Policy Analysis for DiffServ Quality of Service Management

Marinos Charalambides

Submitted for the Degree of Doctor of Philosophy from the University of Surrey

Centre for Communication Systems Research Faculty of Engineering and Physical Sciences University of Surrey Guildford, Surrey GU2 7XH, UK

June 2009

© Marinos Charalambides 2009
Abstract

Policy-based management has often been proposed as a flexible and efficient means for managing Quality of Service (QoS) in IP networks since policies can be dynamically changed without modifying the underlying implementation. Yet despite research projects, standardisation efforts and substantial interest from industry, network providers have been reticent to adopt it in practice. One of the most significant adoption barriers is that it is difficult to analyse policies to ensure the specification is consistent, free of conflicts. Policy analysis encompasses techniques and methodologies that provide the means to detect and resolve conflicts, and remains a poorly explored area.

This thesis comprehensively covers QoS provisioning policies from service management to traffic engineering, and classifies inconsistencies that may arise between them. It presents an integrated framework for policy analysis which is based on formal methods and supported reasoning techniques. The analysis approach has two main aspects: the definition of appropriate rules for determining potential conflicts in policy specifications, and the effective deployment of analysis processes in the context of the managed environment. Detection rules are used to describe the conditions under which a conflict will arise and include information from policies and the managed environment itself to cater for the various QoS management conflicts. A comprehensive set of detection rules together with system-specific information is used by the analysis processes to determine potential inconsistencies.

Analysis processes are distinguished between static and dynamic, which cater for conflicts that can be determined prior to policy enforcement and for conflicts that can only be detected at run-time, respectively. The former is an off-line process initiated by an administrator and searches the policy space for conflicts whose resolution is manual, whereas the invocation of the latter process and the subsequent detection of conflicts are automated. The run-time resolution process is also automated and is based on a pre-defined set of policies. The approach has been implemented in an integrated tool supporting both static and dynamic conflict analysis, which has been extensively tested for scalability over a range of static conflict types using large numbers of policies.

Key words: Policy-based Management, QoS Management Policies, Conflict Detection, Dynamic Conflict Resolution

Email: M.Charalambides@surrey.ac.uk
WWW: http://personal.ee.surrey.ac.uk/Personal/M.Charalambides
In memory of Professor Chris Todd,

a brilliant mind and a good friend
Acknowledgments

I would like to express my sincere gratitude to Professor George Pavlou who has given me the opportunity to pursue a PhD. His knowledge, assistance and supervision have been invaluable. Many thanks also go to Dr. Paris Flegkas for his guidance during the initial stages of my studies.

Additionally, I am grateful to my research collaborators at Imperial College: Dr. Arosha Bandara, Dr. Emil Lupu, Professor Morris Sloman and Dr. Alessandra Russo. Their views and contributions to joint work during the PAQMAN project and beyond have been extremely useful.

I would also like to thank Dr. Javier Rubio Loyola and Professor Joan Serrat from the Technical University of Catalonia for the past and ongoing collaboration on policy refinement issues.

My appreciation goes to my colleagues, as well as friends, in the Networks Group of CCSR: Ning Wang, Alvin Yew, Stelios Gouveris, Stelios Georgoulas, Marco Ballette, Carmelo Ragusa, Roy Ho, Wei Koong Chai, Mina Amin, Aimilios Chourmouziades, and Antonis Hadjiantonis.

Last but not least I would like to thank my parents Iacovos and Athena, and my brother Adonis for their continuous support over the last five years.

The work carried out in this thesis was partly funded by:

- The UK Engineering and Physical Sciences Council (EPSRC) PAQMAN research project (Grant No. GR/S79992/01).
- The EU FP6 EMANICS Network of Excellence project (No. 26854).
## Contents

Abstract ............................................................................................................................... iii  
Acknowledgments ................................................................................................................ v  
Contents .............................................................................................................................. vi  
Figures ................................................................................................................................. xi  
Listings ............................................................................................................................... xiii  
Tables ................................................................................................................................. xv  
Abbreviations .................................................................................................................... xvi  
Publications ....................................................................................................................... xix  

### 1 Introduction ................................................................................................................ 1  
  1.1 Research Motivation ................................................................................................. 2  
  1.2 Thesis Contributions ............................................................................................... 3  
  1.3 Thesis Structure ....................................................................................................... 4  

### 2 Background and Related Work ................................................................................. 6  
  2.1 Policy-Based Management: Frameworks and Languages ........................................ 6  
      2.1.1 IETF Policy Management Framework ................................................................. 7  
      2.1.2 Ponder Policy Framework ................................................................................... 8  
      2.1.3 Policy Management for Autonomic Computing ................................................. 10  
  2.2 Policy Conflicts ...................................................................................................... 11  
      2.2.1 Domain-Independent Policy Conflicts .............................................................. 11  
      2.2.2 Policy Conflicts in Distributed Systems Management .................................... 12  
      2.2.3 Policy Conflicts in Security Management ....................................................... 13  
      2.2.4 Policy Conflicts in QoS Management ............................................................... 14  
      2.2.5 Policy Conflicts in Call Control for Telecommunication Networks ............. 15  
  2.3 Policy Conflict Analysis .......................................................................................... 16  
      2.3.1 Approaches for Conflict Detection .................................................................... 16  
      2.3.1.1 Detection based on meta-policies ................................................................. 16  
      2.3.1.2 Detection based on applicability spaces ....................................................... 17  
      2.3.1.3 Detection based on policy relationships ....................................................... 18
2.3.1.4 Detection based on information models ............................................................... 19
2.3.2 Approaches to Conflict Resolution ............................................................................ 20
  2.3.2.1 Resolution based on precedence ........................................................................ 20
  2.3.2.2 Resolution based on policy ordering ................................................................. 22
  2.3.2.3 Resolution based on conflict prevention ............................................................ 22
2.3.3 Static versus Dynamic Analysis ................................................................................. 23
2.4 Formal Approaches to Policy Specification and Analysis ................................................ 25
  2.4.1 Logic Representation and Analysis of Authorisation Policies ................................... 26
  2.4.2 Logic Representation and Analysis of Obligation Policies ........................................ 27
  2.4.3 The Event Calculus ............................................................................................... 28
2.5 Quality of Service in IP Networks ................................................................................ 30
  2.5.1 Differentiated Services ........................................................................................... 30
  2.5.2 Multi-Protocol Label Switching and Traffic Engineering .......................................... 31
  2.5.3 The TEQUILA Framework .................................................................................... 32
    2.5.3.1 SLS Management ............................................................................................ 33
    2.5.3.2 Traffic Engineering ....................................................................................... 33
    2.5.3.3 Monitoring ..................................................................................................... 34
  2.5.4 DiffServ QoS Policies ............................................................................................ 34
    2.5.4.1 IETF QPIM .................................................................................................. 34
    2.5.4.2 Ponder DiffServ Policies .............................................................................. 35
    2.5.4.3 TEQUILA Policies ....................................................................................... 36
2.6 Summary ....................................................................................................................... 38
3 Policies for DiffServ QoS Management ........................................................................ 39
  3.1 Policy Classification, Specification and Enforcement .................................................. 40
  3.2 Service Management Policy Driven Functionality ...................................................... 43
    3.2.1 SLS Subscription Policies .................................................................................. 43
      3.2.1.1 Service Satisfaction ..................................................................................... 44
      3.2.1.2 Resource Availability Buffer Limit ............................................................... 45
      3.2.1.3 Admission Control Decision ....................................................................... 47
    3.2.2 SLS Invocation Policies ...................................................................................... 47
      3.2.2.1 Module Initialisation .................................................................................... 48
      3.2.2.2 Admission Control ..................................................................................... 49
  3.3 Traffic Engineering Policy Driven Functionality ......................................................... 51
    3.3.1 Network Dimensioning Policies ........................................................................... 51
      3.3.1.1 BW Allocation and Explicit Route Setup ...................................................... 52
      3.3.1.2 Hop Count Derivation ................................................................................. 53
3.3.1.3 Optimisation Algorithm ............................................................... 53
3.3.1.4 Spare/Over-provisioned BW Treatment ............................................. 54
3.3.2 Dynamic Resource Management Policies .................................................. 55
3.3.2.1 Module Initialisation ........................................................................... 55
3.3.2.2 Utilisation Tracking and Resource Allocation ......................................... 55
3.4 Managed Objects and System Behaviour Model ............................................ 57
3.4.1 Service Management ................................................................................. 58
3.4.2 Traffic Engineering ..................................................................................... 61
3.5 Summary and Conclusions ............................................................................ 63

4 Static Policy Conflict Analysis ........................................................................ 65
4.1 Conflict Classification .................................................................................... 66
4.1.1 Service Management Static Conflicts ......................................................... 67
4.1.2 Traffic Engineering Static Conflicts ............................................................ 69
4.2 Formal Representation ................................................................................... 70
4.2.1 Event Calculus ........................................................................................ 70
4.2.2 System Representation .............................................................................. 72
4.2.3 Policy Representation .............................................................................. 75
4.3 Conflict Detection Rules ................................................................................ 78
4.3.1 Module Independent Conflict Rules ............................................................. 79
4.3.2 Service Management Conflict Rules .......................................................... 81
4.3.3 Traffic Engineering Conflict Rules ............................................................. 82
4.4 Analysis of Static QoS Management Conflicts ................................................. 82
4.4.1 Abductive Reasoning for Conflict Detection ............................................... 83
4.4.2 System Architecture ............................................................................... 84
4.5 Static Conflict Analysis Case Study ................................................................. 85
4.5.1 Policy Set ............................................................................................... 86
4.5.2 Static Analysis ......................................................................................... 86
4.6 Summary and Conclusions ........................................................................... 88

5 Dynamic Policy Analysis and Conflict Resolution ........................................... 89
5.1 Conflict Classification .................................................................................... 91
5.1.1 Service Management Dynamic Conflicts ................................................... 91
5.1.2 Traffic Engineering Dynamic Conflicts ..................................................... 92
5.2 Conflict Detection Rules ................................................................................ 93
5.2.1 Service Management Conflict Rules .......................................................... 93
5.2.2 Traffic Engineering Conflict Rules ............................................................ 94
## Contents

5.3 Conflict Resolution ........................................................................................................... 95  
5.3.1 Conflict Resolution Rules........................................................................................... 95  
5.3.2 Support for Resolution Functionality ......................................................................... 97  
5.4 Analysis of Dynamic QoS Management Conflicts ........................................................... 99  
5.4.1 Deductive Reasoning for Conflict Detection .............................................................. 99  
5.4.2 Policy Enforcement Model ....................................................................................... 100  
5.4.3 System Architecture ................................................................................................. 102  
5.5 Dynamic Conflict Analysis Case Study .......................................................................... 103  
5.5.1 Policy Set .................................................................................................................. 104  
5.5.2 Dynamic Analysis .................................................................................................... 105  
5.6 Summary and Conclusions .............................................................................................. 106  
6 Tool Support and Experimental Evaluation .................................................................. 108  
6.1 Policy Management and Analysis Client ........................................................................ 109  
6.1.1 Design....................................................................................................................... 109  
6.1.2 Client Tool Overview ............................................................................................... 110  
6.1.3 Policy Input and Parsing ........................................................................................... 112  
6.1.4 Interfacing with Prolog ............................................................................................. 114  
6.2 Static Conflict Analyser .................................................................................................. 114  
6.2.1 Prolog Implementation ............................................................................................. 114  
6.2.2 Conflict Detection Example ..................................................................................... 116  
6.3 Dynamic Conflict Analyser ............................................................................................ 117  
6.3.1 Prolog Implementation ............................................................................................. 117  
6.3.2 Handling Events ....................................................................................................... 119  
6.3.3 Enforcing Policies .................................................................................................... 120  
6.3.4 Performing Conflict Analysis ................................................................................... 121  
6.4 Analysis Tool Evaluation ................................................................................................ 122  
6.4.1 Static Conflict Detection Performance Analysis ...................................................... 122  
6.4.1.1 Experiment 1 ...................................................................................................... 122  
6.4.1.2 Experiment 2 ...................................................................................................... 123  
6.4.1.3 Experiment 3 ...................................................................................................... 124  
6.4.1.4 Experiment 4 ...................................................................................................... 125  
6.4.2 Dynamic Analysis and Emulated System Behaviour ............................................... 127  
6.5 Summary and Conclusions .............................................................................................. 128  
7 Conclusions and Future Work ...................................................................................... 130  
7.1 Contributions Overview ................................................................................................. 131
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2 Future Research Directions</td>
<td>132</td>
</tr>
<tr>
<td>7.3 Epilogue</td>
<td>134</td>
</tr>
<tr>
<td>Bibliography</td>
<td>135</td>
</tr>
<tr>
<td>Appendix A1 – Service Management Static Conflict Rules</td>
<td>145</td>
</tr>
<tr>
<td>Appendix A2 – Traffic Engineering Static Conflict Rules</td>
<td>146</td>
</tr>
<tr>
<td>Appendix A3 – Service Management Dynamic Conflict Rules</td>
<td>147</td>
</tr>
<tr>
<td>Appendix A4 – Traffic Engineering Dynamic Conflict Rules</td>
<td>147</td>
</tr>
</tbody>
</table>
Figures

Figure 2-1: The IETF/DMTF Policy framework ................................................................. 7
Figure 2-2: The Ponder Policy Management architecture .................................................. 8
Figure 2-3: The PMAC architecture .................................................................................. 10
Figure 2-4: Modality conflict overlaps ............................................................................. 12
Figure 2-5: State diagram for detecting firewall anomalies ............................................... 18
Figure 2-6: Conflict computation ..................................................................................... 19
Figure 2-7: Policy precedence ......................................................................................... 21
Figure 2-8: How the Event Calculus functions ................................................................ 28
Figure 2-9: The TEQUILA QoS management architecture ............................................... 32
Figure 2-10: TE-GUI snapshots for the enforcement of load balancing policies ............... 37
Figure 3-1: Policy-driven QoS management framework .................................................... 39
Figure 3-2: Classification of QoS management policies .................................................... 41
Figure 3-3: Policy enforcement ....................................................................................... 43
Figure 3-4: Resource Availability Buffer ......................................................................... 45
Figure 3-5: Run-time adjustment of service rate and admission control parameters .......... 50
Figure 3-6: Bandwidth tracking of a single QC ................................................................ 56
Figure 3-7: SLS-S module behaviour and action summary ............................................... 58
Figure 3-8: SLS-I module behaviour and action summary ............................................... 60
Figure 3-9: ND module behaviour and action summary .................................................... 61
Figure 3-10: DRsM module behaviour and action summary ............................................. 63
Figure 4-1: Classification of static policy conflicts ............................................................ 67
Figure 4-2: BW allocation (ndMin-ndMax) ...................................................................... 69
Figure 4-3: Event Calculus predicates and axioms ............................................................. 71
Figure 4-4: Partial state chart representation of the SLS-S module behaviour .................... 74
Figure 4-5: Abductive reasoning evaluation tree .............................................................. 84
Figure 4-6: Static conflict analysis system architecture ..................................................... 84
Figure 5-1: Classification of dynamic policy conflicts ....................................................... 91
Figure 5-2: SLS-I module resolution logic ....................................................................... 98
Figure 5-3: DRsM module resolution logic ..................................................................... 99
Figure 5-4: Deductive reasoning evaluation tree ............................................................ 100
Figure 5-5: Policy enforcement model .............................................................. 101
Figure 5-6: Dynamic conflict analysis system architecture ............................. 102
Figure 6-1: Architecture of the policy analysis tool .......................................... 108
Figure 6-2: Class diagram for the policy management and analysis client .......... 110
Figure 6-3: Policy management and analysis tool interface .............................. 111
Figure 6-4: New policy creation using the graphical interface .......................... 112
Figure 6-5: The policy parsing process ............................................................ 113
Figure 6-6: Static analysis engine implementation files ................................. 115
Figure 6-7: Detecting QQL qcPriority conflicts .............................................. 116
Figure 6-8: Dynamic analysis engine implementation files ............................. 117
Figure 6-9: Detection performance with varying number of conflicts ............ 123
Figure 6-10: Redundancy conflicts – Detection performance against number .... 124
Figure 6-11: Detection performance against number of policies with varying QC s 125
Figure 6-12: Detection performance against number of policies for different conflict types 126
Figure 6-13: Analysis example of a dynamic conflict – detection and resolution 127
Listings

Listing 2-1: Authorisation policy example ................................................................. 9
Listing 2-2: Obligation policy example ...................................................................... 10
Listing 2-3: Mobile device adaptation policy using EC ............................................. 29
Listing 2-4: PHB policy rules .................................................................................... 35
Listing 2-5: Policy rule for providing a specific PDB ............................................... 36
Listing 2-6: TEQUILA ND policies ........................................................................... 36
Listing 3-1: Ponder obligation policy example ........................................................ 42
Listing 3-2: Service satisfaction policies ................................................................... 44
Listing 3-3: Resource Availability Buffer policies .................................................... 46
Listing 3-4: Admission Control policy ...................................................................... 47
Listing 3-5: Initialisation policies .............................................................................. 48
Listing 3-6: Admission Control policies ................................................................... 49
Listing 3-7: Explicit BW allocation and LSP setup policies ....................................... 53
Listing 3-8: Hop count derivation policy ................................................................... 53
Listing 3-9: Optimisation algorithm policies .............................................................. 54
Listing 3-10: Post-processing policies ...................................................................... 54
Listing 3-11: Initialisation policy ............................................................................... 55
Listing 3-12: Dynamic resource management policies .............................................. 57
Listing 4-1: Formal representation of managed objects and organisation .................. 73
Listing 4-2: Formal representation of the behaviour model ......................................... 75
Listing 4-3: Rule for detecting modality conflicts ...................................................... 78
Listing 4-4: Rule for detecting redundancy conflicts ............................................... 79
Listing 4-5: Rule for detecting mutual exclusion conflicts ......................................... 80
Listing 4-6: Rule for detecting OQL qcPriority conflicts ........................................... 81
Listing 4-7: Rule for detecting multiplexing conflicts ............................................... 81
Listing 4-8: Rule for detecting diverging allocation conflicts ..................................... 82
Listing 4-9: Rule for detecting hopsExceed conflicts ................................................ 82
Listing 4-10: Policy instances .................................................................................. 83
Listing 4-11: Policy instances .................................................................................. 86
Listing 4-12: Conflict detection trace ....................................................................... 87
Listings

Listing 5-1: Rule for detecting maximum service rate violation conflicts ......................... 94
Listing 5-2: Rule for detecting invocation admission strategy conflicts ............................ 94
Listing 5-3: Rule for detecting ND maximum violation conflicts ...................................... 95
Listing 5-4: Rule for detecting over-allocation conflicts ................................................... 95
Listing 5-5: Policy instances ............................................................................................. 100
Listing 5-6: Formal representation of obligation policy enforcement model ...................... 101
Listing 5-7: Policy instances ............................................................................................. 104
Listing 5-8: Dynamic conflict analysis trace .................................................................... 106
Listing 6-1: Rules for handling system events ................................................................. 120
Listing 6-2: Rule for requesting an action ....................................................................... 120
Listing 6-3: Rules for executing a policy ....................................................................... 121
Listing 6-4: Rules for conflict analysis ......................................................................... 122
Tables

Table 3-1: Conditions and actions for a subscription request decision .............................................. 47
Table 4-1: Functions and predicates for representing the managed system ........................................ 72
Table 4-2: Functions for representing obligation policies ........................................................................ 76
Table 4-3: Sets of mutually exclusive actions ...................................................................................... 80
Table 5-1: Resolution policy actions for the identified conflict types .................................................. 97
Table 5-2: QC associated values ......................................................................................................... 104
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Admission Control</td>
</tr>
<tr>
<td>ACLP</td>
<td>Autonomic Computing Policy Language</td>
</tr>
<tr>
<td>AF</td>
<td>Assured Forwarding</td>
</tr>
<tr>
<td>AM</td>
<td>Autonomic Manager</td>
</tr>
<tr>
<td>AS</td>
<td>Autonomous System</td>
</tr>
<tr>
<td>ASL</td>
<td>Authorisation Specificiation Language</td>
</tr>
<tr>
<td>ATM</td>
<td>Asynchronous Transfer Mode</td>
</tr>
<tr>
<td>BE</td>
<td>Best Effort</td>
</tr>
<tr>
<td>BW</td>
<td>Bandwidth</td>
</tr>
<tr>
<td>COPS</td>
<td>Common Open Policy Service</td>
</tr>
<tr>
<td>DEN-ng</td>
<td>Directory Enabled Networks - next generation</td>
</tr>
<tr>
<td>DiffServ</td>
<td>Differentiated Services</td>
</tr>
<tr>
<td>DMTF</td>
<td>Distributed Management Task Force</td>
</tr>
<tr>
<td>DRsM</td>
<td>Dynamic Resource Management</td>
</tr>
<tr>
<td>DRTM</td>
<td>Dynamic Route Management</td>
</tr>
<tr>
<td>DSCP</td>
<td>DiffServ CodePoint</td>
</tr>
<tr>
<td>EC</td>
<td>Event Calculus</td>
</tr>
<tr>
<td>ECA</td>
<td>Event-Condition-Action</td>
</tr>
<tr>
<td>EF</td>
<td>Expedited Forwarding</td>
</tr>
<tr>
<td>EWMA</td>
<td>Exponentially Weighted Moving Average</td>
</tr>
<tr>
<td>FOL</td>
<td>First Order Logic</td>
</tr>
<tr>
<td>HC</td>
<td>Hop Count</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IntServ</td>
<td>Integrated Services</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ISP</td>
<td>Internet Service Provider</td>
</tr>
<tr>
<td>LDAP</td>
<td>Lightweight Directory Access Protocol</td>
</tr>
<tr>
<td>LSP</td>
<td>Labelled Switched Path</td>
</tr>
<tr>
<td>ME</td>
<td>Mutually Exclusive</td>
</tr>
<tr>
<td>MF</td>
<td>Multiplexing Factor</td>
</tr>
<tr>
<td>MO</td>
<td>Managed Object</td>
</tr>
<tr>
<td>MPLS</td>
<td>Multi-Protocol Label Switching</td>
</tr>
<tr>
<td>MR</td>
<td>Managed Resource</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ND</td>
<td>Network Dimensioning</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OQL</td>
<td>Overall Quality Level</td>
</tr>
<tr>
<td>OSPF</td>
<td>Open Shortest Path First</td>
</tr>
<tr>
<td>PBM</td>
<td>Policy-Based Management</td>
</tr>
<tr>
<td>PCIM</td>
<td>Policy Core Information Model</td>
</tr>
<tr>
<td>PDL</td>
<td>Policy Description Language</td>
</tr>
<tr>
<td>PDB</td>
<td>Per-Domain Behaviour</td>
</tr>
<tr>
<td>PDP</td>
<td>Policy Decision Point</td>
</tr>
<tr>
<td>PDT</td>
<td>Policy Definition Tool</td>
</tr>
<tr>
<td>PEP</td>
<td>Policy Enforcement Point</td>
</tr>
<tr>
<td>PES</td>
<td>Policy Editor Storage</td>
</tr>
<tr>
<td>PHB</td>
<td>Per-Hop Behaviour</td>
</tr>
<tr>
<td>PMA</td>
<td>Policy Management Agent</td>
</tr>
<tr>
<td>PMAC</td>
<td>Policy Management for Autonomic Computing</td>
</tr>
<tr>
<td>PMT</td>
<td>Policy Management Tool</td>
</tr>
<tr>
<td>QC</td>
<td>QoS Class</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>QPIM</td>
<td>QoS Policy Information Model</td>
</tr>
<tr>
<td>RAB</td>
<td>Resource Availability Buffer</td>
</tr>
<tr>
<td>RAM</td>
<td>Resource Availability Matrix</td>
</tr>
<tr>
<td>RPC</td>
<td>Resource Provisioning Cycle</td>
</tr>
<tr>
<td>RSVP</td>
<td>Resource Reservation Protocol</td>
</tr>
<tr>
<td>SIP</td>
<td>Session Initiation Protocol</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SLS</td>
<td>Service Level Specification</td>
</tr>
<tr>
<td>SLS-I</td>
<td>SLS Invocation</td>
</tr>
<tr>
<td>SLS-M</td>
<td>SLS Management</td>
</tr>
<tr>
<td>SLS-S</td>
<td>SLS Subscription</td>
</tr>
<tr>
<td>SNMP</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td>SR</td>
<td>Service Rate</td>
</tr>
<tr>
<td>SU</td>
<td>Subscription Upper</td>
</tr>
<tr>
<td>TCL</td>
<td>Target Critical Level</td>
</tr>
<tr>
<td>TD</td>
<td>Traffic Demand</td>
</tr>
<tr>
<td>TE</td>
<td>Traffic Engineering</td>
</tr>
<tr>
<td>TEQUILA</td>
<td>Engineering for Quality of Service in the Internet at Large Scale</td>
</tr>
<tr>
<td>TF</td>
<td>Traffic Forecast</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>ToS</td>
<td>Type of Service</td>
</tr>
<tr>
<td>TT</td>
<td>Traffic Trunk</td>
</tr>
<tr>
<td>VCL</td>
<td>Very Critical Level</td>
</tr>
<tr>
<td>VoD</td>
<td>Video-on-Demand</td>
</tr>
<tr>
<td>VoIP</td>
<td>Voice-over-IP</td>
</tr>
</tbody>
</table>
Publications

Journal Papers


Peer-reviewed Conference Papers


Chapter 1

1 Introduction

It is evident that modern networks have become increasingly difficult to manage mainly due to the diversity in networking technologies, the vast number of resources, and the high demand for bandwidth-intensive as well as delay-sensitive applications. These complexities pose significant challenges to existing network management models, which can lead to cumbersome administration processes especially when network re-configuration is needed to adapt to new services, unpredicted demand, changing business objectives and application requirements.

Policy-Based Management (PBM) is a management paradigm that has been researched over the past fifteen years and has been proposed as a potential solution for the problems stated above. Under this paradigm, an administrator can manage different aspects of a network or distributed system in a flexible and simplified manner by deploying a set of policies that govern its behaviour. Policies are technology independent rules aiming to enhance the hard-coded functionality of managed devices by introducing interpreted logic that can be dynamically changed without modifying the underlying implementation. This allows for a certain degree of programmability without the need to interrupt the operation of either the managed system or of the management system itself. Furthermore, this approach facilitates scalability since a few policies can manage devices in a collective fashion, thus avoiding the need of specifying multiple vendor-specific scripts for different device technologies as traditionally done.

The advantages offered by PBM as a management technology attracted the attention of both the research community and industry. This has resulted in the development of a number of policy languages and frameworks, commercial products, as well as investigations into their applicability in various application domains. One of the most popular application domains is that of Quality of Service (QoS) management since network providers can realise their objectives through flexible programmability with respect to offered services and treatment of customer traffic. For this reason the application area considered in this thesis is QoS management, focusing on IP networks with DiffServ support. The QoS provisioning policies proposed apply to service management and traffic engineering, and essentially define how traffic should be treated in the network.

As with any new technology however, PBM comes with some drawbacks or unresolved issues, the first being policy refinement. The operational behaviour of a managed system is usually based
on business objectives that can be expressed as high-level policies. These need to be incrementally decomposed (refined) to lower-level policies that enforce management operations supported by the system. The second and most important problem is that of policy conflicts. These are inconsistencies that can arise between policy rules as a result of specification errors, omissions, or contradictory management operations and, in some cases, can have catastrophic effects on the operation of the managed system. The work in this thesis tackles the largely unresolved issue of policy conflicts in the context of QoS management policies for DiffServ networks.

1.1 Research Motivation

Although extensive research has been done in developing policy specification languages, protocols and architectures to support policy-based management, relatively little attention has been devoted to the fundamental issue of policy conflict analysis, which is evidenced by the lack of tool support. This is the main reason why policy-based management has not been widely adopted, despite its potential benefits of flexibility and constrained programmability.

Policy conflicts had been initially studied in the context of generic management policy and were broadly classified as static and dynamic – static conflicts are those that can be detected at policy specification time, whereas dynamic ones refer to those that can only be detected at policy enforcement time. Subsequent research in specific application domains, such as security management and call control in telecommunication networks, mainly focused on static conflicts. This involved simple analysis and resolution was mostly based on the specification of policy precedence rules that may not suit many policy-driven systems. Furthermore, policy conflicts in the domain of QoS management, which was of primary interest to the IETF, have not been considered and, consequently, techniques for their effective detection and resolution have not been developed.

In contrast to static inconsistencies that can be determined by off-line processes before policy deployment, dynamic conflicts can only be detected during system execution since they depend on the current state of the managed system. For example, conflicts can occur between policies for dynamically allocating resources and those setting quotas for users or classes of service. As such, automation should be a key aspect of dynamic analysis mechanisms so that the operational impact of a conflict can be kept to a minimum. This issue has not been adequately addressed in the literature and concrete methodologies for handling run-time inconsistencies have not been developed.

There is limited value in deploying policy-based management systems that do not provide support for conflict detection and resolution. The deficiencies in addressing conflict analysis identified
Chapter 1. Introduction

above motivate the work presented in this thesis, the objective of which is to develop a methodology and supporting tools so that the consistency of QoS management policies can be ensured. In this respect, the various conflicts that might occur in DiffServ environments are identified and categorised, and an approach for both static and dynamic policy analysis is proposed. This is based on the logic formalism of Event Calculus, which serves as the underlying formal representation for both the system and the policies. The logic-based approach allows for advanced reasoning capabilities to cope with the requirements of effectively analysing for conflicts in complex systems such as QoS management.

1.2 Thesis Contributions

In an effort to extend previous work on policy conflicts and to develop new techniques for conflict analysis, several research contributions have been achieved. These can be summarised as follows:

- **QoS management policies**: A small number of QoS management policies have previously been defined, mostly targeting specific QoS provisioning operations. In this thesis a comprehensive set of such policies, from service management to traffic engineering, is provided and the effect of their enforcement on the behaviour of managed modules and associated DiffServ managed objects is described. The specification of these policies using the Ponder format, one of the most widely adopted policy languages, is also provided.

- **QoS management policy conflicts**: A wide range of potential conflicts related to QoS management policies, not been previously reported in the literature, have been identified including the conditions under which they arise. Some of these conditions can apply to other resource management frameworks. The conflicts have been classified based on their level of abstraction, the QoS provisioning subsystem in which they occur, their specificity to the application domain, and the time frame at which they can be detected.

- **Static conflict analysis**: The rules that describe the conditions under which static QoS management conflicts will arise have been defined and a methodology for their detection has been developed. This is based on the formal representation of policies and the managed system components, and the use of abductive reasoning techniques. The use of the latter to not only detect static inconsistencies but also to generate explanations as to how they occur has been demonstrated.
• **Dynamic conflict analysis**: An approach for detecting and resolving dynamic QoS management conflicts has been developed which, as in the case of static analysis, is based on formal methods. The approach allows for the automatic deployment of conflict analysis processes at system execution time by monitoring policy enforcement. Detection is based on deductive reasoning over defined conflict rules and resolution is achieved without human intervention using generic resolution policies. The latter is a flexible solution that overcomes the limitations of traditionally used precedence rules.

• **Tool support and performance evaluation**: The issue of lack for conflict analysis tool support has been addressed by developing the proposed analysis techniques and integrating them in a tool. The tool provides a usable interface that hides the complexity of the underlying formal methods and allows for both static and dynamic consistency checks to be performed. Furthermore, the performance of the static analysis engines has been evaluated through extensive experimentation over a range of conflict types – this is something that has not been adequately addressed in the literature.

### 1.3 Thesis Structure

This chapter introduced the basic principles of policy-based management, described the motivation for investigating policy conflicts and analysis techniques, and presented the research contributions. The remainder of this thesis is organised as follows:

**Chapter 2 – Background and Related Work**: This chapter provides an overview of the background topics and the most important related work associated with this thesis. More specifically, previous work in the areas of policy-based management, conflict analysis, and formal approaches for policy specification and analysis, is presented and discussed. The application domain of QoS provisioning in DiffServ networks is also described.

**Chapter 3 – Policies for DiffServ QoS Management**: This chapter provides a comprehensive set of QoS provisioning policies that can be used to manage different aspects of the chosen application domain, such as admission control and resource management. The specification of the various policies is provided and the influence on the system behaviour as a result of their enforcement is described.

**Chapter 4 – Static Policy Conflict Analysis**: This chapter presents the approach for the detection of static conflicts. Various static inconsistencies related to QoS management policies are identified and how the conditions under which they arise can be encoded into conflict rules is shown. Furthermore, the use of the logic formalism of Event Calculus to represent both policies
and the managed system is described, and the use of abductive reasoning to detect the presence of inconsistencies is demonstrated.

Chapter 5 – Dynamic Policy Analysis and Conflict Resolution: This chapter presents the dynamic conflict analysis approach. Dynamic QoS management conflicts are identified and classified, and the formal representation – which is extended to model policy enforcement – used in conjunction with deductive reasoning to detect inconsistencies at run-time is described. Furthermore, the methodology by which dynamic analysis can be achieved in an automated fashion is presented.

Chapter 6 – Tool Support and Experimental Evaluation: This chapter describes the design and implementation of the tool developed to support the proposed conflict analysis techniques. Evaluation results regarding the performance, scalability and correct operation of the tool are also presented.

Chapter 7 – Conclusions and Future Work: The last chapter concludes this thesis by summarising the work and discussing the contributions. Future directions of this research are also suggested.
Chapter 2

2 Background and Related Work

This chapter lays the foundations for the work presented in the core part of this thesis. It provides the reader with an overview of the background topics and the most important related work in the areas of policy-based management, conflict analysis, and Quality of Service in DiffServ networks.

The chapter begins by introducing the main policy-based frameworks and concepts proposed in the literature along with a brief description of the format and the various components of the policies used. This is followed by a section on policy conflicts, which reports the conflict types identified in various application domains and describes the reasons for their occurrence. The two subsequent sections present prior work relating to techniques for the detection and resolution of conflicts, highlighting the pros and cons. Formal approaches for policy specification and analysis are also discussed here, focusing on those based on first order logic. The latter part of the chapter provides details about the application domain tackled in this thesis, consisting of an overview of QoS in IP networks, an example architecture providing such a capability in DiffServ environments, and relevant work on QoS policies. Finally, the last section summarises the chapter.

2.1 Policy-Based Management: Frameworks and Languages

The increasing complexities and heterogeneity of modern networking technology, and the vast number of resources to be managed, pose significant challenges to network management models. Policy-Based Management (PBM) is a promising solution for these demands, providing the means by which the administration process can be simplified and automated to a large extent. A policy, the basic building block of the policy-based paradigm, is a set of rules that govern the behaviour of a managed system. As these rules constitute interpreted logic, the approach facilitates flexibility and adaptability in that policies can be dynamically changed without modifying the underlying implementation.

