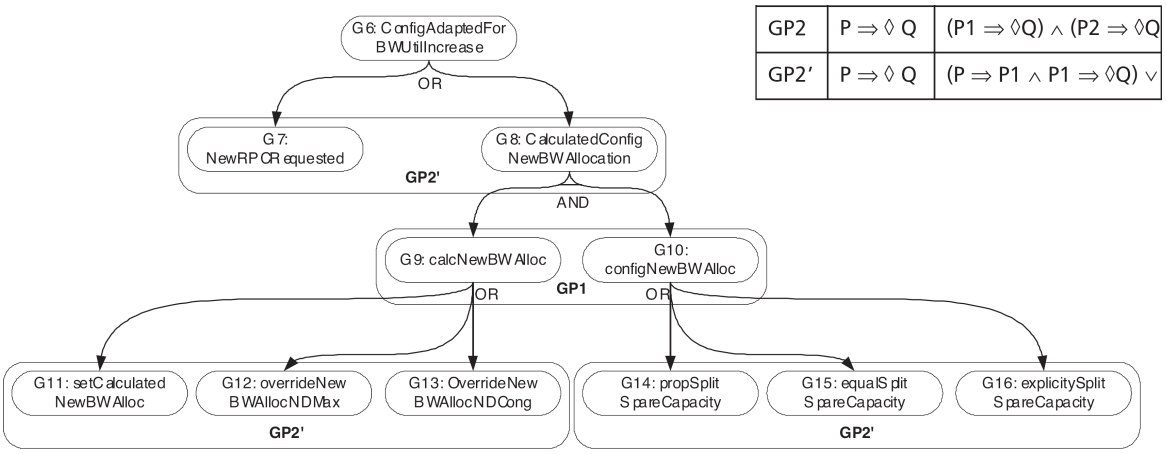


Figure 4.8 KAOS decomposition of the traffic adaptation goal

4.1.8 Goal elaboration using Model checking

Instead of using an abductive reasoning engine, Rubio-loyola et al. [96] investigated the usage of model checking [97] in order to derive operational policies from low-level goals. The refinement framework as presented in figure 4.9 relies also on the KAOS methodology, the use of temporal logic, and the modelling of system behavior in terms of event-based labeled transitions (state charts).

Model checking is a technique of system verification that relies on the counterexample principle:

> In order to prove that property P can hold ($\Leftrightarrow$ P is a fluent), it is sufficient to find a counterexample to the property that states that P can never happen, i.e. $\Box!P$ (Always not P).

The counterexample produced by a model checker not only proves that P can hold but also provides a way to actually achieve it, which in turn is nothing but the low-level policy that needs to be encoded in order to achieve the original goal.

For example, consider again the goal elaboration graph of figure 4.8. A possible realization of this goal is to achieve (G11 $\Rightarrow \Diamond$ G15). So the model checker is set to find a counterexample for $\Box\neg$(G11 $\Rightarrow \Diamond$ G15) $\Leftrightarrow \Box$(G11 $\Rightarrow \Box\neg$ G15).

Since in a goal refinement graph, the original goal can potentially be achieved in several ways (for example (G12 $\Rightarrow \Diamond$ G14) for figure 4.8), the model checker is given one single temporal logic formula which is represents the union (OR operator) of all the potential goal fulfillment paths, that is:

$$\Box(G11 \Rightarrow \Box\neg G14) \parallel \Box(G11 \Rightarrow \Box\neg G15) \parallel \Box(G11 \Rightarrow \Box\neg G15) \parallel \cdots \parallel \Box(G13 \Rightarrow \Box\neg G16) \parallel \Box\neg G7$$

Listing 23 Refinement result of the traffic increase goal

```
G7:  goal NewRPCRequested
  FormalDef alarmRaised(bwUtilIncr, [utilValue, PHB])
    ⇒ requestedNewRPC ∧ requestedNewRPC
    ⇒ ◊ configAdapted.
G8:  goal CalculatedConfigNewBWAllocation
  FormalDef alarmRaised(bwUtilIncr, [utilValue, PHB])
    ⇒ calcAndConfigNewBWAlloc ∧ calcAndConfigNewBWAlloc
    ⇒ ◊ configAdapted.
G9:  goal calcNewBWAlloc
  FormalDef calcNewBWAlloc(newValue)
    ⇒ ◊ configNewBWAlloc.
G10:  goal configNewBWAlloc
  FormalDef configNewBWAlloc
    ⇒ ◊ configAdapted.
G11:  goal setCalculatedNewBWAlloc
  FormalDef calcNewBWAlloc (newValue)
    ⇒ (newValue = calcValue) ∧ (newValue = calcValue)
    ⇒ ◊ configNewBWAlloc.
G12:  goal overrideNewBWAllocNDMax
  FormalDef calcNewBWAlloc (newValue)
    ⇒ (newValue = drsm.ndMaxBWAlloc) ∧ (newValue =
drsm.ndMaxBWAlloc)
    ⇒ ◊ configNewBWAlloc.
G13:  goal overrideNewBWAllocNDCong
  FormalDef calcNewBWAlloc (newValue)
    ⇒ (newValue = drsm.  ndCongBWAlloc) ∧ (newValue =
drsm.ndCongBWAlloc)
    ⇒ ◊ configNewBWAlloc.
G14:  goal propSplitSpareCapacity
  FormalDef configNewBWAlloc
    ⇒ spareCapProportionallySplit ∧ spareCapProportionallySplit
    ⇒ ◊ configAdapted.
G15:  goal equalSplitSpareCapacity
  FormalDef configNewBWAlloc
    ⇒ spareCapEquallySplit ∧ spareCapEquallySplit
    ⇒ ◊ configAdapted.
G16:  goal explicitySplitSpareCapacity
  FormalDef configNewBWAlloc
    ⇒ spareCapExplicitlySplit([splitValues]) ∧
spareCapExplicitlySplit([splitValues])
    ⇒ ◊ configAdapted.
S5:  Strategy G11:  setCalculatedNewBWAlloc && G15:
equalSplitSpareCapacity
  OnEvent alarmRaised(bwUtilIncr, [utilValue, PHB])
  DerivedActions calcValue = drsm.incrAllocBW(PHB,pct) − >
    drsm.configureLink(PHB, calcValue) − >
    drsm.splitSpareCapEqually
  Constraints drsm.incrAllocBW(PHB, pct) <
drsm.ndMaxBWAlloc(PHB).

P3:  inst oblig /policies/adaptTrafficIncreaseAOLSLA_P1
  on alarmRaised(bwUtilIncr, [utilValue, ef]);
  subj s = /routers/FromR1/ToR6/drsmPMAs/;
  targ t = s.drsm;
  do calcValue = t.incrAllocBW(ef, 10) − >
    t.configureLink(ef, calcValue) − >
    t.splitSpareCapEqually;
  when (t.incrAllocBW(ef, 10) < t.ndMaxBWAlloc(ef)) &&
    time.between(``11:00'', ``13:00'');
```

The presented framework uses the SPIN [98,99] model checker and the Promela [99] language for the specification of linear temporal logic formulae. SPIN offers also the possibility to simulate the execution of a counterexample(s) once found.

4.1.9 Goal elaboration using translation primitives

In [74], Rubio-loyola et al. provide a functional approach which extends their KAOS and model-checking-based refinement framework with the use of *design patterns for finite-state verification* [100] and *translation patterns* [101].

The key idea is to consider the constraining of the system to a particular behavior, from among the set of potential behaviors, based on a set of systematically identified refinement patterns. For example, if an achieve goal G1:(P ⇒ ◊Q) is refined using the milestone pattern to (G11:P⇒ ◊R ∧ G12:R⇒ ◊Q), the subsequent decision is to constrain the system behavior to the new temporal relationship [74] "goal G11 *must be fulfilled before* goal G12".

While the refinement patterns used to elaborate goals describe the requirements for a system (e.g. both G11 and G12 must be fulfilled so that G1 is fulfilled), the patterns described in this approach deal with the translation of particular aspects of such requirements (e.g. G11 is achieved *before/after* G12) into formal specifications suitable for finite state verification tools, such as model checking.

Figure 4.10(a) [100] shows a classification of the design patterns for finite-state verification into Occurrence, Order, and compound patterns. *Occurrence* patterns are used to represent

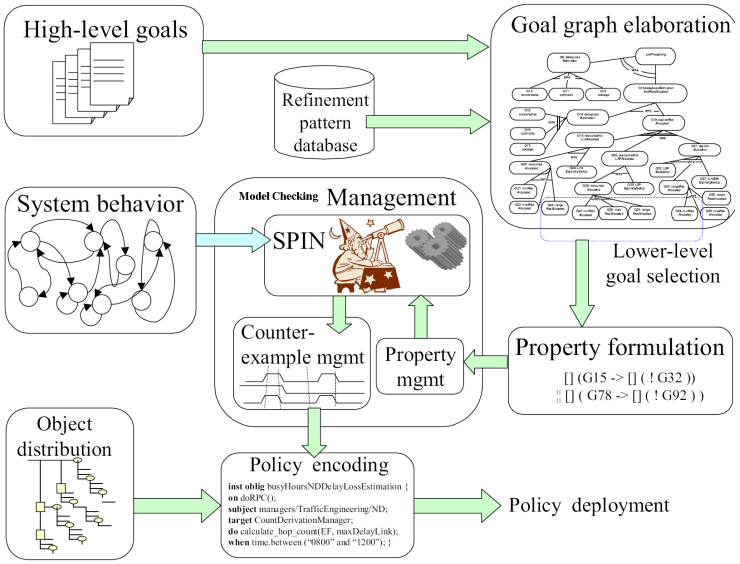


Figure 4.9 KAOS and Model checking-based refinement framework

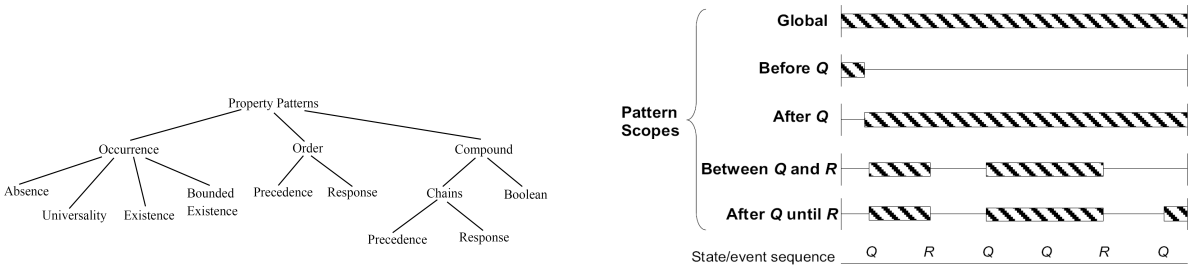
states/events to occur or not to occur (*Existence* and *Absence* patterns), states/events to occur throughout a scope (*Universality* pattern), or a state/event that occurs k times within a scope (*Bounded Existence* pattern). *Order* patterns represent constraints on the order of states/events (*Response* pattern), or specify that a given state/event P has to be always preceded by a state/event Q within a scope (*Precedence* pattern). *Compound* patterns are built up from the combination of the basic occurrence and order patterns.

Each pattern has an equivalent representation in linear temporal logic. For example, to specify that G12 (antecedent) must exist after G11 (consequence), which is a combination of an *existence* pattern with an *after* scope, the corresponding temporal formulae would be: $(\Box\neg G11 \mid \Diamond(G11 \wedge \Diamond G12))$.

Once the KAOS goal graph is elaborated and the temporal relationships between sub-goals identified, the refinement engine makes use of *translation primitives* in order to abstract policies from system trace executions. These primitives take advantage of the fact that the system trace executions indicate the pre- and post-conditions, and the actions taken by the involved managed objects.

Translation primitives specialize the *translation patterns* approach [101], which abstracts policy fields from system process specification, to generate policies automatically from system trace executions.

The first step towards the application of the translation primitives is the identification of transition plans. A *transition plan* is a sub-section of a system trace execution consisting of the following elements [74]:



(a) A Pattern Hierarchy

(b) Pattern scopes

Figure 4.10 Pattern hierarchy and scopes

- A pre-condition in a managed entity S (PSi).
- A state transition TSi,Si+1 in the managed entity S.
- A state transition TQi,Qi+1 in the managed entity Q as a result of transition TSi,Si+1.

Given that S and Q are two different managed objects, the transition plan prescribes that on the occurrence of PSi in the managed object S, preceding the transition TSi,Si+1, the managed object Q must enforce the transition TQi,Qi+1. Considering that the transition TQi,Qi+1 is policy-controlled (i.e. policy-enforceable), the Transition Primitives shown in the right part of Figure 3 enable us to encode the above information into obligation policies (section 3.1.5) in a systematic manner.

Having reviewed the current research in policy refinement methodologies, the next sections will present the currently known approaches to policy analysis.

4.2 Policy inconsistencies

Once a policy set is generated after a refinement process, whether be it manual or automated, it is not sufficient to merely check the syntax of new policies before they are deployed within the system. If conflicting policies are deployed within a system, abnormal behavior will occur and the expected damage is generally difficult to measure. In fact, having a system deployed with inconsistent policies is probably worse than having it without policies at all, as flexibility cannot come ahead of correctness. Thus, policies need to be analyzed for their interactions with each other and with the environment they are going to be deployed into.

Inconsistencies among policies need be *avoided* at design time as much as possible, and if not then conflicts and other abnormal behavior need to be *detected* and *resolved*. Each policy needs also to be *validated* for compatibility towards the environment it is expected to act on.

In the literature, policy inconsistencies are often referred to as "*policy conflict*". This is probably due to the fact that early work in this field was focused on conflict detection and resolution mainly in security policies [102]. However, many types of abnormal policy behavior cannot be described accurately as conflicts. For example, there is no conflict in the situation where the introduction of a new policy, say P1, makes the actions of an existing policy, say P2, completely covered by another set of existing policies augmented by P1. This is a type of inconsistency where the policy P2 is said to be dominated [5] (or shadowed [103]). We therefore prefer the use of the more generic term *Policy Inconsistency*.

Policy inconsistencies [104] can be of two kinds: (i) logical inconsistencies and (ii) statements that can be proved by the theorem prover tool but do not comply with the intended specification.

Lamsweerde et al. present in [105] a comprehensive work on conflict detection and resolution in goal-driven requirements engineering. They extend the KAOS methodology and present a formal solution to a weak form of conflicts called divergence conflicts. However, only a set of heuristics are given in order to deal with conflict detection and resolution in the general case. Conflicts can occur at the very first stage of goal definition as the system administrator might want to refine a goal that is inconsistent with other already enforced goals. This is more likely to occur in large information systems where different system administrators formulate and enforce goals at different locations and/or times.

The specification formalisms used to express policies have an important role in *inconsistency avoidance*. From the study of policy specification formalisms given in section 3.1, it can be deduced that the use of high-level constructs, such as attribute constraints, meta policies (Ponder [104]), action constraints (PDL [106]), roles, domains, domain and policy compatibility rules [35], and other policy structuring techniques help significantly in limiting the spectrum where inconsistencies can occur; as well as the extent of damage that potentially undetected inconsistencies may cause.

Policy inconsistencies can be broadly classified into two categories based on whether they are application-independent or application-dependent. *Application-independent* inconsistencies can be identified with the analysis of the policy specification independently of the semantics of the operations involved.

For example, consider two policies P1 and P2. P1 allocates a gold LSP (MPLS Label Switched Path) to incoming traffic of users of domain A. P2, on the other hand, allocates a best effort LSP to users of domain B. The actions of these two policies will result in a *semantic conflict* if, during system runtime, a user traffic happens to belong temporarily to both domains. In order to detect this type of conflict, there is a need to know the semantics of action and which actions are inconsistent with which. This gets easily complicated when it is not only a matter of

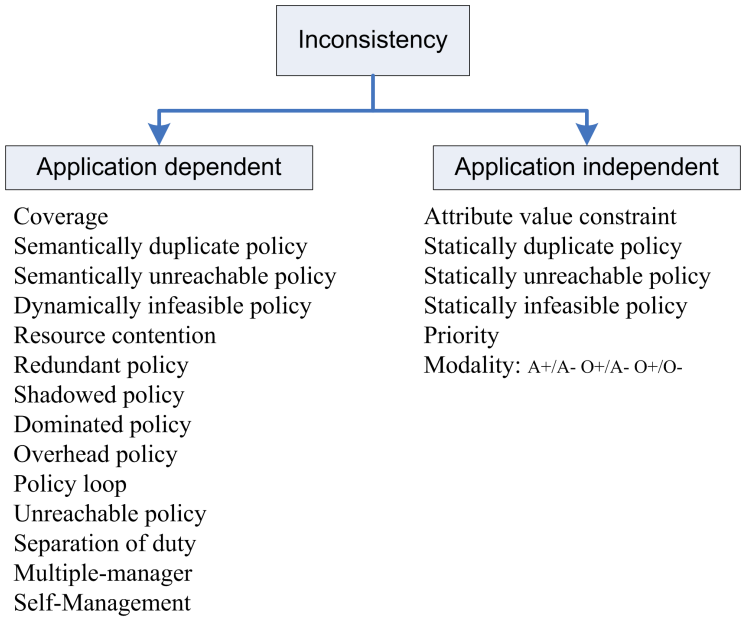


Figure 4.11 Classification of policy inconsistencies

comparing two actions but rather a sequence of events/actions originating from different system components/policies.

In the following, we list the different types of policy inconsistencies that have been identified so far in the literature (Fig. 4.11).

In order to keep the terminology consistent, we have adapted the names of some policy inconsistency types. Each time, a justification for the renaming will be given. In addition, we contribute in this list with four new types of policy inconsistencies, namely *statically* and *semantically duplicate* policy, *redundant* policy, *overhead* policy, and *policy loop* inconsistencies.

Application-independent inconsistencies

Attribute constraint inconsistency This is a type of inconsistency that is not specific to the policy discipline alone. It occurs when an attribute is assigned a value that outside the permitted range. This is referred to as bounds check in [107]. The inconsistency can be identified either at design or runtime depending on the complexity of the constraint and whether it relates to other attributes or not.

Modality conflicts [7] are a special type of application-independent policy inconsistencies that can be syntactically detected. Using Ponder terminology [108], a modality conflict occurs when two policies are specified using the same subjects, targets and actions but are of opposite modality. There are three types of such conflicts:

- *Obligation conflicts* (O+/O-): The subject is obliged to do and obliged not to do an action.
- *Unauthorized Obligation conflicts* (O+/A-): The subject is obliged to carry out an action it is not authorized to do.
- *Authorization conflicts* (A+/A-): The subject is both authorized and unauthorized to carry out an action.

No direct action can be taken at the detection of a modality conflict except to raise an exception to the system administrator to point towards a potential design error.

Duplicate policy inconsistencies.

We consider a policy to be a duplicate if it is *statically equivalent* to another policy in all terms. Syntactic equivalence is the simplest case of a static equivalence. Static equivalence means that the two policies have equivalent expressions defining their event, condition, subject, target, and action components.

For example consider the two policies:

policy p1 = **on** e **if** $((a \wedge b) \vee c)$ **then** {a1→a2}
policy p2 = **on** e **if** $((a \vee c) \wedge (b \vee c))$ **then** {a2→a1}

These two policies have the same event expression e, a logically equivalent condition expression, but different action sequences. They are thus syntactically different. However, if the compiler determines that the execution of actions a1 and a2 is not impacted by their order the two policies become then statically equivalent.

The action to take at the encounter of a duplicate policy is to remove it. Not doing so may result in the policy actions being executed twice each time the event e is triggered, which in turn may have undesirable effects.