PBM has been the subject of extensive research over the past years, evidenced by several research and development efforts in both academia and industry, working groups leading standardisation efforts, technical conferences, and new commercial products. This section presents the three most important PBM frameworks and associated specification languages in the literature. Although all
frameworks have influenced the evolution of policy research in the management community, the work carried out at Imperial College London has paved the way for advances in policy-based approaches.

2.1.1 IETF Policy Management Framework

The joint effort of the IETF [1] and DMTF [2] resulted in a generic policy architecture, which, as illustrated in Figure 2-1, consists of four major functional elements: the Policy Management Tool (PMT), Policy Repository, Policy Decision Point (PDP), and Policy Enforcement Point (PEP).

![Figure 2-1: The IETF/DMTF Policy framework](image)

The PMT is used by an administrator to define or update the policies to be enforced in the managed network. Resulting policies are stored in a repository in a form that must correspond to the information model in [3] so as to ensure interoperability across products from different vendors. When new policies have been added in the repository, or existing ones have been changed, the PMT issues the relevant PDP with notifications, which in turn interprets the policies and communicates them to the PEP. The latter is a component that runs on a policy-aware node and can execute (enforce) the different policies. The components of the architecture can communicate with each other using a variety of protocols. The preferred choice for communicating policy decisions between a PDP and network devices (PEPs) is the Common Open Policy Service (COPS) [4], or SNMP [5], and LDAP [6] for the PMT/PDP‐repository communication.

The simplest approach for policy specification is through a sequence of rules, in which each rule is the form of a simple condition-action pair. The IETF policy framework adopts this approach and considers policies as rules that specify actions to be performed in response to defined conditions:

\[
\text{If } \langle \text{condition(s)} \rangle \text{ then } \langle \text{action(s)} \rangle
\]
The conditional part of the rule can be a simple or compound expression specified in either conjunctive or disjunctive normal form. The action part of the rule can be a set of actions that must be executed when the conditions are true. The IETF does not define a specific language to express network policies but rather a generic object-oriented information model for representing policy information (PCIM) [3]. This model is a generic one, specifying the structure of abstract policy classes by means of association, thus allowing vendors to implement their own set of conditions and actions to be used by the policy rules.

2.1.2 Ponder Policy Framework

Initial work in [7] describes the concept of policies in distributed systems management. Here, policies are viewed as objects which define the relationships between subjects (managers) and targets (managed objects), and are separated from the managers' functionality. This facilitates the dynamic change of the behaviour and adaptivity to new requirements without re-implementing the management applications. In [8] the authors identify that specifying policies for individual managed entities in large-scale systems is not a practical approach. They propose the use of domains as the means of grouping objects representing managed entities to which policies apply, thus partitioning the management responsibility.

The concept of domains is a key aspect of the Ponder policy framework which is depicted in Figure 2-2 [9]. Here, an administrator can create and modify policies using a policy editor. Authorisation policies are disseminated to target agents as specified by the target domains and obligation policies to manager agents (PMAs) as specified by the subject domains. Policies can be subsequently enabled, disabled or removed from the agents. Obligation policies are interpreted by manager agents, which register with the monitoring service to receive events relevant to their activation. Upon receiving an event, the agent queries the domain service to determine the target objects and performs the policy action(s).
Subsequent work on Ponder [10][11] involved the design of a deployment and enforcement model and the development of a toolkit integrating the various components of the framework to support the whole policy life-cycle relating to the specification and management of deployed policies. The toolkit provides a comprehensive policy-based management platform based on an object-oriented Java implementation and has been widely used in the research community. The next part describes the various policy types supported by this toolkit.

The Ponder policy specification language

Ponder is a declarative, object-oriented language [12] that can be used to specify both security and management policies. It supports two main policy types as described below: authorisation and obligation policies.

Authorisation policies define what actions a manager (subject) can perform on target objects. These policies are enforced by access controllers running in the target objects’ environment aiming to protect resources from unauthorised access. A positive authorisation policy is used to define the actions that subjects are permitted to perform on target objects, whereas negative authorisations define the actions that subjects are prohibited from performing. The policy in Listing 2-1 presents the syntax of a positive authorization. In this example, the policy specifies that project managers are granted access to confidential documents, only between office hours as expressed by the condition of the last line.

Obligation policies are event-condition-action (ECA) rules that define the operations that must be performed by managers of the subject domain on objects of the target domain when certain events occur, given some supplementary conditions being true. While authorisations are executed by access controllers, obligation polices are enforced by PMAs which facilitate adaptation of the managed system according to emerging conditions. The events triggering obligation policies can be external events notified by monitoring service components, or internal timer events as in the example of Listing 2-2. Here, an archiving process is instructed to create a backup of documents in the repository every night at 2:00 a.m. The “->” operator in the action part of the policy allows sequential execution of operations and is used here to create a log once the backup process has finished.
2.1.3 Policy Management for Autonomic Computing

The Policy Management for Autonomic Computing (PMAC) platform [13][14] is part of IBM’s initiative on autonomic computing, which defines a framework for self-managing IT systems. PMAC is a generic middleware platform that can be used to manage aspects of large-scale distributed systems including QoS, security and auditing. The architecture of the platform is depicted in Figure 2-3 which provides components for policy creation, policy evaluation, and enforcement at managed resources.

![Figure 2-3: The PMAC architecture [13]](image)

At the highest level, multiple Policy Definition Tools (PDT) are supported for concurrent policy authoring. Policies are stored in a centralised Policy Editor Storage (PES) which can also hold metadata such as templates for policy re-use. The main component of PMAC is the Autonomic Manager (AM), the role of which is similar to that of the PDP in the IETF framework, but supports additional features such as state monitoring, event correlation and notification. AMs obtain policies from the PES and register Managed Resources (MR) that are interested in receiving policy directives from them. MRs provide two interfaces, Sensors (S) and Effectors (E), which represent the attributes that can be read from the resource and the management operations.
that can be performed on the resource respectively. AMs evaluate policies based on the sensed state of resources, which can invoke actions on MRs via the effector interface and consequently changing their behaviour.

Policies in the PMAC framework are specified using the Autonomic Computing Policy Language (ACPL), the structure of which is defined using an XML schema. They are ECA rules, where the conditional part is specified with a generic constraint language, which is also XML-based. The advantage of using such an approach is that the resulting policies can be parsed and type checked by XML parsers, thus making it attractive to applications that can consume XML format. Furthermore, the language can be extended relatively easy with new operations by modifying the schema and adding the extension operators. The problem with an XML representation is that policies can become quite verbose and not easily interpreted by human administrators.

2.2 Policy Conflicts

As with any programmable system, a policy-driven one can suffer from inconsistencies incurred by conflicting rules governing its behaviour. This problem becomes more acute with increasing policy-influenced functionality supported by a managed system and thus the number and types of policies used. Policy conflicts have been described as being analogous to software bugs, which, according to [15], occur when two or more policies are activated simultaneously enforcing contradictory management operations on the system.

Policy conflicts can be broadly classified into domain-independent and application-specific, where the former, as the names suggest, are independent of the policy application, and the latter are bound by the constraints of the application domain. This section presents the main conflict types identified in the literature ranging from simple modality conflicts to more specialised ones in the areas of distributed systems management, security and QoS management, and call control in telecommunication networks.

2.2.1 Domain-Independent Policy Conflicts

One of the most common types of inconsistency cited in the literature is the conflict of modalities [16][17]. This is a generic conflict that can occur in any policy-driven system which supports policies of opposite modalities, as for example positive and negative authorisations.
According to [18], these conflicts arise when two or more policies with modalities of opposite sign refer to the same subjects, actions and targets. This occurs when there is a triple overlap between the sets of subjects, targets and actions as shown in Figure 2-4. Based on the authorisation and obligation policies supported by Ponder, the authors identify three types of modality conflicts as follows:

- **O+/O-**: The subjects are both required and required not to perform the same actions on the target objects.
- **A+/A-**: The subjects are both authorised and forbidden to perform the actions on the target objects.
- **O+/A-**: The subjects are required but forbidden to perform the actions on the target objects.

Practical examples of this conflict type can be found in [17] and [19].

### 2.2.2 Policy Conflicts in Distributed Systems Management

One of the first application domains for which policy conflicts have been considered is that of distributed systems management. The authors of [18] and [20] identify various conflicts in this area and classify them as follows:

- **Conflict of Resources**: This occurs when the number of resources (target objects) available is limited. For example, a policy controlling system backup activities may require more disk space than the pre-allocated amount.

- **Multiple Managers Conflict**: Multiple managers (subjects) may manage the same objects that are shared between several tasks. This conflict will occur if the outcomes of management operations are incongruent with each other. For example, spooling a job to a printer and shutting the same printer down.

- **Self-Management Conflict**: This situation is of a manager managing itself, and will occur if a subject is allowed to retract policies that it is supposed to perform.
• **Conflict of Interest**: This conflict arises when the same subject can perform management tasks on two different sets of targets which are competing. For example, a bank provides investment advice to a client whilst performing a merger for a competing client.

Another common inconsistency cited in the literature is the *conflict of duties*, which is also stated as the requirement to ensure separation of duties. This conflict has been studied in [21] and [22] in the context of access control systems. Such a conflict will arise if the same subject is permitted to perform operations that are not supposed to be carried out by the same entity. For instance, the same user should not be authorised for the operations of submitting, evaluating, and approving the budget in a company's financial system. A specialised form of separation of duty is the *Chinese Wall* policy [23], which prevents a subject performing any conflicting actions on one target, if that subject has already been given permission to perform a conflicting action on a different target.

2.2.3 **Policy Conflicts in Security Management**

There has been considerable work on IP security policy analysis evidenced by a number of publications [24][25][26]. The most representative work in this area is the one presented in [27] and [28], which deals with inconsistencies among legacy firewall policies. In [27] the authors identify the various conflicts that may arise between filtering rules on individual firewalls, which are referred to as *intra-firewall anomalies*. These depend on the relations of filtering rules and their relative ordering in a firewall. In this context, a policy anomaly is defined as the existence of two or more filtering rules that may match the same packet, or the existence of a rule that can never match any packet in the network paths crossing the firewall. Based on these principles the following anomalies have been identified:

- **Shadowing**: A rule is shadowed when a previous rule matches all the packets that match this rule, such that the shadowed rule will never be activated.

- **Correlation**: Two rules are correlated if they have different filtering actions, and the first rule matches some packets that match the second rule and the second rule matches some packets that match the first rule.

- **Generalisation**: A rule is a generalization of a preceding rule if they have different actions, and if the first rule can match all the packets that match the second rule.

- **Redundancy**: A rule is redundant if there is another rule that produces the same matching and action such that if the redundant rule is removed, the security policy will not be affected.

- **Irrelevance**: A filtering rule in a firewall is irrelevant if this rule does match any traffic that may flow through this firewall.
Chapter 2. Background and Related Work

The above anomalies and the reasons as to their occurrence have been cited in other works [29][30] and have been used as the subject for analysis. Apart from inconsistencies in the rule-set of a single firewall, anomalies may also arise between policies applying to different firewalls within an enterprise network. For example, an upstream firewall might block traffic that is permitted by a downstream firewall or vice versa. These are termed inter-firewall anomalies, the various types of which have been classified into shadowing, spuriousness, correlation, and redundancy anomalies [28]. Subsequent work in [31] identified one more inconsistency which applies to large scale environments such as the Internet, where there might exist multiple data paths to the same protected network: a cross-path anomaly refers to the case where some packets denied on one path are accepted through another.

2.2.4 Policy Conflicts in QoS Management

Very recent work in [32] targets the same application domain as the one considered in this thesis. Here, the authors identify conflicts among policies managing QoS in DiffServ networks, but only tackle a small portion of the problem regarding resource management at the router level. The policies involved in this process define the treatment of a traffic flow on network nodes by setting parameter values for BW allocation, queue size, drop method, and priority for the various Per-Hop Behaviours (PHBs). Inconsistencies among these policies are classified according to the scope in which they occur: intra-PHB conflicts arise within the flow properties at a specific node and inter-PHB conflicts occur between policy definitions across different nodes.

Intra-PHB conflicts are further subdivided into two types: (a) Single parameter conflicts are simple inconsistencies that occur due to malformed parameter conditions such as a negative queue length, or a percentile parameter specified with a value greater than a hundred. (b) Multiple parameter conflicts occur as a result of dependencies between policy parameters and the constraints of the application domain. For example, if the priority level is specified by a flow, then the maximum bandwidth should be specified otherwise starvation for other flows will occur.

PHB policies are set such that a flow meets some quality requirements. This implies that equivalent treatment should be exercised at all hops. Inter-PHB conflicts occur when a particular flow meets different behaviours at more than one node along its path from source to destination, which can result in quality reduction. Examples of this conflict type include different bandwidth allocations or different types of forwarding priorities at successive nodes for the same class of service.

In contrast to the above conflicts, the inconsistencies identified in this thesis cover a wider spectrum of QoS policies implementing service management and traffic engineering functions, both at the network and device levels.
2.2.5 Policy Conflicts in Call Control for Telecommunication Networks

Another application domain for which conflict analysis has been addressed is that of telecommunications and more specifically call control. Within this domain, features [33] have been widely used to provide users with some control over calls, which, in a similar fashion to policies, accommodate additional functionality to enhance the base system. Inherent in feature-oriented systems is the problem of feature interactions, where the presence of one feature contradicts another thus causing unexpected behaviour or even failure of calls. Many examples of this have been identified in telecommunications systems and documented in [34], [35] and [36].

With the increasing popularity of policy-based management and the advantages this technology provides, researchers investigated the use of policies to facilitate call control [37][38], as features tend to be low-level and fairly inflexible units of functionality. This work has been extended to investigate conflicts between policies governing the behaviour of call control mechanisms in the context of SIP (Session Initiation Protocol) communications [39][40]. Central to this work is the role of the policy server, which receives environmental information from call requests and responds by executing policy actions that determine how an incoming or outgoing call should be handled (e.g. continue as normal, add a party to the call, fork the call). The authors identify two distinct situations in which call control policies may conflict. The first is in conflicts between policies known to a single policy server, and the second is where conflicts occur between policies associated with two or more policy servers (distributed environment).

Examples of the two conflict types are provided in [40]. To demonstrate a single server conflict, the authors use two user-defined call forwarding policies with the following actions:

(a) \textit{fwdLateCallsVM} – forwards all calls after 3 pm to voicemail.
(b) \textit{fwdLDCallsHome} – forwards long-distance evening calls after 6 pm to home.

In this example, if the user receives a long distance call after 6 pm, both \textit{fwdLateCallsVM} and \textit{fwdLDCallsHome} policies apply to the call, which can potentially lead the policy server to an unstable state. For the case of a distributed setting, the two following policy actions are used:

(a) \textit{noAddPartyOutgoing} – applies to the caller’s (userA) policy server and prohibits other parties from being added to the call.
(b) \textit{addPartyHead} – applies to the callee’s (userB) policy server and requires an additional party to be conferenced into the call, when this originates from userA.

Clearly these two policies conflict with each other. This inconsistency will arise when userA places a call to userB, leading the incoming and outgoing policy servers to disagree.
2.3 Policy Conflict Analysis

To effectively use policies and drive the functionality of a managed system in a consistent manner, it is necessary to check that newly created policies do not conflict with each other or with policies already deployed in the system. To achieve this, detection processes utilise information regarding the conditions under which conflicts can arise to search policy spaces and identify policies that meet the conflict criteria. Resolution is the latter part of policy analysis, which aims at handling detected inconsistencies, preferably in an automated manner, so that consistency among policies can be restored.

A number of detection and resolution approaches have been proposed over the years providing solutions for the various conflict types identified in the literature. This section presents the main approaches and classifies them based on the analysis technique used. A distinction is also made between methodologies that analyse policies statically at compile-time and those that can handle conflicts at run-time.

2.3.1 Approaches for Conflict Detection

Based on the types of conflicts identified in the literature and the different application domains in which they occur, research has concentrated in the development of mechanisms and techniques for their effective detection. Although modality conflicts can be detected by syntactic analysis, more specialised inconsistencies require a precise definition of the conditions for a conflict, which sometimes include domain-specific knowledge, and processes that utilise such information to signal the occurrence of a conflict.

Lupu describes the manner in which modality conflicts can be determined in [41], where a detection process enumerates all subject, action, target tuples which have a different set of policies applying to them. If there are two or more policies applying to a tuple then there is a potential conflict and the policies can be checked to see if they have opposite modalities. Following the relevant example diagram given in the previous section, two policies P1(+) and P2(-) will be signalled as being conflicting due to their common subjects, actions, and targets indicated by the tuple <sc, ac, tc>. This analysis is purely syntactic and does not require understanding of the policies. Below, more advanced methodologies are described catering for more complex conflicting situations.

2.3.1.1 Detection based on meta-policies

Application-specific conflicts arise from the semantics of the policy, which, according to [18], are specified in terms of constraints on attribute values of permitted policies. For example, in the case of the separation of duties described in Section 2.2.2, the conflict is particular to the actions of the
involved policies. To define the conflicting conditions, the authors in [18] make use of meta-policies, i.e. policies about management policies, which capture the various constraints pertaining to a conflict. The separation of duties example can be stated as “there should not be two policies having overlapping subject domains which give rights to submit and approve a company’s budget”, and can be represented as a logical predicate:

\[ \text{intersectSubj}(P_1, P_2) \land (\text{submit} \in P_1.\text{actions}) \land (\text{approve} \in P_2.\text{actions}) \land \\
(P_1.\text{targets} = P_2.\text{targets} = \text{budget}) \land (P_1.\text{mode} = P_2.\text{mode} = \text{A+}) \Rightarrow P_1 \text{ conflicts with } P_2 \]

A set of meta-policies, encapsulating the descriptions for the various conflict types, are manually specified by an administrator and are evaluated during the detection process. The latter iterates through policy specifications on a pairwise basis and signals a conflict if it can match two policies that satisfy the conditions in a predicate. The fact that conflict definitions are separate from the detection process makes the approach scalable, since more meta-policies can be added at any stage in case further policy functionality is introduced and more conflicts are identified. This approach has been used as a general guideline by a number of researchers in this area, which instead propose different representations of conflict conditions and related information.

### 2.3.1.2 Detection based on applicability spaces

The notion of applicability spaces has been proposed in [42] and [43], where the conditional part of a policy, represented by a set of independent terms (such as time), can be looked upon as an independent axis in a hyper-dimensional space. Each policy rule defines a region in the hyper-dimensional space, and separate regions can be associated with a dependent term (such as a QoS class) that is identified by the rule. The detection of a conflict is based on overlaps between spaces that target incompatible dependent terms.

Detection based on the intersection of applicability spaces has been extended by the work of IBM, which focuses on conflicts involved in the management of distributed systems. Agrawal in [14] and [44] describes the policy ratification process which is an integral component of the PMAC platform presented in Section 2.1.3. This is defined as the process by which a new policy is approved before being committed to the system by taking into account its potential interactions with other policies and its deployment environment. Based on the fact that the applicability of a policy relies on a set of conditions, this work proposes the use of Boolean expressions for constraint representation and evaluation. Therefore, the key ratification operation is to determine whether a conjunction of two Boolean expressions is satisfiable (simultaneously true).

To achieve the above objective, policy constraints are mapped to Boolean expressions and subsequently transformed to linear inequalities. The latter are used to build matrix representations
which undergo linear transformations until a feasible solution can be found. If such a solution does not exist, it is assumed that two spaces do not intersect and subsequently no conflicts can be found. Although this approach contributes significantly in the complex task of constraint evaluation, taking into account a wide range of logic and arithmetic operators, it does not elaborate on the important issue of action incompatibility, which is the very reason behind the occurrence of policy conflicts.

2.3.1.3 Detection based on policy relationships

Conflicts in the area of security management come about as a result of the relations between firewall filtering rules and their relative ordering. The relations concern the network policy fields of protocol (TCP/UDP), source, and destination (including address and port), which can be completely disjoint, exactly matching, inclusively matching, partially disjoint, or correlated. Instead of specifying separate rules regarding the conditions under which an anomaly would arise, most works in this area formalise separate rule relations, which are called from a detection process when performing comparisons. More specifically, the most representative work in this domain [27][28] developed an algorithm for discovering rule anomalies by implementing the state transition diagram in Figure 2-5. The latter illustrates the firewall anomaly discovery states for any two rules, $R_x$ and $R_y$, where $R_x$ comes before $R_y$.

![Figure 2-5: State diagram for detecting firewall anomalies](image)

Initially no relationship is assumed by this algorithm. Each field in $R_y$ is compared to the corresponding field in $R_x$ starting with the protocol, then source address and port, and finally destination address and port. The relationship between the two rules is determined based on the result of the aforementioned comparisons. The last step before flagging an anomaly involves equality tests between the action parts of the two rules. If, for example, every field in $R_y$ is a...
subset or equal to the corresponding field in $R$, and both rules have the same action, $R_y$ is redundant to $R_x$, while if the actions are different, $R_y$ is shadowed by $R_x$.

One shortcoming of this approach is the dependence on legacy firewall policies in order to perform anomaly analysis. This issue has been addressed in [29] where network security requirements can be specified using high-level notations whilst still being capable of a range of analysis tasks.

2.3.1.4 Detection based on information models

Motivated by the advantages provided by information models in representing managed entities, such as platform and protocol independency, the authors in [45] and [46] propose their use in the process of conflict detection. More specifically, this work is based on the DEN-ng model [47], which, apart from managed entities, is also used for representing both the policies and the conditions under which these may conflict. The latter are expressed with the Object Constraint Language (OCL) [48] in terms of invariants, pre- and post-conditions associated with attributes and operations modelled by policies.

This work has recently been extended in [49] and [50] to support the overall methodology and implementation of the conflict detection approach. Here, the authors describe a two-phase analysis algorithm which, querying an information model, firstly determines the relationships between a pair of policies and, secondly, applies conflict patterns to determine if the policies should be flagged as conflicting. Policy relationships are expressed in terms of policy subjects, targets and actions, while conflict patterns concern constraints defined in the information model describing policy relationships that must hold for a conflict. Determining a conflict involves transforming the above information into matrices and performing comparisons.

The left part of Figure 2-6 depicts the format of such a matrix providing a common representation to both policy relationships and conflict patterns. The different entries in the matrix concern the relationships between the various fields of policies: rows represent the subjects, targets, events, constraints and actions, while columns indicate the relationship type (subset, superset, equal, correlated, mutually exclusive, conflicting). The right part of Figure 2-6 demonstrates a matrix comparison operation to determine a shadowing filtering policy conflict, where relationships are
asserted by placing 1’s in the relevant fields. The first matrix concerns the relationship between two filtering policies, while the second one represents the pattern for this particular conflict type. The comparison operator combines the values of the two matrices indicating the detection of a conflict if the outcome is 1.

The use of information models is also proposed by Kempter in [51], where invariants extracted from the models are used as indicators for conflicts when they are breached. Although this approach benefits from the inherent advantages of information models, XML representation of policies and conflicting conditions can become very verbose thus posing a cumbersome task for a network administrator if a manual change is required. Furthermore, the use of matrices in [49] limits the definition of relationships to the core fields of a policy. As such, conflicts that arise as a result of inconsistent action parameters, rather than actions, are difficult to detect and the exact reason for their occurrence cannot be provided.

2.3.2 Approaches to Conflict Resolution

The process of resolving conflicts may involve retracting, suppressing, prioritising, or amending policies, and in some cases, enforcing a new policy altogether so that consistency among policy rules can be restored. The methodology in doing so depends heavily on the type of policies involved and the domain in which conflicts occur. Although human intervention is unavoidable in some situations, several research efforts focussed on techniques to automate the resolution process where possible; the main ones are presented and discussed below.

2.3.2.1 Resolution based on precedence

The most popular approach for conflict resolution is that where precedence can be established between policies. This allows two potentially inconsistent policies to coexist within the system and it involves determining which of the two should prevail in the event of a conflict. A number of metrics have been studied for determining the relative priorities between policies, which are summarised in [18] and presented below. Some of these have been implemented in logical precedence mechanisms such as the one supported by the KAoS Policy Administration tool [52].

Precedence based on modality

It is quite common for negative authorisation policies to override positive ones so that a forbidden action will never be permitted [53], but this may not necessarily be always desired. As a general rule, this approach resolves all conflicts in a deterministic way, i.e. negative policies take precedence over positive ones or vice versa, which is not very flexible. Some flexibility may be introduced by adopting a default policy, as proposed in [54], such as “everything is implicitly forbidden,” or “everything is implicitly authorised,” and defining precedence between explicit
Authorisation, explicit denial, implicit authorisation or implicit denial, but this does not really solve the problem.

**Distance between a policy and the managed objects**

The concept of calculating the distance between a policy rule and the objects to which it refers has been introduced in [55] involving authorisation policies in object-oriented databases. Here, priority is given to the policy applying to the closer class in the inheritance hierarchy when evaluating access to an object referenced in a query, which essentially indicates the relevance of a policy to an object. Other types of distances have been considered in [56], such as the importance of objects’ non-common classes in a generalisation hierarchy and the similarity of unique objects’ attributes.

**Specificity related to domain nesting**

This concept was proposed in [57] and constitutes a particular case of the distance metric. The principle here is that a more specific policy overrides a more general one, where specificity is based on the number of objects to which a policy applies, i.e. a sub-domain of objects is more specific than the ancestor domain. This approach is particularly useful in policy systems where objects are grouped in sub-domains to reflect specialisation or any other relationship considered important for management purposes.

![Figure 2-7: Policy precedence [18]](image)

Although this is a flexible approach, Lupu in [18] recognises that it does not apply successfully to all situations as precedence cannot be established in cases where subject and target sets are equal, subject sets are more specific but target sets are less specific or vice versa. The left part of Figure 2-7 depicts overlapping policies, P1 and P2, for which a conflict can be resolved by assigning precedence to the more specific policy P2. The right part of the figure demonstrates the limitations of this approach with examples where precedence cannot be established.
Assigning explicit priorities

An alternative way by which policy precedence can be applied is to assign explicit priority values to policies [58]. Meaningful priorities, however, are notoriously difficult for users to assign and may result in arbitrary priorities which do not really reflect the importance of policies. Furthermore, in distributed settings where multiple administrators specify policies and priorities, inconsistencies among priorities can also be introduced.

These issues were addressed in [44] where the authors developed algorithms to automatically assign the priority values to new policies and to adjust the values of related policies when given only the relative priority of a new policy. The algorithms implement the conflict resolution module of the IBM’s PMAC platform by maintaining ordered lists under policy insertion and deletion operations.

2.3.2.2 Resolution based on policy ordering

Instead of specifying integer numbers that represent policy priorities as in the PMAC platform, conflicts in firewalls can be avoided by maintaining a correctly ordered list of filtering rules. Since the ordering directly impacts the semantics of the firewall security policy, the authors of [27] investigated policy editing techniques to avoid the introduction of anomalies during policy updates.

More specifically, the developed algorithms can automatically determine the correct insertion position of a newly created rule such that no shadowing or redundancy anomalies are created. This is achieved based on the relation of the new rule with existing ones, following the principle that the new rule should be inserted before any rule that is a superset match, and after any rule that is a subset match of this rule. Although some potential inconsistencies can be resolved, this approach does not cater for correlation and generalisation anomalies, for which human intervention is required. Rule removal has much less impact on the firewall policy than insertion. According to [27], a removed rule may change the overall filtering policy semantics, but from a conflict perspective it may only create a redundancy anomaly; this is also handled manually.

2.3.2.3 Resolution based on conflict prevention

The work in [59] describes an alternative approach for handling inconsistencies and follows the validation principle of [60]. The authors propose the use of constraints in a policy to prevent that policy from firing, and consequently prevent a conflict from occurring, if a new configuration parameter is not consistent with an associated system variable.
This approach has been extended in [45] with algorithms to automatically encode appropriate constraints into a policy. In this work, the authors use a BW management scenario where policies aim to allocate finite link resources. The limitation on these resources is expressed with OCL, a specification of which is attached to the allocation operation in the information model. During a refinement process this constraint is conveyed to the conditional part of the policy as follows:

\[ \text{<condition>RouterLink.currentBW + amount < RouterLink.maxBW}</condition> \]

The above condition controls the applicability of the policy, which will not execute if the aggregate value of the current allocation and the amount to be increased exceeds the maximum available capacity. The main disadvantage of this approach is that although it can prevent a conflict, it may also prevent the system from making a potentially essential re-configuration. Additionally, the constraint will need to be evaluated with each policy triggering event, which can, in most cases, induce unnecessary computational overhead.

2.3.3 Static versus Dynamic Analysis

Apart from the distinction between inconsistencies identified in different application domains, conflicts can also be classified into static and dynamic. Static conflicts can be detected through off-line analysis at policy specification time whereas dynamic conflicts can only be detected when policies are enforced as they depend on the current state of the managed system. The various detection and resolution techniques described in previous parts of this section have been applied to static conflicts but, in most cases, cannot meet the requirements posed by run-time inconsistencies. The latter, being the more challenging part of policy analysis, have rather been neglected with little evidence of research work.

The time-frame at which conflicts can be detected influences the analysis methodology for dealing with them. Dynamic conflicts must be detected by a process that monitors policy enforcement and resolution must be achieved automatically in an efficient manner to avoid long run-time processing overheads. The main contribution in this field comes from Dunlop in [61] and [62] which investigate mechanisms and techniques for the analysis of dynamic conflicts in open distributed systems. Central to the development of the detection algorithm in [61] are two indexed databases; the active events index contains a list of run-time events, and the run-time conflict index contains a list of current potential run-time conflicts. These are generated at compile-time and hold references to the policies involved and the precise events that will initiate conflicts. Such events are monitored at run-time, the occurrence of which triggers the conflict detection process.
This work was extended in [62] to address the problem of dynamic conflict resolution. Although the authors propose the use of precedence between conflicting policies as described in Section 2.3.2.1, the contribution of this work is not as to how a conflict is to be resolved, but rather as to when. Based on the fact that a resolution process can be computationally intensive, the authors propose different approaches according to the likelihood of a conflict occurring and the cost of resolving that conflict:

- **Pessimistic conflict resolution**: This approach assumes that both actual and potential inconsistencies will result in conflict at some point in time and must therefore be resolved immediately at compile-time. This involves much initial checking to ensure non-compliance does not occur, but minimises the run-time cost of detection and resolution.

- **Optimistic conflict resolution**: This approach does not involve any preventative measures but relies instead on resolving all conflicts dynamically when they materialise. As a result, the optimistic approach often requires greater effort and cost at run-time.

- **Balanced conflict resolution**: This approach assumes that the likelihood of actual conflict occurring is quite high and thus resolves it statically, thereby reducing the cost of run-time analysis. Identified potential conflicts are monitored and resolved in the run-time environment only if required.

Although, from a computational efficiency point of view, the above approaches have a clear contribution as to the optimisation of static and dynamic analyses, the validity of pessimistic and balanced resolutions can be argued since they violate the principle that dynamic conflicts can only be detected at run-time.

The challenges of dynamic analysis are also addressed in [40] where the authors provide solutions for the conflicts identified in Section 2.2.5 regarding call control policies. These conflicts will arise at run-time, when a call is placed, if policies enforced at different policy servers, between caller and callee, are inconsistent with each other. To facilitate detection and effective resolution in such a setting, a distributed communication mechanism is proposed enabling policy servers to cooperate since the policies they enforce are intended to apply end-to-end over the call. The mechanism resembles the distributed tuple space used by the feature interaction manager in [63], which acts as a temporary store (blackboard) for information relevant to policy decisions applying to a particular call.

Resolution is based on pre-specified rules which are treated in the same manner as policies with respect to storage and enforcement, and their triggering events are a particular combination of call control policy actions. Additional elements in these rules are the conditions for a conflict which are expressed using comparison operators. The actual resolution strategy is encoded in the action
part of the rule as in the example XML representation below, which aims at handling the second conflict described in Section 2.2.5 regarding the addition of a party into a call.

```xml
<actions>
  <orelse/>
  <action>apply_preference</action>
  <action>reject_call</action>
</actions>
```

The first of the above actions takes advantage of the fact that call control policies also incorporate a preference into their specification, e.g. *must not* or *prefer to*, and forces the resolution engine to prioritise the rule that has the strongest preference level (*apply_preference*). In case precedence cannot be established due to the same preference levels, a generic resolution is applied, which in the above example rejects the call.

To achieve distributed communication and ensure correct enforcement of policies and resolutions, the policy store associated with the first policy server assumes the role of the blackboard for the duration of a call. Any policies triggered by a call, instead of being enforced, they are written on the blackboard and their actions are deferred. A reference to the blackboard is carried by the call data ensuring that subsequent policy servers can access and store resolution information relating to the current call. On reaching the callee's policy server, local policies are consulted and those triggered by the call are identified. The resolution process is initiated at the server by first checking for conflicts between local policies and those accumulated on the blackboard. In such a case, the relevant resolution action is identified and enforced. Although the notion of resolution policies is also used in this thesis, the main drawback of this approach is that the detection of conflicts is not supported by a separate process, but the various conditions are encoded within resolution policies instead. Resolution specifications can thus become complex, an aspect amplified by the verboseness of XML representations, and their evaluation can be quite expensive.

### 2.4 Formal Approaches to Policy Specification and Analysis

Apart from high-level languages, research in policy specification has also considered formal logic notations. Although these are, in general, not very easy to understand and use, they provide notations that have well understood semantics thus avoiding potential ambiguities of high-level languages and they support several types of logical reasoning, which can facilitate operations for conflict analysis in a simplified manner. Furthermore, formal approaches allow the effect of policy enforcement on the behaviour of the managed system to be modelled, which can be used as part of the analysis process for the detection of conflicts that are constrained by the state of the system.
A limited number of approaches have proposed logic-based representations for policy specification. Standard Deontic Logic has been considered by some works [64][65], but because of the existence of a number of paradoxes it has not been widely accepted. First Order Logic (FOL) however, does not have this inherent problem and has been used for the specification of both access control and management policies. This section presents the two main approaches that are based on FOL and introduces the logic formalism of Event Calculus which is used in this thesis.

2.4.1 Logic Representation and Analysis of Authorisation Policies

The Authorisation Specification Language (ASL) is based on FOL and is the most representative example for the specification and analysis of authorisation policies. As described in [21] and [22], the basic constructs of the language are a set of predicates the main ones being \texttt{cando}, \texttt{dercando}, \texttt{do} and \texttt{grant}. The first two are used to define explicit and derived positive/negative authorisation rules as in the examples below. The first rule states that all subjects of the group \texttt{CS-Faculty} can read \texttt{file1}, whereas the second rule derives a negative authorisation for a subject \texttt{s} to read \texttt{file2} if there exists another subject \texttt{s'} and a group \texttt{s''} such that \texttt{s} and \texttt{s'} both belong to \texttt{s''}, and \texttt{s'} is authorised to write \texttt{file3}.

\[
\begin{align*}
\texttt{cando(file1, s, +read) } & \leftarrow \texttt{in(s, CS-Faculty)} \\
\texttt{dercando(file2, s, -write) } & \leftarrow \texttt{dercando(file3, s', +write)} \\
& \texttt{& in(s, s') & in(s', s'')} 
\end{align*}
\]

Inconsistencies arising among authorisation rules are a specialised type of the modality conflict described in Section 2.2.1, where subjects are both authorised and forbidden to perform an action. ASL allows for two strategies when resolving such conflicts, giving precedence to either denials or permissions. These are defined by \texttt{do} predicates which state the authorisations the system must consider valid on the basis of existing authorisations, specified or derived. The examples below both assign precedence to permissions but in different ways. The first rule states that a subject can exercise an access on \texttt{file1} if there is a positive authorisation for it, whereas the second states that a subject can exercise an access on \texttt{file2} only if there is not a negative authorisation for it.

\[
\begin{align*}
\texttt{do(file1, s, +a) } & \leftarrow \texttt{dercando(file1, s, +a)} \\
\texttt{do(file2, s, +a) } & \leftarrow \texttt{-dercando(file2, s, -a)} 
\end{align*}
\]

Lastly, resolution predicates are used in the body of access control rules to determine the access decision upon a request based on specified authorisations. The rule below will grant user \texttt{u} access to object \texttt{o}, if the user has a positive authorisation for it.

\[
\texttt{grant(o, u, +a) } \leftarrow \texttt{do(file2, s, +a)}
\]
Chapter 2. Background and Related Work

The main disadvantage of the formal notation used for the development of ASL is that it has not been extended to provide a representation of the managed system state. As such, there is no means to reason about the correctness of policies prior to deployment.

2.4.2 Logic Representation and Analysis of Obligation Policies

Apart from access control, FOL has also been proposed for the specification of obligation policies. One such example is the Policy Description Language (PDL) [66] which is an event-based language formulating obligations as sets of event-condition-action rules. These rules follow the format of the example below, which states that if the event occurs in a situation where the condition is true, then the action is executed.

\[
\text{event causes action if condition}
\]

The event part of the policy rule is an expression of the form \( e_1 \land \ldots \land e_n \) where each \( e_n \) is an event literal, and its occurrence is denoted by the \( \text{occ} \) predicate. Each policy rule can be translated to the implication below, where action \( a \) is executed (\( \text{exec} \) predicate) if the conjunction of a finite set of events and a condition \( c \) evaluates to true.

\[
\text{exec}(a) \iff \text{occ}(e_1) \land \ldots \land \text{occ}(e_n) \land c
\]

Later work by Chomicki et al. [67][68] extends PDL to include the concept of action constraints, which are essentially rules defining the set of actions that can not occur together. Conflicts are captured as violations of action constraints. The expression below is used to specify such constraints and states that the simultaneous execution of actions \( a_1, \ldots, a_n \) is never allowed if condition \( c \) holds. The constraint can be translated to a conflict rule using the \( \text{block} \) and \( \text{exec} \) predicates.

\[
\text{never } a_1 \land \ldots \land a_n \iff c
\]

Detection in this framework is based on monitors for inconsistent policy actions or events, and resolution is achieved by directly or indirectly cancelling actions to obtain a result consistent with the constraints. Taking the latter approach, resolution is specified using the \( \text{blocking} \) and \( \text{accepting} \) rules below. The first ignores any events that cause actions defined in the conflict rule, and the second will only accept a policy action if the event causing it is not blocked.

\[
\text{ignore}(e_1) \lor \ldots \lor \text{ignore}(e_n) \iff \text{occ}(e_1) \land \ldots \land \text{occ}(e_n) \land c \land \text{block}(a)
\]

\[
\text{accept}(a) \iff \text{occ}(e_1) \land \ldots \land \text{occ}(e_n) \land c \land \neg \text{ignore}(e_1) \land \ldots \land \neg \text{ignore}(e_n)
\]

The resolution methodology employed by PDL allows priority ordering among conflicting actions. This is done explicitly during the specification of detection rules where the user can
define which actions are to be blocked in the event of a conflict. These are encoded in the head of a detection rule and are subsequently evaluated in the body of the blocking rule. Taking the example of two conflicting actions, $\text{action}_1$ and $\text{action}_2$, priority to $\text{action}_1$ is assigned by blocking $\text{action}_2$:

$$\text{block(}\text{action}_2\text{) } \leftarrow \text{exec(}\text{action}_1\text{) } \land \text{exec(}\text{action}_2\text{)}$$

Despite its expressiveness, PDL does not support access control policies, and, as in the case of ASL, detection and resolution of policy conflicts can only be achieved at the level of rule execution, i.e. at run-time.

2.4.3 The Event Calculus

The Event Calculus (EC) was introduced by Kowalski and Sergot [69] as a logic formalism for representing and reasoning about events and their effects. The formalism is shown to apply to a variety of domains including those featuring continuous change. Although a number of variations of this formalism exist, most works have used a simplified form consisting of a set of time points, a set of event types, and a set of properties that can vary over the lifetime of the system known as fluents. Additionally, EC defines a set of base predicates that allow the specification of propositions: $\text{initiallyTrue}$, $\text{happens}$, $\text{initiates}$, $\text{terminates}$, and $\text{holdsAt}$.

Because Event Calculus is expressed in FOL, it provides support for a number of reasoning tasks. According to [70], these are broadly classified into deductive, abductive and inductive tasks. With reference to Figure 2-8, in a deductive task, “what happens when” ($\text{initiallyTrue}$, $\text{happens}$) and “what actions do” ($\text{initiates}$, $\text{terminates}$) are given, and “what’s true when” ($\text{holdsAt}$) is required. In an abductive task “what actions do” and “what’s true when” are supplied, and “what happens when” is sought. Finally, in an inductive task, “what’s true when” and “what happens when” are provided, but “what actions do” is required.

In the context of this thesis, and as described in chapters 4 and 5, Event Calculus is used to represent both the policies and the behaviour of the managed system. This allows for the development of analysis processes that cater for static as well as dynamic detection of conflicts. In
Chapter 2. Background and Related Work

In the first case, abductive reasoning provides the means for deriving the sequence of events required to satisfy conflicting conditions, whereas deductive reasoning is used to determine run-time conflicts based on the current state of the system.

Although EC has also been proposed for the purpose of designing and executing protocols [71], its use for policy representation in [72] is more relevant to this thesis. In this work, the authors use the formalism to specify obligation policies so that adaptation on mobile devices can be achieved. More specifically, the defined policy rules consist of system-specific event definitions, a set of fluents controlled by events, conditions expressed in terms of fluents, and actions. The latter represent calls to specific adaptation methods of an application and are executed when the conditional part evaluates to true. Listing 2-3 below demonstrates an example specification of a policy from [72], instructing a mobile device to switch to a GSM network connection once outdoors. The first part of the policy specifies the events in the managed system, and also defines which of those initiate or terminate the outdoors fluent. The action \texttt{switchNetwork(Gsm)} will be executed when one of the events initiating the outdoors fluent occurs.

Listing 2-3: Mobile device adaptation policy using EC

```plaintext
event LeftHome
event LeftOffice
event InHome
event InOffice
fluent Outdoors {
    initiates(LeftHome) or initiates(LeftOffice)
    terminates(InHome) or initiates(InOffice)
}
condition { initiates(Outdoors) }
action { switchNetwork(Gsm) }
```

Thus far in the chapter the main policy-based management frameworks have been described, and the various types of inconsistencies that can occur among policies for different application domains have been identified. A review of the mechanisms and techniques proposed in the literature for conflict detection and resolution has also been provided including logic approaches. The last section of this chapter focuses on the application domain targeted by this thesis, which is Quality of Service provisioning in IP networks.
2.5 Quality of Service in IP Networks

As the Internet evolves toward the global multi-service network of the future the best-effort model, currently employed, is not able to cope with emerging requirements of new services. Bandwidth (BW) intensive applications, like peer-to-peer file sharing, pose significant strains on the usage of resources, and audio/media services, such as Voice-over-IP (VoIP) and Video-on-Demand (VoD), have rather strict delay and packet loss rate requirements. For these reasons, the research community was driven to seek new models to support services with guaranteed resources and Quality of Service (QoS) characteristics. As a result, the Internet Engineering Task Force (IETF) proposed two QoS models, the Integrated [73] and Differentiated Services [74].

The Integrated Services (IntServ) model follows a resource reservation approach to a preferred set of applications in a similar manner to Asynchronous Transfer Mode (ATM) networks. Bandwidth reservation is made with the Resource Reservation Protocol (RSVP) [75], when applications signal their requirements to the network. One of the main strengths of IntServ is that, after a successful RSVP reservation, connections are assured a certain level of end-to-end performance from the network. In addition to the existing best effort class, two more classes of service are supported by this model in an IP network. The guaranteed service class provides quantitative upper bounds on bandwidth and delay experienced by an application, whereas the controlled load class makes a qualitative assurance that the network performance will be equivalent to that of best effort under lightly loaded conditions. Despite its strengths, the IntServ model suffers from scalability problems – the fact that bandwidth reservation is on a per flow basis, it requires every router along a path, from source to destination, to maintain flow state information. The latter increases proportionally with the number of flows, thus placing a huge storage and processing overhead to the routers.

DiffServ overcomes the above shortcomings and provides a scalable approach with which service differentiation can be achieved. This section provides a short overview of the DiffServ approach and also of Multi-Protocol Label Switching as the forwarding technology. It goes on to describe in detail a popular architecture for QoS provisioning in IP DiffServ-enabled networks, and also to present the various policies that have been proposed for the management of this domain.

2.5.1 Differentiated Services

Differentiated services allow the classification of IP traffic into a limited number of service classes that receive different treatment at the router level. The information to perform the differentiation is encoded within the Type of Service (ToS) byte in IPv4 packet headers or Traffic Class in IPv6, which is known as the DiffServ Codepoint (DSCP) [76]. Six out of the eight bits are used for this purpose giving 32 possible combinations that can define the various ways a
packet can be treated by core routers. Each different treatment is known as a *Per-Hop Behaviour* (PHB), which can be used to describe the forwarding behaviour of a network node applied to a collection of packets with the same DSCP. Currently, three types of PHBs are specified for a DiffServ network: (a) the Expedited Forwarding (EF) PHB that is of high priority, (b) the Assured Forwarding (AF) PHB that has four sub-classes offering different levels of forwarding assurances, and, (c) the Default (DE) or Best Effort (BE) PHB available in today’s Internet.

Each of the various PHBs corresponds to a particular DSCP. At the edge of a DiffServ network the access routers have the responsibility of classifying and marking incoming packets to relevant PHBs supported by that network. The forwarding behaviour, in terms of buffer and scheduling management, experienced by a packet in the core of the network corresponds to the PHB assigned to it. This approach pushes the complexity to the edge routers at network boundaries, keeping the core routers simple, resulting in a very efficient forwarding process. Unlike IntServ, it does not require signalling protocols to control the mechanisms used to select different treatment for individual packets. As such, the amount of state information required to be maintained at every node is proportional to the finite number of service classes rather than application flows.

A fundamental aspect of QoS provisioning in a DiffServ network is the Service Level Agreement (SLA). This is a contract, established statically or dynamically, that captures the business relationship between a customer and a service provider. It is used to specify the performance and features of a service but also the penalties that apply in case the service is not provided at the agreed quality. The technical part of the SLA is referred to as the Service Level Specification (SLS). This defines detailed technical parameters for each service level such as the associated QoS class, latency, throughput, and bandwidth allocation.

### 2.5.2 Multi-Protocol Label Switching and Traffic Engineering

Rapid changes in the type (and quantity) of traffic handled by the Internet is putting an enormous strain on the network infrastructure. Since routing protocols have little visibility into the Layer 2 characteristics of the network, particularly in regard to QoS and loading, packet forwarding relies upon Layer 3 to determine the path to the destination. The hop-by-hop packet processing performed by protocols such as OSPF (Open Shortest Path First) can have intensive CPU requirements thus reducing throughput in the network.

Multi-Protocol Label Switching (MPLS) [77] changes the hop-by-hop paradigm by enabling devices to specify paths in the network based upon QoS and bandwidth needs of applications. The forwarding decision is based on the exact-match algorithm using a fixed-length, fairly short label as an index. This enables a simplified and faster forwarding procedure, relative to longest-match forwarding traditionally used at the network layer.
Chapter 2. Background and Related Work

According to [78], Traffic Engineering (TE) is the process where data is routed through the network according to a management view of the availability of resources and the current and expected traffic. The objective of TE is twofold; users expect certain performance from the network, which in turn should attempt to satisfy these expectations. Hence, the target is to accommodate as many traffic requests as possible by optimally using the available network resources.

While TE can be IP-based, where multiple link weights corresponding to different DSCPs can allow routes to be computed per QoS class, an MPLS-based solution allows for sophisticated routing capabilities as well as QoS resource management techniques [79]. MPLS TE relies on an “explicitly routed” paradigm, where a set of routes is computed offline for the various QoS classes. Additionally, appropriate network resources (e.g. BW) may be provisioned along the routes according to predicted traffic requirements. This can allow an administrator to explicitly define paths between source and destination to ensure QoS or have the traffic follow a specified path to reduce traffic loading across certain hops.

2.5.3 The TEQUILA Framework

While the DiffServ QoS model specifies control and data plane mechanisms for providing QoS, there is a need for network and service management functionality, which is an integral part of QoS-based telecommunications networks. The European IST project TEQUILA [80] proposed an architecture for managing QoS in IP Differentiated Services Networks with MPLS capability. This architecture addresses both service and network management issues and can be seen as a detailed decomposition of the concept of a Bandwidth Broker realized as a hierarchical, logically and physically distributed system [79].

![Figure 2-9: The TEQUILA QoS management architecture [79]]
As depicted in Figure 2-9, the architecture is decomposed into three major sub-systems: SLS Management (SLS-M), Traffic Engineering (TE) and Monitoring. SLS Management is responsible for agreeing the customers’ QoS requirements in terms of SLSs, while Traffic Engineering is responsible for fulfilling the contracted SLSs by deriving the parameters for configuring the network devices. The Monitoring sub-system provides the above systems with the appropriate network measurements and assures that the contracted SLSs are indeed delivered at their specified QoS.

The DiffServ/MPLS QoS management framework developed by the TEQUILA project serves as the application domain of the policies defined in the next chapter and of the conflict analysis approach proposed in this thesis. The conflict detection and resolution examples presented in Chapters 4 and 5 demonstrate the applicability of the approach in this domain.

2.5.3.1 SLS Management

SLS Management is decomposed into 3 functional blocks. The SLS Subscription (SLS-S) includes processes of customer registration and long-term SLS admission control. The customer might either be a peer Autonomous System (AS), a business user or residential user. The subscription (or registration) concerns the Service Level Agreement (SLA), which includes prices, terms and conditions and the technical parameters of the SLS. Subscribed SLSs are stored in a local repository and serve as basic input for the Traffic Forecast (TF). The main function of TF is to generate a traffic estimation matrix per QoS class type for the long-term estimated traffic that flows between each ingress-egress pair in the network. TF is the “glue” between the customer-oriented (SLS-M) and the resource-oriented (TE) frameworks of this functional architecture. SLS Invocation (SLS-I) includes the process of dynamically dealing with a flow and is a part of the control plane functionality. Its main responsibility is to perform dynamic admission control on the traffic injected in the network as a response to customer demand. SLS-I receives operational guidelines from SLS-S, its functionality is distributed across the network edges, and has a view on the current spare resources.

2.5.3.2 Traffic Engineering

Traffic Engineering is decomposed into 3 functional blocks. Network Dimensioning (ND) is a centralised off-line component that has a global view of the network. It performs long-to-medium term configuration of the network and is responsible for mapping the expected traffic demand, provided by the traffic matrix, onto the physical network resources. Its objective is to accommodate the expected demand, and therefore meet the SLS performance requirements, without overloading any part of the network. The output configuration is in terms of MPLS Labelled Switched Paths (LSPs) and anticipated loading for each QoS class on all interfaces. This
output is fed to Dynamic Route Management (DRtM) and Dynamic Resource Management (DRsM), and also to SLS-M (in terms of a resource availability matrix) in order to base the admission control decisions for future SLS subscriptions.

DRtM is distributed, operating at each edge router and is responsible for managing the routing processes dynamically according to the guidelines produced by ND on routing traffic according to QoS requirements. This includes setting up routing parameters so that incoming traffic is routed to LSPs proportionally to the bandwidth determined by ND and modifying the routing of traffic according to feedback received from monitoring. DRsM is also distributed, with an instance attached to each router interface and aims to ensure that link capacity is appropriately distributed among the QoS classes sharing the link. This is achieved by dynamically configuring buffer and scheduling parameters according to ND directives, constraints and rules and taking into account the actual experienced load as opposed to required (predicted) resources.

The interactions between SLS-M and TE functions occur only at Resource Provisioning Cycle (RPC) epochs, not at the granularity of every single (or a few) service request(s). That is, a traffic matrix is produced and the network is appropriately engineered only at the start of a RPC. RPCs are relatively long time periods, ranging from hours to days and use traffic forecasts that are usually drawn with long-term perspectives.

2.5.3.3 Monitoring

The state-dependent dynamic SLS-M and TE functions require constant observation of the network in order to apply control actions and drive it to a desired state. Although TEQUILA proposes three types of monitoring – at the node, network and service levels – only the first is relevant to the work presented in this thesis. Node monitoring is a distributed process across all routers, observing resource utilization of each of the QoS classes. Monitoring at core routers can, for example, trigger DRsM functions to adjust the allocation of link resources for the various QoS classes, and monitoring at ingress routers can trigger SLS-I operations to adjust the inbound rate of traffic flows.

2.5.4 DiffServ QoS Policies

Since the IETF initiative on policy-based management, a number of DiffServ policies have been defined in the literature addressing different aspects of QoS management. The main works are described below.

2.5.4.1 IETF QPIM

The IETF Policy Framework Working group produced an object-oriented information model for representing QoS management policies (QPIM) [81]. This is based on the IETF PCIM and defines
a model for QoS enforcement for both IntServ and DiffServ using policies. Concentrating on the later, the model defines a group of actions to control BW allocation and congestion control differentiations, which collectively specify the per-hop behaviour forwarding treatment (QoSPolicyPHBAAction class). The example in Listing 2-4 provides a set of rules that specify PHBs enforced within a DiffServ domain. Here, the AFI condition matches the entire AFI PHB group.

<table>
<thead>
<tr>
<th>Listing 2-4: PHB policy rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (EF) then do EF actions</td>
</tr>
<tr>
<td>if (AF1) then do AF1 actions</td>
</tr>
<tr>
<td>if (AF11) then do AF11 actions</td>
</tr>
<tr>
<td>if (AF12) then do AF12 actions</td>
</tr>
<tr>
<td>if (AF13) then do AF13 actions</td>
</tr>
<tr>
<td>if (DE) then do DE actions</td>
</tr>
</tbody>
</table>

The actions of the corresponding PHBs can be any of the groups of actions mentioned above. The example below shows how a Bandwidth action can be used to define the BW allocation for the EF PHB:

<table>
<thead>
<tr>
<th>QoSPolicyBandwidthAction EF:</th>
</tr>
</thead>
<tbody>
<tr>
<td>qpBandwidthUnits: %</td>
</tr>
<tr>
<td>qpMinBandwidth: 50%</td>
</tr>
</tbody>
</table>

2.5.4.2 **Ponder DiffServ Policies**

The work in [82] and [83] presents a framework for the management of network services using Ponder and focuses on the dynamic adaptation of policies in response to changes within the managed environment. The term “Policy Adaptation” is used to describe the ability of the policy-based management system to modify network behaviour in one of the following ways:

- Adaptation by dynamically changing the parameters of a QoS policy to specify new attribute values for the run-time configuration of managed objects.
- Adaptation by selecting and enabling/disabling a policy from a set of pre-defined QoS policies at run-time. The parameters of the selected network QoS policy are set at run-time.

In this context policies have been defined to provide Per Domain Behaviours (PDB) in a DiffServ environment, to handle service performance degradation, and to support changes in routing or link failures. An example of the former is presented in Listing 2-5 where a PDB policy is specified as a Ponder obligation rule.
Listing 2-5: Policy rule for providing a specific PDB

| inst oblig | /Policies/PDBPolicy1 { |
| subject | /PMAS/DiffServAgent; |
| target | r = /DiffServDomainA/Routers/CoreRouters; |
| on | PDB1_ConfigRequest(DS, max_input_rate, min_output_rate); |
| do | r.applyEFPHB(DS, max_input_rate, min_output_rate); |
| when | max_input_rate <= min_output_rate; |

The actual implementation of the PDB policy, i.e. the implementation of the PHB (or the set of PHBs) that will guarantee the QoS characteristics to the corresponding traffic aggregate, is hidden from the customer. The customer (human or automated agent) is offered the externally observable PDB QoS attributes. The policy rule in this example will configure the core routers within the DiffServ domain to implement the EF PHB on the corresponding traffic aggregate.

2.5.4.3 TEQUILA Policies

The main drawback of the approaches presented in the two preceding sections is that they fail to address network-wide resource management policies as well as policies related to SLS management. This was the objective of the work by Flegkas in [84], which defines policies for the various components of the TEQUILA architecture.

Although a small number of policies target SLS admission control, the main focus of this work concerns policies for network dimensioning, presented in [85] and [86], which were used as proof of concept on the feasibility and benefits of policy-based management. Listing 2-6 shows two example policies, the first of which concerns a rule that creates an explicit LSP following specific nodes in the network with an associated BW of 2 Mbps. The second rule is a resource allocation optimisation policy that prevents parts of the network to get overloaded. This is achieved by directly influencing the dimensioning algorithm and increasing a specific parameter (Exponent) when the load on any link exceeds a specified value (in this case 80%). It should be noted that the policy format closely resembles that of IETF.

Listing 2-6: TEQUILA ND policies

(1) If OA==EF and Ingress==4 and Egress==6 then Setup LSP 4-9-7-6 2Mbps
(2) If maxLinkLoad > 80% then Increase Exponent by 1
The effect of enforcing the two policies above is depicted on the left part of Figure 2-10, which shows the newly created LSP and the even spreading of load across the network (before the enforcement the link between nodes 5 and 6 was heavily loaded – indicated with red colour).

Figure 2-10: TE-GUI snapshots before (above) and after (below) the enforcement of load balancing policies [84]
The next chapter extends this work to cover a wider spectrum of TE and SLS management policies thus providing a richer set of programmable functions. These form the subject of conflict analysis.

2.6 Summary

This chapter presented the necessary background information and prior work on the topics related to this thesis. More specifically, an overview of policy-based management was initially provided, focusing on the main frameworks proposed in the literature. The core part of the chapter described the various conflicts that may arise between policies used in various application domains, and presented the techniques employed by other works for effective detection and resolution. The shortcomings of these approaches were identified serving as a useful basis for comparison when demonstrating the advantages of the proposed approach in subsequent chapters.

The last part of the chapter presented information about QoS in IP networks, which is the application domain of the policies defined in this thesis. In addition to the various technologies discussed, the details of the TEQUILA QoS management framework were presented, and a flavour of QoS policies was given with specific examples.
3 Policies for DiffServ QoS Management

In order to provide a holistic approach for Quality of Service management in DiffServ networks, a range of management operations need to be deployed from traffic engineering and admission control, to dynamic management of resources. Several frameworks have been proposed for this purpose that mainly stemmed from European collaborative research projects including TEQUILA [87], MESCAL [88], and ENTHRONE [89]. All frameworks propose the use of a general model depicted in Figure 3-1, where the QoS management goals are realized by three distinct management blocks: Service Management, Traffic Engineering, and Policy Management. The first is responsible for agreeing the customers’ or peer domain’s QoS requirements in terms of Service Level Specifications (SLSs), while Traffic Engineering is responsible for fulfilling contracted SLSs by deriving the network configuration. Policy Management provides the two aforementioned blocks with a set of policies that guide their functional behaviour to reflect the high-level goals and objectives.

Quality of Service management has always been one of the most popular application domains of policies since Internet Service Providers (ISPs) can realize their objectives through flexible programmability with respect to offered services and treatment of customer traffic in their network. A small number of policies have been defined for some of the components of the QoS management framework described in Section 2.5.3 [85]. This chapter provides a comprehensive
set of QoS management policies and explains how their enforcement influences the behaviour of the managed modules and associated IP DiffServ managed objects. The policies are categorized into service management and traffic engineering policies and follow the two-level hierarchy of Figure 3-1. The latter depicts specific QoS management modules, initially proposed by the TEQUILA framework (see Section 2.5.3), targeted by the policies defined in this chapter. These are SLS Subscription (SLS-S) and SLS Invocation (SLS-I) on the service management side, and Network Dimensioning (ND) and Dynamic Resource Management (DRsM) on the traffic engineering side.

The structure of this chapter is as follows.

Section 3.1 introduces the types of policies that can potentially drive the QoS management framework and classifies them based on their functional applicability and their temporal characteristics. A description of the format used for policy specification is also provided along with the management entities involved in the policy enforcement process.

Section 3.2 describes in detail the functionality of the service management subsystem of the QoS framework, and in particular the SLS-S and SLS-I modules. Using the format provided by the Ponder language, the various policies that can be used to drive the two modules are specified, along with a description of their effect on the system's functional behaviour and the management objectives they can achieve.

Following on, Section 3.3 describes the ND and DRsM modules, which constitute the main body of the traffic engineering subsystem. The policies presented here aim to provide the network with both static and dynamic configurations such that the treatment of traffic meets the QoS guarantees negotiated during the service subscription phase.

To complete the policy influenced functionality of the QoS management framework, Section 3.4 provides state chart representations for the managed modules, which form an essential part of the conflict analysis approach and are used to model the effect of policy enforcement. This section also describes the various managed objects in the framework and the operations they support which can be invoked through policies. Finally, Section 3.5 summarises the chapter.

### 3.1 Policy Classification, Specification and Enforcement

The functionality of the service management and traffic engineering sub-systems is realized by two modules on either side and follows a two-level hierarchy that reflects the off-line and runtime operational mode of the model. As such, individual modules can be categorised both logically based on their functionality, and hierarchically depending on to which layer of the hierarchy they belong. The SLS-I and DRsM modules, located at the lower level, perform
measurement-based dynamic functions, whereas the SLS-S and ND, residing at the upper layer, are time-based off-line modules.

A similar classification can be applied to the policies driving the behaviour of these modules as shown in Figure 3-2. At the first level, policies are classified based on the functional domain of their applicability. Service management policies apply to the SLS-S and SLS-I modules and are mostly related to directives regarding static and dynamic admission control decisions. TE policies on the other hand apply to the ND and DRsM modules and control static and dynamic resource allocation functions. The second level of the classification differentiates policies based on their activation characteristics. Policies that apply to the upper layer of the hierarchy, i.e. to the SLS-S and ND modules, are more time-dependent and their execution depends mostly on the state of the module. Policies specific to SLS-I or DRsM can be either activated by the module state itself or by conditions and events emanating from the network. Off-line policies applying to the dynamic modules are, for example, policies that initialise essential system variables, whereas dynamic policies perform resource management functions at the edge or the core of the network and are triggered by network state notifications received from monitoring components.

As described in Sections 2.1 and 2.5, various ways have been proposed to specify policies. Although the condition-action approach from IETF has been adopted by a number of people in their research work [5][86][90] and parts of the PCIM specification have been implemented in commercial products [91][92], it imposes limitations when designing policies for event-driven
systems such as TEQUILA. To have a complete policy representation for such environments it would be necessary to model events as condition objects. Furthermore, the model only supports a single type of rule and must undergo significant extension to support all the different policy types enforceable in a system. The PMAC specification on the other hand supports such requirements, but the fact that policy definitions have an XML representation makes them verbose and not easily interpreted by a user. The Ponder language overcomes the above issues and has been chosen for the specification of policies in this thesis.

Out of the range of Ponder policy types, obligations are of particular interest to this work. These are event-triggered-condition-action rules and can be used to specify management operations that must be performed when a particular event occurs given some supplementary conditions being true. They are specified in terms of a subject that should perform a particular action on a target when a specified condition is true as in the example of Listing 3-1.

Listing 3-1: Ponder obligation policy example

```
inst oblig /policies/qosmodule/PolID {
  on event(params);
  subj s = qosmodule/PMA;
  targ t = qosmodule/MO;
  do t.action(params);
  when constraints;
}
```

In the context of the QoS management framework, policy triggers can be internal system events generated by a management module indicating operational phases like the start of a new resource provisioning cycle, or network events reporting on the state of managed resources. The different operations supported by the modules are used to encode the action part of the obligation policy. These are exposed, through management interfaces, by managed objects (MOs) which act as the policy targets. The entity responsible for enforcing policies, and thus adopting the subject role, is the Policy Management Agent (PMA) [93] – in our application domain, every instance of a QoS management module is governed by a separate PMA. The last field in the specification example is an optional one and is mainly used to constrain the applicability of the policy with respect to time; it is particularly useful when an administrator needs to specify different network configurations for busy or non-busy hours of the day.
Figure 3-3 depicts the process by which policies are enforced onto a QoS management module. Notifications are received in the PMA by an event handler and passed for processing to the policy enforcer. The latter inquires the local repository about policies that should be activated by the received event. For candidate policies, the relevant MOs are identified as the targets, from the set of MOs supported by a QoS management module, and policy actions are invoked through their management interfaces. The MOs respond by generating one or more low-level operations that realise the objectives of each of the policies being enforced. These drive the functional behaviour of the managed module and can yield new parameters for the configuration of managed resources.

3.2 Service Management Policy Driven Functionality

This section describes in detail the functionality of the SLS Subscription and SLS Invocation modules of the QoS management architecture which essentially perform static and dynamic admission control on network resources, respectively. Using the Ponder format we define the various policies that can be deployed by the SLS-S and SLS-I PMAs, describing their triggering events, constraints (where applicable), and the functionality that can be achieved through their enforcement. Although the MOs associated with individual policy actions are also included in the policy specification for reasons of clarity, their functional description can be found in Section 3.4.1.

3.2.1 SLS Subscription Policies

The main objective of subscription logic is to control the number and type of service subscriptions, aiming to avoid overloading the network, whilst at the same time maximizing subscribed traffic. Service management policies can be used to guide the subscription process by
not only specifying the conditions under which a request is to be accepted, but also defining specific parameters that indirectly influence this decision, thus expressing business objectives. Such parameters are, for example, service multiplexing factors and service quality levels as described in the next section.

3.2.1.1 Service Satisfaction

The confidence with which a service is provided can not be derived in absolute terms. As such, subscriptions can be relatively differentiated (a) on the basis of the confidence with which they will enjoy a target minimum contractual traffic, and/or (b) in terms of how badly they will be affected in case of congestion. To the above end, the following definitions are used [94]:

- **Almost Satisfied Service Rate** ($SR_{AS}$) – Denotes the traffic rate, which if offered to a SLS, the SLS is thought to be “almost satisfied”; expressed in BW units.

- **Fully Satisfied Service Rate** ($SR_{FS}$) – Denotes the traffic rate, which if offered to a SLS, the SLS is thought to be “fully satisfied”; expressed in BW units.

The above service rates are based on factors (multiplexing factors – MFs) that define SLS multiplexing. The latter takes into account traffic descriptors of the SLSs under aggregation, expressed in terms of parameters such as peak bit rate and QoS requirements. Multiplexing factors are defined per QoS class (QC), they range from 0 to 1, and are inversely proportional to QoS requirements.

By defining two MFs through policies, one for each of the service rates, an administrator defines the risk that he/she is willing to take in satisfying the agreed SLS – an increasing MF achieves lower QoS guarantees. The left part of Listing 3-2 gives an example specification of a policy, $P_{1.1}$, setting the almost ($setAlmstSatisf()$) and fully ($setFullSatisf()$) satisfied factors in Ponder format. This policy type is triggered at the start of a new RPC and the action is enforced by the subscription module’s PMA on the managed object (MO) supporting service satisfaction operations ($servSatisfMO$).

**Listing 3-2: Service satisfaction policies**

<table>
<thead>
<tr>
<th>inst oblig /policies/sslss/P1.1</th>
<th>inst oblig /policies/sslss/P1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>on newRPC();</td>
<td>on newRPC();</td>
</tr>
<tr>
<td>subj s = sslssPMA;</td>
<td>subj s = sslssPMA;</td>
</tr>
<tr>
<td>targ t = sslss/servSatisfMO;</td>
<td>targ t = sslss/servSatisfMO;</td>
</tr>
<tr>
<td>do t.setAlmsSatisf(QC, MF)</td>
<td>do t.setFullSatisf(QC, MF);</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In addition to multiplexing factors, service objectives can further be expressed through an *Overall Quality Level* (OQL) parameter. This is associated with the risks in “adequately satisfying” subscriptions and is used as a measure of the degree of quality/satisfaction a provider would opt to provide to subscriptions. For a given SLS, the degree of satisfaction is analogous to the confidence level with which the SLS is to enjoy the agreed QoS at its SRAS. OQL is defined per QoS class as in policy P1.2 of Listing 3-2, and has a range from -1 to 1 as follows:

- **OQL = 0** – In cases of congestion there is high confidence that active SLSs would enjoy their QoS at their SRAS.
- **OQL = 1** – In cases of congestion there is high confidence that active SLSs would enjoy their QoS at their SRFS.
- **OQL = -1** – In cases of congestion no guarantees can be provided for ensuring QoS.
- **OQL increases (0, 1)** – In cases of congestion the higher the OQL the more likely is to ensure that SLS traffic rates higher than SRAS (i.e. near to SRFS) will get their QoS.
- **OQL decreases (-1, 0)** – In cases of congestion the lower the OQL the less likely is to ensure that SLSs will get their QoS at their SRAS.

### 3.2.1.2 Resource Availability Buffer Limit

The subscription admission control logic decisions are based on the Resource Availability Matrix (RAM) calculated by the Traffic Engineering block. The RAM provides an availability estimate per Traffic Trunk (TT), which is expressed in the form of a Resource Availability Buffer (RAB) as shown in Figure 3-4. $R_{\text{min}}^s$ represents the available bandwidth for this TT guaranteed by the network at any time. $R_{\text{min}}^w$ represents the available bandwidth for this TT guaranteed by the network at congestion times. This bandwidth is not hard-reserved by the network and it can be utilised by other TTs but when congestion occurs the Traffic Engineering system will force all TTs to be constrained to their $R_{\text{min}}^w$ bandwidth limit. Finally, $R_{\text{max}}$ represents the maximum available bandwidth for this TT but with no guarantees.