In the case of two policies that are statically inequivalent but semantically equivalent, we refer to each one to be *semantically duplicate* of the other. For example, consider the policies:

policy p3 = **on** e **if** $((a \wedge b) \vee c)$ **then** {paintBlock(3)→paintBlock(1,to,2)}
policy p4 = **on** e **if** $((a \vee c) \wedge (b \vee c))$ **then** {paintBlock(1)→paintBlock(2,to,3)}

Policies p3 and p4 are clearly semantically equivalent although they are not statically equivalent.

The required action in the case of semantically duplicate policies is also to discard them. Although the agents associated with policies p3 and p4 still are able to execute concurrently (they never get to try to paint the same square at the same time, assuming the painting speed is the same), the result is a waste of system resources (and a longer time for the paint to dry too).

Priority inconsistencies occur when a set of two or more policies receive the same priority value while they are inconsistent with each other. Integer priorities [102, 107] are generally used to resolve conflicts at runtime by assigning a privilege to the policy with higher priority values.

However, manual priority assignment only works when the number of involved policies is small. Agrawal et al. [21,43] suggest a consistent priority assignment algorithm which automatically adjusts the integer priority of each policy based on a set of individual relative priorities.

The goal of the consistent priority assignment is to take relative preferences (or in other words, the priority graph) specified by the policy author, including those specified during dominance and conflict checks, and assign integer priorities to the policies so that the number of amortized reassigned priorities is minimized.

Inconsistency resolution with integer priority assignment remains however a limited mechanism as no cycles can be allowed in a priority precedence graph. Earlier in [7], Moffet mentions a case where precedence-based priority assignments are impossible and suggests the use of *meta-policies* as a more elaborate form of specifying non-static precedence between policies.

Infeasible policy inconsistencies occur when a defined policy cannot be executed in the system [5,77]. This can be related to an error in its specification. For example, a policy which tries to access some object properties which do not exist.

We differentiate between *statically-infeasible* and *dynamically-infeasible* policy inconsistencies. A Statically-unfeasible policy inconsistency can be detected at policy design time by querying the system components which the policy interacts with (subjects, targets, and methods invoked on the targets in the policy actions part). If for example an action is called on a component which does not support it, a severe error is detected and the user is invited to double check the policy specification and/or the concerned object specification. Policy validation tools such as the one presented in [77] provide such a facility.

A dynamically-infeasible policy inconsistency on the other hand is related to an infeasibility that can occur at system runtime. This happens, for example, if a policy is triggered to take an action on an object while that object has been dispensed.

Another case of dynamic-infeasibility is related to resource allocation policies. If the policy is triggered to request some resource while that resource is not available, the policy becomes infeasible. Depending on the application, such inconsistencies might

be allowed to occur as a design choice. For example, in the case study chapter (5), the resource allocation strategy employed by the service provider allows resource allocation policies (p2 in listing 32) to be unsatisfied. In this case, either the policy is discarded or it is paused until enough resources become available.

Application-dependent inconsistencies

Unreachable policy inconsistencies occur when the event/condition part of a policy never become true.

A policy is *statically unreachable* if either its condition part logically reduces to false or if the event that causes the policy is not potentially triggered by any policy or system component.

Proving that a policy is unreachable in the general case is difficult. In the case study given in the next chapter, a response time violation policy is automatically generated at an early stage of the policy refinement process (listing 28). However, due to a choice in the resource allocation strategy (section 5.3.2), this policy becomes *semantically unreachable*. This is because system resources are allocated to end customer sessions with a predefined limit on the load on each resource unit, which guarantees a query response time that is always within the contracted values. In this strategy, only the availability violation policy can be triggered in case not enough resource units are available, but the response time violation policy never gets triggered.

Shadowed policy inconsistencies

We say that a policy P is shadowed by another set of policies S if the existence of S makes P become unreachable. Policy shadowing (referred to as dominance in [107]) can occur even between two policies (S has one element). If a newly introduced policy Q consumes the same event as P, is triggered with the same conditions like P, is assigned a priority higher than that assigned to P, and the triggering event can be consumed only once, then P becomes completely shadowed by the introduction of Q.

Dominated policy inconsistencies

We say that a policy P is dominated if its still triggerable at runtime but the impact of its actions are completely covered by another set of policies. This implies that if P is removed no impact will be seen on system behavior.

In the following example, we show how a dominated policy inconsistency is different from a shadow or a duplicate policy inconsistency. Suppose that new policy p7 is introduced in a system where policies p5 and p6 already exist.

policy p5 = **on** e **if** $((a \wedge b) \vee c)$ **then** {a1→a2}
policy p6 = **on** e **if** $(a \vee c)$ **then** {a2}
policy p7 = **on** e **if** $(b \vee c)$ **then** {a1}

If actions a1 and a2 are idempotent and independent of each other, the introduction of p7 makes p5 become a dominated policy. This is different from shadowing as p5 is still triggerable, and different from duplication as no policy is a duplicate of the other.

If the actions a1 and a2 are not independent or idempotent, there is a potential of a harmful dominance. Unfortunately, this type of inconsistency is still not easy to detect and/or resolve.

Redundant policy inconsistency

Two or more policies are said to be redundant if they share a space of action where both have produce the same effect all the time. Such a type of inconsistency, if not done on purpose in case the actions in question are not idempotent, suggests the need to rewrite the policies in order to cancel the redundancy.

For example, policies p8 and p9 below will always result in block 2 being painted twice each time e is triggered. A solution would be to change p9' action to paint only block 1.

policy p8 = **on** e **then** {paintBlocks(2,3)}
policy p9 = **on** e **then** {paintBlock(1,2)}

Coverage inconsistencies [107] are related to the completeness of the policy specification. The following example from [21] illustrates this point:

policy PL1 = **if** (8 AM < time-of-day < 5 PM) ∧ (n < 10): queue = Qh.
policy PL2 = **if** (8 AM < time-of-day < 5 PM) ∧ (10 < n < 30): queue =Qn.
policy PL3 = **if** (8 AM < time-of-day < 5 PM) ∧ (n > 30): queue = Ql.
policy PL4 = **if** (4 PM < time-of-day < 5 PM) ∧ (n > 10): queue = Ql.

This set of policies does not define what to do in the cases time-of-day = 8 AM (or 5PM) and n=30 (or n=5). In this case, the administrator may want to specify non-strict inequalities for time-of-day and n in policy PL2.

Resource contention inconsistencies [102]

These occur when the amount of resources available is limited. Depending on the application type, these types of inconsistencies may be prohibited or allowed to exist.

Separation of duty [102]

A conflict of duties arises if the same subject is permitted to perform operations that, in the context of the application, are defined to be conflicting. For example, in a

company financial system, the operation of entering a request for payment and the operation of approving that request are potentially conflicting if the same user can perform both operations.

Multiple-management inconsistencies [102]

This type of conflict arises when different managers may manage the same objects, either because the objects are shared between several tasks or because different management functions are assigned to different roles. This may constitute a conflict when the management operations to be performed on the target object are not independent. For example an `update` operation may require a service to be temporarily shut down while a `get-configuration` operation may require it to be in service [104].

Self-Management inconsistencies [102]

A manager may not be allowed to retract policies that he is supposed to perform. This can be written as: "there should be no policy authorizing a manager to retract policies of which he is the subject" [104].

Overhead policy inconsistency

We say of a set of policies to cause an overhead if the actions they execute do not contribute to the defined high-level goals. Their existence is hence considered a pure consumption of system resources without any beneficial return. A subtle case of policy overhead inconsistency occurs when at a particular system configuration a set of policies enter an overhead behavior for a temporarily period, while they continue behave normally at other times.

Overheads if detected are indicative of a specification problem that needs to be addressed by the policy designer.

Policy loop inconsistencies

This is a type of policy inconsistency in which a set of policies get trapped into an infinite activity circle with the execution of one policy causing the triggering of the next policy in the loop. We encountered this type of inconsistency in the case study where specific parameter values for policies p2 and p3 of listing 32 (on page 121) enter a loop wherein p2 allocates a server unit resource to an SLA and immediately after p3 frees that resource leading p2 to be triggered again. This loop inconsistency was actually detected only at simulation time. Its detection is made more difficult by the fact that the loop may get automatically broken at the occurrence of specific events

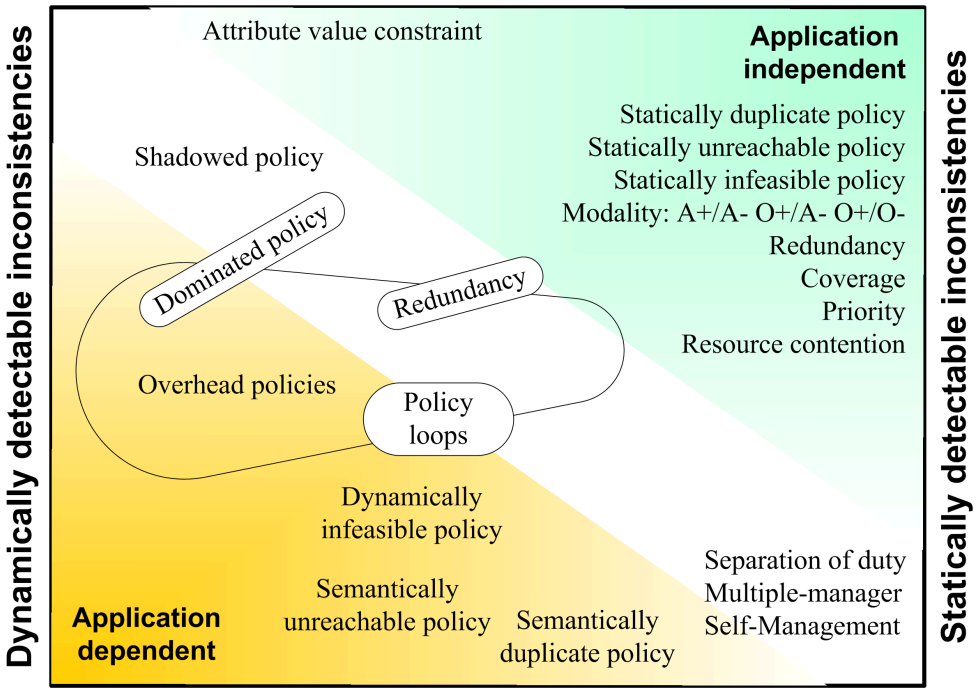


Figure 4.12 A second classification of policy inconsistencies

and the system returns back to normal execution. Hence, during the lifetime of the loop the two policies enter in overhead mode and may cause SLA violations if external events make the loop last for long.

We note also that the existence of policy loops does not necessarily imply a defect in the policy specification as this might be intended by the high-level goal.

Figure 4.13 provides a spacial classification of the henceforth identified policy inconsistency types. In this figure, policies are classified under two criteria based on whether they are statically or dynamically detectable and if they are application-independent or independent. To the right are located policy inconsistencies which are statically detectable. The more leftward we go the more inconsistencies are less easy to detect statically and require a dynamic analysis for them to be detected. Application-independent conflicts tend to be more statically detectable. However, a number of application-independent inconsistencies cannot be detected statically and require a dynamic analysis in order to be detected. Similarly, application-dependent inconsistencies tend to be more complicated and require a dynamic analysis in order to be detected. However, some application-dependent inconsistencies can be statically detectable if appropriate modeling tools for that application offer such a facility or if some theoretical model has been developed for that specific application.

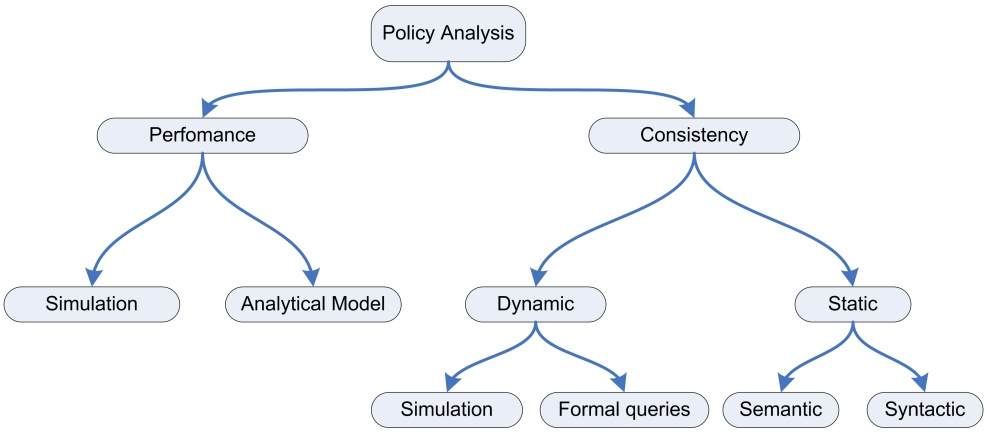


Figure 4.13 Classification of approaches to policy analysis

4.3 Policy Analysis

Policy analysis refers to the process of checking the specification of the policies in a managed system to ensure that they are consistent and fulfill the high-level requirements. For the case of quality assurance policies, an additional aim of the policy analysis would be to check the policies for their performance and seek to identify means to enhance this performance.

The analysis of policies for consistency and goal fulfillment can be performed at any stage during the policy elaboration process, with a more in depth analysis at the final phase where the set of low-level policies is generated.

Policy analysis techniques can be classified into two broad categories. Consistency analysis techniques and performance analysis techniques. Consistency analysis can be further categorized into static and dynamic approaches and the performance analysis into simulation-based and analytical-model-based analysis. Figure 4.13 summarizes this classification. At the leaf nodes, we identify the use of static, formal, analytical, and simulation-based techniques.

Agrawal el al. present in [21] a set of static policy consistency checking algorithms:

- The first algorithm deals with the determination of whether a set of single attribute constraints do not reduce to empty the domain space of that attribute. The algorithm works for both totally ordered (integer, string, real) discrete or continuous domain types as well as for non ordered finite domain types.
- The second algorithm deals with linear constraints over a set of variables and seeks to find a feasible non empty region for them. Constraints can be of the form [swap $\geq$ 2 RAM, boot = 1024, boot+swap< $\frac{1}{4}$HD, RAM>0].
- The third algorithm deals with the solving of the satisfiability of a compound boolean expressions. For example, if we want to check if two policies can be triggered at the same

time where the condition on the first policy is ((X <10) ∨ (X≥10 ∧ X+Y<10)) and that on the second policy is (X>12 ∧ X>2Y); the algorithm will check if the conjunction of the two formulas is satisfiable. The determination of all solutions of a compound boolean expression is however difficult in the general case and the authors provide only a simplified solution based on a Prolog-like backtracking technique.

- The fourth algorithm deals with the priority assignment problem and provides a solution to the precedence-based priority assignment. The algorithm has efficiency criteria that allow it to handle the assignment of priority values to a large number of policies, with a low overhead properties in policy insertions and deletion.

Bandara [77] present a tool for policy analysis. The tool supports querying a set of policies for validation and review queries. *Validation queries* are supported in order to determine the feasibility of a policy. The validation can be carried out on referenced objects, operations interface, attributes interface, and domains. A meta-policy specification is provided in order to detect unsupported operations at system runtime. However, it is not clear how and when this meta-policy can be used. On the other hand, *review queries* are used in order to help the administrator analyze the managed system specification and extract specific types of information. For example, the review queries can be used in order to identify the set of objects that are used in the subject (or object) element of policies that satisfy the query definition. The author also provides the specification of a set of meta-policies which can be used for the detection of modality, separation of duty, and multiple manager conflicts.

In addition, Bandara [77] suggests the use of abducting reasoning and tool developed for event-calculus-based goal elaboration [19, 82] in order to query potential conflicts between policies. The example given is that of querying the obligation policy "Martin should always connect to the VPN server between 9am and 5pm" and the refrain policy "Martin should refrain from connecting to the VPN server when accessing the internet with his PDA" for potential conflict. The abductive reasoning process manages to detect a case in which a conflict arises (Martin connects to the internet with his PDA at 9am). However, this technique remains limited in its applicability because of the inherent difficulty of automating the refinement of a conflict goal.

The policy analysis tool presented by Bandara suffers from a set of limitations, some of them are inherently related to the formal methodology itself [77]. The first limitation is related to the formal notation used by KAOS and which requires more user friendly interfaces. Second, each conflict rule and goal pattern has to be manually identified. Third, the derived strategies do not provide parameter values for management operations. And finally, due to the inherent algorithms used by the ASystem such as the inability to support any level of recursiveness.

Regarding policy performance analysis, and to the best of our knowledge, no previous work has

considered this issue. In fact, for the case of quality assurance policies we should not only be interested in specifying a set of policies that work. The issue of whether another set of policies might produce a better performance merits consideration. Furthermore, even for the same policy set, considerations on the runtime scheduling of policies might also impact the overall performance produced. In the next chapter we present a case where the analytical analysis of a policy set helped in identifying ways to improve their performance.

Analytical models, however, cannot be used at all times, mainly because of the complexity of real case scenarios. In this regard, resorting to simulation as a tool to test the consistency of a policy solution, as well its performance for the case of quality assurance policies, is needed in order to avoid the worst case of deploying a policy solution that might turn out to be a serious problem.

4.4 The $\mathcal{PS}$ Policy Simulator

Although research in policy-based management has been occurring for more than a decade, it is still difficult to put into practice. This limitation is attributed to the theoretical and practical difficulties in proving not only the correctness but also the efficiency of policy-based solutions when it comes to the management of real scale systems with hundreds or even millions of policies interacting in a dynamic way.

A number of policy languages and architectures were proposed. However, effective techniques for refinement and consistency analysis remain to be developed. It is therefore reasonable that venturing into a full policy-based solution for managing one's enterprize infrastructure remains difficult to justify.

With the current state of knowledge in policy-based management, it is possible to do some simple static analysis, mainly for the detection and resolution of conflicts between security policies. However, for the broader range of management policies, including quality assurance policies, there is no established theoretical basis for neither correctness analysis nor efficiency analysis. Furthermore, no work has been done on modeling the dynamics of management policies at system runtime. A policy-based solution should provide, in addition to correct behavior, an adequate performance which justifies its adoption in real scale management systems.

In this regard, resorting to simulation as a low cost testing facility provides a sound alternative. Similar to network simulation tools, which were introduced to cope with the difficulties in modeling the dynamics of queueing systems, we developed the policy simulator $\mathcal{PS}$ to serve as a tool for the simulation and analysis of the dynamics of policy-based management solutions.

This section first presents the architectural components of $\mathcal{PS}$ followed by more details on the implementation and usage of the $\mathcal{PS}$ package. The usage of $\mathcal{PS}$ is demonstrated in the next

chapter with a real case study.

4.4.1 $\mathcal{PS}$ Architecture

$\mathcal{PS}$ is designed in a way to support a business-driven management that has policy support at its core. Figure 4.14 shows the architectural components of $\mathcal{PS}$. Policies are modeled after the Event-Condition-Action (ECA) paradigm and are conceptually stored into the policy repository (bottom-left of figure 4.14) constituting a fast-access knowledge base of actions to take at the occurrence of well specified events and system conditions. The event service (bottom-right of figure 4.14) allows event sources and event listeners to be registered and routes events to their appropriate listener(s). Any $\mathcal{PS}$ component can be an event source and/or listener. An *active* $\mathcal{PS}$ policy is triggered when both its event and condition parts are simultaneously satisfied. When this occurs, the policy state switches from *active* to *triggered* and the policy is sent to the triggered-policies queue (TPQ). There it awaits for the policy decision point (PDP) to decide the actual time of the execution of its *action* part.

The above cycle represents the typical behavior of a conventional policy-based system. It is illustrated by the *policy control loop* of figure 4.14. The effectiveness and adaptiveness that policy-based management promises lies in the appropriate design of policies as well as the algorithms used by the PDP in order to properly orchestrate the queue of triggered policies.

At an upper level, the $\mathcal{PS}$ architecture supports SLAs and high-level business objectives. Low-level policies are generated, using the currently available refinement techniques and domain specific expertise, in a way so as to enforce the set of contracted SLAs and business-level objectives of the service provider. In this regard, $\mathcal{PS}$ represents an implementation of the BDMF framework presented in section 3.4.