![Figure 3-4: Resource Availability Buffer](image)

The decision to accept a new SLS request, is based on a maximum limit in the buffer. This bound, *subscription upper* (SU), can be set by policies and its relative placement in the buffer is determined by the OQL of the associated QoS class. Two main areas can be defined, in the buffer, and interpreted in terms of confidence levels. The buffer up to $R_{\text{min}}^w$ can be used with high
confidence levels because the network has been engineered so that this buffer can be at least available even at times of congestion. The buffer from $R_{\text{min}}^w$ to $R_{\text{max}}^w$ is considered more risky because the network cannot provide any guarantees for the rates in this area; this is due to the fact that other TTs share the same physical resources with the TT under consideration. To this extent, three policies can be defined to set the value of SU (P1.3–P1.5) as shown in Listing 3-3, each expressing a different level of associated risk. The higher the value of SU the more subscriptions are accepted, thus increasing the profit, but the lower the guarantees offered to customers.

### Listing 3-3: Resource Availability Buffer policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>SU Expression</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.3</td>
<td>$SU = R_{\text{min}}^w + (R_{\text{max}}^w - R_{\text{min}}^w) \cdot (1 - OQL)$</td>
<td>OQL &gt; 0</td>
</tr>
<tr>
<td>P1.4</td>
<td>$SU = R_{\text{min}}^w + (R_{\text{max}}^w - R_{\text{min}}^w)$</td>
<td>OQL = 0</td>
</tr>
<tr>
<td>P1.5</td>
<td>$SU = R_{\text{min}}^w + (R_{\text{max}}^w - R_{\text{min}}^w)</td>
<td>OQL</td>
</tr>
</tbody>
</table>

Policies of this type are triggered upon receipt of a new RAM from the TE block and their actions are supported by the managed object responsible for buffer manipulations (bufferMO). These policies set SU in the area between $R_{\text{min}}^w$ and $R_{\text{max}}^w$ following a conservative, moderate or risky strategy depending on the OQL value of the QoS Class associated with a specific TT. The latter is expressed as a constraint in the policy where, for example, the limit associated with high-priority QoS classes (OQL close to 1) is set in a conservative fashion, close to $R_{\text{min}}^w$. The relative value of SU associated with each of the strategies is defined through policy P1.6 and is a function of OQL as follows:

- **Conservative** – $SU = R_{\text{min}}^w + (R_{\text{max}}^w - R_{\text{min}}^w) \cdot (1 - OQL)$ for OQL > 0
- **Moderate** – $SU = R_{\text{min}}^w + (R_{\text{max}}^w - R_{\text{min}}^w)$ for OQL = 0
- **Risky** – $SU = R_{\text{min}}^w + (R_{\text{max}}^w - R_{\text{min}}^w) |OQL|$ for OQL < 0
Chapter 3. Policies for DiffServ QoS Management

3.2.1.3 Admission Control Decision

When a new subscription request arrives, the anticipated traffic demand (TD) of the newly requested SLS is aggregated with the demand of already subscribed SLSs. Aggregating SRAs and SRFS of the new SLS yields new values for minimum and maximum traffic demand denoted by TD\text{min} and TD\text{max} respectively, which are parameters maintained by the subscription logic. The admission control decision is based on policies that define the conditions under which a SLS request is to be accepted, rejected or a counter-offer is to be made, following the guidelines of Table 3-1.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Action when conditions true</th>
<th>Action when conditions false</th>
</tr>
</thead>
<tbody>
<tr>
<td>OQL = 0, TD\text{min} &lt; SU</td>
<td>accept</td>
<td>reject / counter-offer</td>
</tr>
<tr>
<td>OQL = 1, TD\text{max} &lt; SU</td>
<td>accept</td>
<td>reject / counter-offer</td>
</tr>
<tr>
<td>OQL = -1, TD\text{min} &lt; SU</td>
<td>accept</td>
<td>reject / counter-offer</td>
</tr>
<tr>
<td>OQL &gt; 0, TD\text{min} &lt; R\text{max}_{\text{min}}</td>
<td>accept</td>
<td>reject / counter-offer</td>
</tr>
<tr>
<td>1 &gt; OQL &gt; 0, TD\text{min} &lt; SU</td>
<td>accept</td>
<td>reject / counter-offer</td>
</tr>
<tr>
<td>0 &gt; OQL &gt; -1, TD\text{min} &lt; SU</td>
<td>accept</td>
<td>reject / counter-offer</td>
</tr>
</tbody>
</table>

The various conditions for handling a SLS request are encoded as constraints in the policy as in the example of Listing 3-4, which corresponds to the first row of Table 3-1. This policy is triggered once the new anticipated traffic demand has been derived and its action is enforced through the admission control MO (acmo). Similar accept/reject policies can be defined for the rest of the constraints.

Listing 3-4: Admission Control policy

```plaintext
inst oblig /policies/sls/s1.7 {
  on antDemCalced();
  subj s = slsSPMA;
  targ t = sls/acmo;
  do t.accept(SLS);
  when t.getValue(SLS.TT.QC, OQL) == 0 &&
    (t.getValue(SLS.TT, TDmin) < t.getValue(SLS.TT, SU));
}
```

3.2.2 SLS Invocation Policies

The SLS-I module is responsible for the invocation management of already subscribed services. The task of regulating the traffic entering the network encompasses two aspects: (a) control the
number and the type of the active services, and, (b) control the volume and the type of the traffic injected by the active services. The main objectives of invocation management are the following:

- To maximise the use of network resources, and thus the number of admitted services and the QoS they enjoy.
- To prevent QoS deterioration caused by overwhelming the network.

Service management policies can be used to guide the invocation process, which, in contrast to the static nature of the subscription module, is mainly based on run-time events. These are generated by a monitoring component, integrated within the SLS-I module, providing threshold crossing alarms on aggregate traffic (per TT) injected into the network by the local edge. The events act as triggers to policies that regulate the traffic entering the network and aim to achieve the above stated objectives.

### 3.2.2.1 Module Initialisation

At the start of every new RPC some parameters essential for the operation of the module are initialised. These are attributes of TTs originating from a specific ingress node and include the rates that are thought to almost/fully satisfy a service (SRAS, SRFS), minimum and maximum admission control parameters (ACmin, ACmax), as well as monitoring thresholds that indicate target-critical and very-critical levels (TCL, VCL) of traffic flowing into the network. Service rates are aggregates of independent rates of services supported by a TT provided by SLS-S, they remain constant throughout the duration of a RPC, and serve as guidelines for the allocation of resources. Admission control parameters are based on the same RAM as the one used by SLS-S, their initial values range between $R_{\text{min}}^*$ and $R_{\text{max}}^*$, and they influence the probability of accepting new service invocations. Policy P2.1 below, enforced by the SLS-I PMA, encodes the relevant actions that initialise service rates and admission control parameters. The latter are manipulated at run-time by policies described in the next section to reflect the current status of the network.

#### Listing 3-5: Initialisation policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2.1</td>
<td>Initialisation policies</td>
</tr>
</tbody>
</table>

```c
inst oblig /policies/slsI/P2.1 {  
on newRPC();  
subj s = slsIPMA;  
targ t = slsI/initmo;  
do {  
t.setSRas(TT, Value) ->  
t.setSRfs(TT, Value) ->  
t.setACmin(TT, Value) ->  
t.setACmax(TT, Value);  
}
}
```

```c
inst oblig /policies/slsI/P2.2 {  
on newRPC();  
subj s = slsIPMA;  
targ t = slsI/monitormo;  
do {  
t.setTCL(TT, Value) ->  
t.setVCL(TT, Value);  
}
```
Policy P2.2 sets TCL and VCL thresholds through the monitoring MO. These remain constant throughout a RPC and are used to proactively inform the SLS-I module when the levels of admitted traffic can potentially overwhelm the network. TCL and VCL values are relative to $R_{\text{min}}^w$ and $R_{\text{max}}$ and can be derived with the following equation, where $A$ is the percentage of the critical area:

$$\text{threshold} = R_{\text{min}}^w + A \ast (R_{\text{max}} - R_{\text{min}}^w)$$

The relative placement of the thresholds in the buffer defines the objectives of an administrator – the closer they are to $R_{\text{max}}$, the higher the risk of QoS deterioration as notifications will not be issued until the buffer is heavily utilised. The next section describes the various policies triggered at run-time by threshold notifications and their effect on the module’s behaviour.

### 3.2.2.2 Admission Control

The run-time operation of the module is triggered by threshold crossing alarms. These initiate a set of policy actions that control the rates of incoming traffic and change invocation admission control parameters. The latter influence the treatment of new service invocations: the closer the aggregate value of current TT utilization and the requesting SLS traffic rate to $AC_{\text{max}}$ than $AC_{\text{min}}$, the less the chances of the SLS being successfully invoked.

Four event types can emerge from the specified thresholds; two for each of TCL and VCL when crossed upwards or downwards by the traffic entering the network (TRIN). Three of these events act as triggers (slsIAlarMRaised) to the policies of Listing 3-1 which enforce actions supported by the service adjustment managed object (servAdjustMO).

#### Listing 3-6: Admission Control policies

```plaintext
inst oblig /policies/slsI/P2.3 {
  on slsIAlarMRaised(tclUp, TT);
  subj s = slsIPMA;
  targ t = slsI/servAdjustMO;
  do t.decrSR(TT, value) -
      t.decrACmin(TT, value);
  when duration(HH:MM-HH:MM);
}

inst oblig /policies/slsI/P2.4 {
  on slsIAlarMRaised(tclDown, TT);
  subj s = slsIPMA;
  targ t = slsI/servAdjustMO;
  do t.incrSR(TT, value) -
      t.incrACmin(TT, value);
  when duration(HH:MM-HH:MM);
}

inst oblig /policies/slsI/P2.5 {
  on slsIAlarMRaised(vclUp, TT);
  subj s = slsIPMA;
  targ t = slsI/servAdjustMO;
  do t.decrSR(TT, value) -
      t.decrACmin(TT, Value) -
      t.decrACmax(TT, Value);
}
```

49
Upon an upward TCL threshold crossing alarm policy P2.3 decreases the minimum admission control parameter – thus decreasing the probability of accepting new invocations – and the service rate offered to a particular TT. The amount by which these are decreased is expressed as a percentage of their current value. The effect of the policy enforcement is depicted in Figure 3-5 (time t1, t4), where proactive measures are taken to avoid potential congestion built-up.

The opposite actions are enforced when the TCL is crossed downwards by TRIN (Figure 3-5, time t3). Policy P2.3 increases ACmin and the service rate and can potentially restore the normal mode of operation where services enjoy their fully satisfied rates. Upward VCL threshold crossing alarms indicate more severe actions (policy P2.3) whereby service rates are further decreased and can be as low as SRAS (Figure 3-5, time t2). In addition to ACmin, ACmax is also decreased which in effect reduces the probability of accepting new invocations even more. In the event that VCL is crossed downwards no corrective policy actions are defined so as to allow the rate to eventually reduce to lower levels. It should be noted that these policies can also include a time constraint, as in the examples involving TCL events, which can be used to define alternative action parameters for different hours of the day, e.g. peak and off-peak hours, through multiple policies.
3.3 Traffic Engineering Policy Driven Functionality

This section describes in detail the functionality of the Network Dimensioning (ND) and Dynamic Resource Management (DRsM) modules of the QoS management architecture. The overall objective of these modules is the static and dynamic management of resources in terms of BW allocation, route establishment, and resource optimisation with respect to the various QoS classes (QCs) supported in the network. As in the previous section, the specification of the relevant policies is provided in Ponder format along with descriptions of the functionality achieved through their enforcement. A collective description of the MOs involved can be found in Section 3.4.2.

3.3.1 Network Dimensioning Policies

Network Dimensioning performs the provisioning activities of the management system. It is responsible for the long to medium term configuration of the network. Configuration involves the setup of Label Switched Paths (LSPs) as well as the parameters required for the operation of QCs on every link, e.g. bandwidth, priority, weight. The values provided by ND are not absolute but come in the form of a range, constituting directives for the function of the QCs, while for LSPs they come in the form of multiple paths in order to enable multi-path load balancing. The exact configuration values are determined by dynamic TE functions based on the actual state of the network at any point in time as described in Section 3.3.2.

ND runs periodically, by first requesting the predictions (traffic matrix) for the expected traffic per QC in order to be able to compute the provisioning directives. The dimensioning period is typically in the time scale of a week while the forecasting period is in the time scale of hours. The latter is a period in which we have considerably different predictions as a result of the time schedule of the subscribed SLSs. For example, ND might run every Sunday evening and provide multiple configurations i.e. one for each period of the day (morning, evening, night). So, effectively the provisioning cycle is at the same time scale as that of the forecasting period.

The objectives of ND are to optimally distribute the projected traffic over the network resources by minimizing the overall cost and at the same time not overloading parts of the network while others are under-loaded. In general, this problem can be formulated as a network flow optimisation problem [95]. The cost of each link can be defined as the sum of linear functions $f_h(x_{l,h})$ per QC, where $x_{l,h}$ is the load on the link $l$ from QC $h$. The total cost should be the sum of $f_h$ for all QCs over all links; this is the objective function to be minimized. ND goes through three main stages in order to produce a network configuration. The basic functionality of every stage is summarised below, whereas a detailed description of the algorithm can be found in [96].
-- Chapter 3. Policies for DiffServ QoS Management

- **Pre-processing stage**
  - Request traffic forecast per QC, i.e. Traffic Trunks (bandwidth, end-to-end delay, end-to-end loss probability requirements).
  - Obtain statistics for the performance of each QC at each link.
  - Determine the maximum allowable hop count $K$ per TT according to the above statistics.

- **Processing stage**
  For each TT find a set of paths for which:
  - The bandwidth requirements of the TT are met.
  - The delay and loss requirements are met (by using the hop count constraint as an upper bound).
  - The overall cost function is minimized.

- **Post-processing stage**
  - Allocate any extra capacity or reduce any over-allocated bandwidth to the paths up to the maximum link bandwidth according to policy.
  - Sum all the path requirements per link and configure the QCs.
  - Configure the appropriate label switched paths calculated in the optimisation phase.

ND is triggered by time rather than network state events from within the network. The policies applying to this module are resource provisioning policies that influence the way ND calculates the capacity allocation and the path creation and configuration of the network. These policies depend on the input from the Service Management block concerning the predicted volume of traffic and associated QoS classes, and their execution depends on the stage at which the module operates. The next sections describe the policies supported by the ND managed objects.

### 3.3.1.1 BW Allocation and Explicit Route Setup
The bandwidth allocation managed object (bamo) of this module allows for methods that explicitly define the way the BW should be allocated to different QoS classes. This is demonstrated by policy P3.1 (Listing 3-7), where an administrator can define the number of resources to be allocated per QC – giving a minimum ($N_{min}$), a maximum ($N_{max}$), or a range – with the BW value expressed as a percentage of the overall network capacity. Policies of this type are triggered once ND enters the processing stage. Policy P3.2 is supported by the LSP managed object (lspmo) and is invoked during the pre-processing stage of dimensioning. It is useful for setting up explicit paths for traffic that belongs to a particular QC and passes through a set of nodes defined by path with logically assigned bandwidth, $bw$. 

52
3.3.1.2 Hop Count Derivation

An important function of ND is to handle the QoS requirements of the expected traffic in terms of delay and packet loss. The implementation of ND functionality in [96] simplifies the optimisation problem by transforming the delay and loss requirements into constraints for the maximum hop count for each traffic trunk. This transformation is possible by keeping statistics for the delay and loss rate of the QCs per link. The accuracy of the statistics is determined by the period used to obtain them; smoothing methods, such as exponential weighted moving average, can be used.

Listing 3-8: Hop count derivation policy

```
inst oblig /policies/nd/P3.3 {
  on dONDPreProc();
  subj s = ndPMA;
  targ t = nd/hopsMO;
  do t.calHopCountStrg(QC);
}
```

Policy P3.3 of Listing 3-8 is triggered during the pre-processing stage and defines the way to derive the hop count constraint (calHopCountStrg()) for every QC. It allows for different strategies (strg) in achieving this objective by using the minimum (min), maximum (max), or average (avg) delay or loss along a route in order to derive the constraint. It is envisaged that by using the maximum an administrator is too conservative (appropriate for EF traffic), while by using an average the QoS requirements are possibly underestimated.

3.3.1.3 Optimisation Algorithm

The core component of ND is an optimisation algorithm and its objective is to find a set of paths for which the BW requirements of TTs are satisfied, the delay and loss requirements are met by using the hop count constraint as an upper bound, and the overall cost function is minimized. This is a non-linear optimisation problem which is solved by the gradient projection method [96].
Chapter 3. Policies for DiffServ QoS Management

Listing 3-9: Optimisation algorithm policies

| Policies governing the behaviour of the optimisation algorithm are triggered during the processing stage of ND, as in the examples of Listing 3-9, and influence the way the algorithm calculates the output configuration. Policy P3.4, supported by the managed object dealing with hop constraints (hopsMO), sets an upper bound on the number of hops the calculated paths are permitted to have (setMaxHops()). This number may vary depending on the QoS class to which the traffic belongs. Policy P3.5, supported by the LSP managed object, defines the number of alternative paths the optimisation algorithm should allow (setMaxAltPaths()) for every traffic trunk belonging to a specific QC, or even for a specific trunk, for the purpose of load balancing.

3.3.1.4 Spare/Over-provisioned BW Treatment

After the dimensioning algorithm executes, ND enters a post-processing stage where it assigns the residual physical capacity to the various traffic classes or reduces the allocated capacity because the link capacity is not enough to satisfy the predicted traffic requirements.

Listing 3-10: Post-processing policies

| The policies of Listing 3-10 can achieve the above objectives with different strategies (strg). Policy P3.6 defines the distribution of spare capacity (allocSpareBW()) among the traffic classes, which can be done equally (equal) between the QCs, proportionally (prop) to the current allocation, or explicitly (exp1), where the amount of BW is specified as a percentage. Following similar guidelines, policy P3.7 defines the strategy by which to reduce over-provisioned bandwidth (redOverBW()) in order to fit the physical link capacity. |
3.3.2 Dynamic Resource Management Policies

The resulting configuration of ND is based on historical data and customer subscriptions. As such, it is treated as a rough "nominal" value – actual offered traffic should fluctuate around forecasted values. For that reason, dynamic TE functions are deployed by the Dynamic Resource Management (DRsM) module, which has distributed functionality with an instance operating in every router. This module utilizes actual network state and load information in order to optimize network performance in terms of resource utilization while, at the same time, meeting QoS traffic constraints. In particular, DRsM opts for dynamic functions that manage network resources following the guidelines provided by ND and aims to guide the distribution of capacity between the QCs defined on a link.

3.3.2.1 Module Initialisation

At the start of every new RPC some parameters essential for the operation of the module are initialised. These hold resource allocation values which serve as functional constraints and are set by the actions of policy P4.1 below. The first two actions concern the minimum (NDMin) and maximum (NDmax) allocation directives received from the ND module and are used to set the boundaries between which the dynamic allocation of resources assigned to a QC should ideally reside. The third policy action (setRsrcAlloc()) is a general resource management directive that specifies the amount of link resources (as a percentage of link capacity) to be allocated among the various QCs during a DRsM operational cycle. Although the full link capacity should usually be available, an administrator may opt to allow for some BW dedicated to control traffic. It should be noted that the actions enforced by policy P4.1 are valid throughout the duration of a RPC.

Listing 3-11: Initialisation policy

```plaintext
inst oblig /policies/drsm/P4.1 {
  on newRPC();
  subj s = drsmPMA;
  targ t = drsm/initMO;
  do t.setAllocmin(Qc, NDMin)
  t.setAllocmax(Qc, NDmax)
  t.setRsrcAlloc(Link, BW);
}
```

3.3.2.2 Utilisation Tracking and Resource Allocation

In contrast to the static nature of the above policy actions, dynamic resource management policies [97] provide the flexibility to introduce logic on the fly, in the form of directives, for tracking the utilization of a QC and ensuring that the bandwidth allocated to that QC (allocQC) is in accordance with the required BW. The latter is determined according to observed utilization (loadQC). Figure
Chapter 3. Policies for DiffServ QoS Management

3-6 depicts the functionality that can be achieved by the execution of DRsM policies. It shows that when the monitored utilisation exceeds the upper threshold, the allocated bandwidth, upper and lower thresholds are increased. Similarly, when the utilisation crosses the lower threshold, these values are decreased. Monitoring QC utilisation is achieved through a monitoring component, rather than polling instantaneous values, which is integrated within the DRsM module. The triggering of policy actions is based on upper and lower thresholds of the BW consumed by a QC. The monitoring component will raise a threshold crossing alarm when the utilization exceeds the upper threshold or drops below the lower threshold.

![Figure 3-6: Bandwidth tracking of a single QC](image)

The calculation of new threshold and allocation values can be based on an algorithmic approach [98] that takes into account QC priorities as well as trend analysis of historical data. Alternatively, this could be achieved through explicit actions that only apply to the QC for which the alarm was raised. This means that each QC is treated independently through appropriate methods that modify thresholds and allocation upon threshold crossing alarms. Policies P4.2 and P4.3 below, react to an upper threshold crossing alarm (upprTh) where the monitoring managed object (monitormo) increases both upper and lower thresholds (incrThso), and the BW allocation managed object (bamo) increases the allocation (incrAlloc()) for a given QC on a particular link. The opposite actions are enforced (P4.4 and P4.5) when a lower threshold crossing alarm (lowrTh) is issued.

56
Chapter 3. Policies for DiffServ QoS Management

Listing 3-12: Dynamic resource management policies

```plaintext
inst oblig /policies/drsm/P4.2 {
    on drsmAlarmRaised(upprTh, Link, QC);
    subj s = drsMMA;
    targ t = drsm/monitorMO;
    do incrThSStrg(Link, QC, BW);
}

inst oblig /policies/drsm/P4.3 {
    on drsmAlarmRaised(lowrTh, Link, QC);
    subj s = drsMMA;
    targ t = drsm/baMO;
    do incrAllocStrg(Link, QC, BW);
}

inst oblig /policies/drsm/P4.4 {
    on drsmAlarmRaised(lowrTh, Link, QC);
    subj s = drsMMA;
    targ t = drsm/monitorMO;
    do decrThSStrg(Link, QC, BW);
    when duration(HH:MM-HH:MM);
}

inst oblig /policies/drsm/P4.5 {
    on drsmAlarmRaised(lowrTh, Link, QC);
    subj s = drsMMA;
    targ t = drsm/baMO;
    do decrAllocStrg(Link, QC, BW);
    when duration(HH:MM-HH:MM);
}
```

It should be noted that all policy actions of Listing 3-12 are appended with a `strg` keyword that signifies the different strategies with which an action can be achieved. For example, thresholds and allocation can be increased or decreased by an absolute value (Abs – e.g. kbps), a relative value (Rel – e.g. 5%), or by using a specific algorithm (Alg). A well known method would be to use an Exponentially Weighted Moving Average (EWMA) approach providing even more flexibility by setting parameters such as the size of the extrapolation window and the number of historical data to be used in the extrapolation function. Furthermore, policies can also include a time constraint, as in the specification of P4.4 and P4.5, which can be used to differentiate between the handling of threshold crossing alarms during peak and off-peak hours.

3.4 Managed Objects and System Behaviour Model

Implementing a policy-based infrastructure for a real system as complex as the one enabled by the QoS management architecture described is a not an easy task. For the purposes of this thesis, which focuses on policy analysis rather than policy deployment mechanisms, state machines are used to represent system behaviour and model the effect of policy enforcement on the managed modules. This section describes the various MOs and the operations they support in relevance to the presented policies, and provides state machine representations for the modules of the QoS management framework. Policy enforcement is modelled through state transitions, which yield a new set of configuration parameters. The latter constitute properties of TTs and QCs for service management and traffic engineering operations, respectively.
3.4.1 Service Management

To achieve its objectives, the SLS-S module employs three MOs exposing a set of methods that can be invoked through service subscription policies. These are as follows:

- **Service Satisfaction MO** (servsatisfmo) – Handles operations related to service satisfaction parameters. These include setting the SLS multiplexing factors that are used to derive the almost and fully satisfied service rates, and also defining the overall quality level of a service per QoS class.

- **Buffer Allocation MO** (buffermo) – Handles operations that define and execute the strategy with which the maximum limit for accepting new subscription requests is derived with respect to RAB parameters.

- **Admission Control MO** (acmo) – Provides operations that calculate the new anticipated traffic demand for an incoming subscription request and subsequently accept or reject that request based on policy constraints.

The methods provided by the MOs above guide the evolution of the SLS-S module in its operation through a number of states represented by the state machine diagram of Figure 3-7. The transitions between states are managed through policy execution.

The upper part of the state machine represents the initialisation phase of the module which takes place at the start of every new RPC. During this time activated policies set the values for multiplexing factors (M1.1), the service quality level (M1.2), and the maximum buffer limit (M1.3). The lower part represents the actual admission control process where policies guide the module to accept (M1.4) or reject (M1.5) a subscription request. It should be noted that a rejected request can

---

**Figure 3-7: SLS-S module behaviour and action summary**
result in a counter-offer process which is out of the scope of this thesis. Here, it is assumed that if the counter-offer is accepted by the customer the module proceeds to the configuring SLS state, otherwise it returns to idle.

Apart from state transitions, policy enforcement results in new configuration parameters. This is another function of the MOs, which have read/write access to an information store with the following structure:

\[
\text{TT}\{\text{QC(MFAS, MFSS, OQL), R}^{\text{min}}, R^{\text{max}}, \text{TD}^{\text{min}}, \text{TD}^{\text{max}}, \text{SU, SLS[ID}, \text{SRas, SRfs}]\}
\]

The value of one or more of the above parameters, attributes of a TT, can be updated after a policy has been successfully enforced, but can also be passed to the PMA during policy evaluation. This is necessary when evaluating policies with constraints requiring TT related information. Such an example is the policy for setting the upper subscription limit whose evaluation involves instantiating the OQL parameter.

The functionality of the SLS-I module is realised by operations that effectively aim to regulate the traffic entering the network and are supported by the following MOs:

- **Initialisation MO** \((\text{initmo})\) – Provides operations that initialise parameters essential for the operation of the module, including service rates and dynamic admission control parameters per TT.

- **Monitoring MO** \((\text{monitormo})\) – Provides operations that manage the functionality of the monitoring component. These include setting the TCL and VCL thresholds and generating alarms when the latter are crossed by traffic entering the network.

- **Service Adjustment MO** \((\text{servadjustmo})\) – Handles operations that manipulate the service rates and admission control parameters of a TT.

The behavioural representation of the SLS-I module is depicted in Figure 3-8 along with the various methods that result in state transitions once enforced through policies.
Figure 3-8: SLS-I module behaviour and action summary

The on-line admission control process for incoming traffic is triggered by threshold crossing alarms emanating from the monitoring component depicted in the lower part of the figure. These are issued by the monitoring MO to the SLS-I PMA where they are processed by an event handler. Activated policies enforce methods (M2.4 and M2.5) that calculate new values for the service rate and admission control parameters of a TT. Newly generated configuration values are updated in the information store which also holds parameters established through static policies during initialisation:

\[
\text{TT}\{\text{QC, SR, SRAS, SRFS, ACmin, ACmax, TCL, VCL, Rmin, Rmin, Rmax, SLS[ID]}\}
\]

It should be noted that RAB parameters are also maintained in the information store as these can potentially be used to derive the relative placement of thresholds and admission control parameters within the buffer space of a TT.
3.4.2 Traffic Engineering

The objectives of the ND module are achieved through resource provisioning operations supported by the following MOs:

- **BW Allocation MO** (bamo) – Handles operations related to the management of link capacity. These include minimum and maximum BW allocation for different QCs, and also functions for the distribution of spare link capacity and the reduction of over-provisioned BW.

- **LSP MO** (lspmo) – Provides operations that influence the load balancing aspects of the ND optimisation algorithm and for the setup of explicit paths through specific network nodes.

- **Hops MO** (hopsmo) – Provides operations that allow for different strategies when deriving the hop count constraint and also limit the number of hops that routes are permitted to have.

The behavioural representation of the ND module is depicted in Figure 3-9 along with the various methods that result in state transitions once enforced through policies.

![Figure 3-9: ND module behaviour and action summary](image_url)

The ND functionality is encapsulated within the three distinct stages of its operation which are represented by multiple states. Policies pertaining to each of the operational stages are triggered and subsequently enforced upon execution of a particular stage. During the processing stage, for example, policies enforcing methods M3.3-M3.6 define explicit BW allocation and constrain the output of the optimisation algorithm in terms of the maximum number of hops and alternative methods.
paths. The post-processing stage is only activated if the resulting configuration is not an accurate one with respect to available network resources. Here, corrective actions are applied to strategically reduce over-provisioned BW (M3.8) or distribute spare resources (M3.7) between QCs. Configuration parameter values established through policy enforcement are held in the ND information store that has the following structure:

\[
Q \{N_{\text{min}}, N_{\text{max}}, \text{HopCount}, \text{MaxHops}, \text{TT}[\text{ID}, \text{MaxPaths}, \text{LSP}\{\text{Path, BW}\}]\}
\]

It should be noted that the TT field holds multiple instances of TTs and LSPs, the attributes of which are explicitly defined through policies enforcing methods M3.2 and M3.3.

DRsM policies invoke both static and dynamic operations that aim in performing effective resource allocation between the supported QCs, based on guidelines received from the ND module. These operations are offered by the following MOs:

- **Initialisation MO** (initmo) – Provides operations that initialise specific parameters which act as constraints for the operation of the module. These include setting the minimum and maximum allocation of resources per QC and also defining the percentage of link capacity to be collectively made available for consumption.

- **BW Allocation MO** (bamo) – Handles operations that manipulate the BW allocation on a per link basis, which effectively track the utilization of a QC.

- **Monitoring MO** (monitormo) – Provides operations that manage the functionality of the monitoring component. These include generating alarms when upper or lower thresholds are crossed and adjusting their values accordingly.

Policy execution guides the evolution of the DRsM module in its operation through the states depicted in the behavioural representation of Figure 3-10.

The process of tracking resource utilisation is triggered by threshold crossing alarms emanating from the monitoring component depicted in the lower part of the figure. These are issued by the monitoring MO to the DRsM PMA, which, apart from enforcing policies that calculate new resource allocation values (M4.4 and M4.5), it also directs the monitoring MO to adjust the thresholds (M4.6 and M4.7) so that they are in accordance with the new allocation.

In addition to parameters established through static policies during initialisation, the DRsM information store also holds updated threshold and allocation values calculated dynamically through the policies described above:

\[
Q \{\text{Alloc}, \text{AllocCyn}, \text{AllocMax}, \text{Thlowr, Thuppr}\}
\]
3.5 Summary and Conclusions

This chapter defined the various policies that can be used to drive the functionality of the TEQUILA QoS management framework by initially providing a generic classification based on the policy applicability and the temporal characteristics of their enforcement. Although a small number of policies have been defined for this application domain in [84], [85] and [86], these mostly targeted specific operations in the context of the ND and DRsM modules and were used as proof of concept on the feasibility and benefits of policy-based management. Here, we extend the policy definitions to cover all the associated QoS management modules in more detail thus providing a richer set of available functions. This allows to more realistically investigate the impact of a higher degree of programmability on the consistency and stability of the managed system.

More specifically, we have shown the type of policies that can be used to drive the behaviour of service management modules and how these can influence the admission control process. SLS-S policies define essential service quality parameters and express conditional decisions for
performing static admission control on SLS subscriptions, aiming to provide service differentiation based on the QCs associated with the offered SLSs. SLS-I policies on the other hand, are mostly dynamic in nature aiming to regulate the traffic entering the network so that services can receive their subscribed rates and network congestion is prevented. They are used to set monitoring thresholds based on which dynamic functions can be invoked to alter service rates and admission control parameters.

On the traffic engineering side, policies are used to influence the allocation of resources among the supported QCs. ND policies aim to provide static configuration of resources through different strategies for BW allocation and route assignment, which can potentially result in an optimised network with respect to demand and availability. As in the case of SLS-I, most policies for the DRsM module are triggered at run-time based on the network state. They are used to guide the distribution of capacity between the QCs defined on a link thus providing the means for adaptive resource management.

In addition to the policy definitions, this chapter describes the various MOs available in the system and the operations they support, which are used in the policy specification to encode the targets and the actions respectively. The effect of policy enforcement onto the managed modules is modelled through state chart representations. These essentially describe the pre- and post-enforcement conditions which form part of the derivation process when identifying inconsistent situations in the system’s operation.

The defined policies are encoded using the widely adopted language of Ponder and they provide a comprehensive set of directives that serves as the basis for the conflict analysis methodology presented in subsequent chapters. Apart from analysis purposes [97][99], these policies have also been used in practical scenarios when addressing the policy refinement problem, and have been published in [100], [101], [102], and [103], in the context of the collaborative UK EPSRC PAQMAN [104] and European IST EMANICS [105] projects.
Chapter 4

4 Static Policy Conflict Analysis

One of the main reasons that prevent policy-based management from being widely adopted is that it is difficult to analyse policies in order to guarantee configuration stability, given that policies may have conflicts leading to unpredictable effects. The research issue of conflict analysis has been gaining momentum over the past decade and substantial effort has been invested in identifying conflicts and developing techniques for their effective detection and resolution.

Initial work on conflict analysis tried to address the problem in a generic fashion where a number of approaches for the detection and resolution of conflicts pertaining to management policy have been proposed. Although the outcome of this research was very useful for subsequent works, it was recognised that a generic approach could not satisfy the particularities of specific application domains. For this reason the evolution of research around this problem concentrated on providing solutions for distinct application domains, the main ones being security management of IP networks, call control in telecommunication networks, and management of mobile ad-hoc networks. Besides the different types of inconsistencies identified in separate application domains, most of the above works distinguish between static and dynamic conflicts. This distinction is a key aspect as the time frame at which conflicts can be detected poses challenges in the way they are treated; static conflicts can be detected through off-line analysis at policy specification time whereas dynamic conflicts can only be detected when policies are enforced and depend on the current state of the managed system.

While the policies described in the previous chapter can be used to guide the functional behaviour of the various QoS management modules, there is always the likelihood that several policies will be in conflict, either because of a specification error or because of application-specific constraints. This chapter focuses on the analysis of static conflicts by first identifying and classifying possible inconsistencies that may arise among the defined QoS management policies and then presenting techniques and mechanisms for their effective detection before being deployed in the system. The analysis approach is based on formal methods and derivation that caters for the various conflict types we have identified in policy-driven service management and traffic engineering. The logic formalism of Event Calculus was chosen for this purpose as it permits representation of events and persistent properties. The formalism is used for the representation of both policies and the
managed system, providing a uniform description that is amenable to analysis. Analysis relies on specifying detection rules to define the conditions for a conflict.

The structure of this chapter is as follows.