The cycle of observing SLA-level and business-level states and deciding which new low-level control actions or strategies to follow in order to maximize the business profit of the service provider is captured by the *policy business loop* at the upper-level of figure 4.14. This loop is not as straightforward to implement as the lower *policy control loop*.

Metrics and metric probes are used in $\mathcal{PS}$ as a means to track the states of SLAs and business objectives. They represent a central building block for monitoring activities. For this, $\mathcal{PS}$ provides support for the definition of simple as well as compound (high-level) metrics. Several business-level and service-level metrics can be defined. These metrics are generally computed bottom-up from low-level resource metrics. At the top of the *metrics pyramid* lies the *business-profit function* Ψ. This metric reflects the measure of profitability of the business run by the service provider. In the general case, Ψ is a function of several service and business-level parameters including: service profitability, net financial benefit, customer satisfaction, and market share. If carefully

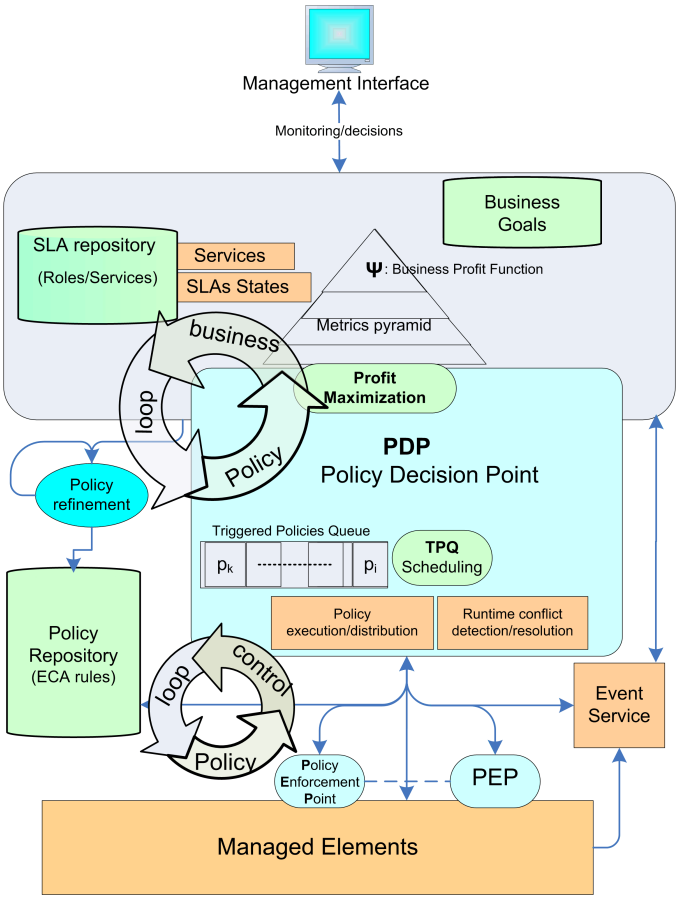


Figure 4.14 Illustration of the $\mathcal{PS}$ architecture

designed, the maximization of this metric should be the ultimate goal of the service provider. In the use-case presented in chapter 5, we consider a very simplistic Ψ which accumulates the net monetary revenue gained from the running SLAs.

Although the goal of maximizing the business profit is clear to state, its implementation is often domain specific. The decoupling of the policy business and control loops offers a high-level of adaptiveness and efficiency to the system without loosing the critical *reactive* property of a conventional policy-based solution.

The next section expands on the usage and implementation of $\mathcal{PS}$ components.

4.4.2 $\mathcal{PS}$ implementation and usage

$\mathcal{PS}$ offers a discrete simulation environment based on the process interaction world view. It builds on the base of the open source package *javaSimulation* [109], which follows very closely the Simula programming language.

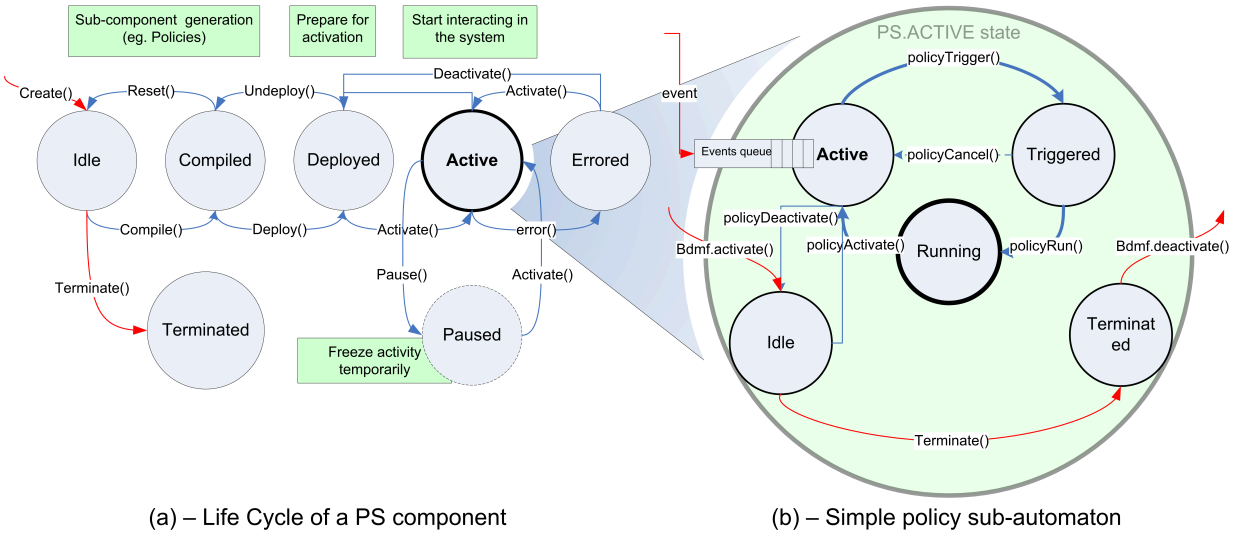


Figure 4.15 Life cycle automaton of a $\mathcal{PS}$ component

Each component of $\mathcal{PS}$ is implemented as a Java class and follows the life cycle automaton of figure 4.15-a. Life cycle state transitions occur through primitive calls triggered by components with the proper access rights. These primitives are implemented as overridden Java methods and are represented by the transition arrows in figure 4.15-a.

Right after it is constructed, a $\mathcal{PS}$ component is in an *idle* state. This means that it does not reserve/consume any system resources and has no responsiveness to events. The *compile()* primitive refers to the subcomponent generation process. The *deployed* state is that of a component which has been completely installed into the system and is only awaiting the green flag to start responding to events and interacting with its environment. A component can only be terminated if it is in the *idle* state. For example, an SLA instance can only be terminated when all of its low-level policies, metrics, and other associated objects have been terminated.

The *compile()* primitive in $\mathcal{PS}$ is equivalent to the refinement process of high-level SLAs, service-level objectives (SLOs), or business-level goals into intermediate or low-level policy rules. In practice, this can be done online, off-line, or in a hybrid way. In the *online refinement* case, an automated or interactive refinement process is triggered at the time the *compile()* method of the $\mathcal{PS}$ object is called. In the *off-line* case, the refinement is done over the class (SLA, SLO, or other high-level class) of the $\mathcal{PS}$ object. This results in the generation of a hierarchy of components together representing the low-level implementation of the initial class, in addition to mapping code within each *compile()* method which indicates how each sub-component of that hierarchy will be generated at runtime. In the *hybrid* case, first an off-line refinement is carried out to generate intermediate-level components. These components are subsequently refined online

whenever this is needed at system runtime. At present, $\mathcal{PS}$ only supports the off-line manual refinement because of the lack of existing off-the-shelf refinement tools.

4.4.3 Policy rules

Policies are modeled according to the Event-Condition-Action paradigm. As a $\mathcal{PS}$ component, a $\mathcal{PS}$ *policy rule* follows the life cycle defined in figure 4.15-a. In addition to this, when in the *active* state, the policy rule evolves within the sub-automaton of figure 4.15-b so as to reflect the behavior of an ECA rule.

At the interception of an event, and when the *condition* of the policy is met, a *policy_triggered* event is generated. By default, this event is intercepted by the policy decision point PDP (core component of figure 4.14). After that, the policy enters the *triggered* state whereby it "sleeps" waiting for the green light to execute its actions part. The PDP proceeds by queuing it (actually, a reference to it) into the TPQ. As long as the policy is in the triggered policies queue it remains in the *triggered* state. When the PDP decides to allow the policy to run, its state changes to *running* and an asynchronous call to method *policy.policyActions()* is made. This allows both the PDP and the policy to evolve independently. Synchronization facilities can be used in the case that the user wants to give more control over policy execution to the PDP. Once the *policyActions()* method returns, the policy returns back to the *active* state where it gets to process the next available event, if any, in its event queue. It also generates an event to notify whether its *actions* part has executed correctly or encountered problems.

Some policies execute only once, such as the policy rule p_1 in figure 32 (page 121). Other policies can be designed to execute a limited or even an unlimited number of times. For example, the policy rule p_2 (figure 32) executes each time its event and condition components are met. The explanation of what p_1 and p_2 actually do is given in section 5.4.5. When a policy reaches the maximum number of executions, it switches to the *idle* (figure 4.15-b) state.

A $\mathcal{PS}$ user should derive his own policy class from *ps.PolicyRule* then override the base methods *event()*, *condition()*, and *policyActions()*. By default, *event()* returns *true* and *condition()* returns *false*. The default dynamics of policy activation are taken care of by $\mathcal{PS}$.

4.4.4 SLA Model

$\mathcal{PS}$ supports the GSLA (section 3.3.2 on page 57) intuitive and generic SLA model, whereby an SLA is made up of a service package and a set of parties, each of which plays a role in delivering or consuming part or all of the service package. A *Role* defines a set of duties of a party. It can contain sets of SLOs and high-level policies. An SLS is the result of an SLA refinement

process. This process is called using the SLA.*compile()* primitive, which recursively calls the compile primitives of its subcomponents. The result of this would be a set of high-level metrics, such as SLO and SLA-level metrics, in addition to a set of low-level policy rules which together represent the compiled SLS.

4.4.5 Metric probes and Graphs

$\mathcal{PS}$ provides support for defining simple as well as compound (high-level) metrics. Metrics are enveloped within metric probes, which are objects able to listen and react to system events.

Basic data types are supported by the built-in *ps.Metric* class. Users can specialize this class to define their own metric types. A compound-metric probe listens to changes in sub-metrics probes as well as to other system events. At the top of the *metrics pyramid* (figure 4.14) lie business-level metrics which provide the state of the system at the business level. Service-level metrics can also be defined and all of the Service, SLA, SLO, and Role classes support the adjunction of metrics that are needed to define their respective states. All possible data types are supported as metric values. The change of a metric value can be signaled to upper and lower metric probes so that they can update their respective values accordingly.

In order to control the propagation of metric updates, six propagation methods are supported: none, parents, children, parentsThenChildren, childrenThenParent, and generic. The generic method subsumes all of the previous ones and is to be used with extra caution. Allowing metric updates to propagate in all possible directions is needed for the generic case, however update propagation loops have to be avoided. To enable the visualization, or simply the recording, of metrics evolution in time, $\mathcal{PS}$ supports graph objects as special metrics which hook on the top of other metrics and record all their new values, time stamps, and any other required data. Graph objects can also write all (time, value) pairs to persistent storage for future analysis.

4.5 Conclusion

This chapter reviewed the efforts conducted in the refinement and analysis of policies. Recently, the interest in policy refinement has grown and important advances have been achieved mainly when wisdom gathered from the requirements engineering discipline has been used.

However, there is still much to do especially for policy inconsistency detection and resolution. The number of policy inconsistency types that we have identified in this chapter showed that not only is there a variety of inconsistencies that a policy designer needs to be concerned about, but that there is also a need to consider researching related fields where these consistencies have been

previously confronted. We believe that many of the identified inconsistencies are not specific to the policy discipline per se.

The $\mathcal{PS}$ policy simulator tool we presented at the end of this chapter is still a work in progress. However, it has a good potential in order to serve for the simulation of policies and help in the inconsistency detection and resolution as well as in the performance evaluation and enhancement of quality assurance policy solutions. The next chapter illustrates this by providing a detailed use case in which $\mathcal{PS}$ has been of use for the detection of a policy loop inconsistency and helped in the performance enhancement of policies.

Chapter 5

Case study

This chapter develops a use case for the business-driven SLA refinement into low-level management policies and provides an implementation which maximizes the business profit of the service provider[1]. The case spans the entire business driven SLA management loop. We show the details of how the refinement process is conducted to produce a policy based SLS. The analysis phases, described in the previous chapter, are then applied to that generated SLS. The *Static Analysis* phase checks the SLS' policy set for consistency and stability. The *Dynamic Analysis* phase addresses the business-driven dynamic analysis of policies in which we emphasize the need for incorporating business (and SLA) related data, encoded mainly within metrics generated during the refinement process, to handle the orchestration of policies at runtime. This analysis proves crucial in making the same set of generated policies (SLS) achieve best performance at runtime.

This chapter proceeds as follows. Section 5.1 presents the generic SLA use case. Section 5.2 defines the business profit function we would like to optimize and section 5.3 presents two different strategies to enforce it. Section 5.4 derives a formal SLS for the generic SLA and shows the different stages of the proposed refinement process. After that, the static analysis is conducted on the generated SLS to detect and resolve a number of inconsistencies. We then conduct in section 5.6 the dynamic analysis and use it to derive better runtime scheduling algorithms of triggered policies. Section 5.8 explains how the use case has been implemented onto the $\mathcal{PS}$ simulation environment and section 5.9 presents summarized performance results of the different policy scheduling algorithms.

Listing 24 Generic application hosting SLA: $\mathcal{SP}$ SLA

1. Customer $\mathcal{C}$ is provided an application hosting service with schedule sc.
2. Maximum capacity is of cp_{max} simultaneous connections.
3. $\mathcal{C}$ is charged $\$ch = a \times cp_{max}$ monthly.
4. Monthly average availability of the hosted service $\geq av_{min}$.
   (a) An i^{th}successive availability violation incurs a reward of $r_i \times ch$.
   (b) At the 3^{rd}successive availability violation, the SLA is considered void.
5. Min average time to process end customer requests $= rt$ ms.
   (a) Otherwise, $\mathcal{C}$ is rewarded $rt.ref \times rt$.

5.1 Generic $\mathcal{SP}$ SLA

We consider an Application Hosting Service Provider $\mathcal{SP}$ which advertises a set of SLAs to its customers. The set of SLAs is derived from the simple generic SLA of listing 24, named $\mathcal{SP}$ SLA. $\mathcal{SP}$ operates in its information infrastructure a pool of $sp.cp$ identical server units. Server units are allocated to each SLA instance to ensure its QoS requirements.

$\mathcal{SP}$ SLA states, in 5 clauses, that $\mathcal{SP}$ offers an application hosting service supporting a load (capacity) of cp_{max} simultaneous end client connections to the system, an availability average of av_{min}, an average response time rt, all with a monthly cost ch. The total set of parameters can be gathered in the tuple (sc, cp_{max}, a, av_{min}, r_1, r_2, r_3, rt, $rt.ref$). Each instance of this tuple generates an *SLA type* the $\mathcal{SP}$ can advertise to its potential customers. An SLA instance is a realization of an SLA type for a particular customer. In the following, sla_i denotes an SLA type and $sla_{i,j}$ denotes SLA instance j of SLA type i, all of which are derived from the generic SLA of listing 24.

Before going further into the SLA refinement process, the first step is to define the business profit function the $\mathcal{SP}$ intends to maximize.

5.2 Defining the Business Profit Ψ

Denoted here as Ψ, the business profit function provides a measure of the profitability of the service provided by the $\mathcal{SP}$. In the general case, Ψ should be a function of several service and business level parameters such as service operation cost, net financial revenue, customer satisfaction, and market share. To keep the use case as simple as possible, we define Ψ to be the sum of the net financial profit gained from each contracted SLA. This implicitly assumes

[1]Published in [110]

that managing $\mathcal{SP}$'s information infrastructure incurs a fixed cost which is independent of the number of contracted SLAs. Hence,

$$\Psi = \sum_{i \in \mathcal{SLA}} \left(\sum_{j \in \mathcal{SLA}_i} (NP(SLA_{i,j})) \right) \tag{5.1}$$

where $\mathcal{SLA}$ represents the set of all SLA types that $\mathcal{SP}$ supports, and $\mathcal{SLA}_i$ the set of contracted instances of SLA type SLA_i. The problem $\mathcal{SP}$ has to solve is to reach $Max(\Psi)$? We will elaborate further on this function in section 5.7.

5.3 Enforcement strategies

The clauses of listing 25 help identify the set of Service Level Objectives (SLOs) the $\mathcal{SP}$ has to enforce. The identification we do is semi-formal as it requires interpreting a textual specification and pouring it into a formal specification. Section 5.4.1 explains how this SLO set has been generated.

Using a policy based approach, $\mathcal{SP}$ still has multiple choices as to how to enforce them. In the following we describe two of them, a guaranteed approach and a lazy one.

5.3.1 Guaranteed enforcement strategy

With this strategy, $\mathcal{SP}$ will pre-allocate for each new SLA instance the exact number of resources (server units) required to enforce its SLOs at maximum load.

We will assume that $\mathcal{SP}$ possesses a mapping function $su(rt)$ which gives the load (number of simultaneous connections) a single server unit (su) instance can handle while still respecting the response time constraint rt (for end customers) as specified in the SLA.

Let $|sla_{i,j}.sus|$ denote the number of server units allocated to $SLA_{i,j}$. The maximum number of server units required by each $SLA_{i,j}$ is then:

$$Max(|sla_{i,j}.sus|) = \lceil cp^i_{max} \times av^i_{min} / su(rt_i) \rceil \tag{5.2}$$

Let $|sla_i|$ be the number of contracted SLAs (instances) of type i. When using the guaranteed approach, we should always have:

$$\sum_{i \in \mathcal{SLA}} \left(|sla_i| \times \lceil cp^i_{max} \times av^i_{min} / su(rt_i) \rceil \right) \leq sp.cp \tag{5.3}$$

We assume that the set of contracted SLAs become activated at the same time. If the system runs with no unexpected failures, the guaranteed enforcement approach will produce business profit:

$$\Psi = \sum_{i \in \mathcal{SLA}} ch_i \left(\sum_{j \in \mathcal{SLA}_i} sc_{i,j}.duration() \right) \tag{5.4}$$

Given a set of defined SLA types, $\mathcal{SP}$ can find the number of instances to contract for each SLA type so as to maximize Ψ by solving the integer programming simplex formed by $Max(eq.5.3)$ and eq.5.4.

However, such approaches prove inefficient in practice where the running SLAs are actually not fully loaded at all times, which is the case for most web applications.

5.3.2 Lazy enforcement strategy

In this approach, $\mathcal{SP}$ allocates server units to each SLA on a per need basis. Conversely, it removes server units from an SLA as soon as this latter starts experiencing low load. Similar to the statistical multiplexing of traffic crossing a shared physical network cable, this technique allows $\mathcal{SP}$ to contract a number of SLAs with a total maximum sever pool capacity beyond the actual capacity it possesses.

At instantiation time, an initial number of $n_0(\geq 1)$ server units is allocated to the SLA. When the connection intensity (number of simultaneous end customer connections) reaches a certain threshold thA of the current SLA capacity, a request is made to the server pool to obtain an additional server unit. Conversely, if a low threshold thR is reached, an action is triggered to release a server unit to the free server units pool. With this technique, $\mathcal{SP}$ aims at a higher business revenue than promised by the guaranteed enforcement approach (eq. 5.3-5.4). Notice that at this stage, three new parameters have been added to the SLA type, which are the thresholds thA and thR and the initial minimal number of server units n_0.