Section 4.1 identifies the types of conflicts that can occur between the policies defined for the various QoS management modules. These conflicts are categorised based on their properties and are described in detail based on the conditions under which they arise.

Section 4.2 presents the logic formalism of Event Calculus and describes how this can be used to represent the various components of the defined obligation policies and the QoS management system with a set of functions and predicates.

Section 4.3 defines a set of rules that can be used to detect the identified conflicts. The rules are specified using the Event Calculus and encode the conditions under which the conflicts occur in the form of constraints. The conflict definitions for the various inconsistencies are part of the static analysis presented in the next two sections.

Section 4.4 describes how system and policy, in their formal representations, can be used by the abductive reasoning technique to derive conflicts in the policy specification. This is demonstrated with a practical example of the derivation process followed by a description of the static conflict detection system architecture highlighting the main components.

Section 4.5 provides a practical demonstration of the conflict detection process through a case study involving a set of ND policies. The output of a conflict detection tool developed is used to show the detected conflicts along with explanations as to their occurrence. Finally, Section 4.6 summarises the chapter.

4.1 Conflict Classification

The fact that policies are downloaded to the QoS management modules, described in the previous chapter, on the fly while the system is operating may cause inconsistencies, since policies have not been tested to coexist with one another or with the rest of the system functionality without conflicts. This section provides a taxonomy of the various static conflict types that have been identified, as presented in Figure 4-1, and describes the conditions under which they arise.

Although it would be possible to classify these conflicts using different characteristics we have chosen to distinguish the categories based on their level of abstraction, the subsystem in which they occur and their specificity to the application domain as we believe these most naturally reflect the scope in which they occur. First we distinguish between conflicts that are module-independent and those specific to the two management subsystems. Module-independent conflicts
may occur among any of the QoS management policies, whilst service management and traffic engineering conflicts are specific to the operations supported by the relevant modules.

Module-independent conflicts represent the simplest forms of inconsistency that may arise between policy specifications and examples include redundancy, and mutual exclusivity. Redundancy conflicts may arise because of duplicate policies or policies with inconsistent action parameters in relation to others and can be detected by syntactic analysis. Mutual exclusion conflicts occur between policies implementing alternative strategies that realize the same goal. Examples of the latter conflict type include SLS-S policies for setting the upper limit in the RAB in a conservative, moderate, or risky fashion, ND policies defining the treatment of spare/over-provisioned BW, and DRSsM policies managing the allocation on link resources through different strategies. The various actions are said to be mutually exclusive since there should not be more than one directive specifying an operation on a particular managed resource. An example of such inconsistency would be between a DRSsM policy incrementing the resource allocation using an absolute value (e.g. 500 kbps) and another one using a relative value. The conflict will materialize if the two policies are triggered by the same event, apply to the same link and QC, and have an overlap in the time constraints.

The next two sub-sections describe the various conflicts relating to service management and traffic engineering policies and the conditions under which they can arise.

4.1.1 Service Management Static Conflicts

Conflicts specific to our application domain primarily occur because of inconsistent attribute values set by policies. It is essential that these are individually identified such that the exact reason for their occurrence can be defined and eventually resolved by a network administrator or in an automated manner.
Conflicts related to the SLS-S module can be detected at specification-time and arise between policies governing the process of static admission control. As described in Chapter 3, the upper limit in the RAB is a major factor for the decision of accepting/rejecting a new subscription request and can be defined with different strategies: risky, moderate, and conservative. A subscription admission strategy (\textit{subscrAdmStrg}) conflict will arise if the resulting value of the conservative approach is greater than that of moderate, or if the latter is greater than the one generated by the risky approach. Multiplexing (\textit{multiplex}) conflicts occur between service satisfaction policies that essentially define multiplexing factors used to derive the rates at which a service is considered almost and fully satisfied. This inconsistency will occur if the fully satisfied multiplexing factor is greater than the almost satisfied multiplexing factor for the same QoS class, as they are inversely proportional to the service rates produced.

The relative priorities between traffic classes can cause inconsistencies to arise between policies defined on the various QCs in the context of any QoS management module. These are termed QoS class priority (\textit{qcPriority}) conflicts and will materialize if the effect of a policy action violates the priority between QCs. Figure 4-1 shows two examples of such a conflict between SLS-S policies for setting service satisfaction and quality levels. A multiplex \textit{qcPriority} conflict will occur if the multiplexing factor of a particular QC is higher than that of another QC with lower priority, whereas an overall quality level (\textit{oql}) conflict will arise if the quality level of a QC with high priority is lower than that of QC with lower priority.

At the start of every new RPC, policies for the SLS-I module initialise minimum and maximum admission control parameters (\textit{AC}_{\text{min}}, \textit{AC}_{\text{max}}) as well as monitoring thresholds that indicate target-critical and very-critical levels (TCL, VCL) of traffic flowing into the network. The relative placement of both thresholds and admission control parameters in the RAB of each trunk allows the administrator to define the initial treatment of invoked services. Incorrect definition of these parameters will lead to an invocation admission strategy (\textit{invcAdmStrg}) conflict, which will occur if \textit{AC}_{\text{min}} is greater than TCL, or if \textit{AC}_{\text{max}} is less than VCL. This is due to the fact that all parameters share the same buffer space and are indirectly related. Violation of the above rules can result in accepting (\textit{AC}_{\text{min}} > \text{TCL}) or rejecting (\textit{AC}_{\text{max}} < \text{VCL}) new invocation requests when the opposite actions should be taken, which can overwhelm the network or compromise the service quality unnecessarily. The reader should note that this conflict type can also occur at run time as AC parameters may be re-calculated on the fly based on threshold crossing alarms. This is discussed in the next chapter.

The last two of the service management conflicts relate to the values of TCL and VCL thresholds. The first is termed a threshold incompatibility (\textit{thrshIncompat}) conflict and can occur between policies setting these thresholds if the value of TCL, aiming to trigger proactive measures, is
greater than that of VCL. The second type is another instance of the $qc_{Priority}$ conflict and will materialise if the value of either threshold for a high priority QC is greater than the corresponding value of a lower priority QC. This is due to the fact that higher priority QCs require more conservative treatment so that early triggering of dynamic AC functions can prevent delay sensitive traffic from suffering congestion.

### 4.1.2 Traffic Engineering Static Conflicts

Conflicts between policies guiding the functional behaviour of Network Dimensioning are all static in nature and occur due to contradicting action parameters of BW allocation and routing policies. With a combination of $set_{NDMin}$ and $set_{NDMax}$ actions, the administrator can specify a range of network resources to be allocated to the various QCs (Figure 4-2a). When these two actions are encoded in two separate policies, there is a possibility that the BW values specified in the actions will not converge to provide the intended BW boundaries for a specific QC. Instead, the values are said to be diverging if $BW_1 > BW_2$, in which case a diverging allocation ($divergAlloc$) conflict should be signalled (Figure 4-2b).

![Figure 4-2: BW allocation (ndMin-ndMax)](image)

The above policies may also cause an over-allocation ($minMaxBW\ overAlloc$) conflict if the sum of the allocation corresponding to all the supported QCs exceeds the network capacity or a higher BW value if over-subscription is permitted. The same rule applies to explicit actions responsible for the distribution of spare resources or the treatment of over-provisioned BW during the post-processing stage of Network Dimensioning: a $spareBW$ conflict will occur if the policy actions distribute more than the available spare capacity, and an $excsBW$ conflict will materialise when the relevant policy actions do not reduce excess BW below the maximum network capacity.

Incorrect specification of routing policies relating to the setup of explicit LSPs and the definition of the maximum number of hops and alternative paths can also lead the ND module to inconsistent states. The parameters set by the $set_{MaxHops}$ and $set_{MaxAltPath}$ policies, in order to meet the QoS characteristics of specific QCs, should not be violated by $setupLSP$ policies. A $hopsExceed$ conflict occurs if the hop count of the path, through which an LSP is set, exceeds the maximum number of allowed hops, provided both policies apply to the same QC. A second
inconsistency termed \textit{altPathExceed} conflict will arise if the instantiated number of LSPs is more than the maximum alternative paths permitted for a specific QC. Apart from the routing conflicts above, explicit LSP policy actions may also violate the constraints imposed by setNDmax policies, in which case a BW and routing (bwRtViolation) conflict should be signalled. The conditions for this inconsistency will be satisfied if the collective BW required by a set of LSPs defined for a particular QC exceeds the maximum permissible.

The last of the traffic engineering conflicts is based on the relative priorities between QCs and is another instance of the \textit{qcPriority} inconsistency. This conflict occurs between policies specifying the strategy with which to derive the hop count constraint (\textit{calcHopCount}), and will materialise if the strategy used for a high priority QC is less conservative in comparison to the one defined for a lower priority QC.

### 4.2 Formal Representation

Although a number of high-level policy languages and frameworks have been developed over the years, they all share a common deficiency in that they do not model the effect of policy enforcement on the behaviour of the managed system. As such, only simple checks can be carried out for the correctness of policies without taking into account the system state, which, in most cases, constrains the applicability of policies. Formal logical notations can allow for the representation of both system and policy and facilitate reasoning techniques to effectively analyse the policy specification.

A small number of formal approaches for policy specification have been proposed in the literature including ASL (Authorisation specification Language) [22], PDL (Policy Description Language) [66], and Rei [109]. Although these approaches have contributed to the progress of policy analysis research, their main limitation is that they do not provide the means to reason about the state of the system and thus can not detect inconsistencies in the policies before they are deployed to the system. This section presents a formalism for representing policies and managed systems, based on Event Calculus, which supports the reasoning methods required for both static and dynamic policy conflict analysis.

#### 4.2.1 Event Calculus

Event Calculus (EC) is a logic formalism for representing and reasoning about dynamic systems. Because it supports a time representation that is independent of any events that may occur, it provides a particularly useful way to specify a variety of event-driven systems including the presented QoS management framework. In the context of this work, EC serves as the underlying
formalism for describing policies and the managed system since it has well understood semantics, and supports all modes of logical reasoning.

Since its initial presentation [69], a number of variations have been presented in the literature [110]. In this work we use the form presented in [111], consisting of (i) a set of time points that can be mapped to the non-negative integers; (ii) a set of properties that can vary over the lifetime of the system, called fluents; and (iii) a set of event types. In addition the language includes a number of base predicates, initiates\((A, B, T)\), terminates\((A, B, T)\), holds\(_{At}(B, T)\), happens\((A, T)\), which are used to define some auxiliary predicates as summarized in Figure 4-3.

**Base predicates**

- initiates\((A, B, T)\)
  - event \(A\) initiates fluent \(B\) for all time \(T > T\).
- terminates\((A, B, T)\)
  - event \(A\) terminates fluent \(B\) for all time \(T > T\).
- happens\((A, T)\)
  - event \(A\) happens at time point \(T\).
- holds\(_{At}(B, T)\)
  - fluent \(B\) holds at time point \(T\). This predicate is useful for defining static rules (state constraints).
- initiallyTrue\((B)\)
  - fluent \(B\) is initially true.
- initiallyFalse\((B)\)
  - fluent \(B\) is initially false.

**Auxiliary predicates**

- clipped\((T1, B, T2)\)
  - fluent \(B\) is terminated between time point \(T1\) and \(T2\).
- declipped\((T1, B, T2)\)
  - fluent \(B\) is initiated between time point \(T1\) and \(T2\).

**Domain independent axioms**

\[\begin{align*}
\text{holds}\(_{At}(B, T1)\) & \iff \text{initiallyTrue}(B) \land \neg \text{clipped}(O, B, T1). \\
\text{holds}\(_{At}(B, T1)\) & \iff \text{initiates}(A, B, T) \land \text{happens}(A, T) \land \\
& \neg \text{clipped}(T, B, T1) \land T \leq T1. \\

\neg \text{holds}\(_{At}(B, T1)\) & \iff \neg \text{initiallyTrue}(B) \land \neg \text{clipped}(O, B, T1). \\

\neg \text{holds}\(_{At}(B, T1)\) & \iff \text{terminates}(A, B, T) \land \text{happens}(A, T) \land \\
& \neg \text{declipped}(T, B, T1) \land T < T1. 
\end{align*}\]

**Figure 4-3: Event Calculus predicates and axioms**

This is the classical form of Event Calculus where theories are written using Horn clauses. The frame problem is solved by circumscription, which allows the completion of the predicates initiates, terminates and happens, leaving open the predicates holds\(_{At}\), initiallyTrue and initiallyFalse. This approach allows the representation of partial domain knowledge (e.g. the initial state of the system). Formulae derived from Event Calculus are in effect derived from the circumscription of the EC representation.
The Event Calculus supports both deductive and abductive reasoning. Deduction uses the description of the system behaviour together with the history of events occurring in the system and the domain independent axioms to derive the fluents that will hold at a particular point in time. Although this reasoning technique can be used to determine conflicting conditions at a given point, partial specifications of the system state limit its applicability. Abduction addresses this issue and, given the descriptions of the behaviour of the system, it can be used to determine the sequence of events that need to occur such that a given set of fluents will hold at a specified point in time. While deduction is useful for dynamic conflict analysis as described in Chapter 5, abduction enables \textit{a priori} static analysis of policies.

### 4.2.2 System Representation

A comprehensive Event Calculus specification of the managed system requires representations for the managed objects, the functional behaviour of the managed modules, and the management information maintained by each module. The main functions and predicates used to represent the managed system are provided on Table 4-1. The format follows standard logic programming conventions where unbound variable names and atoms start with uppercase and lowercase characters respectively.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mgdObj(obj)</td>
<td>Used to specify that obj is a managed object in the system.</td>
</tr>
<tr>
<td>isMember(Obj, Module)</td>
<td>Used to denote that obj is member of a particular module of the managed system.</td>
</tr>
<tr>
<td>operation(obj, OpName, [Params])</td>
<td>Used to denote the management operations specified in a policy function and supported by a managed object, obj.</td>
</tr>
<tr>
<td>state(Module, Value)</td>
<td>Represents the state of a managed module when defining its behaviour model.</td>
</tr>
<tr>
<td>doAction(operation(objTarget, OpName, [Params]))</td>
<td>Represents the event of the action specified in the operation term being performed on the target object, objTarget.</td>
</tr>
<tr>
<td>mgdInfo(Module, Resrc, [Parms])</td>
<td>Used to specify the various types of management information maintained by a module, in terms of resources, such as QCs and TTs.</td>
</tr>
</tbody>
</table>
The managed objects supported by each module of the QoS management framework need to be defined and organised based on membership relationships. This will enable a consistent organisational structure with which managed objects can be later used in the policy specification without ambiguities. This is achieved by the first two predicates of Table 4-1, where predicate \texttt{mgdobj} can be used to represent an instance of a managed object and predicate \texttt{ismember} can be used to define the membership relationship of a managed object with a specific module. To complete the formal description of a managed object, the various operations it supports are defined with the function \texttt{operation}.

Listing 4-1 below demonstrates an example specification of the managed objects related to the SLS-S and DRsM modules. It should be noted that multiple instances of distributed QoS modules, as in the case of DRsM, expose the same set of managed objects with the same functional capabilities. Distinction between the various instances enables an administrator to enforce a different configuration at different physical points in the network. Lines 15 and 16 define the operations supported by the service satisfaction managed object of the SLS-S module.

\textbf{Listing 4-1: Formal representation of managed objects and organisation}

1. \texttt{mgdobj(servSatisfMO)}.
2. \texttt{mgdobj(bufferMO)}.
3. \texttt{mgdobj(acMO)}.
4. \texttt{mgdobj(initMO)}.
5. \texttt{mgdobj(baMO)}.
6. \texttt{mgdobj(monitormO)}.
7. \texttt{ismember(servSatisfMO, slsS)}.
8. \texttt{ismember(bufferMO, slsS)}.
9. \texttt{ismember(acMO, slsS)}.
10. \texttt{ismember(initMO, dRsmRoutr15)}.
11. \texttt{ismember(baMO, dRsmRoutr15)}.
12. \texttt{ismember(monitormO, dRsmRoutr15)}.
13. \texttt{operation(servSatisfMO, setAlmstsatisf, [QC, MFI])}.
14. \texttt{operation(servSatisfMO, setQltlvl, [QC, OQL])}.

The run-time behaviour of the managed modules can be modelled in terms of the changes in state caused by performing operations through managed objects. The state machines described in Chapter 3 represent the different states each of the modules can take as a result of policy enforcement. These can be specified with the function \texttt{state} which is used to represent the pre- and post-conditions for each operation. Performing an operation on the system will modify the state of the system in such a way that, once the operation is complete, there will be some new
fluent that hold, and some other fluents that will cease to hold. This is represented by the
initiates(A, B, T) and terminates(A, B, T) predicates according to the schema below:

(a) initiates(doAction(operation(ObjTarg, OpName, Params)), PostTrue, T) \iff
    \text{Precondition} \land \text{mgdObj(ObjTarg)} \land \text{isMember(ObjTarg, Module)}.

(b) terminates(doAction(operation(ObjTarg, OpName, Params)), PostFalse, T) \iff
    \text{Precondition} \land \text{mgdObj(ObjTarg)} \land \text{isMember(ObjTarg, Module)}.

The first rule above states that when the doAction event occurs at time T, if the Precondition is
ture, then the fluent defined by PostTrue will hold after that time. Under the same conditions, the
second rule states that the fluent defined by PostFalse will hold after time T. In both rules, the
Precondition will be represented by holdsAt predicates. The mgdObj and isMember predicates are
used to verify that the target is a defined object and has a membership relationship with the QoS
management module being modelled. Using these two rules, it is possible to model a transition in
the state of the managed module. The PostTrue fluent in the first rule expresses the new state after
the transition and the Precondition takes the current state value, whereas the second rule
invalidates the current state by setting it as the PostFalse fluent with the same Precondition.

Figure 4-4 represents part of the SLS-S module behaviour during the initialisation phase where
multiplexing factors and the service quality level are set by policies. The first transition is initiated
by the setAlmstSatis() operation driving the module from the idle to the MFactsrsset state, and
the second transition is initiated by the setQltLvl() operation driving the module from the
MFactsrsset to the OQLset state. The module returns to the idle state once the second operation has
completed.
Listing 4-2: Formal representation of the SLS-S behaviour model

```plaintext
1. initiates(doAction(operation(servSatisfMO, setFullSatisf, [QC, MF])),
   state(slss, mFactrsSet), T) \leftarrow
2. holdsAt(state(slss, idle), T) \land
3. mgdObj(servSatisfMO) \land isMember(servSatisfMO, slss).
4. terminates(doAction(operation(servSatisfMO, setFullSatisf, [QC, MF])),
   state(slss, idle), T) \leftarrow
5. holdsAt(state(slss, idle), T) \land
6. mgdObj(servSatisfMO) \land isMember(servSatisfMO, slss).
7. initiates(doAction(operation(servSatisfMO, setQlLV1, [QC, OQL]),
   state(slss, oqlSet), T) \leftarrow
8. holdsAt(state(slss, mFactrsSet), T) \land
9. mgdObj(servSatisfMO) \land isMember(servSatisfMO, slss).
10. terminates(doAction(operation(servSatisfMO, setQlLV1, [QC, OQL])),
    state(slss, mFactrsSet), T) \leftarrow
11. holdsAt(state(slss, mFactrsSet), T) \land
12. mgdObj(servSatisfMO) \land isMember(servSatisfMO, slss).
```

When performing management operations, in addition to changing the state of a module, system resources need to be configured appropriately. Following the example above, the `setFullSatisf` operation with parameters `[ef, 1]` will configure the fully satisfied multiplexing factor of EF traffic to the value of 1 on all traffic trunks. Management information is represented by the `mgdInfo` predicate of Table 4-1 and its parameters are instantiated with the structures associated with a particular module as described in Section 3.4 of the previous chapter.

### 4.2.3 Policy Representation

Although a number of policy types have been used in the literature for different purposes, this thesis is limited to obligation policies as these naturally apply to the QoS management framework under investigation. The formal representation of positive and negative authorisations as well as refrain policies, using Event Calculus, can be found in [112].

The elements of Ponder obligation policies are Subjects, Targets, Events, Actions and Constraints. To provide a complete formal representation of such policies, the information contained in each of these elements must be included in the formalism. Table 4-2 describes the relevant functions used.
Table 4-2: Functions for representing obligation policies

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oblig(P01ID, ObjSubj, operation(objTarg, OpName, [Params]))</td>
<td>Used to specify an obligation policy with identification P01ID, where the subject, ObjSubj, should perform the action OpName on the target object, objTarg.</td>
</tr>
<tr>
<td>systemEvent(E)</td>
<td>Represents events generated by the system which are used as the trigger in the definition of obligation policies.</td>
</tr>
<tr>
<td>clocktick(HH, MM)</td>
<td>Event that represents time in the format of hour : minute.</td>
</tr>
</tbody>
</table>

The Subject and Target elements of an obligation policy refer to the relevant managed objects; the Subject (ObjSubj) is the object representing the Policy Management Agent responsible for a specific module and is responsible for enforcing the policy operations specified in the Action clause, whereas the Target (objTarg) is the managed object on which the operations are performed.

The triggering conditions for obligation policies are expressed with events that are based either on the operational system functionality or time, referred to as system and timer events respectively. System events for static QoS management components (SLS-S, ND) can arise because of a module entering a particular stage of its operation, whereas for dynamic modules (SLS-I, DRsM) they mainly depend on the current state of the underlying network and come in the form on threshold crossing alarms. Timer events can be used to trigger policies at some specific point in time, as in the case of a new resource provisioning cycle that typically takes place on a weekly basis. The two event types are represented by the `systemEvent(E)` and `clocktick(H, M)` functions and they are parameters of the `happens` predicate. The examples below encode (a) the event of an upper threshold crossing alarm for DRsM and (b) the time turning 8 am.

(a) happens(systemEvent(drsMAlarmRaised(upprTh, link2, ef)), T).

(b) happens(clocktick(08, 00), T).

Apart from events, constraints also control the applicability of a policy, which, in the context of our application domain, are either attributes of the managed resources or time periods. Examples of the former include constraints in SLS-S policies for determining the conservative strategy in setting the upper subscription limit in the resource availability buffer, and constraints that decide if a new subscription request should be accepted. As shown below, these constraints are represented by instances of the `mgdinfo` predicate together with comparison operators.
**Chapter 4. Static Policy Conflict Analysis**

suConstraintSatisfy <-
  mgdInfo(slls, tt2, [_, OQL, _]) \ OQL > 0.

acceptConstraintSatisfy <-
  mgdInfo(slls, tt3, [_, OQL, _]) \
  mgdInfo(slls, tt3, [_, TDmin, _]) \
  mgdInfo(slls, tt3, [_, SU, _]) \
  OQL==0 \ TDmin<SU.

Time constraints are useful when an administrator wishes to provide different network configurations during peak and off-peak hours as in the case of DRsM policies described in Chapter 3. To represent this type of constraint it is necessary to specify a rule that allows time values to be compared as shown below. The predicate duration defines a time period and holds between the occurrences of two clocktick events.

```
duration(HH1, MM1, HH2, MM2) <-
  holdSAt(clocktick(HH1, MM1)), T1) \ 
  holdSAt(clocktick(HH2, MM2)), T2) \ 
  T1<T2 \ (HH1<HH2 v (HH1==HH2 A MM1<MM2)).
```

The remaining element of a policy is the Action, which specifies the management operations and is represented with the operation function previously defined. This is used as a parameter of the oblig function of Table 4-2 together with an identifier and a subject to describe the type of the policy and the entity responsible to enforce it.

Having defined the formal specification of the various components, it is now possible to combine them and provide the representation of an obligation policy. For each rule the terms objsubj, ObjTarg, OpName, E, and constraint, can be mapped directly to the subject, target, action, event and constraint clauses described above. Policy rule (a) below is expressed using the initiates predicate and states that if the constraint holds at the time the system event, SystemEvent(E), occurs, then the obligation for the subject to perform the operation on the target object holds. Rule (b) is an instance of a DRsM policy specifying the strategy and the value by which the BW allocation for EF traffic on a specific link should be increased during peak hours when a threshold crossing alarm is raised.

(a) initiates(SystemEvent(E), oblig(PolID, ObjSubj, 
          operation(ObjTarg, OpName, Params)), T) <- Constraint.

(b) initiates(SystemEvent(drsmAlarmRaised(upprTh, link2, ef)), 
          oblig(incrBWPKp, drsmPMMA, operation(bamo, 
          incrAllocRel, [link2, ef, 10])), T) <- 
          duration(8, 0, 18, 0).
Chapter 4. Static Policy Conflict Analysis

4.3 Conflict Detection Rules

According to the description of the conditions under which a conflict in the policy specification may arise, specific rules can be defined to detect such an event. These rules are expressed in the form of logic predicates that encapsulate the conditions to be met for a conflict to occur and are used as conflict fluents in Event Calculus notation. The latter can be considered as goal states that, when they are achieved, they signify the detection of a conflict. The advantage of using such a methodology is that, in addition to detecting possible conflicts, an explanation as to why a conflict occurred will always be provided.

Conflicts identified in the literature come in different forms and levels of abstraction. The most common ones are modality conflicts which are domain independent and, according to [41], arise when two policies are specified using the same subjects, targets and actions but are of opposite modality, as in the case of positive and negative obligations (refrain policies). The conditions for this inconsistency can be encoded in the body of the rule specified below using EC notation (Listing 4-3). The rule specifies that a modality conflict (modlt) will occur at time TC, between the obligation policy Pol ID1 and the refrain policy Pol ID2, if their subjects (S1, S2) are both obliged to, and refrained from performing an operation (opName) on their targets (T1, T2), and if they share the same subjects and targets. The conditions are determined by the isoverlap predicate and the conflicting parameters are used as arguments of the term conflictData. The latter are instantiated during a conflict detection query.

Listing 4-3: Rule for detecting modality conflicts

\[
\text{conflict(} \text{modlt, conflictData(PolID1, PolID2, ObjSubj, ObjTarg, OpName), TC) } \leftarrow \\
\text{holdsAt(} \text{obligr(PolID1, S1, operation(T1, Op1, Params)), TC) } \land \\
\text{holdsAt(} \text{refrain(PolID2, S2, operation(T2, Op2, Params)), TC) } \land \\
\text{isoverlap(S1, S2, ObjSubj) } \land \text{isoverlap(T1, T2, ObjTarg) } \land \\
\text{isoverlap(Op1, Op2, Opname)}. \\
\]

Whilst for modality conflicts the conditions under which the conflicts arise are generic and can be extracted from the policies themselves, for application-specific conflicts these conditions include system-specific data in addition to policy information for correctly capturing conflicting situations and need to be encoded in the body of the rules as additional constraints. The rest of this section presents and describes the various rules defined for the static detection of module independent conflicts and those specific to the two management subsystems.
4.3.1 Module Independent Conflict Rules

Redundancy conflicts will arise between two policies if they are characterised by the same subjects, targets, actions and action parameters (duplicate policies). Furthermore, it is not necessary for all the action parameters to be exactly the same to indicate an anomaly. The matching of some key parameters in the actions will suffice to argue that the two policies are inconsistent with each other. Consider the actions of two ND policy instances where the first policy action specifies that at least 30% of the BW should be allocated to EF traffic and the second allocating a minimum of 40% for the same traffic type. In this case the QC is the key parameter matched, signifying that the two actions will lead to a redundancy conflict irrespective of the associated BW value.

In order to capture redundancy conflicts one has to cater for all policy types driving the QoS management modules and their associated parameters. The rule of Listing 4-4 below is used to detect this particular conflict and is based on the number of policy action parameters. Besides matching subjects, target and actions, the rule performs checks that involve list manipulation and aims to match certain key parameters to signal the occurrence of a conflict. This is achieved with the following predicates: listmembercount determines the length of the parameters list; firstmember determines the first member of the list; and secondmember determines the second. These are used to compare key parameters and cater for policy actions with parameter number ranging from zero (e.g. ND allocspareBWEquai policies) to three (e.g. ND setmaxAltPaths policies).

Listing 4-4: Rule for detecting redundancy conflicts

| conflict(redundancy, conflictData(PolID1, PolID2, ObjSubj, ObjTarg, OpName), TC) <- |
| holdsAt(oblig(PolID1, Sl, operation(T1, Op1, Params1), TC) & |
| holdsAt(oblig(PolID2, S2, operation(T2, Op2, Params2), TC) & |
| isoverlap(S1, S2, objsubj) & isoverlap(T1, T2, objtarg) & |
| isoverlap(Op1, Op2, OpName) & |
| PolID1 \= PolID2. |
| listmembercount(Params1, Params2, 0) v |
| (listmembercount(Params1, Params2, 1) & |
| firstmember(Params1, P1) & firstmember(Params2, P2) & P1 \= P2) v |
| (listmembercount(Params1, Params2, 2) & |
| firstmember(Params1, P3) & firstmember(Params2, P4) & P3 \= P4) v |
| (listmembercount(Params1, Params2, 3) & |
| firstmember(Params1, P5) & firstmember(Params2, P6) & P5 \= P6 & |
| secondmember(Params1, P7) & secondmember(Params2, P8) & P7 \= P8). |

As mentioned in Section 4.1, the functionality of the QoS management framework allows for a choice of methods related to a specific process, i.e. different strategies for realising a goal. Although this approach provides an administrator with the flexibility to follow different strategies
depending on the circumstances and managed resource, care must be taken so that no more than one strategy applies to a specific process and resource at the same time. Table 4-3 summarises the identified actions that are mutually exclusive (ME) between them and classifies them based on the process which they implement.

Table 4-3: Sets of mutually exclusive actions

<table>
<thead>
<tr>
<th>Upper subscription limit (SLS-S)</th>
<th>Spare BW allocation (ND)</th>
<th>Over-provisioned BW reduction (ND)</th>
<th>Hop count derivation (ND)</th>
</tr>
</thead>
<tbody>
<tr>
<td>setSUConsrv</td>
<td>allocSpareBWEqual</td>
<td>redOverBWEqual</td>
<td>calCHOpCountMin</td>
</tr>
<tr>
<td>setSUModert</td>
<td>allocSpareBWPprop</td>
<td>redOverBWPprop</td>
<td>calCHOpCountMax</td>
</tr>
<tr>
<td>setSURisky</td>
<td>allocSpareBWEexpl</td>
<td>redOverBWEexpl</td>
<td>calCHOpCountAvg</td>
</tr>
</tbody>
</table>

Extending the logic formalism, we introduce the meops predicate which organises ME policy actions into domains. Each domain is identified by a process that can be achieved with different strategies and a set of actions relevant to that process. The examples below encode the ME domains and actions for three processes of the ND module.

meops(mutexspareBWAlloc, [allocSpareBWEqual, allocSpareBWPprop, allocSpareBWEexpl]).
meops(mutexoverBWRed, [redOverBWEqual, redOverBWPprop, redOverBWEexpl]).
meops(mutexXHOpCountCalc, [calCHOpCountMin, calCHOpCountMax, calCHOpCountAvg]).

The detection process for a ME conflict between two policy actions involves identifying their membership in a ME actions domain – if they belong to the same domain there is potential for the conflict to materialise. Listing 4-5 shows the general rule for detecting such conflicts, where the member predicate determines the membership of an action in a ME domain of actions.

Listing 4-5: Rule for detecting mutual exclusion conflicts

conflict(mutex, conflictData(PolID1, PolID2, ObjSubj, ObjTarg, MutexDomain, Op1, Op2), TC) ←
holdsAt(oblig(PolID1, S1, operation(T1, Op1, Param1S1)), TC) ∧
holdsAt(oblig(PolID2, S2, operation(T2, Op2, Param2S2)), TC) ∧
meops(MutexDomain, MutexOps) ∧
member(Op1, MutexOps) ∧ member(Op2, MutexOps) ∧
isoverlap(S1, S2, ObjSubj) ∧ isoverlap(T1, T2, ObjTarg) ∧
Op1\=Op2 ∧ PolID1\=PolID2.
4.3.2 Service Management Conflict Rules

The detection of most inconsistencies that are specific to the two management subsystems requires not only information provided by the policy specification, as in the case of redundancies, but also QoS-specific information, such as properties of the managed resources and the supported QoS classes. These are encoded in the detection rules as further constraints that should not be violated. Such an example is provided by the rule for detecting QoS priority conflicts among policies for setting the quality level of traffic classes (Listing 4-6). Here, the constraint related to the relative priorities of QoS classes involved in setQltLV1 policy actions is encoded in the body of the rule. The conflict will be detected if the defined quality level of a QC with high priority is less than that of a QC with lower priority. Priorities are determined using the priority predicate which assigns values to terms Prt1 and Prt2.

Listing 4-6: Rule for detecting OQL qcPriority conflicts

It should be noted that in the above rule, subjects and targets are not explicitly checked for overlaps as in the rules for module-independent conflicts. They are instead assigned the same variable names (subj, Targ) in the policy specification for simplicity. The same applies in subsequent definitions of conflict detection rules.

As described in Section 4.1.1, multiplexing conflicts occur between policies setting the almost and fully satisfied service multiplexing factors. Listing 4-7 aims to match such policies and signal a conflict if the value of the almost satisfied factor, MF1, is less than that of the fully satisfied factor, MF2, for the same QC. The rules for detecting the rest of the identified inconsistencies, encoding application-specific constraints that characterise conflicting conditions, can be found in Appendix A1.

Listing 4-7: Rule for detecting multiplexing conflicts

It should be noted that in the above rule, subjects and targets are not explicitly checked for overlaps as in the rules for module-independent conflicts. They are instead assigned the same variable names (subj, Targ) in the policy specification for simplicity. The same applies in subsequent definitions of conflict detection rules.

As described in Section 4.1.1, multiplexing conflicts occur between policies setting the almost and fully satisfied service multiplexing factors. Listing 4-7 aims to match such policies and signal a conflict if the value of the almost satisfied factor, MF1, is less than that of the fully satisfied factor, MF2, for the same QC. The rules for detecting the rest of the identified inconsistencies, encoding application-specific constraints that characterise conflicting conditions, can be found in Appendix A1.
4.3.3 Traffic Engineering Conflict Rules

Following the examples of the previous sub-section, here we describe the rules for detecting two of the identified conflicts between traffic engineering policies. The first rule concerns conflicts that arise due to inconsistent BW allocation when setting the upper and lower boundaries during the dimensioning process. In Listing 4-8, the conflict predicate will hold if it can match a setNDMin and a SetNDmax policy action applying to the same QoS class but the value of the minimum bound, BW1, is greater than that of the maximum bound, BW2.