5.4 Refinement of the generic $\mathcal{SP}$ SLS

There is currently no approved systematic approach to SLS specification and refinement. The refinement we present here attempts to systemize this process.

The refinement will be done in a set of phases starting by the formal specification of SLOs and high level policy rules. Following this, the Enforcement strategy along with available system resources functionality are used to guide an iterative refinement process until a fixed point is reached where

all SLS statements are directly supported by the resources at hand. Hence, depending on which enforcement strategy is used and/or which system resources are available different SLSs can be generated.

5.4.1 SLO set specification

Service Level Objectives (SLOs) represent logical constraints over SLA parameters the $\mathcal{SP}$ has to respect. Using a straightforward formulation, the set of SLOs of listing 25 correspond respectively to clauses $1, 2, 3, 4$, and 5 of $\mathcal{SP}$ SLA (listing 24), with the sub-conditions $4-a, 4-b$, and $5-a$ excluded. These sub-conditions will be dealt with in the next iteration. As the title of listing 25 indicates, this SLO set corresponds to the guaranteed enforcement approach as it exactly translates (excluding sub-conditions) $\mathcal{SP}$ SLA into formal terms.

Listing 25 SLO set for the guaranteed enforcement of $\mathcal{SP}$ SLA

```
sloSet sloSetG = {
  slo slo_sc = (schedule == sc);
  slo slo_cp = (ws.cp == cp_max);
  slo slo_ch = (payment.sum(month) == ch);
  slo slo_av = (ws.av ≥ av_min);
  slo slo_rt = (ws.rt ≤ rt);
}
```

The capacity SLO slo_{cp} of listing 25 states that the total capacity of allocated server units for an $\mathcal{SP}$ SLA instance should be equal to cp_{max}. For the lazy enforcement, this SLO is a requirement that is stronger than what is actually needed. This is because, in this approach, $\mathcal{SP}$ intends to allocate server units to each SLA on a per need basis. So slo_{cp} needs to be weakened to the new expression of listing 26. With this expression the runtime capacity of an SLA, in terms of the sum of capacities of allocated server units, is allowed to be less than the value contracted in the SLA.

The passage from sloSetG to sloSetL is an example of SLO refinement which is refinement strategy dependent.

Listing 26 SLO set for the Lazy enforcement of $\mathcal{SP}$ SLA

```
sloSet sloSetL extends sloSetG = {
  slo slo_cp = (ws.cp ≤ cp_max); }
```

The next iteration will deal with the tracking of SLO states and how state changes/violations are signaled.

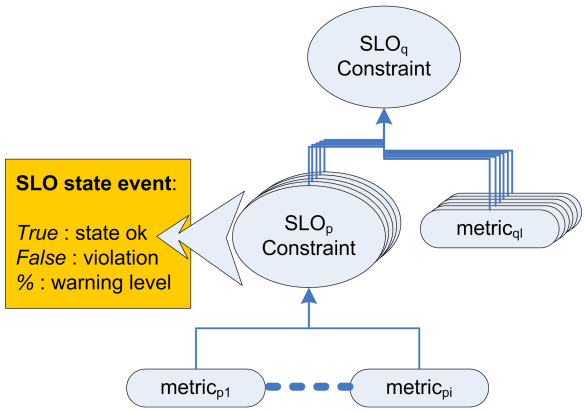


Figure 5.1 SLO state constraints and related metrics

5.4.2 Metrics and SLO constraints for SLO state tracking

In our refinement approach we make use of events as the means to signal SLO violations. When an SLO is violated, an event is generated to inform about the parameters related to the violation, such as the time of its occurrence and/or some log information that might be useful for its processing. To evaluate the constraint defined by the SLO, there is a need to have the values of each of its parameters at hand. Therefore, these parameters need to be defined as metrics that are computed at SLA runtime and serve as input to a constraint tracking object which periodically evaluates and informs about that SLO's state. Such a dependency hierarchy is illustrated in figure 5.1.

The required runtime SLO constraints and metrics and their relationships can hence be identified in a top down fashion with the prior knowledge of what resource level (leaf) metrics are supported by the underlying information infrastructure of the service provider.

The output set of required high level metrics and SLO state constraints is listed in listing 27. This set shows additional parameters that need to be filled for an actual SLS instance. For example, there is a need to define specific values for service deployment and un-deployment times. In addition, the capacity SLO slo_{cp} generates the need to compute a new metric mWSCP. This metric is set to collect the actual runtime load $ws.cp$ of the server units. Each metric is then used at runtime as an input to the SLO evaluation. Similarly, metrics are also generated to track the states of the payment, availability, and response time SLOs.

5.4.3 policies for SLO violation events

The consequences of violating an SLO need to be specified in the SLA. At this phase, a set of policies of the form *on not(slo) do action* is generated based on the generated SLO set (listings 25,

Listing 27 Metrics identification based on available high level system metrics

- slo_{sc} $\Rightarrow$
  define schedule=sc,
  schedule.deployTime=<? >,
  schedule.undeployTime =<? >, ...
- slo_{cp} $\Rightarrow$ **metric** $mWSCP = ws.cp$
- slo_{ch} $\Rightarrow$ **metric** $mMonthlyFee = payment.sum(month)$
- slo_{av} $\Rightarrow$ **metric** $mAv = ws.av$
- slo_{rt} $\Rightarrow$ **metric** $mRT = ws.rt$

26). Since SLO metrics are now defined (listing 27), this set can be automatically generated as is shown in listing 28. In this set, $not(slo)$ is replaced by an event type $\overline{slo}$ which is intended to convey the violation information to all those SLS policies that need it.

Listing 28 Automatic SLO violation policies

- slo_{cp} $\Rightarrow$
  eventType $\overline{slo_{cp}}$;
  policy pmWSCP = {
  on $(mWSCP > cp_{max})$ **do** $generate(\overline{slo_{cp}})$}
- slo_{ch} $\Rightarrow$
  eventType $\overline{slo_{ch}}$;
  policy pmMonthlyFee = {
  on $(mMonthlyFee < ch)$ **do** $generate(\overline{slo_{ch}})$}
- slo_{av} $\Rightarrow$
  eventType $\overline{slo_{av}}$;
  policy pmAv = {
  on $(mAv < av_{min})$ **do** $generate(\overline{slo_{av}})$}
- slo_{rt} $\Rightarrow$
  eventType $\overline{slo_{rt}}$;
  policy pmRT = {
  on $(mRT < rt)$ **do** $generate(\overline{slo_{rt}})$}

The policy rules notation is inspired from [39, 108].

5.4.4 SLO violation policies

Each time an SLO is violated an action is required in order to bring the SLA back to a normal operation state. SLO violations can also trigger actions that are specified in the SLA. In this iteration we generate policies related to SLO violations that are directly deductable from the SLA. Listing 29 shows the policies generated for each SLO violation clause. This set of policies is required by both enforcement strategies.

Availability is computed as a monthly average as specified in the generic SLA. However, the $\mathcal{SP}$ can compute it in several ways. A similar case exists with the semantics of "successive". We will not elaborate on the possible options here in order to keep our discussion focused. We

Listing 29 SLA-specific SLO-violation policies

double av_w = 1 month; // availability window
event $\overline{slo_{av}}$ $e1, e2, e3$; // events of type $\overline{slo_{av}}$

- Clause 4-a ⇒
  policy p_4 = {
  on $e1$
  do $c.credit(r1)$
  where $not(p_5 \vee p_6)$}
  policy p_5 = {
  on $(e1 \rightarrow e2)$ // $e1$ followed by $e2$
  do $c.credit(r2)$
  where $((time(e2) - time(e1) < av_w) \wedge not(p_6))$ }
  policy p_6 = {
  on $(e1 \rightarrow e2 \rightarrow e3)$
  do $\{c.credit(r3)$
  where $((time(e3) - time(e1) < av_w))$ }
- Clause 4-b ⇒
  policy p_{6b} = {
  on $(e1 \rightarrow e2 \rightarrow e3)$
  do {SLA.terminate()
  where $((time(e3) - time(e1) < av_w))$ }
- Clause 5-a ⇒
  policy p_{rtv} = {
  on $\overline{slo_{rt}}$
  do $c.credit(rt.ref)$

assume that $\mathcal{SP}$ computes availability as a sliding window average of length av_{win} (= one month in the SLA). For "successive", the $\mathcal{SP}$ refers to those violations which occur within at least one availability window interval (1 month).

Policies p_4, p_5, and p_6 implement availability violation penalties as specified in clause 4 of the generic $\mathcal{SP}$ SLA of listing 24. p_4 states that on the occurrence of an event e_1 of type $\overline{slo_{av}}$, meaning a violation of slo_{av} at runtime, the action that needs to be carried out is to credit the customer account with $r1$ monetary units. p_5 and p_6 implement respectively the penalty clauses for the 2^{nd} and 3^{rd} successive violations. The where conditions enforce the one month "memory" on successive violations and ensure that only one of p_4, p_5 or p_6 can be triggered at a time.

Listing 30 Policy to enforce at customer side

- Clause 3 ⇒
  policy pC1 = {
  on every month
  do $\mathcal{SP}$.credit(ch)
  start at sc.activationTime }

5.4.5 Enforcement strategy dependent policies

For the guaranteed enforcement approach, $\mathcal{SP}$ has to enforce policy p_{g1}. This policy allocates the number of server units (equation 5.2) that are needed to guarantee availability and response time SLOs at all times, that is, even during maximum load. These policies also make all necessary initializations required by the specific web application of the SLA instance.

At the other end of the SLA life time, the un-deployment time policy p_7 takes care of wrapping up actions which include resource liberation, deactivation of active SLA policies, and some reporting actions.

Listing 31 Guaranteed approach specific policies

- `Clause 1 ∧ Clause 2 ∧ Clause 5 ⇒`
  policy p_{g1} = {
  at $sc.deployTime$
  do $ws.add(\lceil cp_{max} \times av_{min}/su(rt) \rceil)$
  }
- `Clause 1 ⇒`
  policy p_7 = {
  at $sc.undeployTime$
  do `SLA.undeploy()`}

For the lazy approach, policy rule p_{g1} gets replaced (or split) into the three rules of listing 32.

Listing 32 Lazy enforcement dependent policies

double thA = someConstantExpression
  constraint $0 < thA \leq 1$; // %
double thR = someConstantExpression
  constraint $0 < thR < thA$; // %
int n_0 = someConstantExpression // ≥ 1;
  constraint $1 \leq n_0$; // %
policy p_1 **overrides** p_{g1} = {
 at $sc.deployTime$
 do $ws.add(n_0)$}
policy p_2 = {
 on $(ws.load \geq thA)$
 do $ws.add(1)$
 where $(ws.cp \leq cp_{max})$}
policy p_3 = {
 on $(ws.load \leq thR)$
 do $ws.free(1)$
 where $(|ws.su| > 1)$}

p_1 is a deployment time policy which initializes the new SLA by requesting an initial number n_0 of server units from the pool of free server units. p_1 is related to the lazy enforcement strategy and hence overrides policy p_{g1}. p_7 is not in conflict with the lazy enforcement and is kept unchanged.

p_2 and p_3 implement the lazy enforcement approach by tracking the load of the server units that are available to the SLA instance. They execute the necessary actions each time a threshold is

crossed, p_2 by requesting a new server unit at high load times and p_3 by releasing one at low load times.

The determination of the appropriate values of parameters thA and thR are SLS specific and tunable by the $\mathcal{SP}$. In the conducted simulations these two parameters had a strong impact on the generated profit.

Finally, SLO slo_{rt} (listing 26) is implicitly implemented in both strategies by limiting the maximum load on each server unit to $su(rt)$ (see section 5.3.1).

5.4.6 Another iteration for generating metrics/policies

The condition on policies p_2 and p_3 need to be defined in terms of constraints over metric values. Hence, there is a need to detect $ws.load$ through a metric and have an event generated each time this metric crosses one of thA and thR thresholds. Listing 33 shows how p_2 and p_3 are updated.

Listing 33 Additional metric generation and policy set update

- p_2 $\Rightarrow$
  metric $mP2ThAdd = ws.load$
  policy pmP2ThAdd = {
  on $(mP2ThAdd \geq thA)$
  do generate(mP2ThAddEv) }
  update p_2 = { **on** $mP2ThAddEv$ }
- p_3 $\Rightarrow$
  metric $mP3ThRem = ws.load$
  policy pmP2ThRem = {
  on $(mP3ThRem \geq thR)$
  do generate(mP2ThRemEv) }
  update p_3 = { **on** $mP2ThRemEv$ }

5.4.7 Final iteration and Complete SLS specification

So far, this section described the $\mathcal{SP}$ SLS generation process for both the guaranteed and lazy enforcement policies. Each output SLS is the completed service level specification (SLS) of the generic $\mathcal{SP}$ SLA. Listing 34 summarizes the output of this refinement process. The generation of this SLS involved several sub processes, which in the general case are expected to be iterative with a fixed point property.

Note that a first part of the SLS needs to be enforced at $\mathcal{SP}$ side. The $\mathcal{SP}$ responsibilities have been grouped under one *role structure* named $\mathcal{SP}$. The second part of the SLS is related to customer payment policy (pC1 in listing 30) and needs to be enforced at customer side, under role $\mathcal{C}$. Hence, the resulting SLS has two roles for each party of the SLS. $\mathcal{SP}$ role contains a total of 13 policies, 5 metrics, and 3 constraints. Policies p6 and p6b have been blended into

Listing 34 Completed $\mathcal{SP}$ SLS for the lazy enforcement strategy

sls $\mathcal{SLS}$ = {

- // Service Provider Role
  role $\mathcal{SP}$ = {
  schedule sch;
  int n_0; //
  double $thA,\ thR,\ av_w$ = 1 month;
  constraint $0 < thR < thA,\ 0 < thA \leq 1,\ 1 \leq n_0$;
  metric $mWSCP = ws.cp,\ mAv = ws.av$
  metric $mMonthlyFee\ = payment.sum(month)$
  metric $mWSLD = ws.load,\ mRT = ws.rt$;
  eventType $\overline{slo_{cp}},\ \overline{slo_{ch}},\ \overline{slo_{av}},\ \overline{slo_{rt}}$;
  event $\overline{slo_{av}}\ e1, e2, e3$; // events of type $\overline{slo_{av}}$

- // SLO violation notification policies
  policy pmWSCP = {
  on $(mWSCP > cp_{max})$ **do** $generate(\overline{slo_{cp}})$}
  policy pmMonthlyFee = {
  on $(mMonthlyFee < ch)$ **do** $generate(\overline{slo_{ch}})$}
  policy pmAv = {
  on $(mAv < av_{min})$ **do** $generate(\overline{slo_{av}})$}
  policy pmRT = {
  on $(mRT < rt)$ **do** $generate(\overline{slo_{rt}})$}

one policy p6. Metrics mP2ThAdd and mP3ThRem being identical have been blended into one metric mWSLD. Note that we assumed that $\mathcal{SP}$ takes care of all metric computations and not the customer or a third party.

The inclusion of third parties and the distribution of metric computations can also be included in the SLS specification but was not considered here in order to keep the use case simple.

Finally, we would like to emphasize that the final SLS does not contain any SLO definition. All necessary logic has been specified in the sets of policies, metrics, and events. Hence, an SLO can be a component of an SLA or an intermediary SLS, but not the final SLS.

5.5 Static analysis

The second phase after the generic SLS is generated conducts a static analysis in order to test the consistency of the generated policy set. For the static analysis phase we identified and conducted four types of tests:

- action conflicts
- deadlocks
- oscillation (loops)
- unreachable states (dead code (i.e policies))
- erratic behavior

$\mathcal{SP}$ needs to test the generated policies to make sure they are free from any of these defects.

5.5.1 Static conflicts Analysis

Action *conflicts analysis* relates to policies which have conflicting actions and which can execute at the same time on the same system object. By observing the generated policy set, we notice that p_1 and p_2 request additional server units. However, p_1 executes only once at the SLA deployment time while p_2 becomes active only after p_1 has executed correctly. p_3 releases one server unit which is an action expected to be always successful. p_4, p_5 and p_6 cannot execute at the same time (even though this does not cause trouble). In the implementation, this translates to making method c.credit() synchronized. When several SLA instances are running, policies of type p_1 and p_2 can conflict due to lack of sufficient server units. For example, in case only one free server unit is available, only one policy can be executed while the others need to wait until enough resources become available. However, this is a *runtime conflict* that the lazy enforcement strategy, for example, accepts. So, from the static analysis point of view, the policy set is actions conflict free.

5.5.2 Deadlock Analysis

For *deadlock analysis*, it is straightforward that the policy set is deadlock free as there is only one possible blocking action ws.add(). This action requests a number of server units from the pool of free server units. So a deadlock situation at runtime is not possible.

5.5.3 Erratic behavior Analysis

The constraints on the definition of thA and thR (listing 32) have been set following the intuition that the threshold to request a new server unit should be strictly greater than the one which frees an acquired one. Without these constraints and if thR was fixed, possibly due to a mistake of the system operator, to a value greater than thA, the concerned SLA instance might never free any acquired server unit until it is terminated, or on the other extreme, it might show *erratic behavior* in case of a shortage in server units.

For the first case, $ws.load = thA \Rightarrow p_2$ gets triggered

$$\Rightarrow \text{new } ws.load < thA < thR$$

This implies that when the number of connections decreases, no action will be taken by the SLA and it will continue to hold resources that it is actually not using!

The latter case, however, is more harmful. It occurs when p_2 is triggered while no resources are available in the system and $ws.load$ continues to grow until reaching thR. At this moment p_3 is triggered reducing $su.size$ by one.

Since we have

$$ws.load = \frac{|connections|}{(su.cp \times su.size())}$$

This implies that $ws.load$ increases. Hence, p_3 gets triggered again, and so on, until all but one of the server units are freed (because of the **where** clause in p_3). It is then expected that availability violations occur leading possibly to SLA termination (policy p_6). With a slight chance, the SLA can still survive if before p_6 is triggered the load on the unique left server unit diminishes. A way to detect this type of erratic behavior is to specify a rule for the static analyzer which states:

```
//fsupl = free server units pool
policy pErraticTest1 = {
 on (triggered(p6) ∧ (fsupl.size > 0))
 do signal(erraticBehavior);}
```

5.5.4 Unreachable code (policies)

Based on our assumption that one server unit instance is loaded only up to $su(rt)$ simultaneous end customer sessions, the system can deduce that policy p_{rtv} cannot be triggered at runtime. So, theoretically, these policies can be safely removed from the output SLS. However, in practice server machines can become congested due to various causes and, although the constraint on the number of simultaneous sessions is respected, the response time might still be violated. The final decision to remove or keep p_{rtv} has then to be left to the discretion of the SLS designer.

5.5.5 Oscillation Analysis

The next step in this phase is to check for potential *static loops*. We define a system static loop to represent the case when there exists at least one system state $\mathcal{S}_l$, different from the final state $\mathcal{S}_{final}$, which when reached the probability that the system comes back to $\mathcal{S}_l$ after some time $t \geq 1$ is 100%. Formally expressed as:

Definition

A set of system states $\mathcal{S}$ is not static loop free iff

$\exists\ \mathcal{S}_l \in \mathcal{S}, t_l > 0, |\mathcal{S}_l \neq \mathcal{S}_{final} \wedge Pr(\mathcal{S}_0, \mathcal{S}_l) > 0 \wedge Pr(\mathcal{S}_l, \mathcal{S}_l) = 1$

We consider in this section the runtime state of an SLA to be the tuple $(ws.cp, np)$, where $ws.cp$ is the capacity in number of allocated server units, and np the accumulated net profit. np is

affected by operations $SP.credit()$ of policy pC1 (listing 30) and $C.credit()$ of policies p_4 to p_6 (listing 32).