Listing 4-8: Rule for detecting diverging allocation conflicts

\[
\text{conflict(divergAlloc, conflictData(PolID1, PolID2, QC1, BW1, BW2), TC) } \leftarrow \\
\text{holdSAt(oblig(PolID1, subj, operation(Targ, setNDMin, [QC1, BW1]), TC) A} \\
\text{holdSAt(oblig(PolID2, subj, operation(Targ, setNDmax, [QC2, BW2]), TC) A} \\
\text{QC1==QC2 A BW1>BW2).}
\]

The second rule (Listing 4-9) will determine hopsExceed conflicts among policies setting up LSPs and those defining the maximum number of hops, HopNum, to meet the delay constraints of a particular QoS class. The predicate hopcount in the conditional part of the clause determines the hop-count, HC, of the route specified by Path in the setUpLSP policy. This conflict will be detected if HC has a value greater than the permitted number of hops. The definitions of the rules for the other conflicts identified between traffic engineering policies can be found in Appendix A2.

Listing 4-9: Rule for detecting hopsExceed conflicts

\[
\text{conflict(hopsExceed, conflictData(PolID1, PolID2, Path, HC, HopNum), TC) } \leftarrow \\
\text{holdSAt(oblig(PolID1, subj, operation(hopsMO, setmaxHOPS, [QC1, HopNum]), TC) A} \\
\text{holdSAt(oblig(PolID2, subj, operation(lspmo, setUpLSP, [QC2, TT, Path, BW]), TC) A} \\
\text{hopcount(Path, HC) A HC>HopNum A QC1==QC2).}
\]

4.4 Analysis of Static QoS Management Conflicts

The conflict analysis approach presented in this chapter is based on the formal representation of both system and policy has two main aspects: the definition of appropriate rules for determining potential conflicts in policy specifications, and the effective deployment of analysis processes in the context of the managed environment. In addition to the set of conflict detection rules defined in the previous section and system-specific information, an analysis process needs to support the appropriate reasoning capabilities to determine potential inconsistencies. This section describes
how abduction can be used to reason over a set of policies and presents the overall architecture for static conflict detection.

### 4.4.1 Abductive Reasoning for Conflict Detection

Based on the rules that define the conditions for a conflict, abductive reasoning can be used to determine inconsistencies in the policy specification. The logic predicates representing the various conflict types are used as goal states of abductive queries that, when achieved, signify the detection of a conflict. The advantage of using this type of reasoning is that, in addition to detecting possible inconsistencies, it provides an explanation as to why a conflict occurred, i.e. the sequence of events that must take place for the conflict to materialise.

To demonstrate how abduction works, consider the example policies in Listing 4-10 which are candidates for a multiplexing conflict. The first policy sets the almost satisfied multiplexing factor of EF traffic to 0.3 and the second policy sets the fully satisfied factor to 0.4 for the same type of traffic. It should be noted that the integer numbers used in the policy actions represent the fractional part of multiplexing factors as their values range from 0 to 1.

**Listing 4-10: Policy instances**

```prolog
initiateS(SYSteMEVerlt(neWRPC), oblig(pl, slSSPMA,
operation(servSatisfMO, setAlmostsatisf, [ef, 3])), T) :-
duration(8, 0, 18, 0).

initiates(SySteMEvent(newRPC), oblig(p2, slSSPMA,
operation(servSatisfMO, setFullsatisf, [ef, 4])), T) :-
duration(8, 0, 18, 0).
```

The abductive query that checks for a conflict between the two policies above, will attempt to satisfy the goal `conflict(multiplex, conflictData(pol1ID1, pol1ID2, QC1, QC2, MF1, MF2), TC)` as defined in Listing 4-7 of Section 4.3.2. This goal will be satisfied if the goals specified in the conditional part of the conflict rule can be achieved. These include the concurrent activation of the two policies above with matching subjects and targets, and inconsistent multiplexing factors applying to the same QC. Auctive reasoning first checks if these predicates are direct abducibles and if not, it recursively treats each predicate in the body of the rule as a goal of another abductive query. This process is illustrated in Figure 4-5 where each branch of the evaluation tree represents a goal and leaf nodes represent facts or abducibles that satisfy the goals.
conflict(mltplex, conflictData(p1, p2, ef, ef, 3, 4), 1)

→ holdsAt(oblig(p1, slsSPMA, operation(servSatisfMO, setAlmostSatisf, [ef, 3]), 1)
  → initiates(systemEvent(newRPC), 1)
  → duration(8,0, 18,0)
    → happens(clocktick(8,0), 0)
    → happens(clocktick(18,0), 2)
→ holdsAt(oblig(p2, slsSPMA, operation(servSatisfMO, setFullSatisf, [ef, 4]), 1)
→ duration(8,0, 18,0)
  → happens(clocktick(8,0), 0)
  → happens(clocktick(18,0), 2)

ef == ef
3 < 4

Figure 4-5: Abductive reasoning evaluation tree

4.4.2 System Architecture

The architecture of the analysis system developed is outlined in Figure 4-6. Here, the static analysis process is an integral part of a Policy Management Tool (PMT) and is initiated by a network administrator. Our approach is based on the output of a refinement process [113][114], where high-level policy specifications or objectives introduced by a network administrator are decomposed into low-level implementable policies and mapped onto their respective EC representation.
Before their enforcement, policies are analysed by the static analysis engine, which is the main component of the architecture encapsulating the necessary logic and reasoning abilities. This component also incorporates the detection rules defined in the previous section that cater for the identified conflict types. Apart from policies, the analysis engine takes as input the specification of the QoS management modules’ behaviour as well as auxiliary information required by the process. The latter includes the structure of the managed resource representation, e.g. TT(...), QC(...), domain information about the various MOs (the modules they are associated with and the operations they support), QC information regarding their relative priorities, and domain information about mutually exclusive policy actions.

The output of the static analysis process is a set of conflicting policies along with an explanation of their occurrence. The resolution of these conflicts is a manual process that has to be carried out by the administrator as in the case of an oql qcPriority conflict where new quality levels need to be set for one of the QCs. Although some conflict resolution methods based on precedence rules have been proposed in the literature [18][28][30][44], the nature of most static conflicts identified in this thesis does not allow for automation in their resolution. As such, in the event of a conflict, a policy can either be re-specified with the correct parameters or, in the case of redundancies, removed from the system. The details associated with each inconsistency however, can guide the administrator when correcting them.

4.5 Static Conflict Analysis Case Study

This section provides practical examples demonstrating the process of detecting conflicts among a set of policies, some of which are incompatible. In these examples, the conflict fluents defined in Section 4.3 are used as goal states of abductive queries aiming to determine any conflicts in the policy specification. If there are no solutions for a particular conflict fluent, it can be considered that the policy specification is free of that particular conflict type.

The detection process is demonstrated with a tool developed using Prolog [115] in conjunction with the A-System abductive proof engine [116]. The tool takes as input the policy specifications and application-specific information, applies the appropriate detection logic and, in response to the user’s queries, it returns any conflicts that may exist among the policies. The examples considered here aim to determine conflicts that can arise between policies managing the ND module.
4.5.1 Policy Set

The policies used as input are the ones in Listing 4-11, which are in their EC representation and have been deliberately specified with such parameters that can lead to mutex, divergAlloc, hopsExceed, and bwRtViolation conflicts.

In each policy rule, we have added some time constraints that control the applicability of the policy. For example, the first rule states that the ndPMA is obliged to perform the action calcHopcountMin when the time is between 9am and 10am. In this respect, besides the conditions for the identified conflict types that have to be met, a conflict will be signalled if there is also an overlap in the time constraints.

Listing 4-11: Policy instances

```prolog
initiates(systemEvent(doNDPreProc), oblig(p1, ndPMA, operation(hopsMO, calcHopcountMin, [af]), T) :- duration(9, 0, 10, 0).

initiates(systemEvent(doNDPreProc), oblig(p2, ndPMA, operation(hopsMO, calcHopcountAvg, [af]), T) :- duration(9, 30, 10, 30).

initiates(systemEvent(doNDProc), oblig(p3, ndPMA, operation(bamo, setNDMin, [ef, 50]), T) :- duration(16, 0, 20, 0).

initiates(systemEvent(doNDProc), oblig(p4, ndPMA, operation(bamo, setNDMax, [ef, 40]), T) :- duration(18, 0, 22, 0).

initiates(systemEvent(doNDProc), oblig(p5, ndPMA, operation(bamo, setmaXHOPS, [ef, 4]), T) :- duration(13, 0, 19, 0).

initiates(systemEvent(doNDPreProc), oblig(p6, ndPMA, operation(IspMO, SetUpLSP, [ef, r2r15, r2, r4, r6, r8, r9, r11, r15, 45]), T) :- duration(9, 30, 18, 30).

initiates(systemEvent(doNDProc), oblig(p7, ndPMA, operation(bamo, setNDMin, [af, 60]), T) :- duration(16, 0, 20, 0).

initiates(systemEvent(doNDProc), oblig(p8, ndPMA, operation(bamo, setNDMax, [af, 50]), T) :- duration(20, 0, 22, 0).
```

4.5.2 Static Analysis

When performing queries concerning the different conflict types, the tool can indicate if there is a conflict of a particular type and also provide an explanation as to why that specific conflict occurred. Listing 4-12 shows the results of a single general query where the variables type and conflictData are instantiated with the identifiers pertaining to ND conflicts and their associated data respectively.

The results suggest that there is a mutex conflict between policies p1 and p2 because of mutually exclusive actions belonging to the domain mutexHopcountCalc; a divergAlloc conflict between p3
and p4 because of inconsistent BW values for EF traffic; a hopsExceed conflict between p5 and p6 because the hop-count of the specified path between routers r2 to r15 (6) exceeds the maximum number of hops allowed (4); and a bwrtviolation conflict between p4 and p6 because the BW allocated to the LSP is more than the permitted maximum for EF traffic. Additionally, the results describe the sequence of events that need to take place for the conflict to occur – system events mark the execution of particular stages of the dimensioning process controlling the activation of policies, and clocktick events constrain the applicability of policies between certain time periods. It should be noted that there is no conflict detected between p7 and p8. This is because the time constraints for these two policies do not overlap, and therefore there is not a situation in which a conflict may arise.

**Listing 4-12: Conflict detection trace**

```prolog
?- solve(conflict(Type, ConflictData, T)).

Solution found
abduced atoms:
  0-happens(clocktick(9, 0), 0)
  1-happens(clocktick(9, 30), 1)
  2-happens(systemEvent(doNDPreproc), 2)
  3-happens(clocktick(10, 0), 3)
  4-happens(clocktick(10, 30), 4)

Solved query:
conflict(mutex, conflictData(p2, p1, ndPMA, hopsMO, mutexHopCountCalc, calcHopCountAvg, calcHopCountMin), 3)

Solution found
abduced atoms:
  0-happens(clocktick(16, 0), 0)
  1-happens(clocktick(18, 0), 1)
  2-happens(systemEvent(doNDProc), 2)
  3-happens(clocktick(20, 0), 3)
  4-happens(clocktick(22, 0), 4)

Solved query:
conflict(divergAlloc, conflictData(p3, p4, ef, 50, 40), 3)

Solution found
abduced atoms:
  0-happens(clocktick(9, 30), 0)
  1-happens(clocktick(13, 0), 1)
  2-happens(systemEvent(doNDPreProc), 2)
  3-happens(clocktick(13, 30), 3)
  4-happens(clocktick(18, 0), 4)

Solved query:
conflict(hopsExceed, conflictData(p5, p6, [r2, r4, r6, r8, r9, r11, r15], 6, 4), 3)

Solution found
abduced atoms:
  0-happens(clocktick(9, 30), 0)
  1-happens(clocktick(18, 0), 1)
  2-happens(systemEvent(doNDProc), 2)
  3-happens(systemEvent(doNDPreProc), 2)
  4-happens(clocktick(18, 30), 3)
  5-happens(clocktick(22, 0), 4)

Solved query:
conflict(bwRtvViolation, conflictData(p4, p6, ef, 40, 45), 3)
```
4.6 Summary and Conclusions

This chapter presented our approach towards the static analysis of conflicts by first identifying the various inconsistencies that may arise between policies driving the behaviour of the QoS management framework and which can be detected prior to policy enforcement. These were classified into module-independent and module-specific conflicts based on their level of abstraction and their specificity to the application domain. Although recent work in [32] also tackles the problem of conflict analysis in the context of QoS management, the conflicts identified in this chapter have not been previously reported in the literature.

The conflict analysis approach is heavily based on the logic formalism of Event Calculus, which is suited for the representation of event driven systems as in the case of our application domain and the obligation policies that govern its behaviour. Using formal notations we have shown how Event Calculus can be used to represent the various components of the managed system such as the MOs, the operations supported, the state, and the managed information maintained by the QoS management framework. With respect to policies, the formalism was used to represent obligations and their associated events, actions and constraints.

Apart from the system and policies, the formalism is also used to describe conflicting situations. Based on the description of the conditions under which the identified conflict types occur, specific rules have been defined to determine inconsistencies in the policy specification. These are Event Calculus predicates that encode information from policies and the managed system itself signifying the detection of a conflict when they are achieved. Unlike any other conflict detection methodology in the literature, including formal approaches [22][109], we have shown how Event Calculus can be used to not only detect conflicts but also provide explanations as to their occurrence. This was achieved by modelling the effect of policy enforcement through state machine transitions and including the description of the managed system state in the analysis procedure.

The analysis approach presented in this chapter was demonstrated through a case study involving the detection of a range of conflicts among ND policies. The examples show how abductive reasoning can derive the sequence of events that need to occur for the conflicts to materialise, thus generating an explanation. This is particularly important when guiding a network administrator to handle a conflict requiring manual resolution, which is the case for most of the identified conflicts in this chapter. Finally, most of the work presented in this chapter has been published in [99].
Chapter 5

5 Dynamic Policy Analysis and Conflict Resolution

Although research has tackled the problem of policy analysis in the context of several application domains, it has mainly focused on conflicts that can be determined statically at compile-time. The detection process involved simple policy analysis and resolution mostly based on the specification of policy precedence rules that may not suit many policy-driven systems. Although static analysis is very useful for detecting and resolving some conflicts before policies are deployed, it does not cater for the variety of conflicts that can emerge at run-time. Such conflicts occur in dynamic systems as a result of the current state of the resources. In network resource management, for example, policies which increment or decrement allocation of resources may conflict with policies related to setting upper and lower bounds for those resources. Conflicts of this type cannot be detected prior to policy enforcement because they depend on the current state of the managed system.

The time-frame at which conflicts can be detected influences the analysis methodology and requirements for dealing with them. Static conflicts are typically detected through analysis initiated manually by the system administrator; conflicts represent inconsistencies between policies and are typically resolved by amending the policies. In contrast, run-time conflicts must be detected by a process that monitors policy enforcement and detects inconsistent situations in the system’s execution. Resolution must be achieved automatically, for example through enforcing resolution rules. Lack of automation in the handling of run-time conflicts may have catastrophic consequences on the correct system operation, especially when managing QoS for delay sensitive applications.

While the conflicts identified in the previous chapter can be detected statically, policies managing the QoS management modules can also be involved in dynamic inconsistencies. These conflicts occur between policies governing the behaviour of the on-line SLS-I and DRsM modules, and can only be detected dynamically as their manifestation depends on the current state of the underlying managed resources. This chapter is based on the work presented in [97] and addresses the largely unresolved issue of dynamic conflict analysis. We first identify and classify possible inconsistencies that may emerge at run-time along with the description of the conditions under
which they arise, and then define the rules that can be used to detect their occurrence. The methodology presented in this chapter caters for the requirements posed by dynamic conflicts, where detection processes can be invoked automatically during the system's operation and resolution can be achieved without human intervention. As with static conflicts, our approach to dynamic analysis is based on the use of the Event Calculus formalism for the representation of both policies and the managed system, which is extended to model policy enforcement. The latter is linked with the behavioural representation of on-line modules so as to provide information about the run-time state of managed resources, which is essential for the detection of dynamic conflicts.

The structure of this chapter is as follows.

Section 5.1 identifies the types of conflicts that can occur between the policies defined for on-line QoS management modules. These conflicts are categorised based on their properties and are described in detail based on the conditions under which they arise.

Section 5.2 defines a set of rules that can be used to detect the identified run-time conflicts. The rules are specified using the Event Calculus and encode the conditions under which the conflicts occur in the form of constraints.

Section 5.3 presents our approach towards conflict resolution and defines special rules, which, when enforced, can resolve the identified conflict types. This section also describes the additional functionality that is required by the managed system to support resolution logic.

Section 5.4 describes how the formal representation of the system, policy and detection rules can be used by deductive reasoning to determine policy conflicts at run-time. Additionally, this section describes the different components of the dynamic analysis system architecture including a model for policy enforcement.

Section 5.5 provides a practical demonstration of the dynamic analysis process through a case study involving a set of DRsM policies. The output of a conflict analysis tool developed is used to demonstrate the automated detection and resolution of conflicts arising during the system's execution. Finally, Section 5.6 summarises the chapter.
5.1 Conflict Classification

A number of potential conflicts that can only be determined at run-time have been identified and classified as shown in Figure 5-1. Like most static conflicts, these are all specific to the two QoS management subsystems, but they can be further subdivided into conflicts relating to policies for individual modules, and to policies applying to different modules. These are termed *intra-* and *inter-module* conflicts respectively, the latter denoted with green colour in the classification below.

Apart from policies managing specific modules, in a hierarchical system like the presented QoS management framework, policies targeting upper layer modules may also influence the functionality of lower layer modules as a result of their relationship. One such relationship is, for example, between the ND and DRsM modules that constitute the main body of the Traffic Engineering subsystem. ND-specific policies allow the administrator to constrain the amount of network resources which can be allocated for each QC by providing upper and lower bounds. These policies are communicated to the relevant DRsM modules during the refinement process, acting as constraints throughout the dynamic allocation of resources. Violation of these constraints can occur at run-time in case a newly calculated allocation falls outside the nominal values, giving rise to *inter-module* conflicts. The next two sub-sections describe the identified dynamic conflicts along with the conditions under which they can arise.

5.1.1 Service Management Dynamic Conflicts

The first two inconsistencies related to service management policies are inter-module conflicts and occur as a result of the hierarchical relationship between the SLS-S and SLS-I modules. To regulate the traffic entering the network, SLS-I works on guidelines provided by SLS-S. These
come in the form of policies defining the service rates which are thought to almost/fully satisfy a service on a per TT basis. They are enforced during the initialisation of SLS-I acting as constraints for the duration of a RPC and although harmonising the operation of the two modules, they may cause run-time conflicts. A maximum service rate violation (srMaxViolation) conflict will occur if the resulting configuration value of a service invocation policy increasing the service rate of a particular TT exceeds the upper boundary, i.e. the fully satisfied rate, provided by the subscription policy. Similarly, a minimum service rate violation (srMinViolation) conflict will occur if an invocation policy decreases the service rate below the minimum permitted value, i.e. the almost satisfied rate.

In addition to activating service rate policies, TCL and VCL threshold crossing alarms also trigger SLS-I policies that manipulate admission control parameters (ACmin, ACmax) aiming to provide proactive and reactive control over invoked services. The relative position of both thresholds and AC parameters in the RAB of each trunk allows the administrator to adjust the strategy by which services are admitted to the network and potentially avoid the built-up of congestion while maximising resource utilisation. As mentioned in the previous chapter, inconsistencies between thresholds and AC parameters may result to an intra-module invocation admission strategy conflict (invcAdmStrg) that can be detected statically. The same conflict can also arise during the operation of the SLS-I module as AC parameters are re-calculated on the fly. The conflict will materialise if a newly calculated value for ACmin exceeds the TCL threshold, or in case the value of ACmax falls below VCL for a particular trunk.

5.1.2 Traffic Engineering Dynamic Conflicts

Run-time inconsistencies related to policies driving the functional behaviour of traffic engineering modules may arise as a result of enforcing DRsM policies that calculate new BW allocation and threshold values. Two such inconsistencies occur due to ND directives which are communicated and executed by DRsM during its initialisation stage. These policies define upper and lower bounds per QC acting as constraints for the allocation of resources between which DRsM should operate. The enforcement of dynamic actions altering the allocation may violate the constraints leading to inter-module conflicts. This means that a maximum ND violation (ndMaxViolation) conflict will arise if the resulting configuration value of a policy increasing the resource allocation of a particular QC exceeds the upper boundary. Similarly, a minimum ND violation (ndMaxViolation) conflict will occur if a DRsM policy decreases the allocation below the minimum permitted value.

Another high-level directive that is refined down to the DRsM level is a general resource management policy, which explicitly specifies the amount of link resources to be allocated among
the various QCs during a DRsM operational cycle. This implies that a DRsM policy action aiming
to increase or decrease the allocation for a specific QC can violate the above rule as the resulting
allocation may exceed or be less than the specified value. We term these inconsistencies as over-
allocation (overAlloc) and under-allocation (underAlloc) conflicts respectively.

The last DRsM-related conflict is an intra-module conflict and involves policies responsible for
the computation of new thresholds and allocation of resources. The inconsistency arises if the
allocated BW is below its respective upper utilization tracking threshold, in which case a
threshold incompatibility (thrshIncompat) conflict should be signaled.

The next section provides the definitions of the necessary rules for detecting the identified
conflicts encapsulating the policies involved and the conditions under which the conflicts will
materialise.

5.2 Conflict Detection Rules

As with static conflicts, the detection of run-time inconsistencies is also based on the definition of
conflict predicates using the Event Calculus formalism, which, in this case, require additional
information regarding the run-time state of on-line modules. In this context, the conditions under
which a conflict will arise are represented by constraints that depend on the conflict type. The
rules for detecting such conflicts are based on the fact that two policies violate these constraints.

In order to meet the requirements of the dynamic analysis approach presented in subsequent
sections and facilitate an automated process, detection rules need to follow a specific structure.
More specifically, the conditions encapsulated in the body need to be ordered as follows:

- Conflict-causing policy – the policy whose action can potentially violate a constraint
  imposed by another policy when enforced.
- Constraining policy – the policy defining a constraint that should not be violated.
- Set of conditions – these include policy parameters and run-time state information that
  need to be satisfied for the conflict to materialise.

This structure is reflected in the conflict predicates defined in the next two sub-sections.

5.2.1 Service Management Conflict Rules

Based on the description of conditions under which service management conflicts occur, here we
present the rules for detecting two of the identified dynamic conflicts. The first rule concerns the
violation of the constraint imposed by the SLS-S module regarding the maximum service rate to
be allocated to a TT for the duration of a RPC. The conflict predicate in Listing 5-1 aims to
match a SLS-I policy that increases the service rate and a SLS-S refined directive defining the fully satisfied rate (SRfS). The predicate will hold if the maximum permitted value is exceeded by the increased service rate (SR) for a particular trunk. The latter is acquired using the reqSR predicate, which reads the required rate from a management information base.

Listing 5-1: Rule for detecting maximum service rate violation conflicts

The second rule concerns invocation admission conflicts between policies that increase the minimum AC parameter and those setting the TCL threshold. Listing 5-2 aims to match such policies and signal a conflict if the newly calculated value of ACmin is higher than that of TCL for a particular TT. This conflict type can also occur if, during a re-configuration, the maximum AC parameter falls below the value of VCL. The definition of this rule together with the one catching srMinViolation conflicts can be found in Appendix A3.

Listing 5-2: Rule for detecting invocation admission strategy conflicts

5.2.2 Traffic Engineering Conflict Rules

Following the detection rules in the context of dynamic service management conflicts, here we describe the rules for detecting two of the identified conflicts between traffic engineering policies. The first rule (Listing 5-3) determines ndMaxViolation conflicts and aims to match a DRsM policy for increasing the BW allocation of a QC and a directive originating from ND that sets the upper allocation bound (NDmax) for that QC. The conditional part of the rule specifies that such a conflict will be detected if the required allocation (BW), acquired by the reqBW predicate, is higher than the permitted maximum value.
Chapter 5. Dynamic Policy Analysis and Conflict Resolution

Listing 5-3: Rule for detecting ND maximum violation conflicts

```
conflict(ndMaxViolation, conflictData(PolID1, PolID2, QC1, Link, BW, NDmax), TC) ←
holdsAt(oblig(PolID1, Subj, operation(bamo, incrAllOC, [Link, QC1, Val]), TC) ∧
holdsAt(oblig(PolID2, Subj, operation(initmo, setAllocmax, [QC2, NDmax]), TC) ∧
reqBW(Link, QC1, BW) ∧ BW>NDmax ∧ QC1==QC2.
```

The second rule (Listing 5-4) determines overAlloc conflicts and aims to match a DRsM policy for increasing the BW allocation of a QC and the general resource management directive that defines the amount of link resources (Alloc) to be allocated among the various QCs. The conditions for this conflict will be satisfied if the collective allocation of all supported QCs (bw) exceeds the value of the variable Alloc. The definitions of the rules for detecting ndMaxViolation, underAlloc and thrshIncompat conflicts can be found in Appendix A4. As in the examples provided here, the conditional parts of these rules include the policies involved and the run-time state of resources that have to be met for a conflict to occur.

Listing 5-4: Rule for detecting over-allocation conflicts

```
conflict(overAlloc, conflictData(PolID1, PolID2, Link, QC, BW, Alloc), TC) ←
holdsAt(oblig(PolID1, Subj, operation(bamo, incrAlloc, [Link, QC, Val]), TC) ∧
holdsAt(oblig(PolID2, Subj, operation(initmo, setSrcAlloc, [Link, Alloc]), TC) ∧
totalReqBW(Link, BW) ∧ BW>Alloc.
```

5.3 Conflict Resolution

As described in the previous chapter, the resolution of static conflicts requires human intervention since the nature of the identified conflicts does not allow for the deployment of automated resolution techniques. Despite the fact that this process is performed manually, it takes place before policies are deployed in the system and does not impose any run-time overheads on the functionality of on-line modules. Dynamic conflicts however, require system components to both detect and resolve conflicts in real-time, without degrading the performance of the system. This section describes our approach towards the resolution of dynamic conflicts and shows how the system specification can be extended to support such functionality.

5.3.1 Conflict Resolution Rules

Having identified the different inconsistencies that may arise at run-time, a network administrator can pre-specify rules that aim to provide a resolution strategy for each of these conflicts. Unlike other resolution methodologies [18][44][28], the approach presented here does not involve identifying which of the conflicting policies will prevail, but provides separate resolution rules
that handle potential inconsistencies. These rules are effectively obligation policies introduced by
the administrator in Ponder format and their triggering events are conflict occurrences rather than
network events. As with policies driving the behaviour of the QoS management framework,
resolutions are mapped to their Event Calculus representation and enforced when a conflict is
signalled.

The resolution policies for most inconsistencies aim to configure the parameter involved in the
conflict to have a value equal to the violated constraint, i.e. assign the maximum or minimum
permissible value depending on the type of conflict. In the following example relating to the
srmaxviolation conflict, the resolution policy sets the service rate to the maximum permissible
value defined by the relevant SLS-S directive. The resolving value, srfs, for a particular TT can
be acquired from the parameters of the SLS-S policy on the fly – as this quantifies the relevant
variable in the data associated with the conflict predicate – and instantiate the relevant parameter
in the resolution policy action. The latter can be re-used for multiple occurrences of the same
inconsistency alleviating the need for human intervention. A similar policy can be defined to
handle srminviolation conflicts using the minimum permissible service rate in the resolution.

\[
\text{initiates(\text{systemEvent(conflictDetected(srmaxviolation,}
\text{conflictData(PolID1, PolID2, TT, srfs, SR))},
\text{oblig(resPoll, sISIPMA, operation(servAdjustMD, setSR, [TT, srfs])), T}).}
\]

Since the resolution rules are part of the formal description, an analysis engine can determine
which resolution policy applies for a particular conflict predicate based on the information
provided for that conflict. The work in [59] and [107] describes an alternative approach for the
handling of dynamic inconsistencies and follows the validation principle of [60]. The authors
propose the use of constraints to prevent a policy from firing if a new configuration parameter is
not consistent with an associated system variable. Although this approach can prevent a run-time
conflict, it may also prevent the system from making a potentially essential re-configuration to
meet an SLS requirement. Consider, for example, the fully satisfied service rate (srfs) of a trunk
to be 100Mbps, and a policy that increases the rate allocated to that trunk by 20% when executed:
if the policy triggering condition is met when the current allocation is at 90Mbps, the constraint
will prevent the policy from firing and as a result the rate allocation will remain unchanged. Our
approach overcomes this problem and allows for the correct configuration of resources, which, in
this case, is the maximum permissible value of srfs.

The policy actions responsible for the resolution of the rest of the identified conflicts can be found
on Table 5-1. These can be used to encode the action part of resolution policies which take as
events the corresponding conflict type along with the descriptive data values as shown on the left
column of the table. It is worth noticing that in the case of a ND-DRsM ndmaxviolation conflict,
apart from setting the allocation to the maximum permissible value ($ND_{\text{max}}$) defined by the ND directive, additional resolution policies set the upper monitoring threshold to $ND_{\text{max}}$ so that it is consistent with the BW allocation and an alarm is raised notifying the ND module about the event. The latter may decide to initiate a new resource provisioning cycle depending on the frequency of these events. Similar actions are taken to handle an $nd_{\text{minviolation}}$ conflict but in this case the lower monitoring threshold is set to zero as to avoid further decrease in BW allocation.

<table>
<thead>
<tr>
<th>Table 5-1: Resolution policy actions for the identified conflict types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conflict</strong></td>
</tr>
<tr>
<td>$sr_{\text{maxviolation}}(-, -, TT, SRfS, SR)$</td>
</tr>
<tr>
<td>$sr_{\text{minviolation}}(-, -, TT, SRfS, SR)$</td>
</tr>
<tr>
<td>$invC ADMstrg1(-, -, TT, ACmin, TCL)$</td>
</tr>
<tr>
<td>$invC ADMstrg2(-, -, TT, ACmax, VCL)$</td>
</tr>
<tr>
<td>$nd_{\text{maxviolation}}(-, -, QC, Link, BW, ND_{\text{max}})$</td>
</tr>
<tr>
<td>$nd_{\text{minviolation}}(-, -, QC, Link, BW, ND_{\text{min}})$</td>
</tr>
<tr>
<td>$over Alloc(-, -, Link, BW, Alloc)$</td>
</tr>
<tr>
<td>$under Alloc(-, -, Link, BW, Alloc)$</td>
</tr>
<tr>
<td>$thrshIncompat(-, -, QC, Link, BW, Thuppr)$</td>
</tr>
</tbody>
</table>

The resolution of DRsM overAlloc and underAlloc conflicts does not follow the procedure of configuring the value of the violated constraint. These conflicts can be handled with different strategies, as depicted on Table 5-1, where the resolutions are inspired by the policies used in the post-processing stage of the ND module: the distribution of spare capacity, or the reduction of over-provisioned BW can be done equally (equal) between the QCs, proportionally (prop) to the current allocation, or explicitly (expI), where the amount of BW is specified as a percentage.

### 5.3.2 Support for Resolution Functionality

The fact that a new set of policies, not part of the existing system policy specification, is required to handle conflicting situations at run-time, implies that the functionality supported by on-line QoS management modules needs to be extended to facilitate resolution logic. Here, we provide the state machine representations for the DRsM and SLS-I modules that illustrate the new
operations supported and state transitions as a result of enforcing conflict resolution policies. The state machines do not replace the existing functionality of the modules, but they instead build on top of that. Furthermore, the new operations are supported by the existing MOs since they have similar characteristics with the ones previously defined; for example the DRsM MO supporting operations to increase/decrease the BW allocation, now also supports the operation for setting the allocation to a specific value.

Figure 5-2 below, depicts the behaviour of the SLS-I module as a result of incorporating resolution logic. In the event of a conflict, executing any of the three pre-defined resolution policies drives the module into a distinct state in which a conflict is resolved: SRSet state for minimum and maximum service rate violation conflicts, ACmaxSet and ACminSet states for InvCadmStrg conflicts. The operations for setting the service rate and admission control parameters are all supported by the service adjustment MO. Once a conflict has been resolved, the module returns to its idle state.

The resolution behaviour of the DRsM module is represented by two state machines since the monitoring component is treated independently – Figure 5-3. Here, overAlloc and underAlloc conflicts are resolved by enforcing the relevant policies and transitioning to overAllocCrsd or spareCapAllocd states respectively. The resolution of ndmaxViolation and ndminViolation conflicts however, requires the enforcement of multiple policies. Apart from setting the allocation and visiting the AllocSet state, the module alters threshold values in the monitoring component and raises a ND alarm. Finally, a thrshlncompat conflict is resolved after transitioning to the upperThset state. The BW allocation MO is responsible for the enforcement of BW manipulation operations, whereas the monitoring MO supports operations that set the value of thresholds and raise ND alarms.
As with the state machines presented in Chapter 3, the states that on-line modules can take as a result of resolution policy enforcement can be represented in Event Calculus with initiates and terminates predicates.

### 5.4 Analysis of Dynamic QoS Management Conflicts

Having defined the rules for the detection of run-time conflicts and the policies for their resolution, this section describes how these can be used to effectively deploy dynamic analysis processes. Since the detection of dynamic conflicts requires information relating to the current state of the underlying managed resources, we show how such a process can be embedded within policy agents managing on-line modules, and how it can be automatically initiated alleviating the need for human intervention. Additionally, we show how deductive reasoning can be used to determine run-time inconsistencies, the result of which is used to trigger the identified resolution policies pertaining to a conflict.

#### 5.4.1 Deductive Reasoning for Conflict Detection

Although the detection of run-time inconsistencies is also based on predicates that define conflicting conditions, as in the case of static analysis, the two approaches differ in the fact that deductive reasoning is used instead of abduction. This is because complete system specification is available to reason over, which is provided by run-time events and the current state of the managed modules. As such, the defined conflict predicates can be used as goal states of deductive queries to determine the occurrence of dynamic inconsistencies.