We also consider that the state of $\mathcal{SP}$'s system to be the sum of the states of all the contracted SLAs augmented by the state of the free server units pool and all the business metrics the $\mathcal{SP}$ maintains.

For policies p_4 to p_5, an oscillation case is impossible because any of them cannot occur more than once during any availability interval (1 month). Also, if all of them occur during the same availability interval, the corresponding SLA is terminated. Termination is still a safer "state" than a "looping"!

For policies p_1, p_2 and p_3 there is a potential static loop. This is because p_1 and p_2 request additional server units while p_3 requests an operation which nullifies their actions by freeing one server unit. Thus, further analysis is required on these policies.

With this semi-formal analysis, the $\mathcal{SP}$ should be relatively assured that the generated SLS will achieve the goal of the input SLA of listing 24. However, there is still a subtle error which was not discovered until after observing the execution of a batch of randomly generated simulations.

For some randomly generated input parameters which respect all the above stated constraints, the system still enters an infinite loop oscillating between policies p_2 and p_3. By analyzing closely this case we found that the constraint $0 < thR < thA \leq 1$ is not sufficient. This leads us to consider when p_3 can be automatically triggered once p_2 is triggered and vice versa.

First, let's recall that:

- $ws.load = |connections|/ws.cp$, where
  - $|connections|$ is the current number of end customer connections.
  - $ws.cp = su(rt) * su.size$, i.e web server capacity = capacity of one server unit times the number of allocated server units.
- p_2 increments $su.size$ by 1 while p_3 decrements it.

Just before p_2 is triggered, i.e. an end customer is terminating a session, we define:

- $z = su.size$,
- $n = |connections|$, number of connections
- $s = su(rt)$,
- $c = n/su(rt)$, and
- $z_{max} = cp_{max}/su(rt)$

For p_2 to be triggered right after an end customer exits we should have:

$$\frac{n-1}{s \times z} < thA \leq \frac{n}{s \times z} \tag{5.5}$$

The successful execution of p_2 increments z to $z+1$ to translate the adjunction of a new server unit to the SLA instance.

For p_3 to be triggered right after p_2 has executed (creating a loop), we should then have:

$$\frac{n-1}{s \times z} > thR \geq \frac{n-1}{s \times (z+1)} \tag{5.6}$$

By putting these two inequations together and simplifying we obtain the condition:

$$\frac{thA}{thR} \leq 1 + \frac{1}{z}, \forall z | 1 \leq z \leq z_{max}$$

Hence, to avoid a static loop starting from p_2 we prove that it is necessary and sufficient to have ($\rceil$ is the not operator):

$$\rceil (p_2 \rightarrow p_3) \Leftrightarrow thA > 2 \times thR \tag{5.7}$$

The second case is to get the condition to avoid having p_2 automatically triggered right after p_3 has executed. Following similar reasoning we obtain :

$$\frac{thR}{thA} < \left(1 - \frac{1}{z_{max}}\right), \forall z | 2 \leq z \leq z_{max} \tag{5.8}$$

From eq.5.7,5.8 we prove that the necessary and sufficient condition for the policy set of listing 32 to be loop free is :

$$\mathcal{SP} \text{ SLS is loop free} \Leftrightarrow thA > 2 \times thR \tag{5.9}$$

□

At this point the static analysis phase ends. The output of a static analyzer tool, if it existed, should be a recommendation to change the constraint on thR to become compliant with eq.5.9.

The next step explores aspects of the business-driven dynamic analysis of an SLS[2](section 4.3).

5.6 Business-driven dynamic analysis

At normal SLA operation, all triggered policies should execute properly. Conceptually, a triggered policy needs the approval of the policy decision point (PDP) before it can execute [5]. This

[2]The idea of this new type of analysis can be found in [111, 112]

implies the existence of a conceptual queue, or waiting room, for triggered policies which the PDP serves by scheduling them according to some predefined scheme. In practice, the PDP can be implemented as a hierarchy of PDPs distributed within the information infrastructure. Our analysis will be based on the conceptual PDP of $\mathcal{SP}$'s information infrastructure.

In the absence of any further information, the PDP can schedule the Triggered Policies Queue (TPQ) according to two classical algorithms:

FCFS

This is actually a variant of FCFS in which triggered policies are serviced in FCFS as long as there are enough available resources for their actions part. In the case that a triggered policy cannot execute because of unavailable system resources, the policy in question retains its order in the TPQ but the PDP skips it each time it processes the TPQ until resources become available for its execution. For $\mathcal{SP}$ SLS this case can occur for policies p_1 and p_2.

RND

In this algorithm, the PDP picks the next policy to run at random from the set of runnable triggered policies.

If $\mathcal{SP}$ chooses to contract a number of SLAs with a total maximum capacity exceeding the actual capacity it has in its servers pool, it can be proven that by taking additional concern at the TPQ scheduling level better overall business profit can be achieved.

5.7 Business driven TPQ scheduling

In this section we develop a new technique for TPQ scheduling which is intended to provide a better handling of peak utilization times for $\mathcal{SP}$'s resources leading to better overall business revenue. This technique takes into consideration the runtime states of instantiated SLAs in servicing the TPQ.

Since the only policies which may incur delay are p_2 and p_1, we will only consider them for this analysis. The other policies can hence be serviced according to any default discipline (FCFS or RND) without loss of performance or business value.

The decision problem the PDP has to solve when faced with a number of policies in TPQ requesting more resources than available (here server units) is to determine which policies to grant resources to, i.e. allow to execute, and which policies it will delay hoping that enough resources

will be freed. Delaying a triggered policy can lead to a violation of SLOs and violating an SLO can cause penalties paid to the $\mathcal{SP}$ customers. The aim of $\mathcal{SP}$ is to configure its PDP's decision algorithm so as to reflect the goal of maximizing the business profit function Ψ defined in eq.5.1.

5.7.1 The Business aware TPQ scheduling decision problem

Delaying p_1 or p_2 can lead to the violation of the availability SLO (slo_{av} in listing 26). slo_{av} is defined over the monthly average availability of the web application to end customers. Based on listing 24 , availability (= $ws.av$ in listing 26) of each SLA instance is defined as the fraction of successful service requests to the fraction of total service requests of end customers computed over a month time window.

***Definition**: monthly availability of a web service*

$$ws.av = \frac{|\text{processed requests}|_{\text{in}av_w}}{|\text{total number of requests}|_{\text{in}av_w}} \tag{5.10}$$

When confronted with a sequence of p_1 and p_2 policies in the TPQ, the PDP can utilize the information on the availability metric for the SLA to which each policy is associated in order to predict the *impact time* for each delayed policy. The impact time in this case is the time at which a violation of the availability SLO occurs.

We will make use of a *greedy approach* to the maximization of business profit. We approximate the maximization of Ψ to the minimization of the impact (i.e. loss) on Ψ for each decision cycle the PDP performs onto the TPQ.

Let P_{tpq} be the set of policies of type p_1 or p_2 that are queued in TPQ at time t_0. The PDP constructs for each policy $p_i \in TPQ$ a tuple $(p_i, r_i, I(p_i))$. r_i is the time p_i was triggered. $I(p_i)$ is an impact probability function which gives for each $t \geq 0$ the expected impact on Ψ in the case p_i is delayed t time units after the current system time t_0. Based on this sets of tuples, the PDP has to make a decision in order to minimize the impact on Ψ.

5.7.2 Mathematical model of the $\mathcal{SP}$ SLA

Predicting the impact of delaying p_1 or p_2 implies predicting the probability of violating slo_{av} in future time. This implies predicting the evolution of availability ws_{av} over time for each $p_i \in P_{tpq}$.

To do so we model the state of an SLA instance as a tuple $M/M/C_t/C_t\,|A_t|D_t$. $M/M/C_t/C_t$ models a variable capacity markovian queue where:

- λ =rate of end customer requests
- μ =service rate for a single customer request

- the number of available server slots at time t
  $C_t = ws.cp = su(rt) \times su.size()$
- no waiting queue. All requests arriving at 100% load time get lost (rejected).

A_t denotes for each t the number of granted end customer requests (sessions) during the last availability window $[t - av_w, t]$. D_t denotes the number of the denied ones. $T_t = A_t + D_t$ represents the total number of arrivals during the last availability window. Requests arrival is modeled as a poisson source as it reflects the most common type of arrivals. Exponential service times denote the time a customer remains connected to the web service. In the case of a web site this can model the time a customer spends surfing the server web site. A similar case applies to other types of servers such as audio/video streaming servers. Note that this does not contradict with the response time SLO as the response time represents the responsiveness of the web service to end customer queries during one end customer session.

The servers' capacity, in terms of the number of end customer sessions, varies within the discrete set $\{su(rt), su(rt), 2su(rt), \ldots, cp_{max}\}$.

In what follows we will focus more on policy p_2. The study of p_1 follows a similar approach.

Let N_t denote the number of end customer requests being serviced at time t. We have at any time $0 \leq N_t \leq C_t$

Note that all of A_t, D_t, T_t, C_t, and N_t can be easily obtained at runtime by defining corresponding metrics at the SLS level.

Policy p_2 is triggered when the SLA load exceeds thA. Let t_0 denote this time. In the following, when t_0 is used as a subscript the t is omitted for clarity.

Because markovian processes are memoryless, the PDP can take as input to its Impact Prediction Algorithm (IPA) the tuple $(t_0, A_0, D_0, C_0, N_0)$. The output is the probability function I. As a simplification of I, the IPA can determine the time it will take starting from t_0 until av_t drops bellow av_{min}, hence triggering an availability violation.

In spite of all the simplifications done, still remains the difficult problem of predicting the evolution of a Markovian process at transient state.

On page 78 of his book *Queueing Systems, Vol. 1*, Leonard Kleinrock, commenting on the transient solution of an $M/M/1/\infty$ queue, says 'This last expression is most disheartening. What it has to say is that an appropriate model for the simplest interesting queueing system leads to an ugly expression for the time dependent behavior of its state probabilities.'

More recently, Sharma describes in his book *Markovian Queues* [113] a novel approach to the transient analysis problem. He was the first to provide the transient solution for the $M/M/\infty$,

$M/M/N$ and $M/M/2/N$ queues. Sharma states that for higher order queues the problem becomes much more complicated to be handled by currently known Algebraic techniques. This problem is still unsolved.

In order to make the policy decision-making process business aware, we will attempt to find an acceptable solution to the transient analysis of a $M/M/C_t/C_t$ queue that still respects the short time requirement for the PDP' decision. The decision making problem is further complicated by the fact that the PDP is concerned not only with predicting when the servers become fully loaded, but also with the prediction of when after that time the SLA availability drops bellow its minimum contracted value.

The next subsections present two prediction functions and their corresponding impact minimization algorithms, which we have implemented and for which we provide performance results in section 5.9.

5.7.3 Predicting the First Time to Degradation / Violation

As a first approximation we divide the prediction process into two phases, the *fill up* phase and the *time to violation* phase. In the first phase, the system starts with configuration $(t_0, A_0, D_0, C_0, N_0)$ and evolves to the new configuration $(t_f, A_f, D_f = D_0, C_f = C_0, N_f = C_0)$, where t_f represents the time when the SLA server units reach full load ($N_f = C_0$). During interval $[t_0, t_f]$ it is expected that no service request denial occurs as the SLA can still handle more load. The time to violation phase starts at time t_f and lasts until reaching the violation of the availability SLO (slo_{av}). It is described by configuration $(t_v, A_v, D_v, C_v, N_v)$, where

$$\frac{A_v}{A_v + D_v} \overset{\simeq}{\leq} av_{min}$$

Given this information, the PDP can predict that if it delays that p_2 instance the corresponding SLA is expected to experience a violation of slo_{av} at time t_v, i.e. after a duration of $t_v - t_0$ from the current evaluation time.

The problem is hence amenable to providing an approximate but realtime solution of the time to fill up and time to violation phases.

Fill up phase - predicting t_f

The computation of t_f is based on the average behavior of $M/M/C_t/C_t$.

The incoming flow λ of end customer requests is subdivided into two sub flows. A first sub flow of rate $\mu \times N_0$ keeps busy the N_0 occupied server slots. This is because their aggregate average

service rate is $\mu \times N_0$. The remaining sub flow is then

$$\lambda' = \lambda - \mu \times N_0$$

λ' constitutes the new flow of incoming connections for the servers of rank $> N_0$ (after a simple reordering of server slots).

We now consider this new set of server slots separately. At time t the number of connected customers is N'_t. At time $t + dt$ this number will increase by dN'_t where:

$$dN'_t = \lambda' dt - \mu N'_t dt \Rightarrow \frac{dN'_t}{dt} + \mu\, N'_t = \lambda' \tag{5.11}$$

We define $\rho = \frac{\lambda}{\mu}$ and $\rho' = \frac{\lambda'}{\mu} = \rho - N_0$. With the condition $N'_0 = 0$ we get:

$$\Rightarrow N'_t = \rho'(1 - e^{-\mu t}) \tag{5.12}$$

By putting:

$$t_f = t_0 + t'_f \tag{5.13}$$

t'_f is then the solution in t of $N_t = C_0$. Hence,

$$t'_f = \frac{-1}{\mu} ln\left(1 - \frac{C_0}{\rho'}\right) \Rightarrow t'_f = \frac{1}{\mu} ln\left(\frac{\rho - N_0}{\rho - N_0 - C_0}\right) \tag{5.14}$$

Interestingly, this function is independent of t_0 and is related only to N_0, C_0, λ, and μ.

As a side effect of the approximation, this function returns a positive result only if

$$1 - \frac{C_0}{\rho'} > 0 \Rightarrow \rho > C_0 + N_0$$

This indicates that the prediction function we just developed is expected to work when the corresponding SLA instance is using only a small percentage of the maximum number of server units that can be allocated to it.

In the following we will refer to the formula of equation 5.14 as the *FTD function*, for First Time to Degradation.

Time to violation phase - predicting t_v

Following the same reasoning, we assume that λ gets divided into two sub flows. The first sub flow of rate $\mu \times C_0$ works on keeping all server units busy. The second sub flow represents the

loss flow and serves to count down towards availability violation.

As in the first case, we define our modified incoming flow as:

$$\lambda'' = \lambda - \mu \times C_0 \text{ , and } \rho'' = \frac{\lambda''}{\mu} = \rho - C_0$$

All customers in the poisson flow of rate λ'' get rejected. Hence, on average availability is expected to be violated at $t_v = t_f + t'_v$ where:

$$av_{min} = \frac{A_f + \mu \times C_0 t'_v}{T_f + \lambda \times t'_v} \Rightarrow t'_v = \frac{T_f \times av_{min} - A_f}{\mu \times C_0 - \lambda av_{min}}$$

Where $T_f = T_0 + \lambda t'_f$ and $A_f = A_0 + \lambda t'_f$.

$$\Rightarrow t'_v = \frac{T_0(av_{min} - av_0) - \lambda(1 - av_{min})t'_f}{\mu \times C_0 - \lambda av_{min}} \tag{5.15}$$

Let $\triangle av = (av_{min} - av_0)$. The expected absolute time t_v at which a violation of the availability SLO will occur is given by:

$$\begin{aligned}
t_v &= t_f + t'_v = t_0 + t'_f + t'_v \\
&= t_0 + t'_f + \frac{T_0(av_{min} - av_0) - \lambda(1 - av_{min})t'_f}{\mu \times C_0 - \lambda av_{min}} \\
&= t_0 + \frac{1}{\mu \times C_0 - \lambda av_{min}} \times [T_0(av_{min} - av_0) + ((\mu \times C_0 - \lambda av_{min}) - \lambda(1 - av_{min}))\, t'_f] \\
&= t_0 + \frac{1}{\mu \times C_0 - \lambda av_{min}} \times \left(T_0(av_{min} - av_0) + (\mu \times C_0 - \lambda)\, t'_f\right) \\
&= t_0 + \frac{1}{\mu\,(C_0 - \rho\, av_{min})} \times \left(T_0(av_{min} - av_0) + (\mu \times C_0 - \lambda)\, t'_f\right) \\
&= t_0 + \frac{1}{(C_0 - \rho\, av_{min})} \times \left(\frac{T_0}{\mu}(av_{min} - av_0) + (C_0 - \rho)\, t'_f\right) \\
&= t_0 + \frac{1}{(C_0 - \rho\, av_{min})} \times \left(\frac{T_0}{\mu}(av_{min} - av_0) + (C_0 - \rho)\, \frac{1}{\mu} ln\left(\frac{\rho - N_0}{\rho - N_0 - C_0}\right)\right) \\
&= t_0 + \frac{(C_0 - \rho)}{(C_0 - \rho\, av_{min})} \times \left(\frac{T_0}{\mu}\frac{(av_{min} - av_0)}{(C_0 - \rho)} + \frac{1}{\mu} ln\left(\frac{\rho - N_0}{\rho - N_0 - C_0}\right)\right) \\
&= t_0 + \frac{(C_0 - \rho)}{(C_0 - \rho\, av_{min})} \times \left(\frac{T_0 \times \triangle av}{\mu\,(C_0 - \rho)} + \frac{1}{\mu} ln\left(\frac{\rho - N_0}{\rho - N_0 - C_0}\right)\right) \\
&= t_0 + \frac{(C_0 - \rho)}{(C_0 - \rho\, av_{min})} \times \left(\frac{T_0 \times \triangle av}{\mu\,(C_0 - \rho)} - \frac{1}{\mu} ln\left(\frac{\rho - N_0 - C_0}{\rho - N_0}\right)\right) \\
&= t_0 + \frac{(C_0 - \rho)}{\mu\,(C_0 - \rho\, av_{min})} \times \left(\frac{T_0 \times \triangle av}{(C_0 - \rho)} - ln\left(\frac{\rho - N_0 - C_0}{\rho - N_0}\right)\right)
\end{aligned}$$

We hence obtain:

$$t_v \quad = \quad t_0 + \frac{(C_0 - \rho)}{\mu\,(C_0 - \rho\, av_{min})} \times \left(\frac{T_0 \times \triangle av}{(C_0 - \rho)} - ln\left(1 - \frac{C_0}{\rho - N_0}\right)\right) \tag{5.16}$$

□

t_v represents, on average, the time at which a violation is expected to occur if meanwhile p_2 is not granted the execution privilege. It does not necessarily reflect the actual time of first violation at runtime.

It is interesting to note that this formula can be computed in O(1) if the values of $\{C_0, T_0, N_0, av_0\}$ are available. Fortunately, the metrics instantiated from the developed $\mathcal{SP}$ SLS (listing 34) can provide this information instantly at runtime. This has a definite advantage at runtime over any formula which predicts the exact transient evolution of an $M/M/C_t/C_t\,|A_t|D_t$ queue, even though such a formula has not been discovered yet [113].

In the following we will refer to the formula of equation 5.16 as the *FTV function*, for First Time to Violation.

Figure 5.2 shows the actual FTD and FTV times for the availability metric of two $\mathcal{SP}$ SLA instances (taken from $\mathcal{PS}$ output).

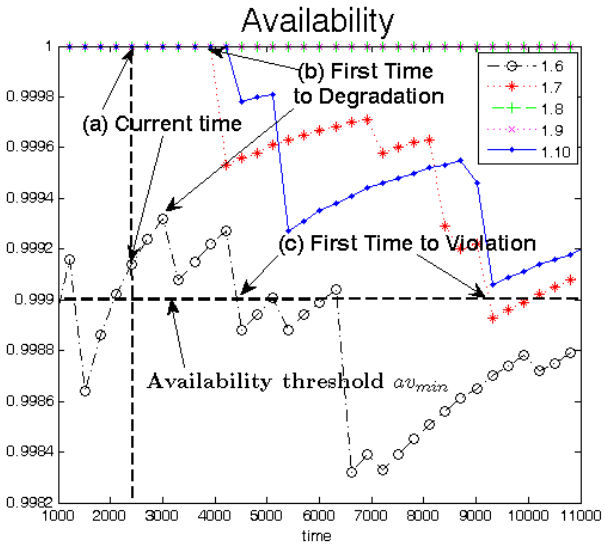


Figure 5.2 Example of actual FTD and FTV for two availability metrics

5.7.4 Impact Minimization Scheduling Algorithms

Provided with an approximation for t_f and t_v, the PDP can be configured to use several possible algorithms for the runtime minimization of impact on the business profit function Ψ. In this work we developed three different algorithms the performance of which is evaluated through simulations in section 5.9.

First, for each triggered policy $p_i \in SLA_{p_i}$, the PDP will create a tuple (p_i, SLA_{p_i}, tt_{p_i}, td_{p_i}, tv_{p_i}, Pn_{pi}), where:

- tt_{p_i} corresponds to the triggering time of p_i,
- td_{p_i} the time of the next service degradation phase (corresponds to t'_f in eq.5.13),
- tv_{p_i} the expected time of availability SLO violation (t_v in eq.5.16) in case p_i is delayed,
- and Pn_{p_i} which is the penalty incurred based on the rules defined in SLA_{p_i} (one of $\{p_4, p_5, p_6\}$ depending on the runtime state of SLA_{p_i}).

the PDP has, among other possibilities, the following set of different scheduling algorithms to select from the TPQ the next policy to execute:

- Select the one with the first(lowest) time to violation, $Min(tv_{p_i})$. We call this algorithm FTVF, for First Time to Violation First.
- Select the one with the first(lowest) time to degradation, $Min(td_{p_i})$. We call this algorithm FTDF, for First Time to Degradation First.
- Select the one with highest penalty first, $Max(Pn_{pi})$. We call this algorithm HFPF, for Highest First Penalty First.

This selection is applied to those policies whose action part can be satisfied in terms of resource availability. Policies whose actions require more resources than available are delayed for the next TPQ iteration.

Next, we will use simulations to evaluate the performance of the proposed algorithms and study how they compare to the default FCFS and RND scheduling.

5.8 Generic $\mathcal{SP}$ SLA simulation package

This package (figure 5.3) was built as a simulation instance which we run over $\mathcal{PS}$.

The $\mathcal{SP}$ generic SLS of figure 34 was implemented as a single class descendent of $\mathcal{PS}$ class GSLA. The class contains two instances of the class Role implementing $\mathcal{SP}$ and $\mathcal{C}$ roles respectively. The same hierarchy is constructed for policy groups, policies, metrics, and events. Each $\mathcal{SP}$ role has a serverGroup instance which manages a set of server units obtained from a serverPool component. A Poisson traffic source is attached to each serverGroup and is used to simulate session requests of end users. At the reception of each session request, the severGroup object tosses an exponential random number to simulate an exponential service time.

Almost all communications between the simulation components are done via events. The event service allows any component to register as a source of a given event type. Time events (timeout counters) are also supported as a special event type. Another component can register as a listener to the same event type from that event source and the event service manages this relationship. The

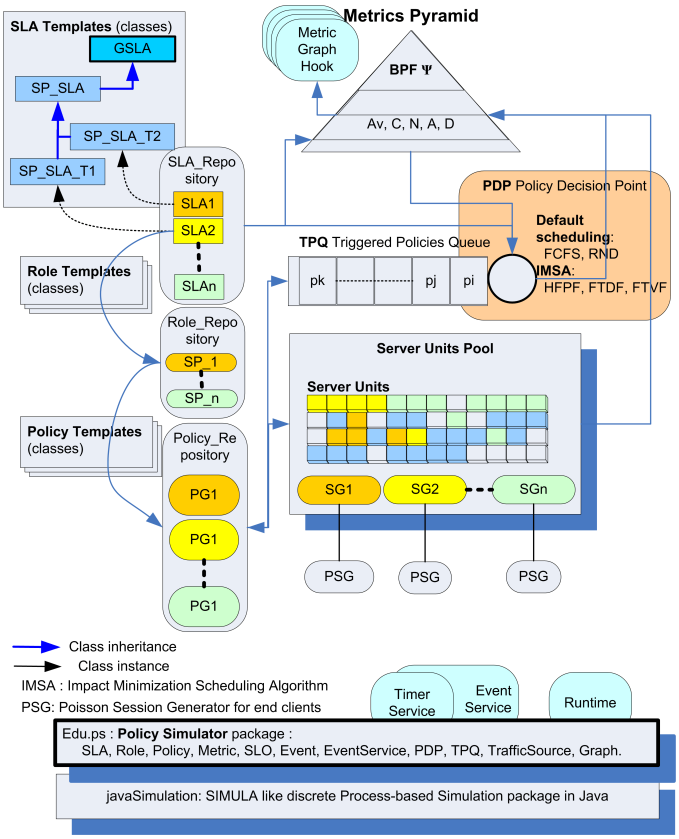


Figure 5.3 $\mathcal{SP}$ testbed over $\mathcal{PS}$

server pool, for example, generates an event each time it receives, accepts, denies or terminates a session. Leaf metrics propagate information they receive from server pool events and other components up to higher level metrics ($A_t, D_t, T_t, Av_t, NP_t, N_t, C_t, etc.$) until reaching the overall business profit function Ψ.

Finally, graph components have been hooked to several metrics to report their evolution in time. MATLAB has been used as a graph plotter because of the significant large size of the generated graph files.

5.9 Simulation Results

In order to validate the policy based implementation of the generic $\mathcal{SP}$ SLS solution and, more importantly, to test the performance of each of the impact minimization scheduling algorithms (HFPF, FTDF, and FTVF) against the basic FCFS and RND, we generated an acceptable number of different simulation instances and analyzed their outputs.

We conducted a number of 330 simulations grouped into batches of five for the five scheduling

algorithms (RND, FCFS, HFPF, FTDF, and FTVF), making a total 66 batches. We used three P4 1.6GH Windows machines, two Sun OS SUNW-Ultra-4 300MHZ machines, and two Sun Ultra 60 (512 MB RAM, 450 MHz) Solaris 8 machines. The simulations run in a total cpu time of ∼245 days with a median runtime per simulation of 12.11 hours.

The simulations were selected from a spectrum of 960 inputs generated by varying a subset of the $\mathcal{SP}$ SLS parameters through ranges of values. Although we selected our use case to be as simple as possible, we still had to deal with more than a two dozen parameters for each simulation. These included the simulation life time (three and six months were used), time granularity (one time unit was used to equal one second), TPQ scheduling algorithm (RND, FCFS, HFPF, FTDF, FTVF), number of SLA types (with each type determined by tuple (cp_{max}, a, rt, $rt.ref$, av_{min}, av_w, λ, μ, $su(rt)$, penalties $\{r_1, r_2, r_3\}$, thA, thR and availability probe interval)), number of instances of each SLA type, and server pool capacity.

Compared to how simple the $\mathcal{SP}$ SLA is, this study gives a practical example of how complex testing and optimizing a policy-based management solution can be.

Figure 5.4-a summarizes the relative performance of each of the studied TPQ scheduling algorithms. The performance of an algorithm is equal to the business profit Ψ it generates. Each slice gives the percentage of time each algorithm performed best compared to the other ones. The inner doughnut slices give the actual number of batches where this happened. As a first observation, it appears that none of the algorithms performed best all the time. HFPF performed best 67% of the time, which is a considerable percentage. Second in the rank is FTVF with 24% , followed by FTDF and FCFS with 18% , and finally RND performing best in 9% of the total number of conducted simulations. The sum of these percentages is greater than 100% because there are cases where different algorithms produced the same highest business profit.

Figure 5.4-f traces, for all simulations batches, the relative performance gain of the best Scheduling Algorithm (SA) compared to FCFS. The gain is computed as:

$$\frac{Max(\Psi_{SA}) - \Psi_{SA}}{|\Psi_{SA|}}\% \tag{5.17}$$

In this figure, a 0% value means an equal performance with FCFS, which occurs 18% of the time (5.4-a). The highest difference was in batch 3 in which HFPF performed best and produced 1000% better than FCFS. In batch 59 FTVF generated the best net profit with a value 780% higher than the one generated by FCFS. The average gain is 119% with a median value of 34%. It is interesting to note that graphs similar to 5.4-a were obtained when comparing the other scheduling algorithms. For example, in batch 18 FTVF performs 1800% better than HFPF, and in batch 55 HFPF performs 1964% times better than FTVF! All these results are summarized in figures 5.4-[g-i]. Figure 5.4-c tells that the percentage of times only one single SA scored best is

82% while in 18% of the times only more than one SA scored the best performance. This shows the importance of collecting knowledge about which algorithm is expected to perform best before actually using it.

Another way to compare the performance of each SA is to track the number of SLAs which failed to continue their execution due to the occurrence of three successive availability violations (clause 4-b of the $\mathcal{SP}$ SLA). The termination of an SLA represents a considerable loss as it implies a cut in customer periodic payments. Figure 5.4-b gives the percentage of times an SA has generated the least loss in terms of the number of dropped SLAs. The service provider can use this business level metric to decide which SA to use instead of using the net profit metric.

Interestingly, figures 5.4-a and 5.4-b, however different, still have a strong similarity. An SLA is lost when it experiences three successive availability violations within the same availability window, i.e one month (figure 24, clause 4-b). Algorithm FCFS scores the lowest number of dropped SLAs 33% of the time while it achieves best business profit only 21% of the time. RND, HFPF, and FTDF also saw their score reduced with RND witnessing the highest relative decrease by going down from 20% to only 9%. Only FTVF generated the smallest loss in SLAs 21% of the time while it achieved best profit 24% of the time which represents an increase of 3%. This property distinguishes FTVF from the other algorithms and allows us to deduce that when FTVF manages to keep a higher number of SLAs alive it also manages to keep the number of experienced violations to the minimum. The random algorithm RND on the other hand performed worst in this regard. In all the 11% cases ($= 20 - 9$) where RND generated a lowest number of dropped SLAs that is shared with at least another SA, RND failed to give a better net profit. This suggests that the random scheduling has to be avoided whenever possible as it fails to manage the TPQ better than the other algorithms.

The analysis of the output graphs for the business profit metric shows that although some algorithms managed to keep a higher number of SLAs alive they still failed to provide best net profit because the remaining SLAs experienced recurrent availability violations causing penalties on the overall business profit.

Figure 5.4-d shows that in 65% of the time, i.e. in 43 batches out of the 66, only one single SA generated the lowest loss in SLAs. Figure 5.4-e shows that 71% of this number, i.e. 30 batches, has been scored by HFPF while FTVF scored a unique lowest SLA loss in only 4 batches. A closer analysis showed that when an algorithm generates a distinctive low loss in SLAs it automatically generates a distinctive highest net profit. This means that the batches related to figure 5.4-e do all belong to the |abs max| area of figure 5.4-c (the 82% pie).

RND and FCFS together performed best 30% of the time. This result shows that there is still room for better scheduling algorithms that are yet to be developed. This case was theoretically

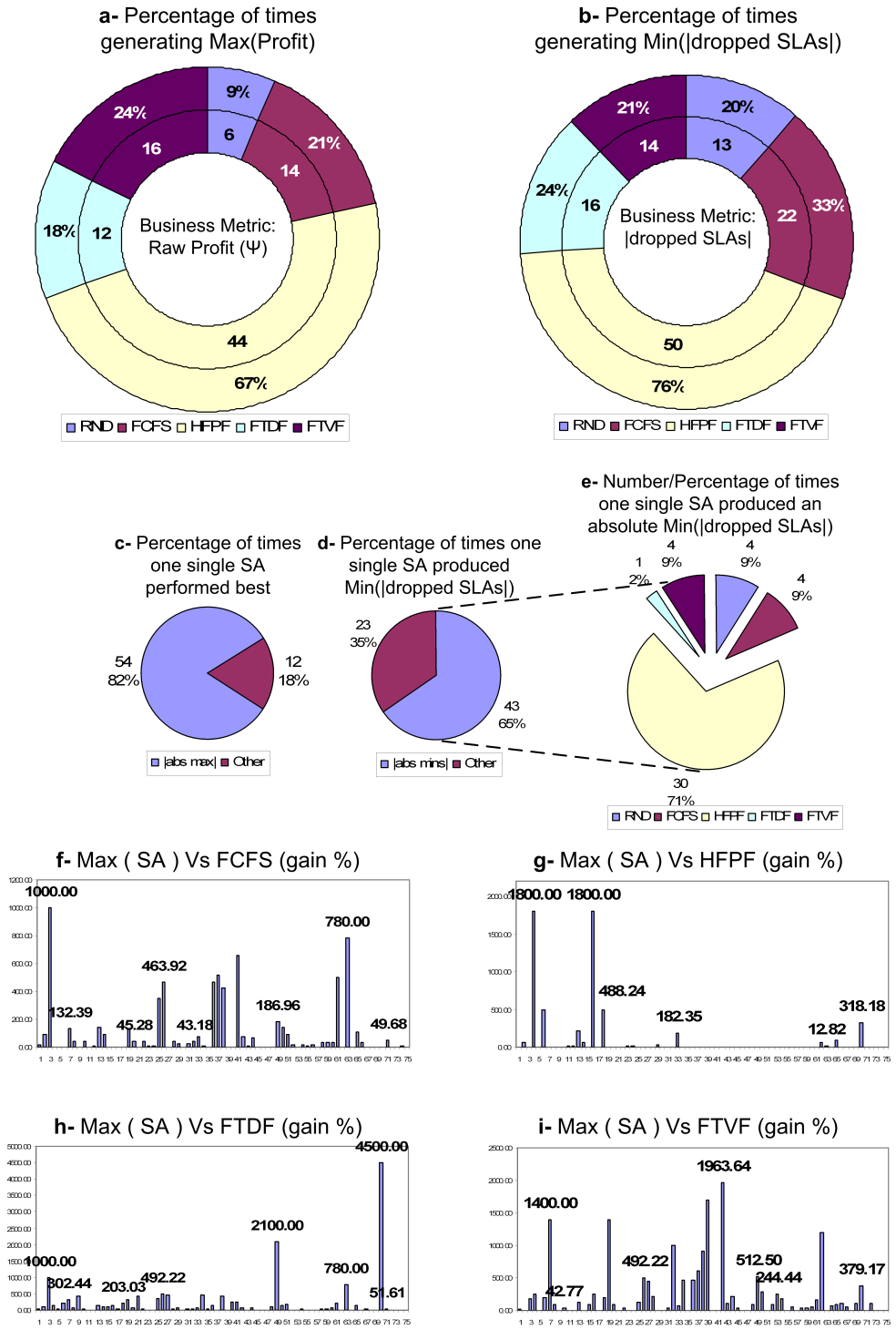


Figure 5.4 Performance results of $\mathcal{SP}$ simulations over $\mathcal{PS}$

predictable as all of HFPF, FTDF, and FTVF make use of a greedy technique that seeks to maximize the business profit Ψ by minimizing the local impact on it by individual policy actions. It is also known that greedy techniques in general have no assurance as to the generation of global optima. Hence, the performance of RND and FCFS supports this theoretical prediction by practical numbers.

Using these two business level metrics the $\mathcal{SP}$ can decide which algorithm to use based on whether it gives greater importance to the mere net profit or rather to keeping the maximum number of SLAs running. Additional business metrics can be taken into consideration, such as the number of experienced penalties, and the selection process can grow more complex than the study presented in here.

The results obtained clearly demonstrate the importance of conducting a simulation before deciding which scheduling algorithm to use. Given the number of parameters to tune, even for this simple SLA case, it was computationally infeasible to determine beforehand the best parameters which lead the best business profit. However, given a certain initial set of SLA types that the $\mathcal{SP}$ intends to advertise and a hardware configuration of the server units pool, it is possible to conduct extensive simulations to determine which scheduling algorithm is best and what SLA admission control policy to use for each SLA type.

5.10 Conclusion

This chapter illustrated the usage of our approach to business-driven policy management through a use case in application hosting environments. The use case showed how it is impossible to maximize the service provider's business profit by considering a policy-based solution wherewith all aspects of policy specification and behavior is fixed in advance at a preliminary off-line refinement process. The consideration of runtime policy dynamics helped in significantly enhancing the generated profit.

The refinement process involved an iterative activity the output of which is a set of metrics and low-level quality assurance policies structured into roles. The *static analysis* phase served in detecting and resolving static inconsistencies (deadlocks, loops, unreachable states, and erratic behaviors) in the generated SLS as well as discovering additional constraints important for the runtime stability of the SLS. In the *dynamic analysis* phase, we attempted to bridge the gap between low-level management actions and the high-level business profit of the service provider. This faced us with the difficult problem of the realtime prediction of the transient state of a variant of an $M/M/C_t/C_t$ queue. We solved this problem through mathematical approximation and used it to derive the policy scheduling algorithms FTDF and FTVF. We also proposed a third

algorithm named HFPF which uses runtime SLO states to decide of the runtime prioritization of the triggered policies.

Using $\mathcal{PS}$, the policy simulator tool we developed for the simulation of policy management solutions, all three algorithms were implemented along with two other default ones (FCFS and RND). For statistical significance, we ran a considerable number of simulations to benchmark the performance of these algorithms. The simulations showed interesting results perhaps the most important of them is that no single algorithm outperformed the others at all times. This confirms, at least for the $\mathcal{SP}$ use case, the importance of simulations before choosing a runtime policy management mechanism for a particular SLA type.

Chapter 6

Conclusion

This thesis investigated the emerging field of business-driven management of information systems. In this management paradigm, the focus is made on linking the details of low-level actions and configurations with the high-level objectives that were initially set out for that system. As information systems continue their growth in complexity and gain an increasingly important place within all aspects of enterprize activity, the need to design methods that make an information system behave according to the high-level intentions, as well as to automatically adapt to their changes, has become more needed than ever.

We identified early in the thesis that for such a management paradigm to be achieved, there is a need for the support of the specification of high-level business goals and Service Level Agreements, the refinement of these goals and SLAs into low-level configuration actions, as well as the validation of these low-level actions against the originally specified high-level behavior. In addition, we demonstrated how policy-based management represents the key to implementing business-driven management due to the potential it gives to alleviate the configuration and maintenance cost of complex computerized systems by providing a management model that separates implementation functionality from the behavioral logic.

In what follows, we will summarize our contributions, identify their limitations, and suggest action plans for future work.

6.1 Summary of contributions

In addition to the review and classification of research efforts in the specification and refinement of SLAs, policies, and business goals, this thesis contributed to the business-driven management of information systems in the following aspects:

SLA specification

As the ability to specify and manipulate Service Level Agreements (SLA) constitutes an important component of business-driven management, we proposed the generic GSLA information model, and a corresponding XML schema GXLA, which allows a policy-driven specification of SLAs.

The GSLA models a contract between two or more parties linked through to (a) service relationship(s) and sets clear measurable common understanding of the role each party agrees to adhere to. The model is role-based and has a native support of policies. A *party role* represents a set of objectives and rules which define the minimal service related expectations, constraints, and obligations it has in relation to other roles. The behavior of a party is ultimately modeled through policies.

Business-Driven Management Framework

The Business-Driven Management Framework (BDMF) we presented extends traditional policy-based management with a wider decision ability that is informed and driven by the business goals and contractual obligations of the service provider. The main idea that the BDMF stresses is the need of a business-level policy management to back the traditional low-level policy dynamics in order to enhance policy performance and achieve a maximization of business goals.

The BDMF features a high-level business and service-driven layer on top of a policy-based resource control layer. The linkage between the two layers is assured in two ways. First, a policy-based refinement engine, supported by the GSLA, ensures the off-line derivation of low-level policy rules from high-level requirements that are expressed in terms of SLAs and business objectives. Second, a business profit maximization engine provides decision support that seeks to maximize the business goals at system runtime whenever it is deemed appropriate.