To demonstrate how deduction can be used to derive a conflict, we use the example of Section 5.3.1 regarding a maximum service rate violation. The rules involved in this inconsistency are presented in Listing 5-5, where the first is a conflict-causing policy that increases the rate of trunk t1 by 20% upon a downward TCL crossing alarm, and the second is a constraining policy setting the value of the fully satisfied service rate for the same trunk, at 100Mbps.
Listing 5-5: Policy instances

```prolog
goal1 = initiates(systemEvent(slSAIAlarmRaised(tcIDown, tt2)),
             oblig(p1, slSIPMA, operation(servAdjustMO, incrSR, [tt2, 20])), T).

goal2 = initiates(systemEvent(newRPC),
             oblig(p2, slSIPMA, operation(initMO, setSRfS, [tt2, 100])), T).
```

Assuming that the current service rate for tt2 is at 90Mbps, the alarm triggering the first policy is raised at T=2, and the second policy has been enforced at the start of a new RPC at T=1. Figure 5-4 shows the evaluation tree of a deductive query aiming to determine a \(srMaxViolation\) conflict. The goal \(\text{conflict}(srMaxViolation, \text{conflictData}(\text{PolMl}, \text{P01D2}, \text{TT}, \text{SRfS, SR}), \text{TC})\) will be satisfied if the goals specified in the conditional part of the conflict rule can be achieved. Deductive reasoning recursively determines the validity of the two policies at T=2, and satisfies the remaining conditions for the conflict as the resulting policy enforcement requires a service rate of 108Mbps, which violates the constraint. During this process, the variables of the conflictData term are unified thus providing all the information pertinent to the conflict.

```
conflict(srMaxViolation, conflictData(p1, p2, tt1, 100, 108), 2)
   | holdsAt(oblig(p1, slSIPMA, operation(servAdjustMO, incrSR, [tt2, 20])), 2)
   |         happens(systemEvent(slSAIAlarmRaised(tcIDown, tt2)), 2)
   |         initiates(systemEvent(slSAIAlarmRaised(tcIDown, tt2)), oblig(p1, slSIPMA, operation(servAdjustMO, incrSR, [tt2, 20])), 2)
   | holdsAt(oblig(p2, slSIPMA, operation(initMO, setSRfS, [tt2, 100])), 2)
   |         happens(systemEvent(newRPC), 1)
   |         initiates(systemEvent(newRPC), oblig(p2, slSIPMA, operation(initMO, setSRfS, [tt2, 100])), 1)
   | reqSR(tt2, 108)
   | 108 > 100
   | tt2 == tt2
```

Figure 5-4: Deductive reasoning evaluation tree

5.4.2 Policy Enforcement Model

It is evident that run-time inconsistencies arise as a result of a change in the state of a managed module, which in turn is caused by the execution of a new policy. As such, it is necessary to define the rules that model policy enforcement. Figure 5-5 below illustrates this process, which is similar to that of Ponder, but does not involve authorisation at the target end.
As shown, a network event from the monitoring component of a QoS management module is received by the Event Handler in the subject’s policy agent, which is forwarded to the Policy Enforcer. The latter enquires the policy repository to determine if that event serves as the trigger for any of the obligation policies. In such a case, a request to perform an operation is dispatched to the target. Here, the operation is evaluated by the relevant managed object which performs the requested action. This causes a state transition in the behavioural representation of the managed module and the derivation of new resource configuration values corresponding to a specific policy action.

Listing 5-6: Formal representation of obligation policy enforcement model

(a) \[ \text{requestAction}(\text{operation}(\text{ObjTarg}, \text{OpName}, \text{Params}), T1) \leftarrow \text{initiates}(\text{Event}, \text{oblig}(\text{PolID}, \text{ObjSubj, operation}(\text{ObjTarg}, \text{OpName}, \text{Params})), T1) \land \text{happens}(\text{Event}, T2) \land (T1>T2). \]

(b) \[ \text{happens}(\text{doAction}(\text{operation}(\text{ObjTarg}, \text{OpName}, \text{Params})), T1) \leftarrow \text{requestAction}(\text{operation}(\text{ObjTarg}, \text{OpName}, \text{Params}), T2) \land \text{operation}(\text{ObjTarg}, \text{OpName}, \text{Params}) \land (T1>T2). \]

The representation of the policy enforcement model in Event Calculus is provided in Listing 5-6. The first rule models the behaviour of the subject’s policy enforcer, where the requestAction predicate holds after an event has triggered an obligation policy. This predicate is used in the conditional part of the second rule, which models the behaviour of a managed object in the target environment. The enforcement will complete by asserting the doAction(...) fluent as an event after
the request, if the operation is supported by the relevant MO. This model is part of the architecture presented in the next sub-section.

5.4.3 System Architecture

As described in the precious chapter, detection engines for static inconsistencies form an integral part of a PMT, analysing policies before they are downloaded to the QoS management modules. The manifestation of dynamic inconsistencies however, depends on the current state of the network and the resulting configuration output of on-line modules. For this reason, the process for handing dynamic conflicts needs to be embedded within policy management agents which have access to the run-time information required.

The architecture of the dynamic analysis system developed is outlined in Figure 5-6, which involves an instance of a PMA and an associated on-line module. The latter is represented by an Event Calculus model which allows the enforcement of policy actions through state transitions and configuration changes, and can also generate events about emerging network conditions.

Policies stored in the PMT repository are translated to their Event Calculus representations and downloaded to a cache local to the PMA. These include resolution policies which have previously been checked for static module-independent conflicts. This is done to remove potential duplicates and resolution actions that are mutually exclusive between them, as in the case of allocating spare BW in the DRsM module. Policy execution is triggered by the Event Handler which processes events from the dynamic network environment and forwards them to the Policy Enforcer.
Successful policy enforcement results to a new resource configuration and state transitions in the behavioural model of the on-line module.

To facilitate a seamless conflict handling environment and avoid driving the managed system into inconsistent states, the deployment of a dynamic analysis process requires automation. This is achieved by processing the detection rules a priori and extracting information about policy actions that can potentially cause a conflict when enforced. These are encoded in the first field in the conditional part of detection predicates, as explained in Section 5.2, and are used to derive the states, from the behavioural representation of a module, that are associated with the enforcement of such actions. The resulting states are used by the Event Handler, which notifies the analysis engine upon intercepting a system event matching one of the identified states, thus initiating the detection process. For example, the policy action causing an ndmaxviolation conflict is associated with the allocincr5 state in the DRsM module. If the enforcement of a new policy drives the module into this state, the detection logic will perform a query to determine the occurrence of an ndmaxviolation conflict. As described in Section 5.4.1, the query is based on deductive reasoning, which makes use of the network events maintained by the Event Handler to evaluate the conditions of a conflict.

If a conflict materialises during the detection process, an event is generated containing the details associated with that conflict and the resolution logic is invoked. Using the DRsM example above, the conflict event has the following format:

```
systemEvent(conflictDetected(ndMaxviolation,
    conflictData(PolID1, PolID2, QC, BW, NDmax)))
```

The resolution engine enters a resolving state which performs a search in the cache repository for a possible resolution pertaining to the detected conflict. If an appropriate resolution policy is identified the Event Handler is notified, which in turn triggers the enforcement of that resolution. The output of dynamic analysis is passed to an interface in the PMT, which can be used to monitor or keep a log of the overall process.

### 5.5 Dynamic Conflict Analysis Case Study

In this section we present an example scenario that demonstrates the use of dynamic logic to detect and resolve conflicts emerging during the operation of the DRsM module. The results presented form the output of the analysis process deployed in the architecture described in the previous section which is based on Prolog [115] and its deductive reasoning capabilities. We assume that two traffic types are defined for the underlying network, namely EF and AFI, for
which the associated values regarding allocation, thresholds and ND constraints on link1 are presented in Table 5-2. All values are expressed as a percentage of the total link capacity.

Table 5-2: QC associated values

<table>
<thead>
<tr>
<th>Link</th>
<th>QC</th>
<th>Alloc</th>
<th>NDmin</th>
<th>NDmax</th>
<th>upprTh</th>
<th>lowrTh</th>
</tr>
</thead>
<tbody>
<tr>
<td>link1</td>
<td>EF</td>
<td>60</td>
<td>40</td>
<td>65</td>
<td>55</td>
<td>45</td>
</tr>
<tr>
<td>link1</td>
<td>AFI</td>
<td>40</td>
<td>20</td>
<td>50</td>
<td>35</td>
<td>25</td>
</tr>
</tbody>
</table>

5.5.1 Policy Set

Listing 5-7 defines a set of policies enforced on DRsM in their EC representation. Policies p1 and p2 specify how the allocation and thresholds are to be increased in case of an upper threshold crossing alarm; policies p3-p6 represent the constraints imposed by ND that are enforced during the initialisation of the module; p7 signifies the full allocation of link capacity; and policies p8-p11 provide resolution strategies for ndmaxviolation, overAlloc and thrshIncompat conflicts respectively.

Listing 5-7: Policy instances

```plaintext
 initiates(SystemEvent(drsmAlarmRaised(upprTh, link1, ef)),
 oblig(p1, drsmPMA, operation(bamo, incrAllocRel, [link1, ef, 20])), T).
 initiates(SystemEvent(drsmAlarmRaised(upprTh, link1, ef)),
 oblig(p2, drsmPMA, operation(monitormo, incrThsRel, [link1, ef, 20])), T).
 initiates(SystemEvent(newRPC),
 oblig(p3, drsmPMA, operation(initmo, setAllocMax, [ef, 65])), T).
 initiates(SystemEvent(newRPC),
 oblig(p4, drsmPMA, operation(initmo, setAllocMin, [ef, 40])), T).
 initiates(SystemEvent(newRPC),
 oblig(p5, drsmPMA, operation(initmo, setAllocMax, [afl, 50])), T).
 initiates(SystemEvent(newRPC),
 oblig(p6, drsmPMA, operation(initmo, setAllocMin, [afl, 20])), T).
 initiates(SystemEvent(newRPC),
 oblig(p7, drsmPMA, operation(initmo, setRsrcAlloc, [link1, 100])), T).
 initiates(SystemEvent(conflictDetected(ndMaxViolation, conflictData(PolID1, PolID2, QC, BW, NDmax))),
 oblig(p8, drsmPMA, operation(bamo, setAlloc, [link, QC, NDMax])), T).
 initiates(SystemEvent(conflictDetected(ndMaxViolation, conflictData(PolID1, PolID2, QC, BW, NDmax))),
 oblig(p9, drsmPMA, operation(monitormo, setThuppr, [Link, QC, NDmax])), T).
 initiates(SystemEvent(conflictDetected(overAlloc, conflictData(PolID1, PolID2, Link, ef, BW, Alloc))),
 oblig(p10, drsmPMA, operation(bamo, redoverBWProp, [Link])), T).
 initiates(SystemEvent(conflictDetected(thrshIncompat, conflictData(PolID1, PolID2, QC, Link, BW, Thup))),
 oblig(p11, drsmPMA, operation(monitormo, setThuppr, [Link, QC, BW])), T).
```
5.5.2 Dynamic Analysis

By using one of the conflict fluents as a goal state of a deductive query, it is possible to detect conflicts during system execution. The queries use the predicates defined for the conflicts we aim to detect in this scenario and are triggered automatically based on the state of the DRsM module, as explained in the previous section. If a particular state triggers the detection of more than one conflict, the sequence with which queries are executed depends on the ordering of the conflict rules. In addition, a single conflict can be handled at a time, which means that a new conflict query will be executed only when the resolution of a previous conflict has completed.

The results of queries indicate if there is a conflict of a particular type; the detection of a conflict causes the system to generate an event containing the conflict information which is passed to the resolution engine. The latter identifies the appropriate policy to handle the conflict which is subsequently enforced. The timeline in Listing 5-8 shows the sequence of events (systemEvent(...)), actions (doAction(...)), and fluents (oblig(...), conflict(...)) that describe the different stages that our system goes through, upon an upper threshold crossing alarm for EF traffic, before producing the appropriate configuration for Tink1.

The generated alarm, at $T=1$, triggers policies $p_1$ and $p_2$, which increase the allocation and thresholds for EF traffic by 20%, thus producing new configuration values and driving the DRsM and monitoring components to states that reflect the enforced policies. At this point, dynamic detection logic is invoked since the state allocIncrsd can potentially lead to any of the three conflicts we provide resolutions for. As the ordering of the detection rules used in this example follows the sequence ndMaxViolation, overAlloc and thrshIncompat, at $T=5$ a query is performed that first checks for an ndMaxViolation conflict. This conflict is detected at $T=6$ between policies $p_1$ and $p_3$ since the required increased BW (72%) exceeds the maximum permissible value of 65% defined by the ND directive. This result acts as a trigger for the relevant resolution policies ($p_8$ and $p_9$), which set the allocation and upper threshold for EF traffic to 65%.

After successful resolution, and at $T=10$, a second query is performed for the detection of an overAlloc conflict. The goal of the query is achieved as the sum of the required BW values for EF and AFI traffic exceed the maximum link capacity (105%). As such, a conflict is signalled between policies $p_1$ and $p_7$, which is subsequently resolved by enforcing policy $p_{10}$. The latter reduces the over-allocated BW proportionally among the two QCs providing a new allocation of 62% and 38% for EF and AFI respectively.

The last query, which is performed at $T=15$, concerns a thrshIncompat conflict. This inconsistency materialises as the increased upper threshold value of 65% exceeds the current allocation for EF traffic. Policy $p_{11}$ resolves the conflict by setting this threshold to 62%. Finally, all newly calculated values are used to configure the link at $T=20$. 

105
### Listing 5-8: Dynamic conflict analysis trace

<table>
<thead>
<tr>
<th>T - Event / Action / Fluent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - systemEvent(drsmAlarmRaised(upprTh, link1, ef))</td>
</tr>
<tr>
<td>2 - oblig(p1, drsmPMA, operation(bamo, incrAllocRel, [link1, ef, 20])) oblig(p2, drsmPMA, operation(monitorMO, incrThsRel, [link1, ef, 20]))</td>
</tr>
<tr>
<td>3 - doAction(operation(bamo, incrAllocRel, [link1, ef, 20])) doAction(operation(monitorMO, incrThsRel, [link1, ef, 20]))</td>
</tr>
<tr>
<td>4 - reqBW(link1, ef, 72) reqThuppr(link1, ef, 66) systemEvent(state(drsm, allocIncrsd)) systemEvent(state(drsmMon, thsIncrsd))</td>
</tr>
<tr>
<td>5 - conflict(ndMaxViolation, conflictData(PolID1, PolID2, QC, Link, BW, NDmax))</td>
</tr>
<tr>
<td>6 - systemEvent(conflictDetected(ndMaxViolation, conflictData(p1, p3, ef, 72, 65)))</td>
</tr>
<tr>
<td>7 - oblig(p8, drsmPMA, operation(bamo, setAlloc, [link1, ef, 65])) oblig(p9, drsmPMA, operation(monitorMO, setThuppr, [link1, ef, 65]))</td>
</tr>
<tr>
<td>8 - doAction(operation(bamo, setAlloc, [link1, ef, 65])) doAction(operation(monitorMO, setThuppr, [link1, ef, 65]))</td>
</tr>
<tr>
<td>9 - reqBW(link1, ef, 65) reqThuppr(link1, ef, 65) systemEvent(state(drsm, allocSet)) systemEvent(state(drsmMon, upprThSet))</td>
</tr>
<tr>
<td>10 - conflict(overAlloc, conflictData(PolID1, PolID2, Link, QC, BW, Alloc))</td>
</tr>
<tr>
<td>11 - systemEvent(conflictDetected(overAlloc, conflictData(p1, p7, link1, ef, 105, 100)))</td>
</tr>
<tr>
<td>12 - oblig(p10, drsmPMA, operation(bamo, redOverBWProp, [link1]))</td>
</tr>
<tr>
<td>13 - doAction(operation(bamo, redOverBWProp, [link1]))</td>
</tr>
<tr>
<td>14 - reqBW(link1, ef, 62) reqBW(link1, afl, 38) systemEvent(state(drsm, overAllocDecrsd))</td>
</tr>
<tr>
<td>15 - conflict(thrshIncompat, conflictData(PolID1, PolID2, QC, Link, BW, ThUp))</td>
</tr>
<tr>
<td>16 - systemEvent(conflictDetected(thrshIncompat, conflictData(p1, p2, ef, link1, 62, 65)))</td>
</tr>
<tr>
<td>17 - oblig(p11, drsmPMA, operation(monitorMO, setThuppr, [link1, ef, 62]))</td>
</tr>
<tr>
<td>18 - doAction(operation(monitorMO, setThuppr, [link1, ef, 62]))</td>
</tr>
<tr>
<td>19 - reqThuppr(link1, ef, 62) systemEvent(state(drsmMon, upprThSet))</td>
</tr>
<tr>
<td>20 - systemEvent(state(drsm, configuringLink))</td>
</tr>
</tbody>
</table>

### 5.6 Summary and Conclusions

Dynamic conflict analysis is known to be the least addressed and one of the most challenging problems in the area of policy-based management. This chapter presented our contributions towards this problem by first identifying the various inconsistencies that may arise between policies driving the behaviour of on-line QoS management modules and which can only be detected at run-time. These were classified into conflicts relating to policies for individual modules (intra-module), and to policies applying to different modules (inter-module) as a result of
their hierarchical relationship, identifying for each conflict type the conditions under which they arise.

The dependency of such conflicts on the run-time state of the managed system guided the design of our approach to deploy analysis engines within policy agents so that state information from the managed resources can be readily available. Although the detection of dynamic inconsistencies shares the same principles with static analysis regarding the definition of conflict predicates, the reasoning technique used is that of deduction rather than abduction. This is because run-time events and the run-time state of resources provide a complete system specification over which to reason. The main novelty of the detection methodology proposed here is that such a process can be initiated automatically during system operation, which is achieved by monitoring for events regarding system states that can potentially lead to a conflict.

The requirements for a fully automated analysis process, so as to prevent the degradation of the system’s run-time performance, are partly fulfilled by the proposed detection method. Our resolution approach completes this objective by providing specific rules that handle the identified conflicts types. These are in the form of pre-defined resolution policies that are triggered and enforced upon successful detection of a conflict, thus alleviating the need for human intervention. Furthermore, they are generic enough with only few required for each conflict type to cater for multiple occurrences of the same inconsistency.

Most approaches in the literature propose the use of precedence rules when resolving a conflict, the most representative being the works in [18] and [44]. Although resolution based on the assignment of priorities to conflicting policies may be useful in some occasions, we believe that this may not be a flexible solution to the problem, especially when application-specific environments are concerned, as demonstrated in our examples where new policies need to be enforced. The approach based on the use of constraints within a policy to prevent it from firing if the resulting configuration is conflicting [107], is also not a viable solution as explained in Section 5.3.1; although it can prevent a run-time conflict, it may also prevent the system from making a potentially essential re-configuration. A methodology similar to the one described in this chapter was proposed in [40] where specific processes handle call control policy conflicts and are based on the notion of resolution policies. The detection of conflicts however is not supported by a separate process, but the various conditions are encoded within resolution policies instead. Resolution specifications can thus become complex and their evaluation quite expensive.
Chapter 6

6 Tool Support and Experimental Evaluation

The proposed analysis techniques have been developed and integrated into a policy conflict analyser. This tool implements static and dynamic analysis engines based on Prolog, supports Ponder policy specifications and has mapping capabilities to formal representation, integrates emulated execution environments of on-line QoS modules, and provides a conflict analysis user interface.

Figure 6-1 shows the architecture of the policy analysis tool developed, which has three main components: the static analyser, the dynamic analyser, and the user interface client application. The first component implements the detection logic for QoS management static policy conflicts making use of Prolog in conjunction with the A-System abductive proof engine. The dynamic analyser implements the detection and resolution logic for conflicts that can arise during system execution. This is based on Prolog's deductive reasoning capabilities and the implementation of an environment that emulates the behaviour of on-line modules. Lastly, the analysis client application provides a graphical interface through which the Prolog engines can be invoked and the analysis output to be acquired.

The remaining three components of the architecture provide the means to store policies in Ponder format, maintain system management information, and represent the behaviour of QoS management modules in Event Calculus notation.
The first three sections detail the implementation aspects of the tool, whereas the latter part of this chapter presents and discusses a number of experimental results that evaluate the performance of the static analyser over a range of conflict types, and validate the functional behaviour of the dynamic analysis process.

6.1 Policy Management and Analysis Client

The logic formalism and conflict analysis techniques presented in this thesis can be difficult to use and understand. The client developed provides an interface through which a user can interact with the Prolog engines in a relatively simple manner, shielding him/her from the complexity of logic specifications and reasoning methods. The client, developed in Java, provides object representations for both policies and conflicts, and implements a graphical interface that allows the user to: (a) display managed system resources, (b) create/retrieve/view QoS management policies in Ponder format, and, (c) perform static and dynamic policy conflict analysis. This section describes the various components and features of the client.

6.1.1 Design

The Java classes that implement the client tool and their relationships are depicted in Figure 6-2. PolicyConflictAnalyser is the main class implementing the graphical interface through which the user can interact with the tool. It defines two modes of operation – static and dynamic – and provides methods to enter/view policies, display the managed resources specified in the information store, and perform static and dynamic conflict analysis. The latter is achieved by invoking the Prolog analysis engines and passing queries and events through Prolog’s JPL interface [117] with Java. Analysis results are added to a list of conflicts maintained by this class through methods addStaticConflictsToList() and addDynamicConflictsToList().

The Policy class provides the representation of a policy as a Java object. It has an ID attribute which is used to distinguish between policy instances, and provides methods to get and set the components of a policy. The latter are implemented by the PolicyEvent, PolicyAction, and PolicyConstraint classes which represent events, actions and constraints respectively. The policy class has a single PolicyEvent and PolicyAction object and a vector of PolicyConstraint objects. This class also provides methods to represent the components of a policy object in text form, both in Ponder format (toString()) and in Event Calculus notation (toPrologString()). The Policy class is used by the PolicyConflictAnalyser for creating and displaying policies and also for converting policy specifications to their Event Calculus representation.
The **PolicyParser** class provides the functionality to read Ponder policy specifications and store them in a dynamic vector as **Policy** objects. The methods of this class cater for a three step parsing process where the contents of the policy file are read line by line and used to populate the vector. This process is described in more detail in Section 6.1.3. The **PolicyParser** is used by the **PolicyConflictAnalyzer** class when loading new policies to the system.

The **Conflict** class models a policy conflict as a Java object and holds information about the conflict type, the policies involved as well as conflicting parameters. It provides a `toString()` method which returns a structured string representation of a conflict's details. It is used by the **PolicyConflictAnalyzer** class to present to the user any conflicts identified during the detection process.

### 6.1.2 Client Tool Overview

Having described the various classes that implement the policy management and analysis client, this sub-section presents the graphical interface of the tool developed and provides an overview of its functionality.

As shown in the screenshot of Figure 6-3 the tool has three main panels corresponding to static analysis, dynamic analysis and presentation of results. The first allows the user to load Ponder policies from a text file (containing multiple policies) and initiate detection queries for static
conflicts choosing among different inconsistencies to check for. The dynamic panel allows the
user to load policies that drive the operation of on-line modules and to interact with the run-time
execution environment by entering network events which can trigger the detection and resolution
of potential conflicts. Apart from the text area for viewing policies loaded into the system, the
dynamic panel also displays information about the operation of on-line modules. This allows the
user to monitor the dynamic execution of policies, the automated invocation of the detection
process and the enforcement of conflict resolutions.

![Policy management and analysis tool interface](Image)

Figure 6-3: Policy management and analysis tool interface

Lastly, the analysis results panel is shared by both processes and displays the output of conflict
analysis by mapping the Event Calculus format to a user friendly representation. This includes a
list of conflicts, explanations for their occurrence and the policy pair involved. The lower part of
the panel displays messages about the operations performed including performance times when
detecting static conflicts.

Apart from interacting with the Prolog conflict analysis engines, the tool (menu bar) allows the
user to create new policies, perform basic Prolog queries, and to access the management
information store and selectively display available entries. The message area on the screenshot of
Figure 6-3, displays information about the four SLSs and the associated TTs currently specified in
the information store.
6.1.3 Policy Input and Parsing

As mentioned above, policies can be introduced into the tool by either loading them directly from plain text files or by creating new ones using the graphical interface provided. The latter process is demonstrated in Figure 6-4 where a policy setting the overall quality level of the EF QoS class to 0.9 is created. This involves three steps that correspond to the relevant tabs of the dialog box: (a) specify the policy ID, triggering event, subject and target, (b) specify the policy action and associated parameters, and, (c) specify any constraints applying to that policy – in this case a time constraint. Once created, the policy is loaded into the system and can subsequently be stored in a new text file or added to an existing one.

When introducing new policy specifications in the tool, by either loading a text file or creating new ones as described above, they undergo a three step parsing process which converts them to Java objects. This process is depicted in Figure 6-5 and described below:

- **Step 1**
  A newly created policy or a text file is initially parsed and stored line-by-line in a vector (linevector). Each line in the vector is then read and split into tokens, which are identified as words surrounded by white spaces. The action part of the policy example above would be split into "do" and "setQItLvl(ef, 0.9)" tokens. These are stored in another vector (tokenvector).
Chapter 6. Tool Support and Experimental Evaluation

- **Step 2**
  The tokenvector is read to determine entries that match the "id" keyword. These entries signify the beginning of a new policy the indices of which are stored in an array (policyIdxArray). Thus, the tokens that hold each policy's specification are contained in the tokenvector's entries that fall inside the range signified by two consecutive policyIdxArray entries.

- **Step 3**
  Based on policyIdxArray, all tokenVector entries describing a policy are read sequentially and keywords signifying policy fields (e.g. "id" or "do") are identified. These are used together with the immediate next entry in the tokenVector to populate a new policy object's fields. In the example, the successive entries of "id" and "1000" are used to create a new policy object (policyobject1) and set its identifier field to 1000. When all fields are populated the new policy is stored in a policyvector. This last step is repeated until the tokenvector has been exhausted.

---

### Figure 6-5: The policy parsing process

In addition to parsing policy specifications and creating Java objects, logic policy representations in Event Calculus notation are also generated. These are communicated to the Prolog engine through the JPL interface where they are loaded to the local predicate database and are subsequently analysed for conflicts or used to drive the behaviour of on-line modules.
6.1.4 Interfacing with Prolog

The client tool allows the user to interact with the Prolog environment in terms of initialising the analysis engines, loading policies, searching for conflicts, and introducing dynamic network events. This communication is achieved through the JPL interface and more specifically with its Query class, which is used to query the Prolog database or to invoke built-in predicates.

Instances of the Query class used in the implementation build queries from the given arguments denoting the goal which is to be called. Arguments can be plain Prolog source text as in (a) below or compound terms following a predefined predicate (b). The latter case is used when committing policies to the predicate database.

(a) \( q = \text{new Query("holdSat(Fluent, T")}); \)

(b) \( q = \text{new Query("assert", compoundPolicyTerm);} \)

The Query class implements the Java Enumeration interface, through which successive solutions can be obtained. The allSolutions() method is used for this purpose returning all query solutions as an array of hash tables, each containing key-value pairs and representing a single solution. Taking the query in (a) as an example, and assuming that fluents \( f1nt1 \) and \( f1nt2 \) hold at time \( T=1 \) and \( T=2 \) respectively, the solutions array would have the following two hash tables:

\[
\begin{align*}
\text{solutions}[0] &= \{ \text{Fluent} => \text{flnt1}, \ T => 1 \} \\
\text{solutions}[1] &= \{ \text{Fluent} => \text{flnt2}, \ T => 2 \}
\end{align*}
\]

The value of specific variables in any of the solutions can be retrieved with a get method, but because these are stored as objects of the Java base class object, they have to be cast back to their original type. Fluent values, for instance, should be cast to String type and \( T \) values to integers.

6.2 Static Conflict Analyser

The static analyser implements the logic for the detection of the various conflict types identified in Section 4.1 and takes as input Ponder policies that have been previously converted into their Event Calculus representation. The reasoning engine iterates through the policies aiming to satisfy the conditions specified in conflict rules, and outputs a set of conflicting policy pairs, along with an explanation of their occurrence.

6.2.1 Prolog Implementation

The main Prolog source files implementing the functionality of the static analysis system are depicted in Figure 6-6. When the engine is initialised through the graphical interface described in Section 6.1.2, the staticAnalyzer file is loaded to the Prolog execution environment. This file
subsequently consults the rest of the source files in the diagram and also adds a set of clauses in
the Prolog database which correspond to the various conflict detection queries that can be
executed. These enable the analysis engine to either search for a specific conflict or for the entire
set of identified conflict types based on the user’s selection from the tool. The example clause
below is used for detecting OQL qcPriority conflicts. In the case of multiple inconsistencies, the
conditional part of the clause contains a conjunction of rules, one for each of the identified
conflict types.

\[
solve(\text{conflict(oql QCPrior, ConflictData, T)}):- \\
\text{conflict(oql QCprior, conflictData(PoID1, PoID2, QC1, QC2, OQL1, OQL2), T).}
\]

![Diagram](image)

**Figure 6-6: Static analysis engine implementation files**

The functionality provided by the rest of the Prolog source files is described below:

- **mgmtInfoLoader** loads the predicates representing system management information in
terms of (a) managed objects, including their association with QoS modules and
supported operations, (b) QoS class properties, e.g. relative priorities among QCs, (c)
domains of mutually exclusive policy actions, and, (d) managed resources e.g.
instantiated SLSs and TTs.

- **ecPreds** specifies the Event Calculus base predicates and axioms used by the detection
engine as listed and described in Section 4.2.1, including the rule which is used to
evaluate the holdsat predicate.

- In addition to the duration predicate used when evaluating policies, the file
**helperFunctions** specifies a set of functions that are used during the evaluation of
conflict rules. Examples include the isoverlap, member, listMemberCount, sumof, and
hopCount functions.

- **conflictRules** contains the logic predicates that encapsulate the conditions for the
occurrence of the various static inconsistencies identified in Section 4.3.
• \texttt{staticModIBehvr} contains the Event Calculus representation of the static modules' behaviour by making use of \texttt{initiates} and \texttt{terminates} predicates, which model state transitions.

6.2.2 Conflict Detection Example

Using a drop-down menu in the static analysis panel of the tool, the user can select whether to submit a general query for any possible inconsistency or to perform detection for a specific conflict type among loaded policies. This feature was added to the tool for demonstration and performance evaluation purposes.

![Figure 6-7: Detecting OQL qcPriority conflicts](image)

The screenshot of Figure 6-7 shows the results of statically detecting the occurrence of OQL \textit{qcPriority} conflicts among four SLS-S policies that set the quality level of EF and AFI traffic classes for different times of the day. The bottom part of the results panel outputs general information about the process, including the analysis time and the number of conflicts detected. The upper part of the panel lists the detected conflicts, the pair of conflicting policies, and the details associated with each conflict. The example above shows the involved policies and the details of the first (out of three) conflict. The solution suggests that in the event of a new RPC, the conflict will occur between policies with IDs 1002 and 1001, because of a time overlap during which, the OQL values set are inconsistent in relation to the priorities of the two QCs involved.
6.3 Dynamic Conflict Analyser

For demonstrating the dynamic analysis approach presented in this thesis, a run-time execution environment that emulates the behaviour of on-line modules through state machines has been implemented. This is an Event Calculus based model of the system which allows the enforcement of policy actions. Dynamic reasoning engines interface with the run-time environment through an event handler, which provides a two-way notification service, allowing for an efficient and automated run-time analysis process, including detection invocation, and conflict resolution.

6.3.1 Prolog Implementation

The Prolog source files implementing the functionality of the dynamic analysis system as well as their relationships are depicted in Figure 6-8 below.

![Diagram](image)

When the system is initialised through the graphical interface, the `initialiser` file is loaded to the Prolog execution environment. This file subsequently `consults` the rest of the source files in the diagram (only four connecting arrows shown here for simplicity) and also adds a set of facts in the Prolog database which correspond to the various states of an on-line module that can potentially lead to a conflict.
Deriving these states is essential for automating the invocation of the conflict detection process. This is achieved in three steps:

- **Step 1**
  The first two lines of each of the conflict detection rules are iteratively parsed and two lists (one for each line) are created as follows:

  \[
  \text{Line1} = [\text{conflict, ConflictType, ConflictData, \ldots}]
  \]

  \[
  \text{Line2} = [\text{holdsAt, oblig, PolID, Subj, operation, MO, OpName, \ldots}]
  \]

- **Step 2**
  For each detection rule, the second member of Line1, as well as the sixth and seventh members of Line2, are used to instantiate the arguments of the following function, which is subsequently asserted as a fact:

  \[
  \text{potconflop(ConflictType, OpName, MO)}
  \]

  Multiple instances of this function represent the policy actions that can potentially cause a conflict when enforced. These actions are associated with a conflict type and a managed object.

- **Step 3**
  Based on the derived actions, the run-time state machines are enquired to determine the states dynamic modules can take as a result of enforcing those actions using the rule below:

  \[
  \text{potconfState(ConflictType, Module, State):-}
  \text{potconflop(ConflictType, OpName, MO), initiates(doAction(operation(MO, OpName, _)), state(Module, State), _).}
  \]

  By querying the Prolog database using this rule, the arguments conflictType, Module, and State are unified and a set of potential conflict-causing states are asserted as facts. The example below represents the state associated with the minimum service rate violation conflict:

  \[
  \text{potconfState(srmInvViolation, slsI, srDecrsd)}
  \]

Once the analysis system has been initialised, the rest of the Prolog source files allow for policy enforcement and subsequently conflict analysis. Their basic functionality is described below:

- As in the case of the static analyser, the mgmtInfoLoader loads the predicates representing system management information (e.g. managed objects and resources), and ecPreds specifies the Event Calculus base predicates and axioms used by the system.
• **dynModlBehvr** contains the Event Calculus representation of the dynamic modules' behaviour by making use of initiates and terminates predicates, which model state transitions.

• **conflictRules** contains the logic predicates that encapsulate the conditions for the occurrence of the various dynamic inconsistencies identified in Section 5.2.

• **eventHandler** specifies a set of rules that define how system events are handled. Apart from state transition and conflict analysis events, the rules can handle user generated events from the client tool. These represent emerging network conditions and lead to the enforcement of policies.

• **polEnforcer** specifies the rule for requesting an action; it evaluates whether a new event triggers the enforcement of a policy operation.

• **moOperations** implements the functionality of the managed objects. The rules specified in this file evaluate requests from the policy enforcer and execute supported operations by deriving new resource configuration values and updating the state of a managed module, as a result of a transition.

• **dynamicAnalyser** implements the dynamic conflict detection and resolution logic. It contains a set of rules for solving the various inconsistencies specified by the **conflictRules** file.

### 6.3.2 Handling Events

As briefly described above, the implementation of the Event Handler manages the various system events that can arise at run-time. These can be of three types:

(a) Network monitoring events generated by the user from the client tool indicating, for example, threshold crossing alarms – `s$IAalarmRaised(tclup, T)`.

(b) State transition events which are generated when executing policies – `state(Module, State)`.

(c) Conflict analysis events which are issued by the dynamic analysis engine after successful detection of a conflict – `conflictDetected(ConflictType, ConflictData)`.

The rules in Listing 6-1 define how the above events are handled. The first rule is the one called from the client tool for type (a) events, but also from a managed object and the dynamic analyser for events of type (b) and (c) respectively. This rule asserts the received event in Prolog's database and subsequently calls the function to handle it (`handleEvent`) by forward chaining. If the received event is a state transition one, the second rule is used to determine a potential association.
with a conflict and the analysis logic is invoked (solve) to perform detection for that conflict. The last rule, which invokes the policy enforcer, is used for handling events of type (a) and (c).