The policy control loop of the BDMF identifies the traditional view of the dynamics of policy execution where policy is determined beforehand prior to system execution through a refinement process. The high-level policy business loop, on the other hand, represents our claim of the additional need of controlling policy dynamics at runtime not through a default treatment but rather through a more elaborate engine that seeks to maximize business profit based on runtime context.

Policy Analysis

We presented a comprehensive review and classification of efforts conducted in the refinement and analysis of policies. The study we conducted on policy specification formalisms showed that they have reached an acceptable level of maturity. This lead us to conclude that

the next important step towards the enabling of business-driven management, in addition to the need for elaborate SLA specification formalisms and Business-driven management frameworks, lies in the researching of effective policy refinement and analysis techniques.

We identified a set of new policy inconsistency types and described how they can be detected in the general case. In addition, we identified the need to consider policy runtime dynamics for the purpose of policy inconsistency detection and resolution; as well as for policy optimization in terms of maximizing the original high-level business goals.

Policy simulator tool

The $\mathcal{PS}$ Policy Simulator tool represents a prototype implementation of the BDMF framework and is in a sense the product of the research conducted in this thesis. It has been developed in order to serve mainly at the policy dynamic analysis phase. The intent is to make of it a general purpose policy analysis tool.

$\mathcal{PS}$ is designed in a way to support a business-driven management that has policy support at its core. It offers a discrete simulation environment based on the process interaction world view and models policies after the Event-Condition-Action paradigm.

Detailed business and policy-driven refinement use case

The use case we investigated, and which is related to a generic application hosting SLA, illustrated the usage of our approach to business-driven policy management. The use case showed how it is not possible to maximize the service provider's business profit by considering a policy-based solution wherewith all aspects of policy specification and behavior is fixed beforehand at a preliminary off-line refinement process. The investigation of runtime policy dynamics helped in significantly enhancing policy performance and the overall business profit.

The refinement process involved an iterative activity the output of which was a set of metrics and low-level quality assurance policies structured into roles. The *static analysis* phase served in detecting and resolving static inconsistencies in the generated SLS as well as discovering additional constraints important for the runtime stability of the SLS. In the *dynamic analysis* phase, we attempted to bridge the gap between low-level management actions and the high-level business profit of the service provider. This faced us with the difficult problem of the realtime prediction of the transient state of a variant of an $M/M/C_t/C_t$ queue. We solved this problem through mathematical approximation and used it to derive several policy scheduling algorithms that proved to be efficient. The simulations conducted using $\mathcal{PS}$ showed interesting results perhaps the most important of them is that no single algorithm outperformed the others at all times; while at the same time significant performance differences were recorded for the majority of simulation instances. This confirmed

the importance of conducting simulations before choosing a runtime policy management mechanism for a particular SLA type.

The use case also showed a real example of the occurrence of the *policy loop* type of inconsistency. The resolution of this conflict required mathematical reasoning over the triggering conditions of the involved policies that is not easy to automate. This proved that this type of inconsistency is difficult to detect and resolve in the general case. This claim is strengthened by the fact that this inconsistency was only discovered at simulation time.

The main result of the use case was in proving the need to consider the investigation of not only the correctness of a quality assurance policy-based solution, but that the performance aspect of this solution also needs to receive an appropriate degree of focus.

6.2 Limitations and Future Work

The work conducted in this thesis presents several drawbacks. These are identified in the following list and each time a weakness is stated a set of remedy actions are suggested.

Limited SLA specification

The GSLA information model and GXLA XML schema we presented are still at the specification level. No actual tool or elaborate user interface has been developed yet in order to make use of them. In addition, the model relies on the existence of a good policy specification language.

The following actions are required to bring the GSLA to practical usage:

- First, there is a need to link the GXLA schema with an existing policy schema.
- An interface needs to be developed in order to help with the easy editing of GXLA specifications
- To avoid XML verbosity, there is a need to develop a language-based notation for the GSLA, possibly by extending an existing policy language specification such as that of Ponder.

Policy inconsistencies

The number of policy inconsistency types that were identified in this thesis showed that not only is there a large variety of inconsistencies that a policy designer needs to be concerned about, but that there is also a need to consider researching related fields where these consistencies have been previously confronted. We believe that many of the identified inconsistencies are not specific to the policy discipline per se. Hence, the development of

tools and specialized libraries to assist in policy inconsistency detection and resolution needs to receive more interest by the policy research community.

Policy simulator

Several features can be added to $\mathcal{PS}$ in order to enhance its operation and usability. For instance, there is a need for a policy and SLA editor front end to facilitate the specification of $\mathcal{PS}$ components: SLAs, SLSs, SLOs, metrics, events, and policy rules. The development of add-on tools to assist in the refinement process would also be of great value. In addition, the interfacing of $\mathcal{PS}$ with some of the formal policy refinement and analysis tools that have been developed recently is particularly desirable.

Furthermore, providing $\mathcal{PS}$ with a more sophisticated event service, which supports more than simple event reuse, would allow the specification of a wider range of policy rules. Finally, the current $\mathcal{PS}$ implementation supports only one PDP. A possible extension to $\mathcal{PS}$ is to include other PDP architectures with distributed policy decision and policy enforcement functions.

6.3 Concluding remarks

This thesis demonstrated the important potential which service level agreements and especially policy-based management possess within the context of the growing interest of the information systems management community in elevating management solutions from bits to business value [114].

While research on service level agreements is still at the specification level, work on policy-based management has shifted recently from policy specification to policy analysis and refinement. Moreover, the use of policies is no longer restricted to the security domain and more and more research is being conducted in applying policies for quality assurance management.

In addition to emphasizing the importance of designing appropriate business-driven models for information systems management and the role of policy analysis and refinement tools, this thesis points at the need to consider not only the consistency and conflict-free aspects of policy, but also the performance and business-profit maximization criteria. The presented ideas as well as the developed policy simulator tool provide a good starting point for this effort.

Appendix A

Abbreviations

BDMF Business Driven Management Framework.

BGP Border Gateway Protocol.

CBR Case-Based Reasoning.

CIM Core Information Model.

CIO Chief Information/Investment Officer.

DiffServ Differentiated Services.

DMTF Distributed Management Task Force.

E2E End-to-End

ECA Event-Condition-Action.

GSLA Generalized Service Level Agreement.

IDPR Inter-Domain Policy Routing

IDRP Inter-Domain Routing Protocol

IETF Internet Engineering Task Force.

IntServ Integrated Services.

IT Information Technology

MANET Mobile Ad-hoc NEtwork.

MBO Management by Business Objectives.

OCL Object Constraint Language.

P2P Peer to Peer.

PCIM Policy Core Information model

PDL Policy Description Language.

PDP Policy Decision Point.

PEP Policy Enforcement Point.

Policy a system goal or rule of behavior.

$\mathcal{PS}$ Policy Simulator.

QoS Quality of Service.

RBAC Role Based Access Control.

RPSL Routing Policy Specification Language

SE Service Element.

SLA Service Level Agreement.

SLS Service Level Specification.

SPO Service Package Objective.

TMF Tele-Management Forum www.tmforum.org.

UML Unified Modeling Language.

VoD Video On Demand.

VPN Virtual Private Network

WMC Wireless Management Community.

Appendix B

GXLA Schema

Listing 35 GXLA Schema

```
<?xml version="1.0" encoding="UTF-8"?>
 <xs:schema targetNamespace="http://lip6.fr/namespace"
 xmlns:sql="urn:schemas-microsoft-com:mapping-schema"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:gxla="http://lip6.fr/namespace"
 elementFormDefault="qualified"
 attributeFormDefault="unqualified">
    <!--                    -->
    <!--Global  -->
    <!--                    -->
 <xs:complexType name="GXLAType">
  <xs:sequence>
   <xs:element ref="gxla:Scope" maxOccurs="unbounded"/>
   <xs:element ref="gxla:Party" maxOccurs="unbounded"/>
   <xs:element ref="gxla:ServicePackage" maxOccurs="unbounded"/>
   <xs:element ref="gxla:Role" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string"/>
 </xs:complexType>
 <xs:element name="GXLA" type="gxla:GXLAType">
 </xs:element>
 <!-- Schedule -->
 <xs:complexType name="ScheduleType">
  <xs:sequence>
   <xs:element name="Temporal">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="Interval">
       <xs:complexType>
        <xs:sequence>
         <xs:element name="begin" type="xs:dateTime"/>
         <xs:element name="end" type="xs:dateTime"/>
        </xs:sequence>
       </xs:complexType>
      </xs:element>
      <xs:element name="period" minOccurs="0" maxOccurs="unbounded">
```

```
      <xs:complexType>
       <xs:sequence>
        <xs:element name="MonthOfYear" minOccurs="0">
         <xs:simpleType>
          <xs:restriction base="xs:string">
           <xs:maxLength value="12"/>
           <xs:pattern value="[0|1]+"/>
          </xs:restriction>
         </xs:simpleType>
        </xs:element>
        <xs:element name="DayOfMonth" minOccurs="0">
         <xs:simpleType>
          <xs:restriction base="xs:string">
           <xs:maxLength value="64"/>
           <xs:pattern value="[0|1]+"/>
          </xs:restriction>
         </xs:simpleType>
        </xs:element>
        <xs:element name="DayOfWeek" minOccurs="0">
         <xs:simpleType>
          <xs:restriction base="xs:string">
           <xs:maxLength value="7"/>
           <xs:pattern value="[0|1]+"/>
          </xs:restriction>
         </xs:simpleType>
        </xs:element>
        <xs:element name="TimeBegin" type="xs:time" minOccurs="0"
         maxOccurs="unbounded"/>
        <xs:element name="TimeEnd" type="xs:time" minOccurs="0"
         maxOccurs="unbounded"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
 <xs:attribute name="Name" type="xs:string" use="required"/>
</xs:complexType>
<xs:element name="Scope" type="gxla:ScheduleType"/>
<!--     -->
<!-- Parties   -->
<!--     -->
<xs:complexType name="ContactType"/>
<xs:simpleType name="ActionType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="Mesure"/>
  <xs:enumeration value="Bridge"/>
  <xs:enumeration value="Costumer"/>
  <xs:enumeration value="ISP"/>
 </xs:restriction>
</xs:simpleType>
<xs:complexType name="PartyDefinitionType">
 <xs:sequence>
  <xs:element name="GroupID" type="xs:ID"/>
  <xs:element name="GroupName" type="xs:string"/>
  <xs:element name="GroupPriority">
   <xs:simpleType>
    <xs:restriction base="xs:integer">
```

```
      <xs:enumeration value="0"/>
      <xs:enumeration value="1"/>
      <xs:enumeration value="2"/>
      <xs:enumeration value="3"/>
      <xs:enumeration value="4"/>
      <xs:enumeration value="5"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
   <xs:element name="UserID" type="xs:ID"/>
   <xs:element name="UserName" type="xs:string"/>
   <xs:element name="UserLastName" type="xs:string" minOccurs="0"
    maxOccurs="2"/>
   <xs:element name="Email" type="xs:string" minOccurs="0"
    maxOccurs="3"/>
   <xs:element name="UserOffice" type="xs:string" minOccurs="0"/>
   <xs:element name="Telephone" type="xs:long" minOccurs="0"
    maxOccurs="3"/>
   <xs:element name="Fax" type="xs:long" minOccurs="0"/>
   <xs:element name="Has_role" maxOccurs="unbounded">
    <xs:complexType>
     <xs:attribute name="ref" type="xs:ID"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:element name="Party" type="gxla:PartyDefinitionType"/>
<!--         -->
<!--services  -->
<!--         -->
<!--          -->
<xs:complexType name="ConstraintType">
 <xs:sequence>
  <xs:element name="Begin" type="xs:dateTime"/>
  <xs:element name="End" type="xs:dateTime"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="SP_ParameterType">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute name="Name" type="xs:string" use="required"/>
   <xs:attribute name="Type" type="xs:string" use="required"/>
   <xs:attribute name="SE_Parameter" type="xs:string"
    use="required"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
<xs:element name="Constraint" type="gxla:ConstraintType"/>
<xs:complexType name="FunctionType">
 <xs:choice>
  <xs:element name="Operand" maxOccurs="unbounded">
   <xs:complexType>
    <xs:choice>
     <xs:element name="Value" type="xs:string"/>
     <xs:element name="Function" type="gxla:FunctionType"
      maxOccurs="unbounded"/>
    </xs:choice>
    <xs:attribute name="Type" type="xs:string" use="required"/>
   </xs:complexType>
```

```
   </xs:element>
  </xs:choice>
  <xs:attribute name="Name" type="xs:string" use="required"/>
  <xs:attribute name="ResultType" type="xs:string"
   use="required"/>
 </xs:complexType>
 <xs:complexType name="DirectiveType">
  <xs:sequence>
   <xs:element name="Period">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="Begin" type="xs:dateTime"/>
      <xs:element name="End" type="xs:dateTime"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="URI" type="xs:string"/>
   <xs:element name="Frequency">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="5mn"/>
      <xs:enumeration value="Hour"/>
      <xs:enumeration value="Day"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="required"/>
  <xs:attribute name="ResultType" type="xs:string"
   use="required"/>
 </xs:complexType>
 <xs:complexType name="SystemMetricType">
  <xs:sequence>
   <xs:choice>
    <xs:element name="Function" type="gxla:FunctionType"
     maxOccurs="unbounded"/>
    <xs:element name="Directive" type="gxla:DirectiveType"
     maxOccurs="unbounded"/>
   </xs:choice>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="SR_ParameterType">
  <xs:sequence>
   <xs:element name="SystemMetric" type="gxla:SystemMetricType"/>
  </xs:sequence>
  <xs:attribute name="Name" use="required"/>
  <xs:attribute name="ref" use="required">
  </xs:attribute>
  <xs:attribute name="DirectiveMesure" use="required">
   <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:enumeration value="Fonction"/>
     <xs:enumeration value="Directive"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
 </xs:complexType>
 <xs:complexType name="ServiceRessourceType">
  <xs:sequence>
```

```
   <xs:element name="SR_Parameter" type="gxla:SR_ParameterType"
    maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="Name" use="required"/>
  <xs:attribute name="ref" use="required">
  </xs:attribute>
 </xs:complexType>
 <xs:complexType name="ServiceElementType">
  <xs:sequence>
   <xs:element name="Constraint" type="gxla:ConstraintType"
    minOccurs="0"/>
   <xs:element name="SE_Parameter" type="xs:string"
    maxOccurs="unbounded"/>
   <xs:element name="ServiceRessource" minOccurs="0"
    maxOccurs="unbounded">
    <xs:complexType>
     <xs:complexContent>
      <xs:extension base="gxla:ServiceRessourceType"/>
     </xs:complexContent>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="required">
  </xs:attribute>
  <xs:attribute name="ref" use="required">
  </xs:attribute>
 </xs:complexType>
 <xs:complexType name="ServicePackageType">
  <xs:sequence>
   <xs:element name="SP_Parameter" type="gxla:SP_ParameterType"
    maxOccurs="unbounded">
    <xs:annotation>
     <xs:documentation>Global parameters</xs:documentation>
    </xs:annotation>
   </xs:element>
   <xs:element name="ServiceElement" maxOccurs="unbounded">
    <xs:annotation>
     <xs:documentation>Services</xs:documentation>
    </xs:annotation>
    <xs:complexType>
     <xs:complexContent>
      <xs:extension base="gxla:ServiceElementType"/>
     </xs:complexContent>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="NamePack" use="required">
  </xs:attribute>
  <xs:attribute name="REF" use="required">
   <xs:simpleType>
    <xs:restriction base="xs:string"/>
   </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="CIM_Service" use="required">
   <xs:simpleType>
    <xs:restriction base="xs:string"/>
   </xs:simpleType>
  </xs:attribute>
 </xs:complexType>
```

```
<!--              -->
<!--     ROLE     -->
<!--              -->
<xs:complexType name="RoleType">
 <xs:sequence>
  <xs:element name="SLO" type="gxla:SLOType"
   maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="Name" use="required">
  <xs:annotation>
   <xs:documentation>Unique</xs:documentation>
  </xs:annotation>
 </xs:attribute>
 <xs:attribute name="GSLAPolicy" use="required">
  <xs:annotation>
   <xs:documentation>L'ensemble des politiques</xs:documentation>
  </xs:annotation>
 </xs:attribute>
</xs:complexType>
<xs:element name="Role" type="gxla:RoleType"/>
<!--     SLO          -->
<!--     PREDICAT Metrique          -->
<xs:complexType name="Predicate1Type">
 <xs:sequence>
  <xs:element name="SE_Parameter" type="xs:string"/>
  <xs:element name="Value" type="xs:string"/>
  <xs:element name="Frequency">
   <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:enumeration value="NewValue"/>
     <xs:enumeration value="5mn"/>
     <xs:enumeration value="Hour"/>
     <xs:enumeration value="Day"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="Predicate2Type">
 <xs:sequence>
  <xs:element name="SR_Parameter" type="xs:string"/>
  <xs:element name="Value" type="xs:string"/>
  <xs:element name="Frequency">
   <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:enumeration value="NewValue"/>
     <xs:enumeration value="5mn"/>
     <xs:enumeration value="Hour"/>
     <xs:enumeration value="Day"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:element>
 </xs:sequence>
 <xs:attribute name="Type" use="required">
  <xs:simpleType>
   <xs:restriction base="xs:string">
    <xs:enumeration value="less "/>
    <xs:enumeration value="greater"/>
    <xs:enumeration value="Equal "/>
```

```
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
</xs:complexType>
<!--     PREDICAT          -->
<xs:complexType name="PredicateType">
 <xs:sequence>
  <xs:element name="SP_Parameter" type="xs:string"/>
  <xs:element name="Value" type="xs:string"/>
  <xs:element name="Event">
   <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:enumeration value="NewValue"/>
     <xs:enumeration value="5mn"/>
     <xs:enumeration value="Hour"/>
     <xs:enumeration value="Day"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:element>
 </xs:sequence>
 <xs:attribute name="Type" use="required">
  <xs:simpleType>
   <xs:restriction base="xs:string">
    <xs:enumeration value="less "/>
    <xs:enumeration value="greater"/>
    <xs:enumeration value="Equal "/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
</xs:complexType>
<!--      SLO           -->
<xs:complexType name="GuarantieType"/>
<xs:complexType name="PolicyType">
 <xs:sequence>
  <xs:element name="Predicate" type="gxla:PredicateType"/>
  <xs:element name="Guarantee">
   <xs:complexType>
    <xs:complexContent>
     <xs:extension base="gxla:GuarantieType">
      <xs:sequence>
       <xs:element name="Party" type="xs:string"/>
       <xs:element name="QualifedAction" maxOccurs="unbounded">
        <xs:complexType>
         <xs:simpleContent>
          <xs:extension base="xs:string"/>
         </xs:simpleContent>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
 <xs:attribute name="Name" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="SLOType">
 <xs:sequence>
  <xs:element name="Policy" type="gxla:PolicyType"
```

```
    maxOccurs="unbounded"/>
   <xs:element ref="gxla:Constraint" minOccurs="0"/>
   <xs:element name="SE_Objetive" type="gxla:SE_ObjectiveType"
    minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="Name" use="required">
  </xs:attribute>
  <xs:attribute name="idf" use="required">
  </xs:attribute>
 </xs:complexType>
 <!--       SEOBJECTIVE          -->
 <xs:complexType name="SE_ObjectiveType">
  <xs:sequence>
   <xs:element name="Party">
   </xs:element>
   <xs:element name="Predicate1" maxOccurs="unbounded">
    <xs:complexType>
     <xs:complexContent>
      <xs:extension base="gxla:Predicate1Type">
       <xs:attribute name="Type" type="xs:string"/>
      </xs:extension>
     </xs:complexContent>
    </xs:complexType>
   </xs:element>
   <xs:element name="SR_Objective" type="gxla:SR_ObjectiveType"
    maxOccurs="unbounded">
   </xs:element>
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="required">
  </xs:attribute>
 </xs:complexType>
 <!--       SROBJECTIVE          -->
 <xs:complexType name="SR_ObjectiveType">
  <xs:sequence>
   <xs:element name="Predicate2" type="gxla:Predicate2Type"/>
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="required">
  </xs:attribute>
 </xs:complexType>
 <xs:element name="ServicePackage"
  type="gxla:ServicePackageType"/>
</xs:schema>
```

Bibliography

[1] Nicodemos Damianou. *A Policy Framework for Management of Distributed Systems.* PhD thesis, Imperial College of Science, April 2002.