**Listing 6-1: Rules for handling system events**

```prolog
nextEvent(Event, T):-
    assert(happens(systemEvent(Event), T)),
    handleEvent(Event, T).

handleEvent(state(Module, State), T1):-
    potconfstate(ConflicType, Module, State),
    T2 is T1+1,
    solve(conflict(ConflicType, ConflicData, T2)).

handleEvent(Event, T1):-
    T2 is T1+1,
    requestAction(operation(ObjTarg, OpName, Params), T2).
```

### 6.3.3 Enforcing Policies

The rule called by the Event Handler in the case of network monitoring or conflict analysis events is presented in Listing 6-2. The conditional part of rule first checks if the new event triggers a policy. The parameters of the operation term are unified upon a positive match, a request for an action is asserted in Prolog's database, and the associated managed object (ObjTarg) is subsequently invoked to perform the action (doAction).

**Listing 6-2: Rule for requesting an action**

```prolog
requestAction(operation(ObjTarg, OpName, Params), T1):-
    happens(systemEvent(Event), T2),
    initiates(Event, oblig(PolID, ObjSubj, operation(ObjTarg, OpName, Params)), T1),
    T1>T2,
    assert(requestAction(operation(ObjTarg, OpName, Params), T1)),
    T3 is T1+1,
    happens(doAction(operation(ObjTarg, OpName, Params)), T3).
```

The rules in Listing 6-3 complete the enforcement process by implementing the execution of a policy. When called by the enforcer, the first rule checks if the operation of the requested action is supported by the managed object. If so, the doAction predicate is asserted as an event and the next two rules are used to update the state of the module and the value of the resource associated with the enforced policy. The updResrc rule is specific to a policy action; multiple rules of this type implement the functionality of the various QoS management policy actions. The example here concerns the operation of decreasing the service rate of a traffic trunk, in the SLS-I module, by a
percentage (value). The resulting rate (\(\text{NewSR}\)) is asserted in Prolog’s database and is later used by the conflict analysis engine.

The new state of a module, as a result of policy enforcement, is derived by the \texttt{updtstate} rule. Using the previously asserted \texttt{doAction} event, the rule searches the behavioural model of a module to determine if a state transition is caused by this event. Upon a positive match, the new state is passed to the Event Handler.

### Listing 6-3: Rules for executing a policy

```
\begin{verbatim}
happens(doAction(operation(ObjTarg, OpName, Params)), T1):-
    requestAction(operation(ObjTarg, OpName, Params), T2),
    operation(ObjTarg, OpName, Params),
    T1>T2,
    assert(happens(doAction(operation(ObjTarg, OpName, Params)), T1)),
    T3 is T1+1,
    updtResrc(module, OpName, Params, NewVal, T3),
    updtState(module, OpName, State, T3).

updtState(module, OpName, State, T1):-
    happens(doAction(operation(ObjTarg, OpName, Params)), T1),
    initiates(doAction(operation(ObjTarg, OpName, Params))),
    state(module, State),
    nextState(state(module, State), T1),
    NewVal is T1+1,
    updtResrc(module, OpName, State, T1),
    updtState(module, OpName, State, T1).

updtResrc(module, OpName, State, T1):-
    mgdinfo(module, TT, [SR, SRP, NeWSR, T1]),
    SR is SR*Value/100,
    NeWSR is SR-SRP,
    assert(reqSR(TT, NeWSR)).
\end{verbatim}
```

### 6.3.4 Performing Conflict Analysis

Conflict detection logic is realised by a set of rules that perform analysis on a per conflict basis, i.e., one \texttt{solve} rule per conflict type. Based on the conflict with which a state transition event may be associated, the Event Handler invokes the detection process by using the \texttt{solve} predicate in the conditional part of the \texttt{handleEvent} rule. The \texttt{conflictType} parameter, unified to a specific conflict, is subsequently used to search for a rule implemented by the Conflict Analyser that solves that conflict. The first rule of Listing 6-4, for example, solves \texttt{srMinviolation} conflicts. This is achieved by calling the \texttt{srMinviolation} conflict predicate from the specified detection rules and using deductive reasoning to satisfy its goal. If the conflict is successfully detected, a \texttt{conflictDetected} event is asserted in Prolog’s database and is used by the resolution logic (\texttt{findRes} rule) to determine if a policy has been specified for resolving that conflict. Upon a positive match, the conflict event is passed to the Event Handler, which triggers the enforcement of the resolving policy.
6. Tool Support and Experimental Evaluation

Listing 6-4: Rules for conflict analysis

```prolog
solve(conflict(srMinViolation, ConflictData, T1)):-
    conflict(srMinViolation, ConflictData(PolID1, PolID2, TT, SRfs, SR), T1),
    assert(happens(systemEvent(conflictDetected(srMinViolation, ConflictData)), T1)),
    findRes(srMinViolation, ConflictData, T1),
    T2 is T1+1,
    nextEvent(conflictDetected(srMinViolation, ConflictData), T2).

findRes(conflictType, ConflictData, T):-
    happens(systemEvent(conflictDetected(conflictType, ConflictData)), T),
    initiates(systemEvent(conflictDetected(conflictType, ConflictData)), Oblig, T).
```

A practical demonstration of the dynamic analysis implementation is provided in the next section. The example involves the enforcement of a SLS-I policy for decreasing the service rate of a traffic trunk, and shows how the tool developed detects and resolves a `srMinViolation` conflict.

6.4 Analysis Tool Evaluation

This section presents the results of a number of experiments conducted to evaluate the performance and scalability of the static analysis engines developed, and to validate the functionality of the dynamic analysis process. All experiments were performed on a Centrino Duo 2GHz processor with 2GB of RAM, and the subject of the conflict analysis were service management policies.

6.4.1 Static Conflict Detection Performance Analysis

The main aim of the performance evaluation experiments concerning static conflict analysis is to determine the relative times taken to detect inconsistencies among varying numbers of policy specifications. Performance is primarily influenced by the evaluation of a conflict predicate in terms of: (a) the cost in evaluating its conditions, and (b) the number of times it is evaluated. The experiments described below indicate that the number of conflicts only has a minor effect on performance, whereas the number of policies, policy types, and QoS-specific information are the main factors affecting the evaluation of a conflict predicate.

6.4.1.1 Experiment 1

This experiment aims at detecting an increasing number of conflicts among a fixed number of policies. We investigate three conflict types each of which applies to a separate set of 1000 policies: `redundancy` and `qcPriority` conflicts among SLS-S policies for setting the quality level of EF traffic, and multiplexing conflicts between SLS-S policies for setting the service satisfaction
factors of AF1 traffic. The conflict number was varied by introducing more inconsistencies in policy action parameters and policy validity time overlaps. As suggested by the results of Figure 6-9, the number of conflicts does not have a significant impact on the performance with an average of 11% increase in analysis times over a range from 0 to 1000 conflicts detected.

![Graph showing detection performance with varying number of conflicts](Image)

**Figure 6-9: Detection performance with varying number of conflicts**

Since there is no formal method in deriving the number policies required for managing a network, the experiments that follow assume that up to 3000 could be in use. This is a reasonable number of policies for large networks taking into account the number of managed devices, the supported QoS classes and the various constraints that can be used (e.g. hours of the day). These policies are not created all at once when the network is initially deployed but are gradually introduced either on an individual basis or collectively through a refinement process.

### 6.4.1.2 Experiment 2

To investigate the impact of policy types in the analysis, a module-independent conflict is required which can detect the same inconsistency among different policy types and can ultimately provide a uniform basis upon which to compare performance. As such, the second experiment concerns redundancy conflicts detected over different numbers of policies. In the first case only one policy type is used – for setting the quality level of a single QoS class. The number of conflicts, although not having a substantial impact on the performance, is kept constant as the number of policies is varied. The number of times the conflict predicate is evaluated is defined by the number of policies since the detection process iteratively compares each policy with the rest in the set. This can be quantified by equation (1) below, where \( L \) is the number of policy types, and \( N_t \) is the number of policies of a particular type.
Figure 6-10 demonstrates the performance of the detection process where the execution time grows quadratically with respect to the number of policies, namely $O(N^2)$. As suggested by (1), for 2500 policies of a single type the detection predicate is evaluated $1999 \times 10^3$ times, which takes 39 seconds. Introducing more policy types, e.g. for setting service satisfaction factors and the upper limit in the RAB, the performance is significantly improved as the number of comparisons decreases, with all the conditions in the detection predicate only being fully evaluated when matching policy actions are found. For 2500 policies of two and three types, there is a performance improvement of 49% and 66% respectively. These results are validated against the theoretical gain provided by (1), which is 50% and 67%.

![Figure 6-10: Redundancy conflicts – Detection performance against number of policies with varying policy types](image)

6.4.1.3 Experiment 3

Another factor that influences analysis performance is application-specific information. This is particularly important when dealing with QoS management conflicts whose occurrence depends on such information, as for example the number of QoS classes supported and their impact on determining qcPriority conflicts among SLS-S policies setting the service quality level. Equation (2) below can be used to calculate the number of times the relevant predicate is evaluated when detecting such conflicts, where $M$ is the number of QCs involved, $L$ is a counter equal to $M-1$, and $N_t$ and $N_w$ are the number of policies setting the quality level of particular QCs. For an example scenario involving three QCs, EF, AF1, and BE, policies setting the OQL of EF traffic are compared against the ones for AF1 and BE, and those for AF1 traffic against the ones for BE. It
can be shown that $(N_1 \times N_2) + (N_1 \times N_3) + (N_2 \times N_3)$ comparisons are performed, where $N_1$, $N_2$, and $N_3$ represent the number of policies associated with each QC.

$$
\sum_{l=1}^{L} \sum_{m=1}^{M} N_l N_m
$$

(2)

Although of the same complexity of $O(N^2)$ as the previous experiment, the detection process for this conflict type is more expensive as indicated by Figure 6-11, especially with an increasing number of QCs. The experimental results indicate an increase of 35% in detection time between two and three QCs, and 53% between two and four, which are comparable to theoretical values of 33% and 50% obtained by (2) respectively.

**Figure 6-11: Detection performance against number of policies with varying QCs**

### 6.4.1.4 Experiment 4

The last experiment compares the performance of various detection predicates. To provide a meaningful comparison this experiment involves a set of policies of the same type which is prone to more than one inconsistency. We consider redundancy/qcPriority conflicts among service quality policies for two QCs, and redundancy/multiplexing conflicts between almost and full satisfaction factor policies also for two QCs. In the first case the performance of the qcPriority predicate is substantially worse than that of redundancy by an average of 52% over a range of 3000 policies (Figure 6-12), despite the fact that it is evaluated half as many times based on equations (1) and (2). This demonstrates the simplicity in detecting redundancies involving the matching of policy actions, and the cost associated with determining relative priorities between potentially conflicting QCs.
The performance in detecting multiplexing conflicts is the most efficient with nearly 100% gain when compared to a redundancy analysis on the same set of policies. Equation (3) below can be used to calculate the number of multiplexing predicate evaluations, where \( L \) is the number of QCs, and \( N_{AS} \) and \( N_{FS} \) are the number of policies for almost and full satisfaction factors of a specific QC.

\[
\sum_{i=1}^{L} N_{AS} N_{FS}
\]

(3)

For 2500 policies – \( N_{AS} = N_{FS} = 625 \) for each of the two QCs – equation (3) results in 781250 predicate evaluations, which is achieved in 9.7 seconds, whereas double the number of comparisons are required to determine redundancy conflicts in 19.6 seconds.

The last experimental result in Figure 6-12 concerns the sequential execution of all three conflict rules, where half of the policy set consists of service quality policies and the other half is equally split between policies for almost and full satisfaction factors; 2 QCs are involved. The combined performance is better than two of the individual predicate evaluations, which is attributed to the policy set and the decreased number of predicate evaluations. The number of policies associated with the expensive \( qc\text{Priority} \) conflict, for example, has halved resulting to a 75% decrease in evaluations of the relevant predicate.
6.4.2 Dynamic Analysis and Emulated System Behaviour

The analysis tool developed interacts with the emulated dynamic behaviour of on-line modules by enforcing policies to anticipate emerging conditions regarding the network status and eventually handle potential inconsistencies at run-time. In contrast to static detection, dynamic analysis aims at discovering a single inconsistency at a time and enforcing the appropriate resolution policy. For this reason, searching the entire policy space for a conflict may not be required, thus improving the detection performance in comparison to static analysis. The evaluation of this approach is mainly in terms of correct functional behaviour in the event of a conflict.

To demonstrate the functionality of the dynamic analysis engine, we consider a scenario involving the SLS-I module, which is loaded with 100 policies (reasonable number for a single router), and concentrate on managing the service rate of a specific TT. The current allocation for this TT is 120Mbps with the almost and fully satisfied service rates set by SLS-S policies at 100Mbps and 150Mbps respectively. Conflict specifications loaded in the system concern $sr_{MinViolation}$ and $sr_{MaxViolation}$ inconsistencies. The screenshot on Figure 6-13 shows the response of the analysis engine when an upper threshold crossing alarm is received.

![Figure 6-13: Analysis example of a dynamic conflict – detection and resolution](image-url)
The analysis process is initialised at T=0, at which point it sets the state of the SLS-I module to idle and determines potential conflict-causing states from the loaded conflict specifications. In the example above, two such states are determined, which relate to service rate violations (srDecrsd and srIncrsd). At T=1 a user generated threshold crossing alarm is entered in the system, which triggers a policy for decreasing the service rate by 25%. The request for this action is issued by the policy enforcer at T=2, which is subsequently executed at T=4 by calculating the new rate (90Mbps) and deriving the new state of the module (srDecrsd). The latter activates the analysis engine for a potential srMinViolation at T=5, which detects the occurrence of the conflict and also identifies a resolution policy for this inconsistency. At T=6 the conflict detection event is issued, which leads to the execution of the resolution policy. This configures the service at the minimum acceptable (almost satisfied) rate and derives the new state of the module (srSet) at T=9. The cycle is completed with the SLS-I module returning to idle state consuming not more than 10ms. The delay introduced can be argued as being acceptable, even for the strict requirements of EF traffic, as long as conflicts do not occur extremely frequently. The lower part of Figure 6.13 shows the specifics of the detected conflict including an explanation of the inconsistency, the policies involved, and the resolution enforced.

6.5 Summary and Conclusions

This chapter presented the design, implementation and evaluation of the conflict analysis tool that realises the approach for static and dynamic policy analysis. After presenting the overall architecture, a detailed description of the three main components of the tool was provided.

The client application, which provides a graphical interface for the user to create, retrieve and view policies, display management information such as TTs and SLSs, but also to perform conflict analysis among a set of policies loaded in the system, was initially described. Details about the various classes implementing the functionality of the client were provided and the processes of policy input and policy parsing were illustrated with examples. The interface through which the client interacts with the underlying Prolog engines was also presented.

The implementation of the static conflict analyser was then described, which focused on the functionality provided by the various Prolog source files and their collective use in detecting static inconsistencies. A practical example of detecting OQL qcPriority conflicts using the client application was given, demonstrating the ease by which the user can perform queries and the type of output he/she is presented with.

In a similar manner, the Prolog implementation of the dynamic conflict analyser was described giving emphasis on the manner with which the analysis engine can be automatically invoked during system execution in order to detect the occurrence of a conflict and subsequently resolve it.
More specifically, we presented the main rules that are responsible for handling emerging system events, enforcing policies, and performing conflict detection and resolution.

The last part of the chapter described a set of experiments that have been carried out in order to evaluate the tool developed. These allowed to identify the main reasons that influence the performance of the static analysis engines, and also to validate the functional behaviour of the dynamic analysis process. While the number of conflicts in a policy set has only a minor effect on the detection performance, the number of policies in the set, their type, and application-specific information, such as the number of QoS classes supported, constitute the main performance factors. These control the number of conflict predicate evaluations and consequently the analysis times. Furthermore, the various conflict types investigated in the experiments exhibit different execution times for the same number of policies. This is attributed to the different requirements of individual detection predicates when evaluating their conditions.
Chapter 7

7 Conclusions and Future Work

Despite its potential benefits in managing complex systems, policy-based management has so far not been widely adopted. This is mainly due to the shortage of tools and techniques for analysing policies in order to guarantee configuration stability given that policies may have conflicts leading to unpredictable effects.

This thesis presented an approach towards policy conflict analysis based on the formalisation and reasoning provided by Event Calculus and its application in the domain of QoS management for DiffServ networks. The subject of the analysis techniques presented is a set of management policies that can be used to influence/control the behaviour of key modules in the process of QoS provisioning. The various inconsistencies that can arise between these policies have been identified and classified based on their characteristics, which are used to describe the reasons and the conditions under which a conflict will arise.

Conflicts that can occur between policies applied to a single management module, or between policies specified for different modules as a result of their hierarchical relationship have been defined. The main characteristic distinguishing between conflicts however, is the time-frame at which they can be detected. This has driven the design and specification of two different methods to address the issues associated with the analysis of conflicts that can be detected statically, at policy specification-time, and those that can only be determined dynamically, during system execution, based on feedback regarding the current state of the managed system. These techniques have been implemented and integrated in a conflict analysis tool aiming to provide a network administrator with a usable interface through which to interact with the management system and perform both static and dynamic consistency checks.

The sections that follow provide an overview of the contributions, point out potential future directions of this work and conclude this thesis with some remarks.
7.1 Contributions Overview

This section re-visits the main contributions of this work, as described in the introductory part of this thesis, which are discussed further.

It is always beneficial to demonstrate the applicability of a new approach within a valid domain. In this respect, Chapter 3 presented a comprehensive set of QoS provisioning policies that can be used to guide the behaviour of IP DiffServ networks. Although QoS policies have been previously studied in the literature, they were focusing on basic management operations serving as proof of concept for the use of PBM in this domain. The policies defined in this thesis allow for a richer set of programmable operations, they are specified using the format provided by the Ponder language, and their effect on the managed system's functional behaviour has been described. Furthermore, the majority of these policies are generic enough to apply to other QoS and resource management frameworks, where functions for admission control and bandwidth management are essential.

Conflicts specific to the application domain of QoS management primarily occur because of inconsistent attribute values set by policies. It is essential that these are individually identified such that the exact reason for their occurrence can be defined and eventually resolved by a network administrator or in an automated manner. Based on the QoS management policies defined, Chapters 4 and 5 provided a comprehensive review of potential conflicts that may arise between them, and classified those conflicts based on their properties. Furthermore, the rules defining the conditions that will result in conflicts have been specified and were therefore used, together with system-specific information, in analysis processes to determine the presence of inconsistencies. Since most of the defined policies are generic to a certain extent, re-use of the conflict detection rules in other QoS frameworks could be possible.

The principal challenges in detecting policy conflicts are being able to account for the constraints that limit the applicability of a given policy to specific states of the managed system and the effects of enforcing policies on the states of the managed system. To achieve this, it is necessary to use formal reasoning techniques and formal models of the QoS management system behaviour, policy enforcement mechanisms and the policy rules themselves. Since the QoS management system and the policy enforcement mechanisms concerned are event-based reactive systems, Event Calculus has been used as the underlying formal representation. In addition to having built-in representations for events and persistence of properties, Event Calculus is a suitable formalism because it supports both deductive and abductive reasoning. The latter has been used when analysing for static inconsistencies, as described in Chapter 4, and provides the means to not only identify a conflict but also generate an explanation as to how that conflict occurred. This is
particularly important when guiding a network administrator to handle inconsistencies requiring manual resolution, as in the case of the static conflicts identified.

Despite the fact that the resolution of static conflicts is performed manually, this process takes place before policies are deployed in the system and does not impose any run-time overheads on the functionality of on-line modules. Dynamic conflicts however, require system components to both detect and resolve conflicts in real-time, without degrading the performance of the system. This has been the main motivation behind the dynamic analysis approach presented in Chapter 5, which provides an automated run-time analysis process. This can be automatically invoked based on run-time network events, can provide a resolution if a conflict materialises, and also instruct the appropriate entity for the enforcement of that resolution. The latter is in the form of pre-defined policies that are generic enough with only few required for each conflict type to cater for multiple occurrences of the same inconsistency.

The approach has been implemented in an integrated tool supporting both static and dynamic conflict analysis. As described in Chapter 6, the developed client application hides the complexity of the underlying logic formalism and reasoning techniques, allowing the user to perform analysis queries in a simplified manner and presenting the results in an easily understood form. Finally, the tool has been used to perform extensive experiments through which it was possible to identify the main reasons that influence the performance of the static analysis engines; the correct operation of the dynamic analysis process was also validated with a case study involving a service management conflict.

7.2 Future Research Directions

This section presents ways in which the work in this thesis can be extended. Potential future directions are summarised below:

- **Enhancements to the tool**: Although not very research oriented, some enhancements to the tool would further simplify its use. The conflict rules and the system behaviour representations are currently specified manually by a user who is familiar with the underlying logic formalism. An interface could be developed to input conflict rules in a higher level format closer to the user’s understanding, which are subsequently translated to Event Calculus representations. A similar translation approach could be developed for system behaviour specifications which can be derived from UML state diagrams.
• **Integration with policy refinement**: The analysis approach is currently performed on a set of policies that can be derived from high-level goals through a refinement process. Static conflict detection could be an integral part of policy refinement since both are offline processes and achieve complementary objectives. A solution where potential conflict states are avoided at every step of the refinement process is envisaged.

• **Inter-domain QoS policy analysis**: The conflicts identified in this thesis concern policies that manage QoS within a single administrative domain. A challenging future direction is in the domain of collaborative QoS management, where neighbouring network providers set-up service-level agreements aiming to create an end-to-end chain for the delivery of QoS sensitive applications. The negotiation process is envisaged to be one where each provider tries to force its own policies in terms of requirements and objectives resulting in conflicting situations. A collaborative negotiating process would act as a mediator where an optimal solution, satisfying both entities, would be achieved through conflict analysis. Initial work has been done by investigating game theoretic approaches for the negotiation and policy harmonisation processes.

• **Performance evaluation**: Further experimentation with the dynamic analysis engines involving a bigger number of policies could be performed. Separate performance evaluations for the detection and resolution phases could be carried out, and the main factors influencing the performance of both could be identified. Furthermore, network traces could be used as input to the analysis process so that the frequency of run-time inconsistencies under realistic conditions, as well as possible conflict patterns, can be determined.

• **Dynamic conflict analysis**: The dynamic analysis approach could be extended by considering the requirements of more volatile application environments where policies are continuously changing to achieve adaptation based on emerging conditions. Initial work in ubiquitous networked environments has been carried out, which focused on conflicts arising between policies originating from different managing entities [108]. A mechanism to detect such conflicts has been developed, which employs an automated resolution process once a resolution strategy for each conflict type is agreed. The strategy depends on the contractual agreement between management entities and/or the business model of the managed network.
7.3 Epilogue

It is evident that there is limited value in developing policy-based management systems that do not provide support for conflict detection and resolution. The importance of conflict analysis is accentuated by the recent view of policies as the enabling technology in autonomic networking where the diversity of programmable management functions can be considerably high. Future adoption of autonomic networking principles will require support for conflict analysis so that stability in the control loop can be ensured.

Research has shown that with conflict analysis approaches not taking into account the system state, which, in most cases, constrains the applicability of policies, only simple checks can be carried out for the correctness of policies. As described in this thesis, formal logic notations can address this issue and also facilitate reasoning techniques to effectively analyse for conflicts. The ideas presented contribute to the effort for a wider uptake of policy-based management as a technology that can potentially simplify the management of complex systems.
Bibliography


Appendix A1 – Service Management Static Conflict Rules

conflict(subscrAdmStrg, conflictData(PolID1, PolID2, Strg1, Strg2, Val1, Val2), TC) ⇐
holdsAt(oblig(PolID1, Subj, operation(Targ, defSuStrg, [Strg1, Val1])), TC) ∧
holdsAt(oblig(PolID2, Subj, operation(Targ, defSuStrg, [Strg2, Val2])), TC) ∧
moreConsrvty(Strg1, Strg2) ∧ Val1>Val2 ∧
polID1==polID2.

conflict(mltplexQCPrior, conflictData(PolID1, PolID2, QC1, QC2, MF1, MF2), TC) ⇐
holdsAt(oblig(PolID1, Subj, operation(Targ, setAlmstSatisf, [QC1, MF1])), TC) ∧
holdsAt(oblig(PolID2, Subj, operation(Targ, setAlmstSatisf, [QC2, MF2])), TC) ∧
priority(QC1, Prtl) ∧ priority(QC2, Prt2) ∧
((Prtl>Prt2 ∧ MF1>MF2) v
(Prt2>Prtl ∧ MF2>MF1)) ∧
polID1==polID2.

conflict(invcAdmStrg, conflictData(PolID1, PolID2, TT1, Val1, Val2), TC) ⇐
holdsAt(oblig(PolID1, Subj, operation(initmo, setACmin, [TT1, Val1])), TC) ∧
holdsAt(oblig(PolID2, Subj, operation(monitormo, setTCL, [TT2, Val2])), TC) ∧
TT1==TT2 ∧ Val1>Val2.

conflict(thrshlncompat, conflictData(PolID1, PolID2, TT1, Val1, Val2), TC) ⇐
holdsAt(oblig(PolID1, Subj, operation(Targ, setTCL, [TT1, Val1])), TC) ∧
holdsAt(oblig(PolID2, Subj, operation(Targ, setVCL, [TT2, Val2])), TC) ∧
TT1==TT2 ∧ Val1>Val2.

conflict(thrshQCPrior, conflictData(PolID1, PolID2, TT1, TT2, Val1, Val2), TC) ⇐
holdsAt(oblig(PolID1, Subj, operation(Targ, setTCL, [TT1, Val1])), TC) ∧
holdsAt(oblig(PolID2, Subj, operation(Targ, setTCL, [TT2, Val2])), TC) ∧
mgdInfo(slsS, TT1, QC1, -1) ∧
mgdInfo(slsS, TT2, QC2, -1) ∧
priority(QC1, Prtl) ∧ priority(QC2, Prt2) ∧
((Prtl>Prt2 ∧ Val1>Val2) v
(Prt2>Prtl ∧ Val2>Val1)) ∧
polID1==polID2.
Appendix A2 – Traffic Engineering Static Conflict Rules

<table>
<thead>
<tr>
<th>Rule Description</th>
<th>Input Data</th>
<th>Output Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conflict (Alt Path Exceed)</td>
<td>PolID1, PolID2, TT, IC, PathNum</td>
<td>TC</td>
</tr>
<tr>
<td>Conflict (BW Rt Violation)</td>
<td>PolID1, PolID2, QC1, BW1, BW2</td>
<td>TC</td>
</tr>
<tr>
<td>Conflict (NDQ Prior)</td>
<td>PolID1, PolID2, QC1, Prtl, QC2, Prt2</td>
<td>TC</td>
</tr>
<tr>
<td>Conflict (Spare BW)</td>
<td>PolID1, PolID2, PolID3, BW1, BW2, BW3</td>
<td>TC</td>
</tr>
<tr>
<td>Conflict (Exc SBW)</td>
<td>PolID1, PolID2, PolID3, BW1, BW2, BW3</td>
<td>TC</td>
</tr>
<tr>
<td>Conflict (MinMax BW)</td>
<td>PolID1, PolID2, PolID3, BW1, BW2, BW3</td>
<td>TC</td>
</tr>
</tbody>
</table>

**Conflicts (Alt Path Exceed)**
\[
\text{conflict(altPathExceed, conflictData(PolID1, PolID2, TT, IC, PathNum), TC) } \\
\text{<-- holdsAt(oblig(PolID1, Subj, operation(Targ, setMAAltPaths, [QC1, TT, PathNum]), TC) } \\
\text{^ holdsAt(oblig(PolID2, Subj, operation(Targ, setupLSP, [QC2, TT, Path, BW]), TC) } \\
\text{^ instCount(operation(Targ, setupLSP, [QC2, TT, _ , _]), IC) } \\
\text{^ IC>PathNum ^ QC1==QC2.}
\]

**Conflicts (BW Rt Violation)**
\[
\text{conflict(bwRtViolation, conflictData(PolID1, PolID2, QC1, BW1, BW2), TC) } \\
\text{<-- holdsAt(oblig(PolID1, subj, operation(bAMO, setNDmax, [QC1, BW1]), TC) } \\
\text{^ holdsAt(oblig(PolID2, subj, operation(IspMO, setupLSP, [QC2, TT, Path, BW2]), TC) } \\
\text{^ BW2>BW1 ^ QC1==QC2.}
\]

**Conflicts (NDQ Prior)**
\[
\text{conflict(ndqCPrior, conflictData(PolID1, PolID2, QC1, Prtl, QC2, Prt2), TC) } \\
\text{<-- holdsAt(oblig(PolID1, subj, operation(Targ, opi, [QC1]), TC) } \\
\text{^ holdsAt(oblig(PolID2, subj, operation(Targ, op2, [QC2]), TC) } \\
\text{^ meOps(mutexHopCountCalc, MutexOps) } \\
\text{^ member(op1, MutexOps) ^ member(op2, MutexOps) } \\
\text{^ priority(QC1, Prtl) ^ priority(QC2, Prt2) } \\
\text{^ ((Prt1>Prt2 ^ moreConsrvtv(Op2, Op1)) v } \\
\text{^ (Prt2>Prt1 ^ moreConsrvtv(Op1, Op2))) ^ polID1==polID2.}
\]

**Conflicts (Spare BW)**
\[
\text{conflict(spareBW, conflictData(PolID1, PolID2, PolID3, BW1, BW2, BW3, BW), TC) } \\
\text{<-- holdsAt(oblig(PolID1, subj, operation(Targ, allocSpareBWExp1, [ef, BW1]), TC) } \\
\text{^ holdsAt(oblig(PolID2, subj, operation(Targ, allocSpareBWExp1, [af, BW2]), TC) } \\
\text{^ holdsAt(oblig(PolID3, subj, operation(Targ, allocSpareBWExp1, [be, BW3]), TC) } \\
\text{^ sumOf(BW1, BW2, BW3, BW) ^ BW>100.}
\]

**Conflicts (Exc SBW)**
\[
\text{conflict(excSBW, conflictData(PolID1, PolID2, PolID3, BW1, BW2, BW3, BW), TC) } \\
\text{<-- holdsAt(oblig(PolID1, subj, operation(Targ, redOverBWExp1, [ef, BW1]), TC) } \\
\text{^ holdsAt(oblig(PolID2, subj, operation(Targ, redOverBWExp1, [af, BW2]), TC) } \\
\text{^ holdsAt(oblig(PolID3, subj, operation(Targ, redOverBWExp1, [be, BW3]), TC) } \\
\text{^ sumOf(BW1, BW2, BW3, BW) ^ BW<100.}
\]

**Conflicts (MinMax BW)**
\[
\text{conflict(minMaxBW, conflictData(PolID1, PolID2, PolID3, BW1, BW2, BW3, BW), TC) } \\
\text{<-- holdsAt(oblig(PolID1, subj, operation(Targ, setNDmax, [ef, BW1]), TC) } \\
\text{^ holdsAt(oblig(PolID2, subj, operation(Targ, setNDmax, [af, BW2]), TC) } \\
\text{^ holdsAt(oblig(PolID3, subj, operation(Targ, setNDmax, [be, BW3]), TC) } \\
\text{^ sumOf(BW1, BW2, BW3, BW) ^ BW>100.}
\]
Appendix A3 – Service Management Dynamic Conflict Rules

\[
\text{conflict}(srMinViolation, \text{conflictData}(\text{PolID1}, \text{PolID2}, TT1, SRas, SR), TC) \leftarrow \\
\text{holdsAt}(\text{oblig}(\text{PolID1}, \text{Subj}, \text{operation(srvAdjustMO, decrSR, [TT1, Val1]})), TC) \wedge \\
\text{holdsAt}(\text{oblig}(\text{PolID2}, \text{Subj}, \text{operation(initMO, setSRas, [TT2, SRas]})), TC) \wedge \\
\text{reqSR}(TT1, SR) \wedge SR<SRas \wedge TT1==TT2.
\]

\[
\text{conflict}(invCAdmstrg2, \text{conflictData}(\text{PolID1}, \text{PolID2}, TT1, ACmax, VCL), TC) \leftarrow \\
\text{holdsAt}(\text{oblig}(\text{PolID1}, \text{Subj}, \text{operation(srvAdjustMO, decrACmax, [TT1, Val1]})), TC) \wedge \\
\text{holdsAt}(\text{oblig}(\text{PolID2}, \text{Subj}, \text{operation(monitorMO, setVCL, [TT2, VCL]})), TC) \wedge \\
\text{reqACmax}(TT1, ACmax) \wedge ACmax<VCL \wedge TT1==TT2.
\]

Appendix A4 – Traffic Engineering Dynamic Conflict Rules

\[
\text{conflict}(ndMinViolation, \text{conflictData}(\text{PolID1}, \text{PolID2}, QCl, Link, BW, NDmin), TC) \leftarrow \\
\text{holdsAt}(\text{oblig}(\text{PolID1}, \text{Subj}, \text{operation(bamo, decrAlloc, [Link, Qc1, Val1]})), TC) \wedge \\
\text{holdsAt}(\text{oblig}(\text{PolID2}, \text{Subj}, \text{operation(initMO, setAllocMin, [Qc2, NDmin]})), TC) \wedge \\
\text{reqBW}(Link, Qc1, BW) \wedge BW<NDmin \wedge Qc1==Qc2.
\]

\[
\text{conflict}(underAlloc, \text{conflictData}(\text{PolID1}, \text{PolID2}, Link, QC, BW, Alloc), TC) \leftarrow \\
\text{holdsAt}(\text{oblig}(\text{PolID1}, \text{Subj}, \text{operation(bamo, decrAlloc, [Link, QC, Val]})), TC) \wedge \\
\text{holdsAt}(\text{oblig}(\text{PolID2}, \text{Subj}, \text{operation(initMO, setRsCAlloc, [Link, Alloc]})), TC) \wedge \\
\text{totalReqBW}(Link, BW) \wedge BW<Alloc.
\]

\[
\text{conflict}(thriShIncompat, \text{conflictData}(\text{PolID1}, \text{PolID2}, QCl, Link, BW, ThUp), TC) \leftarrow \\
(\text{holdsAt}(\text{oblig}(\text{PolID1}, \text{Subj}, \text{operation(bamo, incrAlloc, [Link, Qc1, Val1]})), TC) \wedge \\
\text{holdsAt}(\text{oblig}(\text{PolID2}, \text{Subj}, \text{operation(monitorMO, incrThs, [Link, QC2, Val2]})), TC)) \vee \\
(\text{holdsAt}(\text{oblig}(\text{PolID1}, \text{Subj}, \text{operation(bamo, decrAlloc, [Link, QC1, Val1]})), TC) \wedge \\
\text{holdsAt}(\text{oblig}(\text{PolID2}, \text{Subj}, \text{operation(monitorMO, decrThs, [Link, QC2, Val2]})), TC)) \wedge \\
\text{reqBW}(Link, Qc1, BW) \wedge reqThUppr(Link, QC2, ThUp) \wedge \\
\text{ThUp}>BW \wedge QC1==QC2.
\]

147