[2] Dakshi Agrawal, Seraphin Calo, Kang-Won Lee, and Jorge Lobo. Issues in designing a policy language for distributed management of it infrastructures. In *IFIP/IEEE 10th International Symposium on Integrated Network Management (IM 2007): Moving from bits to business value*, pages 30–39. IEEE, May 21-25 2007.

[3] IDC Market Intelligence. http://www.idc.com,

[4] Gartner information technology research and advisory company. http://www.gartner.com,

[5] Dinesh C. Verma. *Policy-Based Networking: Architecture and Algorithms.* Technology series. Sams, 2000 edition, Novermber 14 2000.

[6] Issam Aib, Mathias Salle, Claudio Bartolini, and Abdel Boulmakoul. A business driven management framework for utility computing environments. Technical Report HPL-2004-171, Hewlett Packard Laboratories, oct-12 2004.

[7] Jonathan D. Moffet and Morris Sloman. Policy hierarchies for distributed systems management. *IEEE Journal on Selected Areas in Communications, Special Issue on Network Management*, 11(9), 11 1993.

[8] Miriam J. Maullo and Seraphin B. Calo. Policy management: an architecture and approach. In *Systems Management, 1993., Proceedings of the IEEE First International Workshop on*, pages 13–26, Los Angeles, CA, USA, 14-16 Apr 1993. IEEE.

[9] Seraphin Calo and Jorge Lobo. A basis for comparing characteristics of policy systems. In *POLICY '06: Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06)*, pages 183–194, Washington, DC, USA, June 2006. IEEE Computer Society.

[10] Morris Sloman. Policy-based management: The holy grail? Panel, 7-9 June 2004.

[11] Raouf Boutaba and Simon Znaty. An architectural approach for integrated network and systems management. *SIGCOMM Comput. Commun. Rev.*, 25(5):13–38, 1995.

[12] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Terminology for policy-based management. IETF RFC 3198, November 2001.

[13] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy core information model – version 1 specification. IETF RFC 3060, February 2001.

[14] Ed. B. Moore. Policy core information model (PCIM) extensions, rfc 3460, January 2003.

[15] Mark Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. In *16th IFIP/IEEE International Workshop on Distributed Systems Operations and Management, DSOM*, Barcelona, Spain, October 24-26 2005.

[16] John Strassner. How policy empowers business-driven device management. In IEEE, editor, *IEEE Third International workshop on Policies for Distributed Systems and Networks (POLICY'02)*, pages 214– 217, 2002.

[17] John Strassner. Policy based management, thoughts and observations from a network management perspective. Panel, IEEE Workshop on Policies for Distributed Systems and Networks (POLICY'04), June 7-9 2004.

[18] TeleManagementForum. SLA management handbook. Public evaluation 1.5, TMF, June 2001.

[19] Arosha K Bandara, Emil C Lupu, Alessandra Russo, Naranker Dulay, Morris Sloman, Paris Flegkas, Marinos Charalambides, and George Pavlou. Policy refinement for diffserv quality of service management. *IEEE eTransactions on Network and Service Management (eTNSM)*, 3(2):12, 2nd quarter 2006.

[20] Javier Rubio-Loyola, Joan Serrat, Marinos Charalambides, Paris Flegkas, and George Pavlou. A methodological approach toward the refinement problem in policy-based management systems. *IEEE Communications Magazine*, 44(10):60–68, October 2006.

[21] Dakshi Agrawal, James Giles, Kang won Lee, and Jorge Lobo. Policy ratification. In IEEE, editor, *IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2005)*, IBM Thomas J Watson Research Center, Yorktown Heights, New York, June 2005.

[22] Dakshi Agrawal, James Giles, Kang won Lee, Kaladhar Voruganti, and Khalid Filali-Adib. Policy based validation of SAN configuration. In IEEE, editor, *IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2004)*, June 2004.

[23] Issam Aib, Nazim Agoulmine, and Guy Pujolle. A multi-party approach to SLA modeling, application to WLANs. In *Second IEEE Consumer Communications and Networking Conference (CCNC)*, pages 451 – 455. IEEE, Jan 3-5 2005.

[24] Issam Aib, Mathias Salle, Claudio Bartolini, Abdel Boulmakoul, Raouf Boutaba, and Guy Pujolle. Business aware policy-based management. In *The First IEEE/IFIP International Workshop on Business-Driven IT Management (BDIM 2006), in conjunction with NOMS 2006*. IEEE, April 7 2006.

[25] K. Lougheed and Y. Rekhter. A border gateway protocol (bgp). IETF Network Working Group RFC 1105, May 1989.

[26] M. Steenstrup. Idpr: An approach to policy routing in large diverse internetworks. *J. High Speed Nets.*, page 81–105, 1994.

[27] D. Meyer et al. Using RPSL in practice. IETF Network Working Group RFC 2650, Aug 1999.

[28] C. Kunzinger. Protocol for the exchange of inter-domain routing information among intermediate systems to support forwarding of iso 8473. IETF working draft ISO 10747, Apr 1994.

[29] IETF. Policy framework. http://www.ietf.org.

[30] DMTF. CIM core model, version 2.4. DMTF White Paper, DMTF 2000, 2000.

[31] J. Strassner Intelliden R. Cohen Ntear LLC B. Moore IBM. Y. Snir, Y. Ramberg Cisco Systems. Policy quality of service (qos) information model. IETF Internet standards track protocol, Nov 2003.

[32] Y. Snir, Y. Ramberg Cisco Systems, J. Strassner Intelliden, R. Cohen Ntear LLC, and B. Moore IBM. Policy qos inofrmation model (qpim). IETF Internet Draft, Expired May 2002, Oct 2001.

[33] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss and. An architecture for differentiated services. IETF Internet Draft 2475, Differentiated Services group, Dec 1998.

[34] S. Shenker and J. Wroclawski. General characterization parameters for integrated service network elements. IETF Internet Draft 2215, Integrated Services group, Dec 1998.

[35] Issam Aib, Nazim Agoulmine, Mauro Sergio Fonseca, and Guy Pujolle. Analysis of policy management models and specification languages. In *Net-Con*, volume 261 of *IFIP Conference Proceedings*, pages 26–50, Muscat, Oman, oct 17 2003. Kluwer.

[36] Solidum Inc. Pax pattern description language reference guide, April 2002.

[37] Nevil Brownlee. SRL: A language for describing traffic flows and specifying actions for flow groups. IETF Internet draft, Expird February 2000, Aug 1999.

[38] Gary N. Stone, Bert Lundy, and Geoffrey G. Xie. Network policy languages: a survey and a new approach. *IEEE Network*, 15(1):10–21, January/February 2001.

[39] Jorge Lobo, Randeep Bhatia, and Shamim Naqvi. A policy description language. In *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference*, pages 291–298, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[40] Chitta Baral, Michael Gelfond, and Alessandro Provetti. Representing actions: Laws, observations and hypothesis. *Journal of logic programming*, 31(1-3):201–243, 1997.

[41] Chitta Baral and Jorge Lobo. Formal characterization of active databases. In *Logic in Databases*, pages 175–195, 1996.

[42] Naranker Dulay, Emil Lupu, Morris Sloman, and Nicholas Damianou. A Policy Deployment Model for the Ponder Language. In *Integrated network management: 2001 IEEE/IFIP integrated management strategies for the new millennium, Seattle*, pages 529–544, May 2001.

[43] Dakshi Agrawal, Kang-Won Lee, and Jorge Lobo. Policy-based management of networked computing systems. *IEEE Communications Magazine*, 43(10), October 2005.

[44] IBM. Autonomic computing: Creating self-managing computing systems. http://www.ibm.com/autonomic/, 2004.

[45] D. Agrawal et al. Autonomic computing expression language. IBM DeveloperWorks Tutorial, Mar 2005.

[46] Mark Burgess. Cfengine concepts, version 2.1.10. Technical report, Faculty of Engineering, Oslo University College, Norway, 2004.

[47] Mark Burgess. Cfengine: a system configuration engine. Technical report, University of Oslo, 1993.

[48] Mark Burgess. *A site configuration engine*. 1995.

[49] N. Damianou, A. Bandara, M. Sloman, and E. Lupu. A survey of policy specification approaches, 2002.

[50] Mark Burgess. Recent developments in cfengine. In *Unix.nl conference*, 2001.

[51] M. Mont, A. Baldwin, and C. Goh. Power prototype: Towards integrated policy-based management. In *IEEE/IFIP Network Operations and Management Symposium NOMS*, 2000.

[52] James Bret Michael, Vanessa L. Ong, and Neil C. Rowe. Natural-language processing support for developing policy-governed software systems. In IEEE Computer Society Press, editor, *Proceedings of the 39th International Conference on Technology for Object-Oriented Languages and Systems TOOLS39*, pages 263– 274, Santa Barbara, California, July 2001.

[53] Sandrine Duflos, Gladys Diaz, Valérie Gay, and Eric Horlait. A comparative study of policy specification languages for secure distributed applications. In Lecture Notes In Computer Science, editor, *13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management - DSOM*, number 2506, pages 157 – 168, 2002.

[54] Hanan Lutfiyya, Gary Molenkamp, Michael Katchabaw, and Michael A. Bauer. Issues in managing soft qos requirements in distributed systems using a policy-based framework. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 185–201, London, UK, 2001. Springer-Verlag.

[55] Issam Aib, Mathias Sallé, Claudio Bartolini, and Abdel Boulmakoul. Capturing adaptive B2B service relationships management through a generalized service information model. In *HPOVUA*, page 16, Paris, France, June 2004. HP Labs.

[56] Ron Sprenkels and Aliko Pras. Service level agreements. Technical Report Deliverable D2.7, Internet NG, April 2001.

[57] Alexander Keller and Heiko Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Networks and Systems Management*, 11(1), 2003.

[58] Jim Martin and Arne Nilsson. On service level agreements for ip networks. In *Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies - INFOCOM 2002*, volume 2, pages 855– 863. IEEE, 2002.

[59] Li jie Jin, Vijay Machiraju, and Akhil Sahai. Analysis on service level agreement for web services. Technical report, Hewlett Packard Laboratories, June 2002.

[60] E. Marilly, O. Martinot, S. Betge-Brezetz, and G. Delegue. Requirements for service level agreement management. In IEEE, editor, *IP Operations and Management, 2002 IEEE Workshop*, pages 57–62, 2002.

[61] Vijay Machiraju, Akhil Sahai, and Aad van Moorsel. Web services management network: An overlay network for federated service management. In *Proceedings of the VIII IFIP/IEEE IM conference on network management*, 2003.

[62] D. Lamanna, J. Skene, and W. Emmerich. Slang: A language for defining service level agreements, 2003.

[63] Keller A. and Ludwig H. Web service level agreement (WSLA) language specification, version 1.0, revision wsla-2003/01/28. Technical report, IBM Research Division, 2003.

[64] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Aad P. A. van Moorsel, and Fabio Casati. Automated SLA monitoring for web services. In Metin Feridun, Peter G. Kropf, and Gilbert Babin, editors, *DSOM*, volume 2506 of *Lecture Notes in Computer Science*, pages 28–41. Springer, November 16 2002.

[65] Mathias Salle and Claudio Bartolini. Management by contract. In *IEEE/IFIP NOMS*, 2004.

[66] Markus Debusmann and Alexander Keller. SLA-driven management of distributed systems using the common information model. In *Proceedings of the VIII IFIP/IEEE IM conference on network management*, page 563, 2003.

[67] G.D. Rodosek. A generic model for it services and service management. In *IFIP/IEEE Eighth International Symposium on Integrated Network Management, 2003*, pages 171– 184. IEEE, 24-28 March 2003.

[68] Lupu E. Sloman M. Tonouchi T. Damianou N., Dulay N. Tools for domain-based policy management of distributed system. In *8th Network Operations and Management Symposium (NOMS 2006)*, page 157. IEEE ComSoc, 15-19 Apr 2002.

[69] DMTF. CIM policy model v.2.8. Technical report, Distributed Management Task Force DMTF, Jan 2004.

[70] Claudio Bartolini, Mathias Salle, and David Trastour. It service management driven by business objectives an application to incident management. In IEEE, editor, *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 45 – 55, 2006.

[71] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Fifth IEEE International Symposium on Requirements Engineering, 2001.*, pages 249–262, Toronto, Ont., Canada, 2001.

[72] Ian Sommerville. Integrated requirements engineering: A tutorial. *IEEE Softw.*, 22(1):16–23, Jan.-Feb 2005.

[73] Pamela Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, 8(3):250–269, 1982.

[74] Javier Rubio-Loyola, Joan Serrat, Marinos Charalambides, Paris Flegkas, and George Pavlou. A functional solution for goal-ooriented policy refinement. In *POLICY '06: Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06)*, pages 133–144, Washington, DC, USA, 2006. IEEE Computer Society.

[75] Marinos Charalambides, Paris Flegkas, George Pavlou, Arosha K. Bandara, Emil C. Lupu, Alessandra Russo, Naranker Dulay, Morris Sloman, and Javier Rubio-Loyola. Policy conflict analysis for quality of service management. *policy*, 00:99–108, 2005.

[76] Mandis S. Beigi, Seraphin Calo, and Dinesh Verma. Policy transformation techniques in policy-based systems management. In *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*, page 13, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[77] Arosha Bandara. *A Formal Approach to Analysis and Refinement of Policies*. PhD thesis, University College London, University of London, July 2005.

[78] Linying Su, David W. Chadwick, Andrew Basden, and James A. Cunningham. Automated decomposition of access control policies. In *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, pages 3–13, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[79] Dinesh Verma and Mandis S. Beigi. Network prediction in a policy-based ip network. In IEEE, editor, *Global Telecommunications Conference (GlobeCom 2001)*, volume 4, pages 2522–2526, San Antonio, TX, USA, Nov 25-29 2001.

[80] David B. Leake. *Case-Based Reasoning: Experiences, Lessons, And Future Directions*. second printing 2000 AAAI Press/MIT Press, 2 edition, 1996.

[81] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications*, 7(1):39–52, 1994.

[82] Arosha K. Bandara, Emil C. Lupu, Jonathan Moffett, and Alessandra Russo. A goal-based approach to policy refinement. *policy*, 00:229, 2004.

[83] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. Grail/kaos: an environment for goal-driven requirements engineering. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 612–613, New York, NY, USA, 1997. ACM Press.

[84] Robert Darimont and Axel van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 179–190, New York, NY, USA, 1996. ACM Press.

[85] A. Van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In *Second IEEE International Symposium on Requirements Engineering (RE'95)*, page 194, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[86] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[87] B. Delcourt A. van Lamsweerde, A. Dardenne and F. Dubisy. The kaos project: Knowledge acquisition in automated specification of software. In *AAAI Spring Symposium Series, Track: "design of Composite Systems"*, pages 59–62, Stanford University, March 1991.

[88] Ronald J. Brachman. *Readings in Knowledge Representation*. Morgan Kaufmann, August 1985.

[89] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurent Systems*. 1992.

[90] Ben Potter, Jane Sinclair, and David Till. *Introduction to Formal Specification and Z*. Prentice Hall PTR, 2 edition, July 11 1996.

[91] Arosha K Bandara, Emil C Lupu, Alessandra Russo, Naranker Dulay, Morris Sloman, Paris Flegkas, Marinos Charalambides, and George Pavlou. Policy refinement for diffserv quality of service management. *IEEE eTransactions on Network and Service Management (eTNSM)*, 3(2):12, 2nd quarter 2006.

[92] Robert Darimont. *Process Support for Requirements Elaboration*. PhD thesis, Université catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, June 23 1995.

[93] R Kowalski and M Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, 1986.

[94] Bert Van Nuffelen and C. Kakas, Antonis. A-System : Programming with abduction. In *Logic Programming and Nonmonotonic Reasoning, LPNMR 2001, Proceedings*, volume 2173 of *LNAI*, pages 393–396. Springer Verlag, 2001.

[95] Bert Van Nuffelen. *Abductive constraint logic programming: implementation and applications*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, June 2004.

[96] Javier Rubio-Loyola, Joan Serrat, Marinos Charalambides, Paris Flegkas, George Pavlou, and Alberto Lluch-Lafuente. Using linear temporal model checking for goal-oriented policy refinement frameworks. In *POLICY*, pages 181–190. IEEE Computer Society, 2005.

[97] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled.

[98] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[99] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Technology & Industrial Arts. Addison-Wesley Professional, 2004.

[100] Matthew Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.

[101] V. Danciu and Kempter B. From processes to policies — concepts for large scale policy generation. In R. Boutaba and S-B. Kim, editors, *Managing Next Generation Convergence Networks and Services*, pages 17–30. IFIP/IEEE, IEEE Publishing, apr 2004.

[102] Jonathan D. Moffett and Morris S. Sloman. Policy conflict analysis in distributed system management. *Journal of Organizational Computing*, 4(1):1–22, 1994.

[103] Ehab Al-Shaer and Hazem Hamed. Modeling and management of firewall policies. *IEEE Trans. Network and Service Management*, 1(1), Apr 2004.

[104] Emil C. Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering, Special issue on inconsistencies management*, 25(6):852–869, 1999.

[105] Axel van Lamsweerde, Robert Darimont, and Emmanuel Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998.

[106] Jan Chomicki, Jorge Lobo, and Shamin Naqvi. Programming approach to conflict resolution in policy management. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2000)*, pages 121–132. Morgan Kaufmann, San Francisco, CA, USA 2000.

[107] Dinesh C. Verma. Simplifying network administration using policy-based management. In *Network, IEEE*, volume 16, pages 20–26, Mar/Apr 2002.

[108] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on*

Policies for Distributed Systems and Networks, pages 18–38, London, UK, 2001. Springer-Verlag.

[109] Keld Helsgaun. Discrete event simulation in java. Technical Report 1-1, Department of Computer Science, Roskilde University, DK-4000 Roskilde, Denmark, March 2004.

[110] Issam Aib and Raouf Boutaba. Business-driven optimization of policy-based management solutions. In *IFIP/IEEE 10th International Symposium on Integrated Network Management (IM 2007): Moving from bits to business value*, pages 254–263. IEEE, May 21-25 2007. Best student paper award.

[111] Issam Aib, Mathias Salle, Claudio Bartolini, and Abdel Boulmakoul. A business driven management framework for utility computing environments. In *proceedings of the Ninth IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*. IEEE, May 16-19 2005.

[112] Issam Aib and Raouf Boutaba. Utility driven generation and scheduling of management actions. In Karima Boudaoud, Nicolas Nobelis, and Thomas Nebe, editors, *13th HP OpenView University Association workshop (HP-OVUA 2006)*, Nice, France, May 2006. HP Labs, Infonomics-Consulting (May 15, 2006). short paper.

[113] O.P. Sharma. *Markovian Queues*. Mathematics and its applications. Ellis Horwood, 1990.

[114] Gabi Dreo Rodosek and Edgar Aschenbrenner, editors. *Integrated Network Management X (IM 2007), Moving from Bits to Business value*, 2007.

Index

This thesis has been typeset using LaTeX 2_ε *and the WinEdt© editor*